Category Theory Applied to Functional Programming

Juan Pedro Villa Isaza

Departamento de Informática y Sistemas
Escuela de Ingeniería
Universidad EAFIT
Medellín, Colombia
2014

"Why is a raven like a writing-desk?"

—Carroll (2004, p. 79)

Category Theory Applied to Functional Programming

Juan Pedro Villa Isaza

Systems Engineering undergraduate project

Supervisor: Andrés Sicard Ramírez

Departamento de Informática y Sistemas
Escuela de Ingeniería
Universidad EAFIT
Medellín, Colombia
2014

To Abuelita

Acknowledgements

I'm very grateful to my supervisor, Andrés Sicard Ramírez, especially for introducing me to category theory and functional programming, to my parents, Ana María Isaza Builes and Carlos Fernando Villa Gómez, and to my boyfriend and proofreader, Nicolás Arbeláez Estrada. I thank you all for your support and encouragement, and for not running out of patience with me. And last but not least, I'm very thankful to the examiners, Francisco José Correa Zabala and Juan Francisco Cardona McCormick, to the previous and current Systems Engineering undergraduate projects coordinators, Hernán Darío Toro Escobar and Edwin Nelson Montoya Múnera, respectively, to the EAFIT Logic and Computation seminar 2013 attendees, and to my psychologist, María Paula Valderrama López.

Errata

- February 15, 2019: In example 5.2.4, `concat` is a natural transformation from the `[] . []` functor and into the `[]` functor (instead of from and into the `[]` functor). Thanks to Paulo Villela for reporting!

Deixa-te levar pela criança que foste.

—Saramago (2006)

Sempre chegamos ao sítio aonde nos esperam.

—Saramago (2008)

Abstract

We study some of the applications of category theory to functional programming, particularly in the context of the Haskell functional programming language, and the Agda dependently typed functional programming language and proof assistant. More specifically, we describe and explain the concepts of category theory needed for conceptualizing and better understanding algebraic data types and folds, functors, monads, and parametrically polymorphic functions. With this purpose, we give a detailed account of categories, functors and endofunctors, natural transformations, monads and Kleisli triples, algebras and initial algebras over endofunctors, among others. In addition, we explore all of these concepts from the standpoints of categories and programming in Haskell, and, in some cases, Agda. In other words, we examine functional programming through category theory.

Keywords: Agda, category theory, functional programming, Haskell.

Contents

Chapter 1

Introduction

> Race de Caïn, au ciel monte,
> Et sur la terre jette Dieu !
>
> —Baudelaire (1857, v. 16)

Category theory is a branch of mathematics developed in the 1940s which has come to occupy a central position in computer science. Broadly, it is a convenient and powerful conceptual framework for structures and systems of structures (Mac Lane 1998, p. vii; Marquis 2013, p. 1; Wolfram 2002, p. 1154).

In computer science, category theory allows us to formulate definitions and theories of concepts, or to analyze the coherence of existing formulations, to carry out proofs, to discover and exploit relations with other fields, to deal with abstraction and representation independence, to formulate conjectures and research directions, and to unify concepts (Goguen 1991, pp. 49–50). Moreover, category theory contributes to areas such as automata theory, domain theory, functional programming, polymorphism, semantics, type theory, among others (Marquis 2013, p. 23; Pierce 1991, p. xi; Poigné 1992, p. 415). In particular, category theory can be viewed as a formalization of operations on data types and as a foundational theory of functions, which provides a sound basis for functional programming (Poigné 1992, p. 414; Wolfram 2002, p. 1154).

According to Elkins (2009, p. 73) and Yorgey (2009, pp. 50–51), several concepts of functional programming languages, such as Agda (Norell 2007; The Agda Team 2014), Haskell (Peyton Jones 2003), Miranda (Turner 1985), ML (Milner 1984), among others, are based on concepts of category theory, but one can be a perfectly competent functional programmer without knowledge of these theoretical foundations. In spite of that, category theory can be

applied to functional programming with the purpose of, for instance, better understanding concepts such as algebraic data types, applicative functors, functors, monads, and polymorphism, and thus becoming a better programmer.

In this regard, we aim to explore the category-theoretical foundations of some of the concepts of functional programming listed above. In other words, we aim to examine functional programming through category theory.

1.1 Summary of the Project

This is an undergraduate project submitted in partial fulfillment of the requirements for the degree of Systems Engineering at EAFIT University[1]. As a summary of the project proposal, our objective is to study some of the applications of category theory to functional programming in Haskell and Agda. More specifically, our goal is to describe and explain the concepts of category theory needed for conceptualizing and better understanding algebraic data types, functors, monads, and polymorphism.

1.2 Audience and Prerequisites

The reader of this project is a mathematically inclined functional programmer. We assume a working knowledge of mathematics and functional programming in Haskell and Agda. If any further background material is required, some suggestions can be found in the references in Section 1.4 or in the references at the end of each chapter.

1.3 Overview of the Project

This project is divided into six chapters organized like the tree diagram in Figure 1.1.

- In Chapter 2 (Categories), we define categories and commutative diagrams, and the categories which will allow us to relate category theory to functional programming in Haskell and Agda.

- In Chapter 3 (Constructions), we describe some basic constructions in categories (isomorphisms, initial and terminal objects, and prod-

[1] http://www.eafit.edu.co.

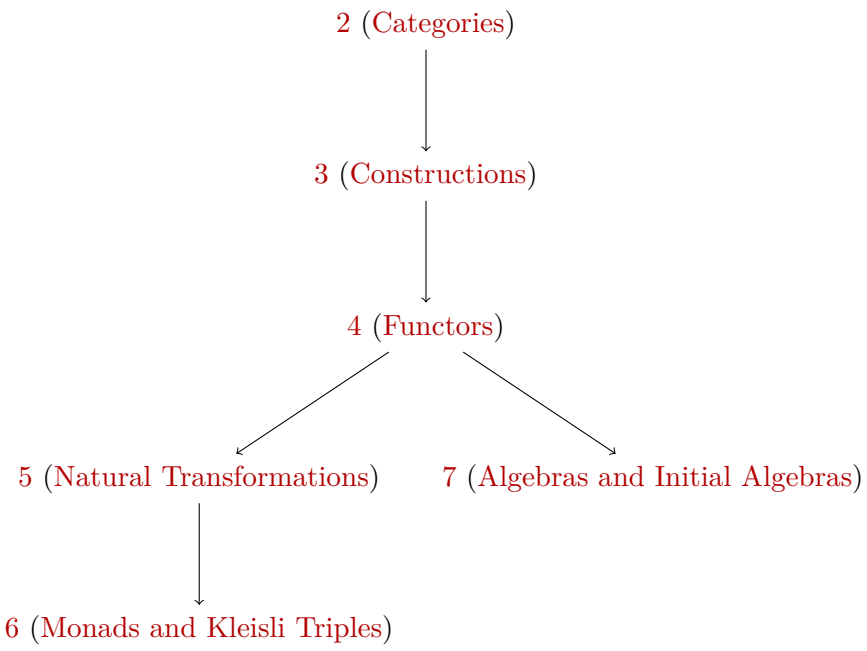


Figure 1.1: Overview of the project.

ucts and coproducts), which we use for describing some concepts and examples in all subsequent chapters.

- In Chapter 4 (Functors), we study functors and endofunctors in order to conceptualize and better understand functors in Haskell and Agda.
- In Chapter 5 (Natural Transformations), we explore the connection between natural transformations and polymorphic functions in Haskell.

  This chapter completes the trinity of concepts category, functor, and natural transformation, which are the basis of category theory (Mac Lane 1998, p. vii).
- In Chapter 6 (Monads and Kleisli Triples), we give a detailed account of monads and Kleisli triples, and their correspondence to monads in Haskell and Agda.
- In Chapter 7 (Algebras and Initial Algebras), we describe algebras and initial algebras over endofunctors, and their relation to algebraic data types in Haskell.

- Finally, Chapter 8 (Conclusions) contains our conclusions and some ideas for future work.

Each chapter, except Chapter 3, is further subdivided into two or three sections concerning some concepts of category theory, and their relation to functional programming in Haskell and, in some cases, Agda. Moreover, each chapter ends with a section of references which collects the sources of information for all definitions and examples.

1.4 References

This section includes a list of suggestions for further reading, as well as some bibliographical remarks. For more thorough category-theoretical reading guides, see (Marquis 2013, pp. 48–56; Pierce 1991, § 4).

Category theory

- Most of our definitions are based on (Mac Lane 1998), which is a standard reference on category theory, as well as (Awodey 2010).

- Marquis (2013) includes an interesting description of the history of category theory, some of its applications, an analysis of its philosophical significance, and, perhaps more relevant, a useful programmatic reading guide.

- As far as history is concerned, categories, functors, and natural transformations were discovered by Eilenberg and MacLane (1942), but category theory was devised in (Eilenberg and MacLane 1945). Some of our definitions were compared with those of the latter.

- Bird and de Moor (1997) is a standard reference on the applications of category theory to computer science.

- Some of our definitions and examples are based on (Pierce 1991), which covers the basic concepts of category theory and some of its applications to computer science. Besides, it includes a chapter with interesting reading suggestions.

- Many of our definitions and some of our examples are based on (Poigné 1992), which is a complete reference on the applications of category theory to computer science.

- Some works on category theory are not easily accessible. Fortunately, Reprints in Theory and Applications of Categories[2] contains some interesting and useful books, including (Adámek, Herrlich, and Strecker 2006; Barr and Wells 2005; Barr and Wells 2012).

- The *n*Lab[3], a wiki-lab for collaborative work on mathematics, physics, and philosophy, is a helpful source of information about category theory.

Haskell

- Among the many tutorials on Haskell, (Lipovača 2011; O'Sullivan, Goerzen, and Stewart 2008) are highly recommended.

- Yorgey (2009) is an introduction to the standard Haskell type classes, including categories, functors, and monads, and (Elkins 2009) is an interesting article about how to use category theory in Haskell.

- For more information, see the Haskell wiki[4].

Agda

- For an introduction to Agda, see (Bove and Dybjer 2009; Norell 2009).

- For more information, see the Agda wiki[5].

Additional references

- Weisstein (2014), which is an extensive mathematics resource, was frequently consulted for supplementary information.

- Some Stack Exchange[6] sites, notably MathOverflow and Stack Overflow, were consulted during the development of this project.

[2] http://www.tac.mta.ca/tac/reprints/.
[3] http://ncatlab.org/nlab/.
[4] http://www.haskell.org.
[5] http://wiki.portal.chalmers.se/agda/.
[6] http://stackexchange.com.

1.5 Notes

Frontispiece The illustration in page ii, John Tenniel's *Hatter engaging in rhetoric*, is taken from the Tenniel illustrations for Lewis Carroll's *Alice's Adventures in Wonderland*[7].

Haskell and Agda code

- The Haskell code was tested with GHC 7.6.3. Most of it corresponds to or is based on standard Haskell code, and can be used as it is.
- The Agda code was tested with Agda 2.3.2.2 and the Agda standard library 0.7 (Danielsson et al. 2013). We omit a lot of details such as import declarations.
- All the code can be found in a Git repository which is available at `https://github.com/jpvillaisaza/abel`.

Links

- This document is available for download at `http://bit.ly/1cq5fwN`.
- A printable version is available at `http://bit.ly/1hDomqv`.

For more information, send an email to `jvillai@eafit.edu.co`.

[7] `http://www.gutenberg.org/ebooks/114`.

Chapter 2

Categories

> And God saw every thing that he had made, and, *behold*, it was very good.
>
> —God (1769, Genesis 1:31)

In this chapter we study categories in order to be able to study functors and natural transformations, which are the basic concepts of category theory. More interestingly, we see that, up to a point, the types and functions of a functional programming language such as Haskell yield a category which will allow us to relate category theory to functional programming.

To motivate categories, consider the subset of Haskell types and functions depicted by the diagram in Figure 2.1, excluding composite functions. More specifically, as types, take the unit type, `()`, the Boolean type, `Bool`, and the natural number type, `Nat`[1], and, as functions, take the constants-as-functions `()`, `True`, `False`, and `Zero`, the functions `Succ`, `isZero`, and `not`, the identity functions, `id`, and composite functions such as `not . True`.

Now, observe that, according to the semantics of the language, some of the functions, such as `isZero . Zero` and `True`, are identical. Additionally, we could prove, for instance, that `id . Succ = Succ = Succ . id`, which exemplifies that identity functions are identities for the composition of functions, which is associative.

In terms of category theory, the types and functions of this subset of Haskell represent the objects and morphisms of a category, respectively, and the statement that composition of functions is associative and has identities means that we are in fact dealing with a category.

[1] Note that, at least in GHC 7.6.3, Haskell does not have a natural number data type by itself.

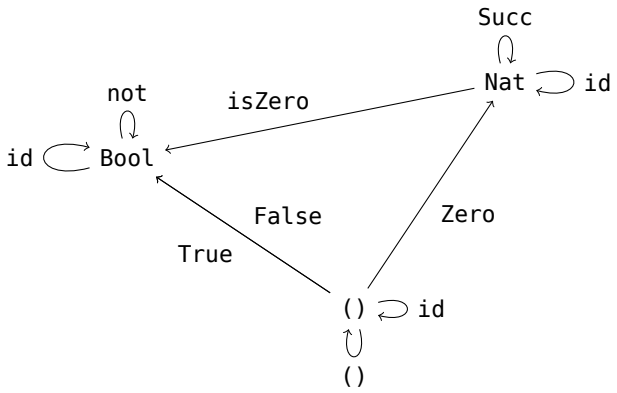


Figure 2.1: A subset of Haskell.

In the remainder of this chapter, we define the concepts of category and commutative diagram, and give some examples of categories, including the category of sets and functions. In addition, we describe categories of types and functions for Haskell and Agda.

2.1 Categories

We begin with the concept of category, which will be found everywhere in our development.

Definition 2.1. A category $\mathcal{C}$ consists of:

- Objects a, b, c, ...
- Morphisms or arrows f, g, h, ...
- For each morphism f, domain and codomain objects $a = \mathrm{dom}(f)$ and $b = \mathrm{cod}(f)$, respectively. We then write $f : a \to b$.
- For each object a, an identity morphism $\mathrm{id}_a : a \to a$.
- For each pair of morphisms $f : a \to b$ and $g : b \to c$, a composite morphism $g \circ f : a \to c$. That is, for each pair of morphisms f and g with $\mathrm{cod}(f) = \mathrm{dom}(g)$, a composite morphism $g \circ f : \mathrm{dom}(f) \to \mathrm{cod}(g)$. We may then draw a diagram like that of Figure 2.2.

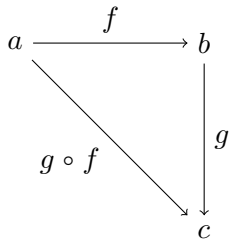


Figure 2.2: Composition of morphisms.

Composition of morphisms associates to the right. Therefore, for all morphisms $f : a \to b$, $g : b \to c$, and $h : c \to d$, $h \circ g \circ f$ denotes $h \circ (g \circ f)$.

The category is subject to the following axioms:

- For all morphisms $f : a \to b$, $g : b \to c$, and $h : c \to d$,

$$h \circ (g \circ f) = h \circ g \circ f = (h \circ g) \circ f, \tag{2.1}$$

that is, composition of morphisms is associative or, equivalently, the diagram in Figure 2.3a is commutative.

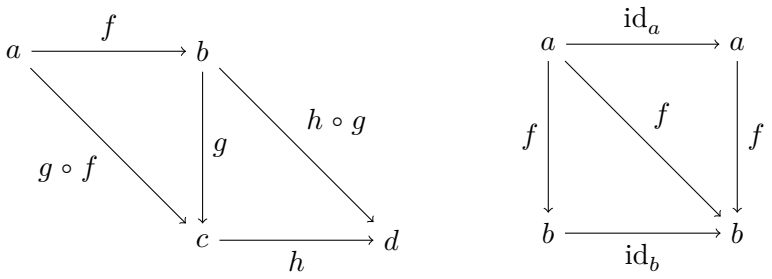
- For all morphisms $f : a \to b$,

$$\mathrm{id}_b \circ f = f = f \circ \mathrm{id}_a\,, \tag{2.2}$$

that is, identity morphisms are identities for the composition of morphisms or, equivalently, the diagram in Figure 2.3b is commutative.

> "And what is the use of a book without pictures or conversations?"
>
> —Carroll (2004, p. 13)

Definition 2.2. We often use diagrams consisting of objects and morphisms of a category, like those of Figures 2.2 and 2.3. Such a diagram in a category $\mathcal{C}$ is said to be commutative, or to commute, when, for each pair of objects a and b, any two paths leading from a to b yield, by composition, equal



(a) The associativity axiom. (b) The identity axiom.

Figure 2.3: Axioms for categories.

morphisms from a to b. For instance, if we say that the diagram in Figure 2.4 is commutative, we mean that

$$g' \circ f' = g \circ f.$$

In this way, "commutative diagrams are just a convenient way to visualize equalities of morphisms" (Poigné 1992, p. 434).

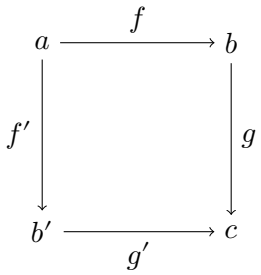


Figure 2.4: A commutative square.

Remark 2.1. Moreover, commutative diagrams are closed under composition of diagrams in that a diagram commutes if all its subdiagrams commute. For example, if we say that the inner triangles of the diagram in Figure 2.5 commute, then the outer diagram commutes as well, that is, if

$$f' = h \circ f \quad \text{and} \quad g = g' \circ h,$$

then

$$g' \circ f' = g' \circ h \circ f = g \circ f.$$

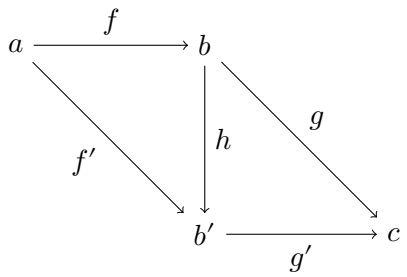


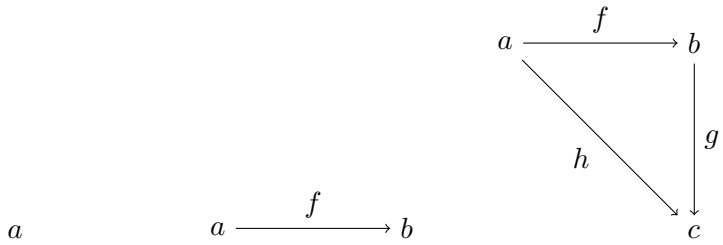
Figure 2.5: Commutative triangles.

As examples, we consider some common categories, including the empty category, the trivial category, discrete categories, the category of sets and functions, which is the motivating example of category theory, monoids, which help to better understand the idea of sets with structure as categories, and deductive systems, which is an interesting change of perspective.

Example 2.1.1. The empty category, **0**, has neither objects nor morphisms. It looks like, well, nothing. The trivial category, **1**, has one object and one (identity) morphism. It looks like the diagram in Figure 2.6a, which is a diagram of a category, excluding the identity morphisms. The category **2** has two objects, two identity morphisms and one (non-identity) morphism which looks like the morphism of the category in Figure 2.6b. And the category **3** has three objects, three identity morphisms, and three (non-identity) morphisms which look like the morphisms of the category in Figure 2.6c. In each case, there is only one way to define composition of morphisms, and the axioms for categories obviously hold.

Example 2.1.2. A discrete category is a category whose only morphisms are the identity morphisms. Given a set A, we get a discrete category $\mathcal{C}$ in which the objects are the elements of A and the morphisms are the identity morphisms, one for each $x \in A$, which are uniquely determined by the identity axiom. A discrete category is so determined by its objects, which correspond exactly to its identity morphisms.

Example 2.1.3. Set is the category of sets and functions. Its objects are sets A, B, C, ..., and its morphisms are functions f, g, h, ... Each function



(a) The category **1**. (b) The category **2**. (c) The category **3**.

Figure 2.6: Trivial categories.

$f : A \to B$ is composed of a domain $A = \mathrm{dom}(f)$, a codomain or range $B = \mathrm{cod}(f)$, and a rule which assigns to each element $x \in A$ an element $f(x) \in B$. For each set A, there is an identity function $\mathrm{id}_A : A \to A$ such that, for all $x \in A$,

$$\mathrm{id}_A(x) = x, \tag{2.3}$$

and, for each pair of morphisms $f : A \to B$ and $g : B \to C$, there is a composite function $g \circ f : A \to C$ such that, for all $x \in A$,

$$(g \circ f)(x) = g(f(x)). \tag{2.4}$$

Now, let us prove that this is a category. In the first place, we prove that the associativity axiom holds for **Set**. Given functions $f : A \to B$, $g : B \to C$, and $h : C \to D$, then, for all $x \in A$:

$$\begin{aligned}
&(h \circ g \circ f)(x) \\
&\quad = \quad \text{(by (2.4))} \\
&h((g \circ f)(x)) \\
&\quad = \quad \text{(by (2.4))} \\
&h(g(f(x))) \\
&\quad = \quad \text{(by (2.4))} \\
&(h \circ g)(f(x)) \\
&\quad = \quad \text{(by (2.4))} \\
&((h \circ g) \circ f)(x)
\end{aligned}$$

In the second place, we prove that the identity axiom holds for **Set**. Given a function $f : A \to B$, then, for all $x \in A$:

$$
\begin{aligned}
&(\mathrm{id}_B \circ f)(x) \\
&\quad = \quad \text{(by (2.4))} \\
&\mathrm{id}_B(f(x)) \\
&\quad = \quad \text{(by (2.3))} \\
&f(x) \\
&\quad = \quad \text{(by (2.3))} \\
&f(\mathrm{id}_A(x)) \\
&\quad = \quad \text{(by (2.4))} \\
&(f \circ \mathrm{id}_A)(x)
\end{aligned}
$$

Remark 2.2. To some extent, we are considering the objects and morphisms of **Set** to be the sets of all sets and all functions, respectively, which would lead to a paradox such as the set of all sets not members of themselves. For this reason, we ought to assume, for instance, that there is a big enough set, the universe, and take the objects of **Set** to be the sets which are members of the universe, that is, small sets. However, we shall not go into detail about mathematical foundations of category theory[2].

Example 2.1.4. A monoid is a category with just one object. A monoid is thus determined by its morphisms, its identity morphism, and its rule for the composition of morphisms. A monoid may then be described as a usual monoid, that is, as a set (of morphisms) that is closed under an associative binary operation which has an identity (morphism). More formally, given a category $\mathcal{C}$ with just one object a, we get a usual monoid $C = (\mathcal{C}_{\mathrm{M}}, \circ, \mathrm{id}_a)$ where the elements of $\mathcal{C}_{\mathrm{M}}$ are the morphisms of $\mathcal{C}$. Conversely, given a usual monoid $M = (M, *, e)$, we get a category $\mathcal{M}$ with just one object M, morphisms the elements of M, composition $*$ and identity morphism e.

Remark 2.3. Mon is the category of monoids and monoid homomorphisms, which we shall not describe here. This category is just one example of the fact that any notion of sets with structure together with morphisms that preserve that structure define a category.

[2]For a full account on mathematical foundations of category theory, see (Awodey 2010, § 1.8; Mac Lane 1998, § I.6).

Example 2.1.5. We can look at categories as deductive systems with objects formulas and morphisms deductions. In this way, domains and codomains are premises and conclusions, respectively. A morphism is thus a proof of the fact that its codomain is deducible from its domain. In particular, identity morphisms are instances of the reflexivity axiom, and composition of morphisms is a rule of inference asserting that deductions are transitive. Note that this is just a change of vocabulary.

Before we move on, let us revisit the motivating example of the subset of Haskell, which we shall complete in the following section.

Example 2.1.6. The motivating example of the subset of Haskell corresponds to a category. Its objects are the types `()`, `Bool`, and `Nat`, and its morphisms are the constants-as-functions

- `() :: () -> ()`,
- `True` and `False :: () -> Bool`, and
- `Zero :: () -> Nat`,

the functions

- `Succ :: Nat -> Nat`,
- `not :: Bool -> Bool`, and
- `isZero :: Nat -> Bool`,

the identity functions, `id`, and composite functions

- `not . True`, `not . (not . True) :: () -> Bool`, ...,
- `Succ . Zero`, `Succ . (Succ . Zero) :: () -> Nat`, ...,
- `isZero . Zero`, `isZero . (Succ . Zero) :: () -> Bool`, ...,
- and so on.

We shall not prove the associativity and identity axioms yet, but, evidently, this subset of Haskell is a category. What would result if we considered all of Haskell?

2.2 A Category for Haskell

Having described a subset of Haskell types and functions as a category, let us now construct a general category for Haskell. Intuitively, if we keep adding types and functions to the category of Example 2.1.6, we get what we want. But we have to be careful because there are some features of Haskell which we cannot omit.

In a nutshell, **Hask** is the category of Haskell types and functions. As expected, the objects of this category are Haskell types, that is, nullary type constructors or type expressions with kind *[3]. For instance, the Boolean type, `Bool`:

```
data Bool = False | True
```

The natural number type, `Nat`:

```
data Nat = Zero | Succ Nat
```

The integer types, `Int` and `Integer`, the floating-point number types, `Float` and `Double`, the Unicode character type, `Char`, lists of types such as `[Bool]` and `[Nat]`, `Maybe` types such as `Maybe Bool` and `Maybe Nat`:

```
data Maybe a = Nothing | Just a
```

But not `[]` and `Maybe`, which are unary type constructors or type expressions with kind `* -> *`.

Convention 1. However, Haskell types have bottom. For example:

```
undefined :: a
undefined = undefined
```

As a consequence, values of type `Bool` include `True` and `False`, but also `undefined`, values of type `Nat` include `Zero` and `Succ Zero`, but also `undefined`, and so forth, which is a difficulty[4]. For this reason, by "Haskell types" we mean "Haskell types without bottom values," which is why **Hask** is sometimes considered a platonic category.

[3] In Haskell, type expressions are classified into different kinds, which are like types of types. See (Peyton Jones 2003, § 4.1.1).

[4] See http://www.haskell.org/haskellwiki/Hask.

As anticipated, the morphisms of **Hask** are Haskell functions. For instance, `not`:

```
not :: Bool -> Bool
not False = True
not True  = False
```

And `isZero`:

```
isZero :: Nat -> Bool
isZero Zero = True
isZero _    = False
```

For each type `a`, there is an identity function, `id`:

```
id :: a -> a
id x = x
```

And, for each pair of morphisms `f :: a -> b` and `g :: b -> c`, there is a composite function, `g . f`:

```
infixr 9 .

(.) :: (b -> c) -> (a -> b) -> a -> c
g . f = \x -> g (f x)
```

As a result, the associativity axiom becomes, whenever it makes sense:

```
h . (g . f) = h . g . f = (h . g) . f
```

And the identity axiom becomes, for all functions `f :: a -> b`:

```
id . f = f = f . id
```

Both axioms follow immediately from rewriting using definitions. In the first place:

```
(h . (g . f)) x
```

```
  =  (by definition of (.))
h ((g . f) x)
  =  (by definition of (.))
h (g (f x))
  =  (by definition of (.))
(h . g) (f x)
  =  (by definition of (.))
((h . g) . f) x
```

In the second place:

```
(id . f) x
  =  (by definition of (.))
id (f x)
  =  (by definition of id)
f x
  =  (by definition of id)
f (id x)
  =  (by definition of (.))
(f . id) x
```

From now on, we shall use **Hask** as described above as Haskell's category[5].

2.3 A Category for Agda

In Agda, types are called sets and there is an infinite hierarchy of universes `Set₀`, `Set₁`, `Set₂`, ..., such that `Set₀` is of type `Set₁`, `Set₁` is of type `Set₂`, and so on. The first universe, `Set₀` or `Set`, which is called the universe of small types, and the second universe, `Set₁`, will be the only universes necessary for our development (Bove and Dybjer 2009, pp. 57, 61; Norell 2009, p. 231).

Let us now construct a category analogous to **Hask**, the category of Haskell types and functions, in Agda. The objects of this category are small types, that is, types of type `Set`. For example, the Boolean type, `Bool`:

[5] Note that this is not an attempt to answer the question of Haskell's category.

```
data Bool : Set where
  true  : Bool
  false : Bool
```

And the natural number type, `Nat`:

```
data ℕ : Set where
  zero : ℕ
  succ : ℕ → ℕ
```

And the morphisms of this category are functions between small types. For instance, `not`:

```
not : Bool → Bool
not true  = false
not false = true
```

For each small type `A`, there is an identity function defined as follows:

```
id : {A : Set} → A → A
id x = x
```

And, for each pair of functions `f : A → B` and `g: B → C`, there is a composite function, `g ∘ f`:

```
infixr 9 _∘_

_∘_ : {A B C : Set} → (B → C) → (A → B) → A → C
g ∘ f = λ x → g (f x)
```

For both of these definitions, see the module Abel.Function.

Unlike in Haskell, the associativity and identity axioms for this category are declared and proved in Agda (see the module Abel.Function.Category):

```
associativity : {A B C D : Set} {f : A → B} {g : B → C} {h : C → D}
                (x : A) → (h ∘ g ∘ f) x ≡ ((h ∘ g) ∘ f) x
associativity _ = refl
```

```
identity : {A B : Set} {f : A → B}
           (x : A) → (id ∘ f) x ≡ f x × (f ∘ id) x ≡ f x
identity _ = refl , refl
```

From now on, we shall use this category as Agda's category[6] and call it **Agda**.

2.4 References

The definition of category is based on (Awodey 2010, pp. 4–5; Mac Lane 1998, pp. 7–8, 289). We defined categories directly, but they can be defined in many ways. For instance, Eilenberg and MacLane (1945) defined them as aggregates of objects and mappings, and Mac Lane (1998) did it in terms of metacategories and in terms of sets, as sets of objects and arrows. Other ways include defining them as sets of objects and morphisms, just a set of morphisms, and in terms of collections of morphisms or hom-sets.

The definition of commutative diagram is based on (Mac Lane 1998, p. 8; Poigné 1992, pp. 434–435), but almost any reference on category theory contains an equivalent definition. Additionally, the examples of categories are based on (Awodey 2010, pp. 7–8; Mac Lane 1998, pp. 8–12, 21; Pierce 1991, Example 1.1.14; Poigné 1992, § 1.2.2).

Finally, the motivating example of the subset of Haskell is based on (Pierce 1991, Example 1.1.15; Pitt 1986, pp. 7–11). For more information about categories in Haskell, including **Hask**, see (Elkins 2009, p. 74; Yorgey 2009, pp. 49–51).

[6] Note that this is not an attempt to answer the question of Agda's category.

Chapter 3

Constructions

In this chapter we explore some basic constructions in categories. We shall use these constructions for describing some concepts and examples in all the following chapters.

3.1 Isomorphisms

We shall often find the concept of isomorphism and the property of uniqueness up to isomorphism.

Definition 3.1. Let $\mathcal{C}$ be a category. A morphism $f : a \to b$ is an isomorphism if there is an inverse morphism $f^{-1} : b \to a$ such that

$$f^{-1} \circ f = \mathrm{id}_a \quad \text{and} \quad f \circ f^{-1} = \mathrm{id}_b \,. \tag{3.1}$$

Objects a and b are isomorphic if there is an isomorphism $f : a \to b$. Isomorphic objects are often said to be identical up to isomorphism. Similarly, an object with some property is said to be unique up to isomorphism if every object satisfying the property is isomorphic to it.

3.2 Initial and Terminal Objects

We define initial and terminal objects, which we shall use for describing algebras and initial algebras.

Definition 3.2. Let $\mathcal{C}$ be a category. An object 0 is the initial object of $\mathcal{C}$ if, for all objects a, there is a unique morphism $0 \to a$.

Definition 3.3. Let $\mathcal{C}$ be a category. An object 1 is the terminal object of $\mathcal{C}$ if, for all objects a, there is a unique morphism $a \to 1$.

Lemma 3.1. *Initial and terminal objects are unique up to isomorphism.*

Proof. Let $\mathcal{C}$ be a category with initial objects 0 and $0'$. There are unique morphisms $0_{0'} : 0 \to 0'$ and $0'_0 : 0' \to 0$, and $0_0 = \mathrm{id}_0$ and $0'_{0'} = \mathrm{id}_{0'}$. Hence, $0'_0 \circ 0_{0'} = \mathrm{id}_0$ and $0_{0'} \circ 0'_0 = \mathrm{id}_{0'}$. That is, 0 is unique up to isomorphism.

Similarly, let $\mathcal{C}$ be a category with terminal objects 1 and $1'$. There are unique morphisms $1_{1'} : 1' \to 1$ and $1'_1 : 1 \to 1'$, and $1_1 = \mathrm{id}_1$ and $1'_{1'} = \mathrm{id}_{1'}$. Hence, $1_{1'} \circ 1'_1 = \mathrm{id}_1$ and $1'_1 \circ 1_{1'} = \mathrm{id}_{1'}$. That is, 1 is unique up to isomorphism.

□

As examples, we consider initial and terminal objects in the categories **Set**, **Hask**, and **Agda**.

Example 3.2.1. In **Set**, the empty set $\emptyset$ is the initial object. Given a set A, the empty function is the unique function $\emptyset \to A$. Additionally, any singleton set $\{x\}$ is a terminal object. Given a set A, the function which assigns x to each element of A is the unique function $A \to \{x\}$.

Example 3.2.2. In **Hask**, the empty or uninhabited type is the initial object of the category:

```
data Void
```

The `absurd` function, as defined, for instance, in (Kmett 2012), is the unique function required to show that `Void` is indeed the initial object:

```
absurd :: Void -> a
absurd = absurd
```

Note that, in the absence of Convention 1, we would also have:

```
absurd' :: Void -> a
absurd' _ = undefined
```

Additionally, the unit type is the terminal object of the category:

```
data () = ()
```

The `unit` function is the unique function required to show that `()` really is the terminal object:

```
unit :: a -> ()
unit _ = ()
```

Without Convention 1, we would also have:

```
unit' :: a -> ()
unit' _ = undefined
```

Example 3.2.3. In **Agda**, the empty type, defined in Data.Empty, is the initial object:

```
data ⊥ : Set where
```

The `⊥-elim` function, defined in Abel.Data.Empty, is the unique function required to show that `⊥` is indeed the initial object:

```
⊥-elim : {A : Set} → ⊥ → A
⊥-elim ()
```

The unit type, defined in Data.Unit, is the terminal object:

```
record ⊤ : Set where
  constructor tt
```

Or, equivalently, the unit type defined in Data.Unit.Core:

```
data Unit : Set where
  unit : Unit
```

Finally, the `unit` function, defined in Abel.Data.Unit, is the unique function required to show that `Unit` really is the terminal object:

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

Example 3.2.4. In **Set**, the elements of a set A can be considered as functions from a terminal object, that is, any singleton set, to A. More specifically, if $x \in A$ and 1 is a terminal object, then x can be considered as a function $x : 1 \to A$ which assigns x to the element of 1.

3.3 Products and Coproducts

In this section, we describe the concepts of product and coproduct, which correspond to Cartesian products and disjoint unions, respectively. As examples, we consider both constructions in the categories **Set**, **Hask**, and **Agda**.

Definition 3.4. A product of objects a and b in a category $\mathcal{C}$ consists of a product object $a \times b$, and projection morphisms $\pi_1 : a \times b \to a$ and $\pi_2 : a \times b \to b$, such that, for all objects c, and morphisms $f : c \to a$ and $g : c \to b$, there is a unique morphism $\langle f, g \rangle : c \to a \times b$ such that

$$\pi_1 \circ \langle f, g \rangle = f \quad \text{and} \quad \pi_2 \circ \langle f, g \rangle = g, \tag{3.2}$$

that is, the diagram in Figure 3.1 is commutative.

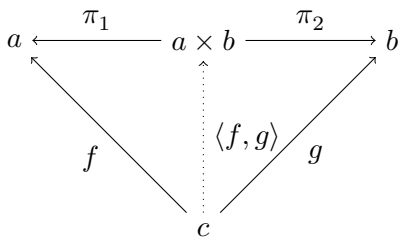


Figure 3.1: A product.

Example 3.3.1. In **Set**, the product of two sets A and B consists of the Cartesian product

$$A \times B = \{(x, y) \mid x \in A \quad \text{and} \quad y \in B\}, \tag{3.3}$$

and two projection functions $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$ such that, for all $(x, y) \in A \times B$,

$$\pi_1(x, y) = x \quad \text{and} \quad \pi_2(x, y) = y.$$

Given a set C, and two functions $f : C \to A$ and $g : C \to B$, there is a unique function $\langle f, g \rangle : C \to A \times B$ defined by

$$\langle f, g \rangle(z) = (f(z), g(z))$$

for all $z \in C$. Equations (3.2) hold.

Example 3.3.2. In **Hask**, tuples are products:

```
data (,) a b = (,) a b
```

The projection functions are `fst` and `snd`, which extract the first and second components of a pair, respectively:

```
fst :: (a,b) -> a
fst (x,_) = x

snd :: (a,b) -> b
snd (_,y) = y
```

The `fork` function is the function required to show that tuples are indeed products:

```
fork :: (c -> a) -> (c -> b) -> c -> (a,b)
fork f g z = (f z,g z)
```

Example 3.3.3 (See module Abel.Data.Product)**.** In **Agda**, products and their projection functions are defined as follows:

```
data _×_ (A B : Set) : Set where
  _,_ : A → B → A × B

proj₁ : {A B : Set} → A × B → A
proj₁ (x , _) = x

proj₂ : {A B : Set} → A × B → B
proj₂ (_ , y) = y
```

The required function to satisfy the definition of products is:

```
⟨_,_⟩ : {A B C : Set} → (C → A) → (C → B) → C → A × B
⟨_,_⟩ f g z = f z , g z
```

Definition 3.5. A coproduct of objects a and b in a category $\mathcal{C}$ consists of a coproduct object $a + b$, and injection morphisms $\iota_1 : a \to a + b$ and $\iota_2 : b \to a + b$, such that, for all objects c, and morphisms $f : a \to c$ and $g : b \to c$, there is a unique morphism $[f, g] : a + b \to c$ such that

$$[f, g] \circ \iota_1 = f \quad \text{and} \quad [f, g] \circ \iota_2 = g, \tag{3.4}$$

that is, the diagram in Figure 3.2 is commutative.

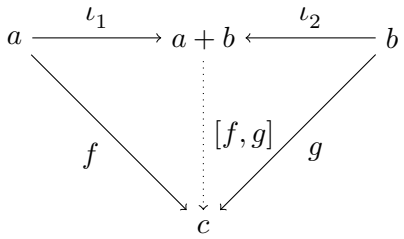


Figure 3.2: A coproduct.

Example 3.3.4. In **Set**, the coproduct of two sets A and B consists of the disjoint union

$$A + B = (\{1\} \times A) \cup (\{2\} \times B), \tag{3.5}$$

and two injection functions $\iota_1 : A \to A + B$ and $\iota_2 : B \to A + B$ such that, for all $x \in A$ and $y \in B$,

$$\iota_1(x) = (1, x) \quad \text{and} \quad \iota_2(y) = (2, y). \tag{3.6}$$

Given a set C, and two functions $f : A \to C$ and $g : B \to C$, there is a unique function $[f, g] : A + B \to C$ defined by

$$[f, g](\iota_1(x)) = f(x) \quad \text{and} \quad [f, g](\iota_2(y)) = g(y) \tag{3.7}$$

for all $x \in A$ and $y \in B$. Equations (3.4) hold.

Example 3.3.5. In **Hask**, coproducts and their injection functions are defined by the `Either` type:

```
data Either a b = Left a | Right b
```

The `either` function is the function required to satisfy the definition of a coproduct:

```
either :: (a -> c) -> (b -> c) -> Either a b -> c
either f _ (Left x)  = f x
either _ g (Right y) = g y
```

Example 3.3.6 (See module Abel.Data.Sum)**.** In **Agda**, sums or disjoint unions are coproducts:

```
data _+_ (A B : Set) : Set where
  inj₁ : (x : A) → A + B
  inj₂ : (y : B) → A + B
```

Finally, the required function to show that sums are indeed coproducts is defined as follows:

```
[_,_] : {A B C : Set} (f : A → C) (g : B → C) → A + B → C
[_,_] f _ (inj₁ x) = f x
[_,_] _ g (inj₂ y) = g y
```

3.4 References

This chapter is based on (Pierce 1991, pp. 15–19; Poigné 1992, pp. 439–440, 444; Mac Lane 1998, p. 63).

Chapter 4

Functors

> *Category* has been defined in order to be able to define *functor*...
>
> —Mac Lane (1998, p. 18)

In this chapter we explore mathematical functors, and functors in Haskell and Agda. Mapping over lists, which is accomplished with the `map` function, is "a dominant idiom in Haskell" (Lipovača 2011, p. 146). The type signature of the `map` function is:

```
map :: (a -> b) -> [a] -> [b]
```

According to (Marlow 2010, p. 190), given a function `f` and a list `xs`, `map f xs` is the list obtained by applying `f` to each element in `xs`. In other words:

```
map f [x1, x2, ..., xn] = [f x1, f x2, ..., f xn]
```

Or, better:

```
map f [x1, x2, ...] = [f x1, f x2, ...]
```

The definition of the `map` function is:

```
map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```

Even though this is the correct definition of the `map` function (it applies a function to all the elements of a list), it is possible to implement alternative definitions. For instance:

```
map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : f x : map f xs
```

This alternative `map` function applies a function to each element in a list and duplicates each result.

Deciding whether the former or the latter `map` function is the correct one for mapping over lists requires a more general approach. This is achieved with the definition of the `Functor` type class, which is used for types that can be mapped over and which generalizes the `map` function as a *uniform* action over a parameterized type such as `[a]` or `Maybe a`.

However, the definition of the `Functor` type class is not enough to determine what a "uniform action over a parameterized type" is. On the other hand, a comment in (Peyton Jones 2003, p. 88) states that all instances of the `Functor` type class *should* satisfy the functor laws. These laws, which are not part of the definition of functors in Haskell, guarantee that a "uniform action over a parameterized type" is actually uniform.

Functors in Haskell implement mathematical functors (that is, functors in category theory) and the functor laws correspond to the conditions that a mathematical functor *must* satisfy in order to be a functor. Studying mathematical functors may not be necessary for uniformly mapping over a parameterized type, but it may be very useful for better understanding what that means.

4.1 Functors

Basically, a functor or morphism of categories maps the objects and morphisms of a category to objects and morphisms of another category.

Definition 4.1. Let $\mathcal{C}$ and $\mathcal{D}$ be categories. A functor $\mathsf{F} : \mathcal{C} \to \mathcal{D}$ assigns to each object a in $\mathcal{C}$ an object $\mathsf{F}_{\mathrm{O}}(a)$ in $\mathcal{D}$, and to each morphism $f : a \to b$ in $\mathcal{C}$ a morphism $\mathsf{F}_{\mathrm{M}}(f) : \mathsf{F}_{\mathrm{O}}(a) \to \mathsf{F}_{\mathrm{O}}(b)$ in $\mathcal{D}$, such that, for all objects a in $\mathcal{C}$,

$$\mathsf{F}_{\mathrm{M}}(\mathrm{id}_a) = \mathrm{id}_{\mathsf{F}_{\mathrm{O}}(a)}\,, \tag{4.1}$$

and, for all morphisms $f : a \to b$ and $g : b \to c$ in $\mathcal{C}$,

$$\mathsf{F}_{\mathrm{M}}(g \circ f) = \mathsf{F}_{\mathrm{M}}(g) \circ \mathsf{F}_{\mathrm{M}}(f). \tag{4.2}$$

A functor from a category to itself is called an endofunctor.

A simple example of functors is the power set operation, which yields an endofunctor in **Set**.

Example 4.1.1. The power set endofunctor $\mathsf{P} : \mathbf{Set} \to \mathbf{Set}$ assigns to each set A the set of all subsets of A, that is,

$$\mathsf{P}_{\mathrm{O}}(A) = \{X \mid X \subseteq A\}, \tag{4.3}$$

and to each function $f : A \to B$ a function $\mathsf{P}_{\mathrm{M}}(f) : \mathsf{P}_{\mathrm{O}}(A) \to \mathsf{P}_{\mathrm{O}}(B)$ such that, for all $X \in \mathsf{P}_{\mathrm{O}}(A)$,

$$\mathsf{P}_{\mathrm{M}}(f)(X) = \{f(x) \mid x \in X\}. \tag{4.4}$$

In order to see that this defines a functor, we prove, in the first place, that (4.1) holds for all $X \in \mathsf{P}_{\mathrm{O}}(A)$:

$$\begin{aligned}
&\mathsf{P}_{\mathrm{M}}(\mathrm{id}_A)(X) \\
&\quad = \quad \text{(by (4.4) with } f = \mathrm{id}_A\text{)} \\
&\{\mathrm{id}_A(x) \mid x \in X\} \\
&\quad = \quad \text{(by (2.3))} \\
&\{x \mid x \in X\} \\
&\quad = \\
&X \\
&\quad = \quad \text{(by (2.3) with } A = \mathsf{P}_{\mathrm{O}}(A) \text{ and } x = X\text{)} \\
&\mathrm{id}_{\mathsf{P}_{\mathrm{O}}(A)}(X)
\end{aligned}$$

In the second place, we prove that (4.2) holds for all $X \in \mathsf{P}_{\mathrm{O}}(A)$:

$$\begin{aligned}
&\mathsf{P}_{\mathrm{M}}(g \circ f)(X) \\
&\quad = \quad \text{(by (4.4) with } f = g \circ f\text{)} \\
&\{(g \circ f)(x) \mid x \in X\} \\
&\quad = \quad \text{(by (2.4))} \\
&\{g(f(x)) \mid x \in X\}
\end{aligned}$$

$$
\begin{aligned}
&= \quad \text{(by (4.4) with } f = g \text{ and } X = \{f(x) \mid x \in X\}) \\
&\mathsf{P}_{\mathrm{M}}(g)(\{f(x) \mid x \in X\}) \\
&= \quad \text{(by (4.4))} \\
&\mathsf{P}_{\mathrm{M}}(g)(\mathsf{P}_{\mathrm{M}}(f)(X)) \\
&= \quad \text{(by (2.4) with } f = \mathsf{P}_{\mathrm{M}}(f),\ g = \mathsf{P}_{\mathrm{M}}(g), \text{ and } x = X) \\
&(\mathsf{P}_{\mathrm{M}}(g) \circ \mathsf{P}_{\mathrm{M}}(f))(X)
\end{aligned}
$$

"There are many functors between two given categories, and the question of how they are connected suggests itself" (Marquis 2013, p. 11). For instance, there is always the identity endofunctor from a category to itself.

Example 4.1.2. Let $\mathcal{C}$ be a category. The identity endofunctor $\mathsf{I} : \mathcal{C} \to \mathcal{C}$ sends all objects and morphisms of $\mathcal{C}$ to themselves. In detail, for all objects a,

$$\mathsf{I}_{\mathrm{O}}(a) = a, \tag{4.5}$$

and, for all morphisms f,

$$\mathsf{I}_{\mathrm{M}}(f) = f. \tag{4.6}$$

In order to see that this defines a functor, we prove, in the first place, that (4.1) holds:

$$
\begin{aligned}
&\mathsf{I}_{\mathrm{M}}(\mathrm{id}_a) \\
&= \quad \text{(by (4.6) with } f = \mathrm{id}_a) \\
&\mathrm{id}_a \\
&= \quad \text{(by (4.5))} \\
&\mathrm{id}_{\mathsf{I}_{\mathrm{O}}(a)}
\end{aligned}
$$

In the second place, we prove that (4.2) holds:

$$
\begin{aligned}
&\mathsf{I}_{\mathrm{M}}(g \circ f) \\
&= \quad \text{(by (4.6) with } f = g \circ f) \\
&g \circ f \\
&= \quad \text{(by (4.6) with } f = g \text{ and (4.6))} \\
&\mathsf{I}_{\mathrm{M}}(g) \circ \mathsf{I}_{\mathrm{M}}(f)
\end{aligned}
$$

4.2 Functors in Haskell

Functors in Haskell are defined by "the most basic and ubiquitous type class in the Haskell libraries" (Yorgey 2009, p. 18), the `Functor` type class, which is exported by the Haskell Prelude:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

It is used for types that can be mapped over, and generalizes the `map` function on lists with a uniform action over a parameterized type. Intuitively, functors represent containers or computational contexts, but we are more interested in their definition and its relation to that of mathematical functors.

It is important to note that `f` is a type constructor rather than a type (its kind is `* -> *`, not `*`). `[]` and `Maybe` are examples of such type constructors. The result of applying a type constructor to a type (for example, `Int` or `Bool`) is a type or concrete type (that is, something of kind `*`). Therefore, the kinds of `[] Int` (that is, `[Int]`) and `Maybe Int` are both `*`. As another example, since the kind of `Either` is `* -> * -> *`, it cannot be declared as an instance of the `Functor` type class. However, a type constructor can be partially applied and something like `Either a` can be declared as an instance of `Functor`.

The fact that `f` is a type constructor can be made explicit using the `KindSignatures` language option:

```
class Functor (f :: * -> *) where
  fmap :: (a -> b) -> f a -> f b
```

The kind signature of `f` shows that it corresponds to the object mapping of a mathematical functor (that is, F_O): it sends objects of **Hask** (types) to objects of **Hask** (types).

The `fmap` function is curried and can be rewritten with extra (and unnecessary) parentheses:

```
class Functor (f :: * -> *) where
  fmap :: (a -> b) -> (f a -> f b)
```

The type of `fmap` shows that it corresponds to the morphism mapping of a mathematical functor (that is, F_{M}): it sends morphisms of **Hask** (functions) to morphisms of **Hask** (functions).

Instances of the `Functor` type class correspond to functors (more precisely, endofunctors) from **Hask** to **Hask** with the type constructor and the `fmap` function as the required object and morphism mappings.

Although functors *should* obey the functor laws, this is not mandatory when declaring an instance of the `Functor` type class. The first law can be stated as:

```
fmap id = id
```

Polymorphism in Haskell allows us to write just `id` in both sides of this law, but the types of each `id` are different (which is more obvious when comparing this with (4.1)). A more precise way of stating the first law in Haskell follows:

```
fmap (id :: a -> a) = (id :: f a -> f a)
```

The second law can be stated as:

```
fmap (g . f) = fmap g . fmap f
```

This law corresponds to (4.2). Proving both of these laws amounts to proving that an instance of the `Functor` type class is actually a functor.

Remark 4.1. In the following examples, we shall prove the functor laws by hand. A different approach is that of (Jeuring, Jansson, and Amaral 2012), which develops a framework that supports testing such laws using QuickCheck.

We have already described the identity endofunctor for any category. In particular, there is the identity functor of **Hask**.

Example 4.2.1. The identity functor of **Hask**, which is just an instance of the identity endofunctor as described in Example 4.1.2, is defined as follows[1]:

[1] Using the `InstanceSigs` language option.

```
newtype Identity a = Identity {unIdentity :: a}

instance Functor Identity where
  fmap :: (a -> b) -> Identity a -> Identity b
  fmap f (Identity x) = Identity (f x)
```

Common examples of functors in Haskell include `Maybe` and lists. We shall use these functors again when studying natural transformations and monads.

Example 4.2.2. A basic example of functors in Haskell is the `Maybe` functor. Its type constructor is the `Maybe` type constructor:

```
data Maybe a = Nothing | Just a
```

Its `fmap` function is defined as follows:

```
instance Functor Maybe where
  fmap :: (a -> b) -> Maybe a -> Maybe b
  fmap _ Nothing  = Nothing
  fmap f (Just x) = Just (f x)
```

Basically, the `Maybe` type constructor sends types `a` to types `Maybe a`, a value of type `Maybe a` either contains a value of type `a` or is empty, and the `fmap` function sends functions `a -> b` to functions `Maybe a -> Maybe b`. This instance satisfies the laws. First, we prove that (4.1) holds.

Case `Nothing`:

```
fmap id Nothing
```
= (by definition of `fmap`)
```
Nothing
```
= (by definition of `id`)
```
id Nothing
```

Case `(Just x)`:

```
fmap id (Just x)
```

```
  =  (by definition of fmap)
Just (id x)
  =  (by definition of id)
Just x
  =  (by definition of id)
id (Just x)
```

Second, we prove that (4.2) holds.

Case `Nothing`:

```
(fmap g . fmap f) Nothing
  =  (by definition of (.))
fmap g (fmap f Nothing)
  =  (by definition of fmap)
fmap g Nothing
  =  (by definition of fmap)
Nothing
  =  (by definition of fmap)
fmap (g . f) Nothing
```

Case `Just x`:

```
fmap (g . f) (Just x)
  =  (by definition of fmap)
Just ((g . f) x)
  =  (by definition of (.))
Just (g (f x))
  =  (by definition of fmap)
fmap g (Just (f x))
  =  (by definition of fmap)
fmap g (fmap f (Just x))
  =  (by definition of (.))
(fmap g . fmap f) (Just x)
```

Example 4.2.3. Another common example of functors in Haskell is the `[]` (list) functor. Its type constructor is `[]`, and its `fmap` function is the `map` function:

```
instance Functor [] where
  fmap :: (a -> b) -> [a] -> [b]
  fmap _ []     = []
  fmap f (x:xs) = f x : fmap f xs
```

Or:

```
instance Functor [] where
  fmap :: (a -> b) -> [a] -> [b]
  fmap = map
```

The `[]` type constructor sends types `a` to types `[a]`, that is, lists of `a`, and the `fmap` or `map` function, which we talked about earlier, sends functions of type `a -> b` to functions of type `[a] -> [b]`. In order to see that this instance satisfies the laws, we begin by proving (4.1) by induction.

Case `[]`:

```
fmap id []
```
  `=` (by definition of `fmap`)
```
[]
```
  `=` (by definition of `id`)
```
id []
```

Case `(x:xs)`:

```
fmap id (x:xs)
```
  `=` (by definition of `fmap`)
```
id x : fmap id xs
```
  `=` (by definition of `id`)
```
x : fmap id xs
```
  `=` (by inductive hypothesis)
```
x:xs
```

```
  =  (by definition of id)
id (x:xs)
```

Second, we prove (4.2), also by induction.

Case `[]`:

```
(fmap g . fmap f) []
  =  (by definition of (.))
fmap g (fmap f [])
  =  (by definition of fmap)
fmap g []
  =  (by definition of fmap)
[]
  =  (by definition of fmap)
fmap (g . f) []
```

Case `(x:xs)`:

```
fmap (g . f) (x:xs)
  =  (by definition of fmap)
(g . f) x : fmap (g . f) xs
  =  (by definition of (.))
g (f x) : fmap (g . f) xs
  =  (by inductive hypothesis)
g (f x) : (fmap g . fmap f) xs
  =  (by definition of (.))
g (f x) : fmap g (fmap f xs)
  =  (by definition of fmap)
fmap g (f x : fmap f xs)
  =  (by definition of fmap)
fmap g (fmap f (x:xs))
  =  (by definition of (.))
(fmap g . fmap f) (x:xs)
```

Additional examples of functors in Haskell include products and coproducts, which are some of the constructions we discussed in the previous chapter, and functions, which is an interesting case for better understanding the type signature of `fmap`.

Example 4.2.4. Another usual example of functors in Haskell is the product functor. Its type constructor is `(,) a` (see Example 3.3.2), and its `fmap` function is uniquely defined as follows:

```
instance Functor ((,) a) where
  fmap :: (b -> c) -> (a,b) -> (a,c)
  fmap f (x, y) = (x, f y)
```

Let us prove that this instance satisfies the functor laws. In the first place, we show that (4.1) holds.

```
fmap id (x, y)
```
  = (by definition of `fmap`)
```
(x, id y)
```
  = (by definition of `id`)
```
(x, y)
```
  = (by definition of `id`)
```
id (x, y)
```

In the second place, we show that (4.2) holds:

```
(fmap h . fmap g) (x, y)
```
  = (by definition of `(.)`)
```
fmap h (fmap g (x, y))
```
  = (by definition of `fmap`)
```
fmap h (x, g y)
```
  = (by definition of `fmap`)
```
(x, h (g y))
```
  = (by definition of `(.)`)
```
(x, (h . g) y)
```
  = (by definition of `fmap`)

```
fmap (h . g) (x, y)
```

Example 4.2.5. In a similar way, the coproduct functor is a usual example of functors in Haskell. Its type constructor is `Either a` (see Example 4.2.5), and its `fmap` function is uniquely defined as follows:

```
instance Functor (Either a) where
  fmap :: (b -> c) -> Either a b -> Either a c
  fmap _ (Left x)  = Left x
  fmap g (Right y) = Right (g y)
```

In order to see that this instance obeys the laws, we prove, in the first place, that (4.1) holds.

Case `(Left x)`:

```
fmap id (Left x)
```
  = (by definition of `fmap`)
```
Left x
```
  = (by definition of `id`)
```
id (Left x)
```

Case `(Right y)`:

```
fmap id (Right y)
```
  = (by definition of `fmap`)
```
Right (id y)
```
  = (by definition of `id`)
```
Right y
```
  = (by definition of `id`)
```
id (Right y)
```

In the second place, we prove that (4.2) holds.

Case `(Left x)`:

```
(fmap h . fmap g) (Left x)
```
  = (by definition of `(.)`)

```
fmap h (fmap g (Left x))
```
  = (by definition of `fmap`)
```
fmap h (Left x)
```
  = (by definition of `fmap`)
```
Left x
```
  = (by definition of `fmap`)
```
fmap (h . g) (Left x)
```

Case `(Right y)`:

```
(fmap h . fmap g) (Right y)
```
  = (by definition of `(.)`)
```
fmap h (fmap g (Right y))
```
  = (by definition of `fmap`)
```
fmap h (Right (g y))
```
  = (by definition of `fmap`)
```
Right (h (g y))
```
  = (by definition of `(.)`)
```
Right ((h . g) y)
```
  = (by definition of `fmap`)
```
fmap (h . g) (Right y)
```

Example 4.2.6. An interesting example of functors in Haskell is the function functor. Its type constructor is `(->) a`, and its `fmap` function is function composition:

```
instance Functor ((->) a) where
  fmap :: (b -> c) -> (a -> b) -> a -> c
  fmap g f = \x -> g (f x)
```

Or, equivalently:

```
instance Functor ((->) a) where
  fmap :: (b -> c) -> (a -> b) -> a -> c
  fmap = (.)
```

Let us see that this instance satisfies the functor laws. In the first place, we prove that (4.1) holds.

```
fmap id f
```
= (by definition of `fmap`)
```
id . f
```
= (by (2.2))
```
f
```
= (by definition of `id`)
```
id f
```

In the second place, we prove that (4.2) holds.

```
(fmap h . fmap g) f
```
= (by definition of `(.)`)
```
fmap h (fmap g f)
```
= (by definition of `fmap`)
```
fmap h (g . f)
```
= (by definition of `fmap`)
```
h . (g . f)
```
= (by (2.1))
```
(h . g) . f
```
= (by definition of `fmap`)
```
fmap (h . g) f
```

Since the functor laws are not part of the definition of the `Functor` type class, we can declare instances which do not satisfy the laws. In the following examples, we consider incorrect definitions of the `Maybe` and `[]` functors.

Example 4.2.7. It is possible to incorrectly define the `Maybe` functor (see Example 4.2.2) as follows:

```
instance Functor Maybe where
  fmap :: (a -> b) -> Maybe a -> Maybe b
  fmap _ Nothing  = Nothing
  fmap _ (Just _) = Nothing
```

Even though this definition of the `fmap` function is accepted by the Haskell type checker, the following counterexample proves that it violates the first functor law:

```
> fmap id (Just 0)
Nothing
> id (Just 0)
Just 0
```

Note that this is the only way to incorrectly declare the `Maybe` functor within **Hask**.

Example 4.2.8. We can wrongly define the `[]` (list) functor too (see Example 4.2.3). Here is the declaration we discussed at the beginning of the chapter:

```
instance Functor [] where
  fmap :: (a -> b) -> [a] -> [b]
  fmap _ []     = []
  fmap f (x:xs) = f x : f x : fmap f xs
```

But this is not the only way. For instance:

```
instance Functor [] where
  fmap :: (a -> b) -> [a] -> [b]
  fmap _ []     = []
  fmap f (x:xs) = [f x]
```

However, neither of these instances satisfy the first functor law, as demonstrated by the following counterexamples. In the first case:

```
> fmap id [0,1]
[0,0,1,1]
> id [0,1]
[0,1]
```

In the second case:

```
> fmap id [0,1]
[0]
> id [0,1]
[0,1]
```

4.3 Functors in Agda

The Agda standard library defines functors in Agda just like functors in Haskell, that is, without the functor laws[2]. Abel defines functors in the module Abel.Category.Functor, which includes the functor laws[3]:

```
record Functor (F : Set → Set) : Set₁ where

  constructor mkFunctor

  field

    fmap    : {A B : Set} → (A → B) → F A → F B

    fmap-id : {A : Set} (fx : F A) → fmap id fx ≡ id fx

    fmap-∘  : {A B C : Set} {f : A → B} {g : B → C}
              (fx : F A) → fmap (g ∘ f) fx ≡ (fmap g ∘ fmap f) fx
```

The inclusion of the functor laws makes it impossible to define a functor which is not really a functor because all instances must prove that `F` and `fmap` satisfy the laws.

As examples, we consider all instances described in Haskell in the previous section.

Example 4.3.1 (See module Abel.Data.Maybe.Functor)**.** The `Maybe` functor in Agda is defined as follows:

```
functor : Functor Maybe
functor = mkFunctor fmap fmap-id fmap-∘
```

[2] See (Danielsson et al. 2013, module Category.Functor).

[3] We refer to propositional (intensional) equality, as defined in (Danielsson et al. 2013, module Relation.Binary.PropositionalEquality).

```
  where
    fmap : {A B : Set} → (A → B) → Maybe A → Maybe B
    fmap f (just x) = just (f x)
    fmap _ nothing  = nothing

    fmap-id : {A : Set} (mx : Maybe A) → fmap id mx ≡ id mx
    fmap-id (just _) = refl
    fmap-id nothing  = refl

    fmap-∘ : {A B C : Set} {f : A → B} {g : B → C}
             (mx : Maybe A) → fmap (g ∘ f) mx ≡ (fmap g ∘ fmap f) mx
    fmap-∘ (just _) = refl
    fmap-∘ nothing  = refl
```

This functor corresponds to the `Maybe` functor in Haskell (see Example 4.2.2).

Example 4.3.2 (See module Abel.Data.List.Functor)**.** The following instance corresponds to the `List` functor in Agda:

```
functor : Functor List
functor = mkFunctor fmap fmap-id fmap-∘
  where
    fmap : {A B : Set} → (A → B) → List A → List B
    fmap _ []       = []
    fmap f (x ∷ xs) = f x ∷ fmap f xs

    fmap-id : {A : Set} (xs : List A) → fmap id xs ≡ id xs
    fmap-id []       = refl
    fmap-id (x ∷ xs) = cong (_∷_ x) (fmap-id xs)

    fmap-∘ : {A B C : Set} {f : A → B} {g : B → C}
             (xs : List A) → fmap (g ∘ f) xs ≡ (fmap g ∘ fmap f) xs
    fmap-∘             []       = refl
    fmap-∘ {f = f} {g} (x ∷ xs) = cong (_∷_ (g (f x))) (fmap-∘ xs)
```

This definition matches that of the `[]` functor in Haskell (see Example 4.2.3).

Example 4.3.3 (See module Abel.Data.Product.Functor)**.** Here is the declaration of the product functor in Agda (see Example 3.3.3):

```
functor : {A : Set} → Functor (_×_ A)
functor {A} = mkFunctor fmap fmap-id fmap-∘
  where
    fmap : {B C : Set} → (B → C) → A × B → A × C
    fmap g (x , y) = x , g y

    fmap-id : {B : Set} (x,y : A × B) → fmap id x,y ≡ id x,y
    fmap-id (x , y) = refl

    fmap-∘ : {B C D : Set} {g : B → C} {h : C → D}
             (x,y : A × B) → fmap (h ∘ g) x,y ≡ (fmap h ∘ fmap g) x,y
    fmap-∘ (x , y) = refl
```

Compare this with the product functor in Haskell (see Example 4.2.4).

Example 4.3.4 (See module Abel.Data.Sum.Functor)**.** The coproduct functor in Agda (see Example 3.3.6) is defined as follows:

```
functor : {A : Set} → Functor (_+_ A)
functor {A} = mkFunctor fmap fmap-id fmap-∘
  where
    fmap : {B C : Set} → (B → C) → A + B → A + C
    fmap _ (inj₁ x) = inj₁ x
    fmap g (inj₂ y) = inj₂ (g y)

    fmap-id : {B : Set} (x+y : A + B) → fmap id x+y ≡ id x+y
    fmap-id (inj₁ _) = refl
    fmap-id (inj₂ _) = refl

    fmap-∘ : {B C D : Set} {g : B → C} {h : C → D}
             (x+y : A + B) → fmap (h ∘ g) x+y ≡ (fmap h ∘ fmap g) x+y
    fmap-∘ (inj₁ _) = refl
    fmap-∘ (inj₂ _) = refl
```

Compare this with the coproduct functor in Haskell (see Example 4.2.5).

Example 4.3.5 (See module Abel.Function.Functor)**.** Here is the definition of the function functor in Agda, which corresponds to the Haskell functor described in Example 4.2.6:

```
functor : {A : Set} → Functor (λ B → A → B)
functor = mkFunctor (λ g f → g ∘ f) (λ _ → refl) (λ _ → refl)
```

Example 4.3.6. We can try to define the alternative `Maybe` functor of Example 4.2.7 in Agda:

```
functor : Functor Maybe
functor = mkFunctor fmap fmap-id fmap-∘
  where
    fmap : {A B : Set} → (A → B) → Maybe A → Maybe B
    fmap f (just x) = nothing
    fmap _ nothing  = nothing

    fmap-id : {A : Set} (mx : Maybe A) → fmap id mx ≡ id mx
    fmap-id (just _) = ?
    fmap-id nothing  = refl

    fmap-∘ : {A B C : Set} {f : A → B} {g : B → C}
             (mx : Maybe A) → fmap (g ∘ f) mx ≡ (fmap g ∘ fmap f) mx
    fmap-∘ (just _) = refl
    fmap-∘ nothing  = refl
```

But this code does not type check because there is a proof missing. As we saw in Example 4.2.7, the first functor law does not hold for this definition of the `Maybe` functor, so there is no way to make this instance type check in Agda.

4.4 References

The definition of a functor is based on (Mac Lane 1998, p. 13; Poigné 1992, p. 428), the power set and identity functors are taken from (Poigné 1992, p. 431) and (Marquis 2013, p. 11), respectively, and the study of functors in Haskell is based on (Lipovača 2011, pp. 146–150, 218–227; Yorgey 2009, pp. 18–23).

Chapter 5

Natural Transformations

> ... and *functor* has been defined in order to be able to define *natural transformation*.
>
> —Mac Lane (1998, p. 18)

In this chapter we explore natural transformations and their relation to polymorphic functions in Haskell. Despite their name, natural transformations might be "a first stumbling block" in the study of category theory, "simply because the examples tend to raise the level of mathematical sophistication" (Poigné 1992, p. 433). But natural transformations are indeed natural, especially in functional programming. It is typical to explain this idea with examples of parametrically polymorphic functions such as appending an element to a list, extracting the first component of a pair, reversing a list, among others.

In Haskell, one such function is the `head` function, which extracts the first element of a list[1]:

```
head :: [a] -> Maybe a
head []    = Nothing
head (x:_) = Just x
```

This is not any function, but a function between functors. More specifically, it is a function from the `[]` (list) functor into the `Maybe` functor. Besides, it is not just a function, but rather a family of functions indexed by Haskell types.

[1] Note that this is not the standard Haskell `head` function.

The `head` function is natural or uniform in the sense that, given two types `a` and `b`, a function `f :: a -> b`, and a list of elements of `a`, mapping the function over the list and then extracting the first element of the result is the same as extracting the first element of the list and then mapping the function over the result:

```
head . fmap f = fmap f . head
```

That is, the diagram in Figure 5.1 is commutative.

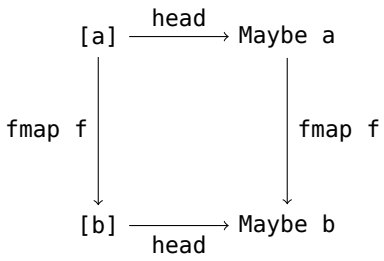


Figure 5.1: Naturality of the `head` function.

Even if this is an intuitive property of the `head` function and proving it is quite simple, this property says a lot about the behavior of all the functions that are part of the family defined by `head`. Most importantly, all of these facts are abstracted by natural transformations.

5.1 Natural Transformations

Having defined categories and functors (morphisms of categories), let us now define natural transformations (morphisms of functors).

Definition 5.1. Let F and $\mathsf{G} : \mathcal{C} \to \mathcal{D}$ be functors for two categories $\mathcal{C}$ and $\mathcal{D}$. A natural transformation

$$\tau : \mathsf{F} \to \mathsf{G} : \mathcal{C} \to \mathcal{D}$$

assigns to each object a in $\mathcal{C}$ a morphism $\tau_a : \mathsf{F}_{\mathrm{O}}(a) \to \mathsf{G}_{\mathrm{O}}(a)$ in $\mathcal{D}$, called a component of the natural transformation, such that, for all morphisms $f : a \to b$ in $\mathcal{C}$,

$$\tau_b \circ \mathsf{F}_{\mathrm{M}}(f) = \mathsf{G}_{\mathrm{M}}(f) \circ \tau_a, \tag{5.1}$$

that is, the diagram in Figure 5.2 is commutative.

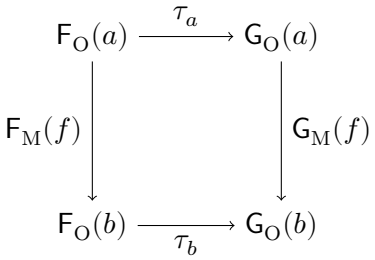


Figure 5.2: Naturality of a natural transformation.

As an example, the identity morphisms of a category are the components of a natural transformation, and, interestingly, the identity axiom of the category is the naturality of the transformation.

Example 5.1.1. Given a category $\mathcal{C}$, the identity natural transformation $\mathrm{id} : \mathsf{I} \to \mathsf{I} : \mathcal{C} \to \mathcal{C}$ assigns to each object a the identity morphism $\mathrm{id}_a : a \to a$. This is a natural transformation from and into the identity endofunctor (see Example 4.1.2). Naturality is the commutativity of the diagram in Figure 5.3a, which holds by (2.2). More explicitly, naturality corresponds to the commutativity of the diagram in Figure 5.3b, which includes the identity endofunctor.

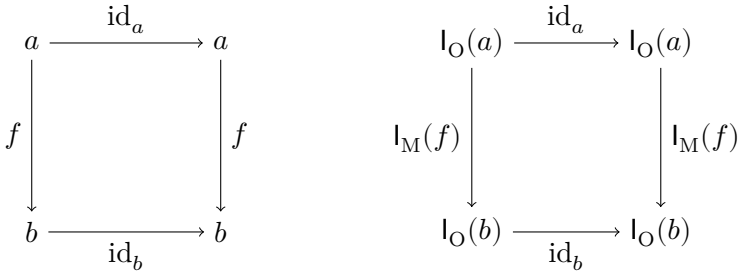


Figure 5.3: Naturality of the identity natural transformation.

As an example of natural transformations as morphisms of functors, the identity and power set functors are related in a natural manner.

Example 5.1.2. In **Set**, $\eta : \mathsf{I} \to \mathsf{P} : \mathbf{Set} \to \mathbf{Set}$ is a natural transformation which assigns to each set A a function $\eta_A : \mathsf{I}_{\mathrm{O}}(A) \to \mathsf{P}_{\mathrm{O}}(A)$ such that, for all $x \in A$,

$$\eta_A(x) = \{x\}. \tag{5.2}$$

This is a natural transformation from the identity functor into the power set functor (see Examples 4.1.2 and 4.1.1, respectively). Naturality is the commutativity of the diagram in Figure 5.4, which we shall prove as follows. For a function $f : A \to B$ and an element $x \in A$:

$$\begin{aligned}
&(\eta_B \circ \mathsf{I}_{\mathrm{M}}(f))(x) \\
&\quad = \quad \text{(by (2.4) with } f = \mathsf{I}_{\mathrm{M}}(f) \text{ and } g = \eta_B) \\
&\eta_B(\mathsf{I}_{\mathrm{M}}(f)(x)) \\
&\quad = \quad \text{(by (4.6))} \\
&\eta_B(f(x)) \\
&\quad = \quad \text{(by (5.2) with } A = B \text{ and } x = f(x)) \\
&\{f(x)\} \\
&\quad = \quad \text{(by (4.4) with } X = \{x\}) \\
&\mathsf{P}_{\mathrm{M}}(f)(\{x\}) \\
&\quad = \quad \text{(by (5.2))} \\
&\mathsf{P}_{\mathrm{M}}(f)(\eta_A(x)) \\
&\quad = \quad \text{(by (2.4) with } f = \eta_A \text{ and } g = \mathsf{P}_{\mathrm{M}}(f)) \\
&(\mathsf{P}_{\mathrm{M}}(f) \circ \eta_A)(x)
\end{aligned}$$

$$\begin{array}{ccc}
\mathsf{I}_{\mathrm{O}}(A) & \xrightarrow{\eta_A} & \mathsf{P}_{\mathrm{O}}(A) \\
\Big\downarrow{\scriptstyle \mathsf{I}_{\mathrm{M}}(f)} & & \Big\downarrow{\scriptstyle \mathsf{P}_{\mathrm{M}}(f)} \\
\mathsf{I}_{\mathrm{O}}(B) & \xrightarrow[\eta_B]{} & \mathsf{P}_{\mathrm{O}}(B)
\end{array}$$

Figure 5.4: Naturality of the η natural transformation.

5.2 Natural Transformations in Haskell

Polymorphic functions in functional programming correspond to natural transformations. The kind of polymorphism we refer to is parametric polymorphism. A parametrically polymorphic or generic function is a function whose "parameters can have more than one type." Such a function "works uniformly on a range of types," which is "achieved by type parameters" (Cardelli and Wegner 1985, p. 476).

In Haskell, polymorphic functions can be thought of as functions between functors in order to relate them to natural transformations. Given two functors with type constructors `f` and `g`, a natural transformation `tau` is a polymorphic function `tau :: f a -> g a` or, using the `ExplicitForAll` language option, a family of functions indexed by Haskell types:

```
tau :: forall a. f a -> g a
```

More precisely, each `tau` function is a component of a natural transformation. The naturality of `tau` is the commutativity of the diagram in Figure 5.5, that is, for all functions `f :: a -> b`:

```
tau . fmap f = fmap f . tau
```

An intuitive idea behind naturality is that "terms evaluated in related environments yield related values" (Wadler 1989, p. 347).

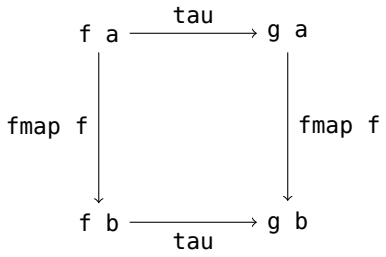


Figure 5.5: Naturality in Haskell.

Naturality in Haskell is the "theorem for free" of (Wadler 1989), which is also known as parametricity due to its relation with parametric polymorphism. Given the type of a polymorphic function, it is possible to conclude that it satisfies its naturality. However, it is important to note that even

though parametricity guarantees that a polymorphic function satisfies its naturality, it does not provide a proof of it. Parametricity does not require the definition of a polymorphic function (only its type), but proving naturality does.

As examples, we consider the identity natural transformation, the `head` and `last` functions, which show that proofs of naturality differ according to the definition, and the `concat` function, which is one of the examples of theorems from types in (Wadler 1989, p. 349).

Example 5.2.1. The identity function of Haskell is a natural transformation. Naturality is the identity axiom. See Example 5.1.1, which establishes that the identity morphisms are the components of a natural transformation regardless of which category it belongs to.

Example 5.2.2. We have already talked about the `head` function, which is a natural transformation from the `[]` (list) functor into the `Maybe` functor. Naturality is the commutativity of the diagram in Figure 5.1. Here is its proof.

Case `[]`:

```
(head . fmap f) []
```
  = (by definition of `(.)`)
```
head (fmap f [])
```
  = (by definition of `fmap`)
```
head []
```
  = (by definition of `head`)
```
Nothing
```
  = (by definition of `fmap`)
```
fmap f Nothing
```
  = (by definition of `head`)
```
fmap f (head [])
```
  = (by definition of `(.)`)
```
(fmap f . head) []
```

Case `(x:xs)`:

```
(head . fmap f) (x:xs)
```

```
  =  (by definition of (.))
head (fmap f (x:xs))
  =  (by definition of fmap)
head (f x : fmap f xs)
  =  (by definition of head)
Just (f x)
  =  (by definition of fmap)
fmap f (Just x)
  =  (by definition of head)
fmap f (head (x:xs))
  =  (by definition of (.))
(fmap f . head) (x:xs)
```

Example 5.2.3. Another natural transformation from the `[]` (list) functor into the `Maybe` functor is the `last` function, which extracts the last element of a list[2]:

```
last :: [a] -> Maybe a
last []     = Nothing
last (x:[]) = Just x
last (_:xs) = last xs
```

Its naturality is the commutativity of the diagram in Figure 5.6, that is, for all functions `f :: a -> b`:

```
last . fmap f = fmap f . last
```

Intuitively, this means that mapping a function over a list and then extracting the last element of the list is equivalent to extracting the last element of the list and then applying the function. In this case, proving naturality requires induction, as follows.

Case `[]`:

```
(last . fmap f) []
```

[2]Note that this is not the standard Haskell `last` function.

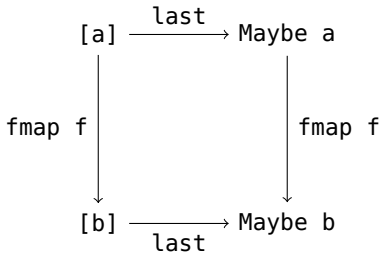


Figure 5.6: Naturality of the `last` function.

```
  =  (by definition of (.))
last (fmap f [])
  =  (by definition of fmap)
last []
  =  (by definition of last)
Nothing
  =  (by definition of fmap)
fmap f Nothing
  =  (by definition of last)
fmap f (last [])
  =  (by definition of (.))
(fmap f . last) []
```

Case `[x]`:

```
(last . fmap f) (x:[])
  =  (by definition of (.))
last (fmap f (x:[]))
  =  (by definition of fmap)
last (f x : fmap f [])
  =  (by definition of fmap)
last (f x : [])
  =  (by definition of last)
```

```
Just (f x)
```
  = (by definition of `fmap`)
```
fmap f (Just x)
```
  = (by definition of `last`)
```
fmap f (last (x:[]))
```
  = (by definition of `(.)`)
```
(fmap f . last) (x:[])
```

Case `(x:y:ys)`:

```
(last . fmap f) (x:y:ys)
```
  = (by definition of `(.)`)
```
last (fmap f (x:y:ys))
```
  = (by definition of `fmap`)
```
last (f x : fmap f (y:ys))
```
  = (by definition of `last`)
```
last (fmap f (y:ys))
```
  = (by definition of `(.)`)
```
(last . fmap f) (y:ys)
```
  = (by inductive hypothesis)
```
(fmap f . last) (y:ys)
```
  = (by definition of `(.)`)
```
fmap f (last (y:ys))
```
  = (by definition of `last`)
```
fmap f (last (x:y:ys))
```
  = (by definition of `(.)`)
```
(fmap f . last) (x:y:ys)
```

Example 5.2.4. We can think of the `concat` function, which concatenates a list of lists, as a natural transformation from the `[] . []` functor and into the `[]` functor. The type signature of this function is:

```
concat :: [[a]] -> [a]
```

In this case, naturality is given by, for all functions `f :: a -> b`:

```
fmap f . concat = concat . fmap (fmap f)
```

That is, the diagram in Figure 5.7 is commutative, which holds by parametricity.

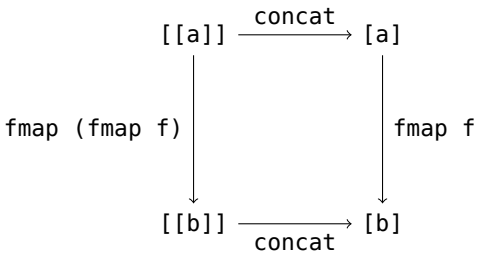


Figure 5.7: Naturality of the `concat` function.

5.3 References

The definition of a natural transformation is based on (Mac Lane 1998, p. 16; Poigné 1992, pp. 435–436), the η natural transformation is taken from (Marquis 2013, p. 11), and the statement that polymorphic functions in functional programming correspond to natural transformations is based on, for instance, (Bird and de Moor 1997, p. 34; Elkins 2009, p. 78; Poigné 1992, pp. 435, 436; Rydeheard 1986b, pp. 48, 49; Rydeheard and Burstall 1988, p. 113; Wadler 1989, p. 350).

Chapter 6

Monads and Kleisli Triples

> La Monade, dont nous parlerons icy, n'est autre chose qu'une substance simple...
>
> —Leibniz (1714, par. 1)

In Haskell, given two types `a` and `b`, the Cartesian product of a list `xs` of elements of type `a` and a list `ys` of elements of type `b` is defined to be the list of tuples `(x,y)` of type `(a,b)` for which `x` belongs to `xs` and `y` belongs to `ys`, that is, using a list comprehension:

```
cartesian :: [a] -> [b] -> [(a,b)]
cartesian xs ys = [(x,y) | x <- xs, y <- ys]
```

Or, equivalently, desugaring the list comprehension:

```
cartesian xs ys = xs >>= \x -> ys >>= \y -> return (x,y)
```

This is but one simple example to show that "a monad is often an obvious and useful tool to help solve a problem" (O'Sullivan, Goerzen, and Stewart 2008, p. 325), and that "many common programming patterns have a monadic structure" (O'Sullivan, Goerzen, and Stewart 2008, p. 328).

In this chapter, we explore monads and Kleisli triples in order to be able to conceptualize and better understand monads in functional programming. As motivation, in **Hask**, "we distinguish the object `a` of values (of type `a`) from the object `m a` of computations (of type `a`) (...). In particular, we identify the type `a` with the object of values (of type `a`) and obtain the object

of computations (of type `a`) by applying a unary type constructor `m` to `a`. We call `m` a notion of computation," which is just a qualitative description of a computation (Moggi 1989, p. 17), "since it abstracts away from the type of values computations may produce" (Moggi 1991, pp. 57–58). There are many notions of computation. For instance, the `Maybe` and `[]` (list) type constructors represent the notions of partiality and nondeterminism, respectively. Instead of studying a specific `m`, we focus on monads, which describe the general properties common to such notions of computation (Moggi 1991, p. 58).

In the remainder of this chapter, we define the concepts of monad and Kleisli triple, prove their equivalence, and study both constructs in Haskell and Agda. We should note that, terminologically, category-theoretical monads and monads in Haskell, which actually correspond to Kleisli triples, are not the same thing.

6.1 Monads and Kleisli Triples

In this section, we define the concepts of monad and Kleisli triple, and prove their equivalence. Kleisli triples "are easy to justify from a computational perspective," but monads "are more widely used in (...) category theory and have the advantage of being defined only in terms of functors and natural transformations, which make them more suitable for abstract manipulation" (Moggi 1991, p. 60).

6.1.1 Monads

First, we describe monads. Despite the fact that monads in functional programming correspond to Kleisli triples, categorical monads are likely more appropriate for analyzing them from the perspective of category theory.

Definition 6.1. Let $\mathcal{C}$ be a category. A monad $\mathsf{T} = (\mathsf{T}, \eta, \mu)$ in $\mathcal{C}$ consists of an endofunctor $\mathsf{T} : \mathcal{C} \to \mathcal{C}$, together with two natural transformations

$$\eta : \mathsf{I} \to \mathsf{T} : \mathcal{C} \to \mathcal{C} \tag{6.1}$$

and

$$\mu : \mathsf{T} \circ \mathsf{T} \to \mathsf{T} : \mathcal{C} \to \mathcal{C}, \tag{6.2}$$

called unit and multiplication of the monad, respectively, such that, for all objects a,

$$\mu_a \circ \mu_{\mathsf{T}_{\mathrm{O}}(a)} = \mu_a \circ \mathsf{T}_{\mathrm{M}}(\mu_a), \tag{6.3}$$

$$\mu_a \circ \eta_{\mathsf{T}_{\mathrm{O}}(a)} = \mathrm{id}_{\mathsf{T}_{\mathrm{O}}(a)}\,, \tag{6.4}$$

and

$$\mu_a \circ \mathsf{T}_{\mathrm{M}}(\eta_a) = \mathrm{id}_{\mathsf{T}_{\mathrm{O}}(a)}\,, \tag{6.5}$$

that is, the diagrams in Figures 6.1 and 6.2[1] are commutative. Since η and μ are natural transformations, then, for all morphisms $f : a \to \mathsf{T}_{\mathrm{O}}(b)$,

$$\eta_{\mathsf{T}_{\mathrm{O}}(b)} \circ f = \mathsf{T}_{\mathrm{M}}(f) \circ \eta_a \tag{6.6}$$

and

$$\mu_{\mathsf{T}_{\mathrm{O}}(b)} \circ \mathsf{T}_{\mathrm{M}}(\mathsf{T}_{\mathrm{M}}(f)) = \mathsf{T}_{\mathrm{M}}(f) \circ \mu_a, \tag{6.7}$$

that is, the diagrams in Figures 6.3 and 6.4 are commutative.

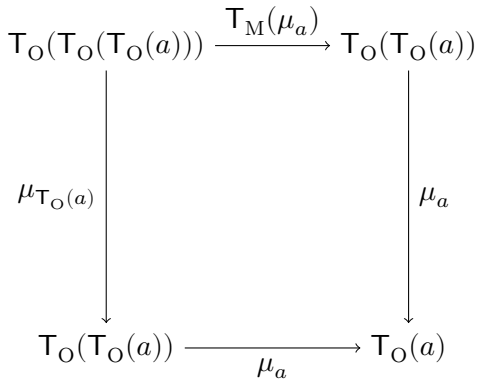


Figure 6.1: Monadic associativity.

Remark 6.1. Formally, the definition of a monad is like that of a monoid as described in Example 2.1.4. Let $\mathsf{T} = (\mathsf{T}, \eta, \mu)$ be a monad in a category $\mathcal{C}$. The endofunctor T is the set of elements of a monoid $M = (\mathsf{T}, \mu, \eta)$. The multiplication natural transformation, μ, is the associative binary operation of the monoid, and it has an identity, the unit natural transformation, η. Thus, the diagram in Figure 6.1 is the associative law for the monad, while the diagram in Figure 6.2 expresses the left and right unit laws (Mac Lane 1998, p. 138).

[1] We use double arrows in commutative diagrams to represent equality of objects.

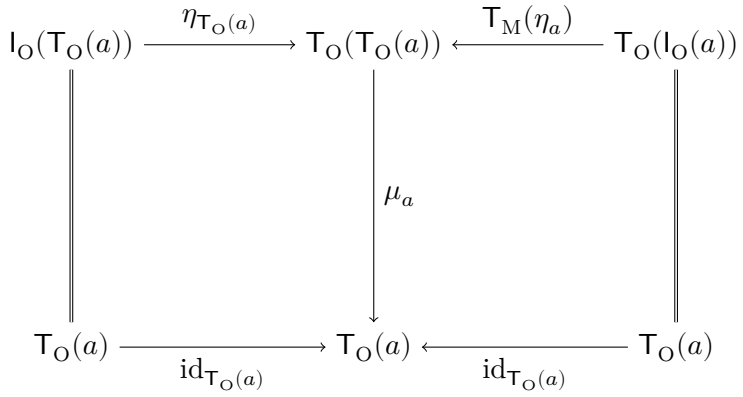


Figure 6.2: Monadic unity.

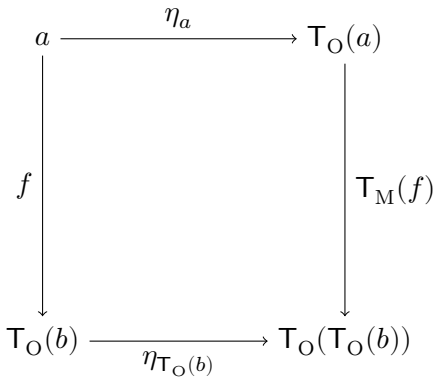


Figure 6.3: Naturality of the η natural transformation.

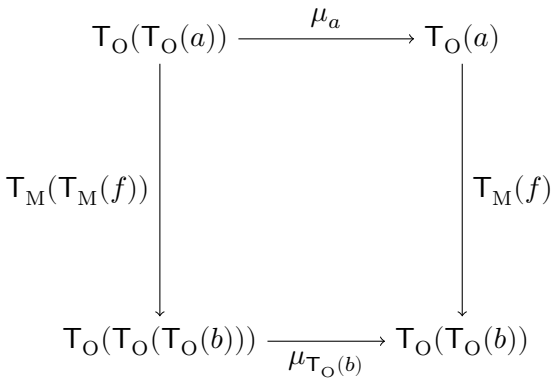


Figure 6.4: Naturality of the μ natural transformation.

As examples of monads, we consider the identity monad, which is just a reformulation of the identity functor for a given category.

Example 6.1.1. Let $\mathcal{C}$ be a category. The identity or trivial monad of $\mathcal{C}$ is $\mathsf{I} = (\mathsf{I}, \mathrm{id}, \mathrm{id})$, that is, the identity endofunctor (see Example 4.1.2) and the identity mapping of $\mathcal{C}$. Equations (6.3), (6.4), and (6.5) hold by (2.2), and (6.6) and (6.7) hold because id is a natural transformation (see Example 5.1.1).

6.1.2 Kleisli Triples

Now, we describe Kleisli triples, which "are just an alternative description for monads" (Moggi 1991, p. 60).

Definition 6.2. Let $\mathcal{C}$ be a category. A Kleisli triple $\mathsf{T} = (\mathsf{T}_{\mathrm{O}}, \eta, \_^*)$ in $\mathcal{C}$, where η is a transformation (6.1), assigns to each object a an object $\mathsf{T}_{\mathrm{O}}(a)$, and to each morphism $f : a \to \mathsf{T}_{\mathrm{O}}(b)$ a morphism $f^* : \mathsf{T}_{\mathrm{O}}(a) \to \mathsf{T}_{\mathrm{O}}(b)$, such that, for all morphisms $f : a \to \mathsf{T}_{\mathrm{O}}(b)$ and $g : b \to \mathsf{T}_{\mathrm{O}}(c)$,

$$g^* \circ f^* = (g^* \circ f)^*, \tag{6.8}$$

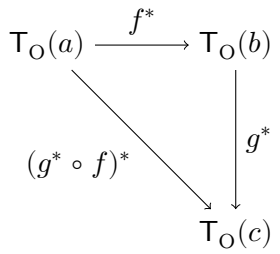
for all morphisms $f : a \to \mathsf{T}_{\mathrm{O}}(b)$,

$$f^* \circ \eta_a = f, \tag{6.9}$$

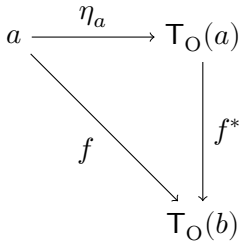
and, for all objects a,

$$\eta_a^* = \mathrm{id}_{\mathsf{T}_{\mathrm{O}}(a)}, \tag{6.10}$$

that is, the diagrams in Figures 6.5a and 6.5b are commutative, and (6.10).



(a) Kleisli triple associativity.



(b) Kleisli triple unity.

Figure 6.5: Kleisli triple laws.

Remark 6.2. From a computational perspective, "η_a is the inclusion of values into computations and f^* is the extension of a function f from values to computations to a function from computations to computations, which first evaluates a computation and then applies f to the resulting value" (Moggi 1991, p. 59).

Remark 6.3. Unlike the definition of a monad, a Kleisli triple does not require an endofunctor, just an object mapping, and its unit is not required to be defined as a natural transformation, just a transformation.

As examples, we describe the identity or trivial monad as a Kleisli triple.

Example 6.1.2. Let $\mathcal{C}$ be a category. The identity or trivial Kleisli triple of $\mathcal{C}$ is $\mathsf{I} = (\mathsf{I}_{\mathrm{O}}, \mathrm{id}, \mathsf{I}_{\mathrm{M}})$, that is, the object mapping of the identity endofunctor (see Example 4.1.2), the identity mapping, and the morphism mapping of the identity endofunctor of $\mathcal{C}$. Equation (6.8) holds by (2.1), and (6.9) and (6.10) hold by (2.2). This is just an alternative description of the identity monad (see Example 6.1.1).

6.1.3 Equivalence of Monads and Kleisli Triples

Algebraic theories in clone form (Manes 1976, p. 24), which we shall refer to as monads in clone form, are yet another alternative description for monads. Manes (1976, pp. 26–29) thoroughly proved the equivalence between monads and monads in clone form, and Moggi (1991, p. 61) stated the equivalence between monads and Kleisli triples, and proved it without going into details. The following theorems demonstrate in a thorough manner that monads and Kleisli triples are equivalent. As illustrated in Figure 6.6, these

proofs are enough for stating the equivalence between the three alternative descriptions.

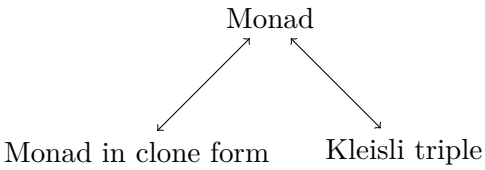


Figure 6.6: Equivalence of monads, monads in clone form, and Kleisli triples.

First, we state that a Kleisli triple can be obtained from a monad. Since monads are defined in terms of functors and natural transformations, the following lemma is simpler than its converse.

Lemma 6.1. *Let* $\mathsf{T} = (\mathsf{T}, \eta, \mu)$ *be a monad in a category* $\mathcal{C}$*. Then*

$$\mathsf{T} = (\mathsf{T}_{\mathrm{O}}, \eta, \_^*),$$

where T_{O} *is the object mapping of the endofunctor* T*,* η *is the underlying transformation of the natural transformation* η*, and* $\_^*$ *assigns to each morphism* $f : a \to \mathsf{T}_{\mathrm{O}}(b)$ *a morphism* $f^* : \mathsf{T}_{\mathrm{O}}(a) \to \mathsf{T}_{\mathrm{O}}(b)$ *defined by*

$$f^* = \mu_b \circ \mathsf{T}_{\mathrm{M}}(f), \tag{6.11}$$

is a Kleisli triple in $\mathcal{C}$*.*

Proof. First, we prove that (6.8) holds:

$$\begin{aligned}
&g^* \circ f^* \\
&\quad = \quad \text{(by (6.11) with } f = f \text{ and } f = g) \\
&\mu_c \circ \mathsf{T}_{\mathrm{M}}(g) \circ \mu_b \circ \mathsf{T}_{\mathrm{M}}(f) \\
&\quad = \quad \text{(by (6.7) with } f = g) \\
&\mu_c \circ \mu_{\mathsf{T}_{\mathrm{O}}(c)} \circ \mathsf{T}_{\mathrm{M}}(\mathsf{T}_{\mathrm{M}}(g)) \circ \mathsf{T}_{\mathrm{M}}(f) \\
&\quad = \quad \text{(by (4.2) with } g = \mathsf{T}_{\mathrm{M}}(g)) \\
&\mu_c \circ \mu_{\mathsf{T}_{\mathrm{O}}(c)} \circ \mathsf{T}_{\mathrm{M}}(\mathsf{T}_{\mathrm{M}}(g) \circ f) \\
&\quad = \quad \text{(by (6.3) with } a = c) \\
&\mu_c \circ \mathsf{T}_{\mathrm{M}}(\mu_c) \circ \mathsf{T}_{\mathrm{M}}(\mathsf{T}_{\mathrm{M}}(g) \circ f) \\
&\quad = \quad \text{(by (4.2) with } f = \mathsf{T}_{\mathrm{M}}(g) \circ f \text{ and } g = \mu_c)
\end{aligned}$$

$$
\begin{aligned}
&\mu_c \circ \mathsf{T}_{\mathrm{M}}(\mu_c \circ \mathsf{T}_{\mathrm{M}}(g) \circ f) \\
&\quad = \quad \text{(by (6.11) with } f = g \text{ and } f = g^* \circ f) \\
&(g^* \circ f)^*
\end{aligned}
$$

Now, we prove that (6.9) holds:

$$
\begin{aligned}
&f^* \circ \eta_a \\
&\quad = \quad \text{(by (6.11))} \\
&\mu_b \circ \mathsf{T}_{\mathrm{M}}(f) \circ \eta_a \\
&\quad = \quad \text{(by (6.6))} \\
&\mu_b \circ \eta_{\mathsf{T}_{\mathrm{O}}(b)} \circ f \\
&\quad = \quad \text{(by (6.4) with } a = b) \\
&\mathrm{id}_{\mathsf{T}_{\mathrm{O}}(b)} \circ f \\
&\quad = \quad \text{(by (2.2))} \\
&f
\end{aligned}
$$

Finally, we prove that (6.10) holds:

$$
\begin{aligned}
&\eta_a^* \\
&\quad = \quad \text{(by (6.11) with } f = \eta_a) \\
&\mu_a \circ \mathsf{T}_{\mathrm{M}}(\eta_a) \\
&\quad = \quad \text{(by (6.5))} \\
&\mathrm{id}_{\mathsf{T}_{\mathrm{O}}(a)}
\end{aligned}
$$

□

The object mapping of a Kleisli triple can be extended to define an endofunctor, as proven in the following lemma.

Lemma 6.2. *Let $\mathcal{C}$ be a category. If $\mathsf{T} = (\mathsf{T}_{\mathrm{O}}, \eta, \_^*)$ is a Kleisli triple in $\mathcal{C}$, then $\mathsf{T} = (\mathsf{T}_{\mathrm{O}}, \mathsf{T}_{\mathrm{M}})$, which assigns to each morphism $f : a \to b$ a morphism $\mathsf{T}_{\mathrm{M}}(f) : \mathsf{T}_{\mathrm{O}}(a) \to \mathsf{T}_{\mathrm{O}}(b)$ defined by*

$$\mathsf{T}_{\mathrm{M}}(f) = (\eta_b \circ f)^*, \tag{6.12}$$

is an endofunctor in $\mathcal{C}$.

Proof. In the first place, we prove that (4.1) holds:

$$
\begin{aligned}
&\mathsf{T}_{\mathrm{M}}(\mathrm{id}_a) \\
&\quad = \quad \text{(by (6.12) with } f = \mathrm{id}_a\text{)} \\
&(\eta_a \circ \mathrm{id}_a)^* \\
&\quad = \quad \text{(by (2.2) with } f = \eta_a\text{)} \\
&\eta_a^*
\end{aligned}
$$

In the second place, we prove that (4.2) holds:

$$
\begin{aligned}
&\mathsf{T}_{\mathrm{M}}(g \circ f) \\
&\quad = \quad \text{(by (6.12) with } f = g \circ f\text{)} \\
&(\eta_c \circ g \circ f)^* \\
&\quad = \quad \text{(by (6.9) with } f = \eta_c \circ g\text{)} \\
&((\eta_c \circ g)^* \circ \eta_b \circ f)^* \\
&\quad = \quad \text{(by (6.8) with } f = \eta_b \circ f \text{ and } g = \eta_c \circ g\text{)} \\
&(\eta_c \circ g)^* \circ (\eta_b \circ f)^* \\
&\quad = \quad \text{(by (6.12) with } f = f \text{ and } f = g\text{)} \\
&\mathsf{T}_{\mathrm{M}}(g) \circ \mathsf{T}_{\mathrm{M}}(f)
\end{aligned}
$$

□

Now, since T is an endofunctor, we state and prove that η is a natural transformation.

Lemma 6.3. *If $\mathsf{T} = (\mathsf{T}_{\mathrm{O}}, \eta, \_^*)$ is a Kleisli triple in a category $\mathcal{C}$, then the transformation η is natural.*

Proof. We prove that (5.1) holds for η:

$$
\begin{aligned}
&\eta_{\mathsf{T}_{\mathrm{O}}(b)} \circ f \\
&\quad = \quad \text{(by (6.9) with } f = \eta_{\mathsf{T}_{\mathrm{O}}(b)} \circ f\text{)} \\
&(\eta_{\mathsf{T}_{\mathrm{O}}(b)} \circ f)^* \circ \eta_a \\
&\quad = \quad \text{(by (6.12))} \\
&\mathsf{T}_{\mathrm{M}}(f) \circ \eta_a
\end{aligned}
$$

□

Next, we define the multiplication of a monad given a Kleisli triple, and prove that it is a natural transformation.

Lemma 6.4. *Let $\mathcal{C}$ be a category. If $\mathsf{T} = (\mathsf{T}_\mathrm{O}, \eta, \_^*)$ is a Kleisli triple in $\mathcal{C}$, then a transformation μ which assigns to each object a a morphism $\mu_a : \mathsf{T}_\mathrm{O}(\mathsf{T}_\mathrm{O}(a)) \to \mathsf{T}_\mathrm{O}(a)$ defined by*

$$\mu_a = \mathrm{id}^*_{\mathsf{T}_\mathrm{O}(a)} \tag{6.13}$$

is a natural transformation $\mu : \mathsf{T} \circ \mathsf{T} \to \mathsf{T} : \mathcal{C} \to \mathcal{C}$.

Proof. We prove that (5.1) holds for μ:

$$\begin{aligned}
&\mu_{\mathsf{T}_\mathrm{O}(b)} \circ \mathsf{T}_\mathrm{M}(\mathsf{T}_\mathrm{M}(f)) \\
&\quad = \quad \text{(by (6.13) with } a = \mathsf{T}_\mathrm{O}(b)) \\
&\mathrm{id}^*_{\mathsf{T}_\mathrm{O}(\mathsf{T}_\mathrm{O}(b))} \circ \mathsf{T}_\mathrm{M}(\mathsf{T}_\mathrm{M}(f)) \\
&\quad = \quad \text{(by (6.12))} \\
&\mathrm{id}^*_{\mathsf{T}_\mathrm{O}(\mathsf{T}_\mathrm{O}(b))} \circ \mathsf{T}_\mathrm{M}((\eta_{\mathsf{T}_\mathrm{O}(b)} \circ f)^*) \\
&\quad = \quad \text{(by (6.12) with } f = (\eta_{\mathsf{T}_\mathrm{O}(b)} \circ f)^*) \\
&\mathrm{id}^*_{\mathsf{T}_\mathrm{O}(\mathsf{T}_\mathrm{O}(b))} \circ (\eta_{\mathsf{T}_\mathrm{O}(\mathsf{T}_\mathrm{O}(b))} \circ (\eta_{\mathsf{T}_\mathrm{O}(b)} \circ f)^*)^* \\
&\quad = \quad \text{(by (6.8) with } f = \eta_{\mathsf{T}_\mathrm{O}(\mathsf{T}_\mathrm{O}(b))} \circ (\eta_{\mathsf{T}_\mathrm{O}(b)} \circ f)^* \text{ and } g = \mathrm{id}_{\mathsf{T}_\mathrm{O}(\mathsf{T}_\mathrm{O}(b))}) \\
&(\mathrm{id}^*_{\mathsf{T}_\mathrm{O}(\mathsf{T}_\mathrm{O}(b))} \circ \eta_{\mathsf{T}_\mathrm{O}(\mathsf{T}_\mathrm{O}(b))} \circ (\eta_{\mathsf{T}_\mathrm{O}(b)} \circ f)^*)^* \\
&\quad = \quad \text{(by (6.9) with } f = \mathrm{id}_{\mathsf{T}_\mathrm{O}(\mathsf{T}_\mathrm{O}(b))}) \\
&(\mathrm{id}_{\mathsf{T}_\mathrm{O}(\mathsf{T}_\mathrm{O}(b))} \circ (\eta_{\mathsf{T}_\mathrm{O}(b)} \circ f)^*)^* \\
&\quad = \quad \text{(by (2.2) with } f = (\eta_{\mathsf{T}_\mathrm{O}(b)} \circ f)^*) \\
&((\eta_{\mathsf{T}_\mathrm{O}(b)} \circ f)^*)^* \\
&\quad = \quad \text{(by (2.2) with } f = (\eta_{\mathsf{T}_\mathrm{O}(b)} \circ f)^*) \\
&((\eta_{\mathsf{T}_\mathrm{O}(b)} \circ f)^* \circ \mathrm{id}_{\mathsf{T}_\mathrm{O}(a)})^* \\
&\quad = \quad \text{(by (6.8) with } f = \mathrm{id}_{\mathsf{T}_\mathrm{O}(a)} \text{ and } g = \eta_{\mathsf{T}_\mathrm{O}(b)} \circ f) \\
&(\eta_{\mathsf{T}_\mathrm{O}(b)} \circ f)^* \circ \mathrm{id}^*_{\mathsf{T}_\mathrm{O}(a)} \\
&\quad = \quad \text{(by (6.12))} \\
&\mathsf{T}_\mathrm{M}(f) \circ \mathrm{id}^*_{\mathsf{T}_\mathrm{O}(a)} \\
&\quad = \quad \text{(by (6.13))} \\
&\mathsf{T}_\mathrm{M}(f) \circ \mu_a
\end{aligned}$$

□

Now, we state that a monad can be obtained from a Kleisli triple using the constructions from the three lemmas above.

Lemma 6.5. *Let* $\mathsf{T} = (\mathsf{T}_{\mathrm{O}}, \eta, \_^*)$ *be a Kleisli triple in a category* $\mathcal{C}$*. Then*

$$\mathsf{T} = (\mathsf{T}, \eta, \mu),$$

where T *is the endofunctor in* $\mathcal{C}$ *defined by Lemma 6.2,* η *is the transformation* η *regarded as a natural transformation (see Lemma 6.3), and* $\mu : \mathsf{T} \circ \mathsf{T} \to \mathsf{T} : \mathcal{C} \to \mathcal{C}$ *is the natural transformation defined by Lemma 6.4, is a monad in* $\mathcal{C}$*.*

Proof. First, we prove that (6.3) holds:

$$\begin{aligned}
&\mu_a \circ \mu_{\mathsf{T}_{\mathrm{O}}(a)} \\
&\quad = \quad \text{(by (6.13) with } a = a \text{ and } a = \mathsf{T}_{\mathrm{O}}(a)) \\
&\mathrm{id}^*_{\mathsf{T}_{\mathrm{O}}(a)} \circ \mathrm{id}^*_{\mathsf{T}_{\mathrm{O}}(\mathsf{T}_{\mathrm{O}}(a))} \\
&\quad = \quad \text{(by (6.8) with } f = \mathrm{id}_{\mathsf{T}_{\mathrm{O}}(\mathsf{T}_{\mathrm{O}}(a))} \text{ and } g = \mathrm{id}_{\mathsf{T}_{\mathrm{O}}(a)}) \\
&(\mathrm{id}^*_{\mathsf{T}_{\mathrm{O}}(a)} \circ \mathrm{id}_{\mathsf{T}_{\mathrm{O}}(\mathsf{T}_{\mathrm{O}}(a))})^* \\
&\quad = \quad \text{(by (2.2) with } f = \mathrm{id}^*_{\mathsf{T}_{\mathrm{O}}(a)}) \\
&(\mathrm{id}^*_{\mathsf{T}_{\mathrm{O}}(a)})^* \\
&\quad = \quad \text{(by (2.2) with } f = \mathrm{id}^*_{\mathsf{T}_{\mathrm{O}}(a)}) \\
&(\mathrm{id}_{\mathsf{T}_{\mathrm{O}}(a)} \circ \mathrm{id}^*_{\mathsf{T}_{\mathrm{O}}(a)})^* \\
&\quad = \quad \text{(by (6.9) with } f = \mathrm{id}_{\mathsf{T}_{\mathrm{O}}(a)}) \\
&(\mathrm{id}^*_{\mathsf{T}_{\mathrm{O}}(a)} \circ \eta_{\mathsf{T}_{\mathrm{O}}(a)} \circ \mathrm{id}^*_{\mathsf{T}_{\mathrm{O}}(a)})^* \\
&\quad = \quad \text{(by (6.8) with } f = \eta_{\mathsf{T}_{\mathrm{O}}(a)} \circ \mathrm{id}^*_{\mathsf{T}_{\mathrm{O}}(a)} \text{ and } g = \mathrm{id}_{\mathsf{T}_{\mathrm{O}}(a)}) \\
&\mathrm{id}^*_{\mathsf{T}_{\mathrm{O}}(a)} \circ (\eta_{\mathsf{T}_{\mathrm{O}}(a)} \circ \mathrm{id}^*_{\mathsf{T}_{\mathrm{O}}(a)})^* \\
&\quad = \quad \text{(by (6.13))} \\
&\mu_a \circ (\eta_{\mathsf{T}_{\mathrm{O}}(a)} \circ \mu_a)^* \\
&\quad = \quad \text{(by (6.12) with } f = \mu_a) \\
&\mu_a \circ \mathsf{T}_{\mathrm{M}}(\mu_a)
\end{aligned}$$

Now, we prove that (6.4) holds:

$$\begin{aligned}
&\mu_a \circ \eta_{\mathsf{T}_{\mathrm{O}}(a)} \\
&\quad = \quad \text{(by (6.13))}
\end{aligned}$$

$$
\begin{aligned}
&\mathrm{id}^*_{\mathsf{T}_{\mathrm{O}}(a)} \circ \eta_{\mathsf{T}_{\mathrm{O}}(a)} \\
&\quad = \quad (\text{by (6.9) with } f = \mathrm{id}_{\mathsf{T}_{\mathrm{O}}(a)}) \\
&\mathrm{id}_{\mathsf{T}_{\mathrm{O}}(a)}
\end{aligned}
$$

Finally, we prove that (6.5) holds:

$$
\begin{aligned}
&\mu_a \circ \mathsf{T}_{\mathrm{M}}(\eta_a) \\
&\quad = \quad (\text{by (6.13)}) \\
&\mathrm{id}^*_{\mathsf{T}_{\mathrm{O}}(a)} \circ \mathsf{T}_{\mathrm{M}}(\eta_a) \\
&\quad = \quad (\text{by (6.12) with } f = \eta_a) \\
&\mathrm{id}^*_{\mathsf{T}_{\mathrm{O}}(a)} \circ (\eta_{\mathsf{T}_{\mathrm{O}}(a)} \circ \eta_a)^* \\
&\quad = \quad (\text{by (6.8) with } f = \eta_{\mathsf{T}_{\mathrm{O}}(a)} \circ \eta_a \text{ and } g = \mathrm{id}_{\mathsf{T}_{\mathrm{O}}(a)}) \\
&(\mathrm{id}^*_{\mathsf{T}_{\mathrm{O}}(a)} \circ \eta_{\mathsf{T}_{\mathrm{O}}(a)} \circ \eta_a)^* \\
&\quad = \quad (\text{by (6.9) with } f = \mathrm{id}_{\mathsf{T}_{\mathrm{O}}(a)}) \\
&(\mathrm{id}_{\mathsf{T}_{\mathrm{O}}(a)} \circ \eta_a)^* \\
&\quad = \quad (\text{by (2.2) with } f = \eta_a) \\
&\eta^*_a \\
&\quad = \quad (\text{by (6.10)}) \\
&\mathrm{id}_{\mathsf{T}_{\mathrm{O}}(a)}
\end{aligned}
$$

□

Finally, in the following theorem, we prove that monads and Kleisli triples are equivalent.

Theorem 6.6. *Monads and Kleisli triples are coextensive.*

Proof. The correspondence between monads and Kleisli triples is given by Lemma 6.1, which proves that a Kleisli triple can be derived from a monad, and Lemma 6.5, which proves that a monad can be derived from a Kleisli triple.

□

6.2 Monads and Kleisli Triples in Haskell

When discussing monads, there are three possibilities: monads (in monoid form), Kleisli triples (monads in extension form), and monads in clone form. Kleisli triples are easier to justify from a computational point of view and correspond to the representation that is found in functional programming languages like Haskell and the Agda standard library. On the other hand, monads have some mathematical advantages and are more intuitive in some cases. In this section, we describe both representations in Haskell.

6.2.1 Monads in Haskell

In **Hask**, a monad consists of an endofunctor, together with two parametrically polymorphic functions, as follows[2]:

```
class Functor m => Monad' m where
  return :: a -> m a
  join   :: m (m a) -> m a
```

The endofunctor `m` corresponds to the endofunctor T of a monad, and the `return` and `join` functions correspond to the unit and multiplication natural transformations of a monad, η and μ, respectively. Equation (6.3), monadic associativity, becomes the commutativity of the diagram in Figure 6.7:

```
join . join = join . fmap join
```

Equations (6.4) and (6.5), monadic unity, become the commutativity of the diagram in Figure 6.8:

```
join . return = id = join . fmap return
```

In addition, by parametricity (see Section 5.2), the naturality of the unit natural transformation, (6.6), becomes the free theorem for the `return` function, that is, for all functions `f :: a -> m b`:

```
return . f = fmap f . return
```

[2]Note that this is not a standard Haskell type class.

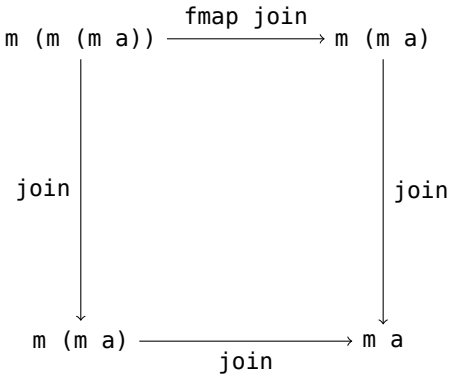


Figure 6.7: Monadic associativity in **Hask**.

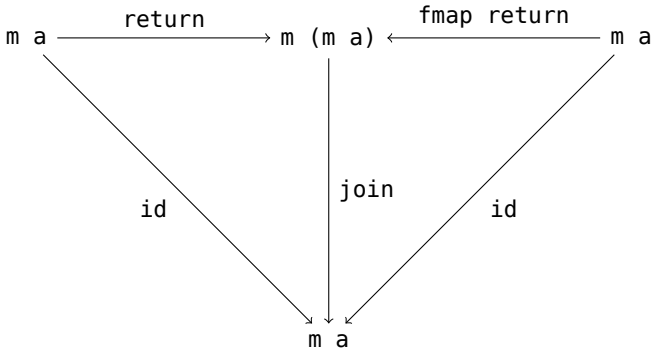


Figure 6.8: Monadic unity in **Hask**.

Similarly, the naturality of the multiplication natural transformation, (6.7), becomes the free theorem for the `join` function, that is:

```
join . fmap (fmap f) = fmap f . join
```

As examples, we consider the identity or trivial monad, which is an intuitive way of learning to use the `Monad'` type class, and two of the most common monads in Haskell, `Maybe` and `[]` (list).

Example 6.2.1. In **Hask**, the `Identity` or trivial monad, which uses the identity endofunctor (see Example 4.2.1), is defined as follows[3]:

[3]Using the `InstanceSigs` language option.

```
instance Monad' Identity where
  return :: a -> Identity a
  return = Identity

  join :: Identity (Identity a) -> Identity a
  join (Identity mx) = mx
```

This instance satisfies the monad laws, as proved in Example 6.1.1.

Example 6.2.2. In **Hask**, the `Maybe` monad, which uses the `Maybe` endofunctor (see Example 4.2.2), is defined as follows:

```
instance Monad' Maybe where
  return :: a -> Maybe a
  return = Just

  join :: Maybe (Maybe a) -> Maybe a
  join Nothing   = Nothing
  join (Just mx) = mx
```

Let us see that this instance satisfies the monad laws. First, we prove that (6.3) holds:

Case `Nothing`:

```
(join . join) Nothing
```
  = (by definition of `(.)`)
```
join (join Nothing)
```
  = (by definition of `join`)
```
join Nothing
```
  = (by definition of `fmap`)
```
join (fmap join Nothing)
```
  = (by definition of `(.)`)
```
(join . fmap join) Nothing
```

Case `(Just mmx)`:

```
(join . join) (Just mmx)
```

```
  =  (by definition of (.))
join (join (Just mmx))
  =  (by definition of join)
join mmx
  =  (by definition of join)
join (Just (join mmx))
  =  (by definition of fmap)
join (fmap join (Just mmx))
  =  (by definition of (.))
(join . fmap join) (Just mmx)
```

Now, we prove that (6.4) holds:

```
(join . return) mx
  =  (by definition of (.))
join (return mx)
  =  (by definition of return)
join (Just mx)
  =  (by definition of join)
mx
  =  (by definition of id)
id mx
```

Finally, we prove that (6.5) holds:

Case `Nothing`:

```
(join . fmap return) Nothing
  =  (by definition of (.))
join (fmap return Nothing)
  =  (by definition of fmap)
join Nothing
  =  (by definition of join)
Nothing
```

= (by definition of `id`)

```
id Nothing
```

Case `(Just x)`:

```
(join . fmap return) (Just x)
```

= (by definition of `(.)`)

```
join (fmap return (Just x))
```

= (by definition of `fmap`)

```
join (Just (return x))
```

= (by definition of `join`)

```
return x
```

= (by definition of `return`)

```
Just x
```

= (by definition of `id`)

```
id (Just x)
```

Example 6.2.3. In **Hask**, the list monad, which uses the list endofunctor (see Example 4.2.3), is declared as follows:

```
instance Monad' [] where
  return :: a -> [a]
  return x = [x]

  join :: [[a]] -> [a]
  join = concat
```

The `concat` function is defined as follows:

```
concat :: [[a]] -> [a]
concat []       = []
concat (xs:xss) = xs ++ concat xss

(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : xs ++ ys
```

Let us see that this instance satisfies the monad laws. In the first place, we prove, by induction, that (6.3) holds:

Case `[]`:

```
(join . join) []
```
= (by definitions of `(.)` and `join`)
```
concat (concat [])
```
= (by definition of `concat`)
```
concat []
```
= (by definition of `fmap`)
```
concat (fmap concat [])
```
= (by definitions of `(.)` and `join`)
```
(join . fmap join) []
```

Case `(xss:xsss)`:

```
(join . join) (xss:xsss)
```
= (by definitions of `(.)` and `join`)
```
concat (concat (xss:xsss))
```
= (by definition of `concat`)
```
concat (xss ++ concat xsss)
```
= (see below)
```
concat xss ++ concat (concat xsss)
```
= (by inductive hypothesis)
```
concat xss ++ concat (fmap concat xsss)
```
= (by definition of `concat`)
```
concat (concat xss : fmap concat xsss)
```
= (by definition of `fmap`)
```
concat (fmap concat (xss:xsss))
```
= (by definitions of `(.)` and `join`)
```
(join . fmap join) (xss:xsss)
```

In this proof, we use the fact that `concat` distributes over `(++)`:

```
concat (xss ++ yss) = concat xss ++ concat yss
```

We shall not prove this property, which could be done by induction on `xss`. Now, we prove that (6.4) holds:

```
(join . return) xs
  =   (by definitions of (.) and join)
concat (return xs)
  =   (by definition of return)
concat [xs]
  =   (by definition of concat)
xs
  =   (by definition of id)
id xs
```

Finally, we prove, by induction, that (6.5) holds:

Case `[]`:

```
(join . fmap return) []
  =   (by definitions of (.) and join)
concat (fmap return [])
  =   (by definition of fmap)
concat []
  =   (by definition of concat)
[]
  =   (by definition of id)
id []
```

Case `(x:xs)`:

```
(join . fmap return) (x:xs)
  =   (by definitions of (.) and join)
concat (fmap return (x:xs))
  =   (by definition of fmap)
```

```
concat (return x : fmap return xs)
```

= (by definition of `concat`)

```
return x ++ concat (fmap return xs)
```

= (by inductive hypothesis)

```
return x ++ id xs
```

= (by definitions of `return` and `id`)

```
[x] ++ xs
```

= (by definition of `(++)`)

```
(x:xs)
```

= (by definition of `id`)

```
id (x:xs)
```

6.2.2 Kleisli Triples in Haskell

In **Hask**, a Kleisli triple consists of a type constructor, and two functions, as follows[4]:

```
class Monad'' m where
  return :: a -> m a
  bind   :: (a -> m b) -> m a -> m b
```

The type constructor `m` corresponds to the object mapping T_O of a Kleisli triple, and the `return` and `bind` functions correspond to the unit natural transformation and the extension mapping of a Kleisli triple, η and $\_^*$, respectively. Equation (6.8), Kleisli triple associativity, becomes the commutativity of the diagram in Figure 6.9a:

```
bind g . bind f = bind (bind g . f)
```

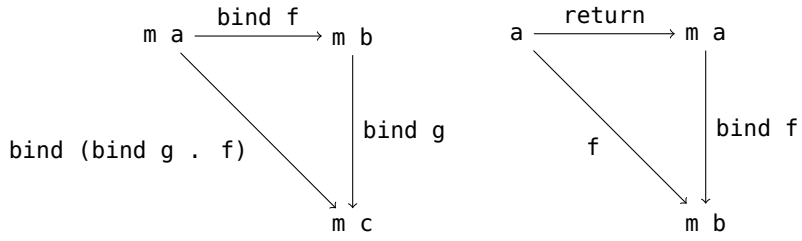
Equation (6.9), Kleisli triple left-unity, is the commutativity of the diagram in Figure 6.9b:

```
bind f . return = f
```

[4] Note that this is not the standard Haskell `Monad` type class.

And (6.10), Kleisli triple right-unity, becomes:

```
bind return = id
```



(a) Kleisli triple associativity. (b) Kleisli triple unity.

Figure 6.9: Kleisli triple laws in **Hask**.

Remark 6.4. Our `Monad''` type class corresponds to the minimal declaration of the standard Haskell `Monad` type class, which is not defined in terms of `bind`:

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b

  (>>)   :: m a -> m b -> m b
  m >> k = m >>= \_ -> k

  fail   :: String -> m a
  fail = error
```

Given a monad with a `bind` function, we can get `(>>=)` as follows:

```
(>>=) :: Monad'' m => m a -> (a -> m b) -> m b
mx >>= f = bind f mx
```

In terms of standard Haskell, the `bind` function is `(<<=)`, which is the same as `(>>=)`, but with the arguments interchanged. Now, let us rewrite the `Monad''` laws using `(>>=)`. In the first place, since

```
(bind g . bind f) mx
```
= (by definition of `(.)`)
```
bind g (bind f mx)
```
= (by definition of `(>>=)`)
```
(bind f mx) >>= g
```
= (by definition of `(>>=)`)
```
(mx >>= f) >>= g
```

and

```
bind (bind g . f) mx
```
= (by definition of `(>>=)`)
```
mx >>= (bind g . f)
```
= (by definition of `(.)`)
```
mx >>= (\x -> bind g (f x))
```
= (by definition of `(>>=)`)
```
mx >>= (\x -> f x >>= g)
```

Kleisli triple associativity becomes

```
(mx >>= f) >>= g = mx >>= (\ x -> f x >>= g)
```

In the second place, since

```
(bind f . return) x
```
= (by definition of `(.)`)
```
bind f (return x)
```
= (by definition of `(>>=)`)
```
return x >>= f
```

Kleisli triple left-unity becomes

```
return x >>= f = f x
```

Finally, since

```
bind return mx
```

= (by definition of `(>>=)`)

```
mx >>= return
```

Kleisli triple right-unity becomes

```
mx >>= return = mx
```

Even though our `Monad''` type class corresponds just to `Monad` with `return` and `(>>=)`, it is worth mentioning the `(>>)` and `fail` functions, neither of which are part of the mathematical definition of a Kleisli triple. In the first place, "`(>>)` is a specialized version of `(>>=)`" (Yorgey 2009, p. 30). In the second place, the `fail` function is a hack used by Haskell to enable failure in a `do` expression, which is a special syntactic construct for `Monad` (Lipovača 2011, p. 273; Peyton Jones 2003, p. 88; Yorgey 2009, p. 30). Besides, the `error` function is closely related to `undefined`[5], which we discussed in Convention 1.

As examples, we consider the `Identity`, `Maybe`, and `[]` monads as Kleisli triples.

Example 6.2.4. In **Hask**, the `Identity` or trivial Kleisli triple is defined as follows:

```
instance Monad'' Identity where
  return :: a -> Identity a
  return = Identity

  bind :: (a -> Identity b) -> Identity a -> Identity b
  bind f (Identity x) = f x
```

This instance satisfies the Kleisli triple laws, as proved in Example 6.1.2.

Example 6.2.5. In **Hask**, the `Maybe` Kleisli triple is defined as follows:

```
instance Monad'' Maybe where
  return :: a -> Maybe a
  return = Just
```

[5] See (Peyton Jones 2003, § 3.1).

```
bind :: (a -> Maybe b) -> Maybe a -> Maybe b
bind _ Nothing  = Nothing
bind f (Just x) = f x
```

Let us see that this instance satisfies the Kleisli triple laws. We prove that (6.8) holds:

Case `Nothing`:

```
bind return Nothing
```
  = (by definition of `bind`)
```
Nothing
```
  = (by definition of `id`)
```
id Nothing
```

Case `(Just x)`:

```
bind return (Just x)
```
  = (by definition of `return`)
```
return x
```
  = (by definition of `return`)
```
Just x
```
  = (by definition of `id`)
```
id (Just x)
```

Now, we prove that (6.9) holds:

```
(bind f . return) x
```
  = (by definition of `(.)`)
```
bind f (return x)
```
  = (by definition of `return`)
```
bind f (Just x)
```
  = (by definition of `bind`)
```
f x
```

Finally, we prove that (6.10) holds:

Case `Nothing`:

```
(bind g . bind f) Nothing
```
= (by definition of `(.)`)
```
bind g (bind f Nothing)
```
= (by definition of `bind`)
```
bind g Nothing
```
= (by definition of `bind`)
```
Nothing
```
= (by definition of `bind`)
```
bind (bind g . f) Nothing
```

Case `(Just x)`:

```
(bind g . bind f) (Just x)
```
= (by definition of `(.)`)
```
bind g (bind f (Just x))
```
= (by definition of `bind`)
```
bind g (f x)
```
= (by definition of `(.)`)
```
(bind g . f) x
```
= (by definition of `bind`)
```
bind (bind g . f) (Just x)
```

Example 6.2.6. In **Hask**, the `[]` (list) Kleisli triple is declared as follows:

```
instance Monad'' [] where
  return :: a -> [a]
  return x = [x]

  bind :: (a -> [b]) -> [a] -> [b]
  bind f xs = concat (map f xs)
```

We shall not prove that this instance satisfies the Kleisli triple laws. Since we have already proved that the list monad satisfies the monad laws (see Example 6.2.3) and that both representations are coextensive, we know that this instance does satisfy the Kleisli triple laws.

6.2.3 Equivalence of Monads and Kleisli Triples in Haskell

We have already proved that monads and Kleisli triples are equivalent. In this subsection, we shall see their correspondence in terms of Haskell. First, let us see that a `Monad''` (and a `Monad`) can be obtained from a `Monad'`, which we demonstrated in Lemma 6.1:

```
bind :: Monad' m => (a -> m b) -> m a -> m b
bind f mx = join (fmap f mx)

(>>=) :: Monad' m => m a -> (a -> m b) -> m b
mx >>= f = join (fmap f mx)
```

Second, let us see that a `Monad'` can be obtained from a `Monad''` (and from a `Monad`). The type constructor of a `Monad''` can be extended to define an endofunctor, which we showed in Lemma 6.2:

```
fmap :: Monad'' m => (a -> b) -> m a -> m b
fmap f mx = bind (return . f) mx

liftM :: Monad m => (a -> b) -> m a -> m b
liftM f mx = mx >>= (return . f)
```

This definition of `fmap` is just (6.12). Finally, the `bind` function of a `Monad''` can be used to define the `join` function of a `Monad'`, which corresponds to Lemma 6.4 or (6.13):

```
join' :: Monad'' m => m (m a) -> m a
join' mmx = bind id mmx

join :: Monad m => m (m a) -> m a
join mmx = mmx >>= id
```

Remark 6.5. In (Kmett 2014), there is a definition of bindable functors (that is, monads without `return`) which includes both `bind` and `join`. Thus, yet another type class declaration for monads and Kleisli triples is:

```
class Functor m => Monad''' m where
  return :: a -> m a
```

```
bind :: (a -> m b) -> m a -> m b
bind f = join . fmap f

join :: m (m a) -> m a
join = bind id
```

6.3 Monads and Kleisli Triples in Agda

In this section, we describe monads and Kleisli triples in Agda. Since we have already described both representations in category theory and Haskell, we shall not go into a lot of detail here.

6.3.1 Monads in Agda

Monads in **Agda** are defined by the `Monad'` record, which can be found in the module Abel.Category.Monad. This declaration includes the monad laws, even naturalities, and the `bind` function, which corresponds to the fact that a Kleisli triple can be obtained from a monad.

```
record Monad' {M : Set → Set} (functor : Functor M) : Set₁ where

  constructor mkMonad'

  open Functor functor using (fmap)

  field

    return : {A : Set} → A → M A

    join   : {A : Set} → M (M A) → M A

    associativity : {A : Set} (mmmx : M (M (M A))) →
                    join (join mmmx) ≡ join (fmap join mmmx)

    unity-left    : {A : Set} (mx : M A) → join (return mx) ≡ mx

    unity-right   : {A : Set} (mx : M A) → join (fmap return mx) ≡ mx
```

```
    naturality-return : {A B : Set} {f : A → M B} (x : A) →
                        return (f x) ≡ fmap f (return x)

    naturality-join   : {A B : Set} {f : A → M B} (mmx : M (M A)) →
                        join (fmap (fmap f) mmx) ≡ fmap f (join mmx)

  bind : {A B : Set} → (A → M B) → M A → M B
  bind f = join ∘ fmap f
```

As examples, we consider all the monads from Section 6.2, that is, the `Identity`, `Maybe`, and `List` monads.

Example 6.3.1 (See module Abel.Data.Identity.Monad)**.** In **Agda**, the `Identity` or trivial monad, which is an instance of the identity monad (see Example 6.1.1), is defined as follows:

```
monad' : Monad' functor
monad' = mkMonad' return join associativity unity-left unity-right
                  naturality-return naturality-join
  where
    return : {A : Set} → A → Identity A
    return = identity

    join : {A : Set} → Identity (Identity A) → Identity A
    join (identity x) = x

    open Functor functor using (fmap)

    associativity : {A : Set}
                    (x : Identity (Identity (Identity A))) →
                    join (join x) ≡ join (fmap join x)
    associativity (identity _) = refl

    unity-left : {A : Set} (x : Identity A) → join (return x) ≡ x
    unity-left _ = refl

    unity-right : {A : Set} (x : Identity A) →
                  join (fmap return x) ≡ x
    unity-right (identity _) = refl
```

```
    naturality-return : {A B : Set} {f : A → Identity B} (x : A) →
                        return (f x) ≡ fmap f (return x)
    naturality-return _ = refl

    naturality-join : {A B : Set} {f : A → Identity B}
                      (x : Identity (Identity A)) →
                      join (fmap (fmap f) x) ≡ fmap f (join x)
    naturality-join (identity _) = refl
```

Example 6.3.2 (See module Abel.Data.Maybe.Monad)**.** In **Agda**, the `Maybe` monad, which corresponds to the `Maybe` monad in **Hask** (see Example 6.2.2), is defined as follows:

```
monad' : Monad' functor
monad' = mkMonad' return join associativity unity-left unity-right
                  naturality-return naturality-join
  where
    return : {A : Set} → A → Maybe A
    return = just

    join : {A : Set} → Maybe (Maybe A) → Maybe A
    join (just mx) = mx
    join nothing   = nothing

    open Functor functor

    associativity : {A : Set} (mmmx : Maybe (Maybe (Maybe A))) →
                    join (join mmmx) ≡ join (fmap join mmmx)
    associativity (just _) = refl
    associativity nothing  = refl

    unity-left : {A : Set} (mx : Maybe A) → join (return mx) ≡ mx
    unity-left _ = refl

    unity-right : {A : Set} (mx : Maybe A) →
                  join (fmap return mx) ≡ mx
    unity-right (just _) = refl
    unity-right nothing  = refl

    naturality-return : {A B : Set} {f : A → Maybe B} (x : A) →
```

```
                      return (f x) ≡ fmap f (return x)
    naturality-return _ = refl

    naturality-join : {A B : Set} {f : A → Maybe B}
                      (mmx : Maybe (Maybe A)) →
                      join (fmap (fmap f) mmx) ≡ fmap f (join mmx)
    naturality-join (just _) = refl
    naturality-join nothing  = refl
```

Example 6.3.3 (See module Abel.Data.List.Monad)**.** In **Agda**, the `List` monad, which corresponds to the list monad in **Hask** (see Example 6.2.3), is defined as follows:

```
monad' : Monad' functor
monad' = mkMonad' return join associativity unity-left unity-right
                  naturality-return naturality-join
  where
    return : {A : Set} → A → List A
    return x = x ∷ []

    join : {A : Set} → List (List A) → List A
    join = concat

    open Functor functor using (fmap)

    associativity : {A : Set} (xsss : List (List (List A))) →
                    join (join xsss) ≡ join (fmap join xsss)
    associativity []                        = refl
    associativity ([] ∷ xsss)               = associativity xsss
    associativity (([] ∷ xss) ∷ xsss)       =
      associativity (xss ∷ xsss)
    associativity (((x ∷ xs) ∷ xss) ∷ xsss) =
      cong (_∷_ x) (associativity ((xs ∷ xss) ∷ xsss))

    unity-left : {A : Set} (xs : List A) → join (return xs) ≡ xs
    unity-left []       = refl
    unity-left (x ∷ xs) = cong (_∷_ x) (unity-left xs)

    unity-right : {A : Set} (xs : List A) →
                  join (fmap return xs) ≡ xs
```

```
  unity-right []       = refl
  unity-right (x :: xs) = cong (_::_ x) (unity-right xs)

  naturality-return : {A B : Set} {f : A → List B} (x : A) →
                      return (f x) ≡ fmap f (return x)
  naturality-return _ = refl

  naturality-join : {A B : Set} {f : A → List B}
                    (xss : List (List A)) →
                    join (fmap (fmap f) xss) ≡ fmap f (join xss)
  naturality-join         []               = refl
  naturality-join         ([] :: xss)      = naturality-join xss
  naturality-join {f = f} ((x :: xs) :: xss) =
    cong (_::_ (f x)) (naturality-join (xs :: xss))
```

6.3.2 Kleisli Triples in Agda

Kleisli triples in **Agda** are defined by the `Monad''` record, which can be found in the module Abel.Category.Monad. The following definition corresponds to the `Monad` type class declaration in Haskell. As usual, this definition includes the appropriate associativity and unity laws, and naturalities, which means that any instance of the `Monad` type class *is* a Kleisli triple.

```
record Monad'' (M : Set → Set) : Set₁ where

  constructor mkMonad''

  field

    return : {A : Set} → A → M A

    bind   : {A B : Set} → (A → M B) → M A → M B

    associativity : {A B C : Set} {f : A → M B} {g : B → M C}
                    (mx : M A) →
                    bind g (bind f mx) ≡ bind (bind g ∘ f) mx

    unity-left    : {A B : Set} {f : A → M B} (x : A) →
                    bind f (return x) ≡ f x
```

```
    unity-right   : {A : Set} (mx : M A) → bind return mx ≡ mx

  infixr 1 _=<<_

  _=<<_ : {A B : Set} → (A → M B) → M A → M B
  _=<<_ = bind

  infixl 1 _>>=_ _>>_

  _>>=_ : {A B : Set} → M A → (A → M B) → M B
  mx >>= f = bind f mx

  _>>_ : {A B : Set} → M A → M B → M B
  mx >> my = mx >>= λ _ → my

  fmap : {A B : Set} → (A → B) → M A → M B
  fmap f = bind (return ∘ f)

  join : ∀ {A} → M (M A) → M A
  join = bind id
```

As with Kleisli triples in **Hask**, we use the `bind` function instead of the `_>>=_` operator because the former is easier to use for abstract manipulation. However, the latter is included in the definition, as well as the `_=<<_` operator, which corresponds exactly to the `bind` function. Additionally, this record includes the `fmap` and `join` functions, which correspond to the fact that a monad can be obtained from a Kleisli triple.

As examples, we consider the `Identity`, `Maybe`, and `List` monads as Kleisli triples.

Example 6.3.4 (See module Abel.Data.Identity.Monad)**.** In **Agda**, the `Identity` or trivial Kleisli triple, which is an instance of the identity Kleisli triple (see Example 6.1.2), and which corresponds to the identity Kleisli triple in **Hask** (see Example 6.2.4), is defined as follows:

```
monad : Monad'' Identity
monad = mkMonad'' return bind associativity unity-left unity-right
  where
    return : {A : Set} → A → Identity A
```

```
  return = identity

  bind : {A B : Set} → (A → Identity B) → Identity A → Identity B
  bind f (identity x) = f x

  associativity : {A B C : Set} {f : A → Identity B}
                  {g : B → Identity C} (x : Identity A) →
                  bind g (bind f x) ≡ bind (bind g ∘ f) x
  associativity (identity _) = refl

  unity-left : {A B : Set} {f : A → Identity B} (x : A) →
               bind f (return x) ≡ f x
  unity-left _ = refl

  unity-right : {A : Set} (x : Identity A) → bind return x ≡ x
  unity-right (identity _) = refl
```

Example 6.3.5 (See module Abel.Data.Maybe.Monad)**.** In **Agda**, the `Maybe` Kleisli triple, which corresponds to the `Maybe` Kleisli triple in **Hask** (see Example 6.2.5), is defined as follows:

```
monad : Monad'' Maybe
monad = mkMonad'' return bind associativity unity-left unity-right
  where
    return : {A : Set} → A → Maybe A
    return = just

    bind : {A B : Set} → (A → Maybe B) → Maybe A → Maybe B
    bind f (just x) = f x
    bind _ nothing  = nothing

    associativity : {A B C : Set} {f : A → Maybe B} {g : B → Maybe C}
                    (mx : Maybe A) →
                    bind g (bind f mx) ≡ bind (bind g ∘ f) mx
    associativity (just _) = refl
    associativity nothing  = refl

    unity-left : {A B : Set} {f : A → Maybe B} (x : A) →
                 bind f (return x) ≡ f x
    unity-left _ = refl
```

```
    unity-right : {A : Set} (mx : Maybe A) → bind return mx ≡ mx
    unity-right (just _) = refl
    unity-right nothing  = refl
```

Example 6.3.6 (See module Abel.Data.List.Monad)**.** In **Agda**, the `List` Kleisli triple, which corresponds to the list Kleisli triple in **Hask** (see Example 6.2.6), is defined as follows:

```
monad : Monad'' List
monad = mkMonad'' return bind associativity unity-left unity-right
  where
    return : {A : Set} → A → List A
    return x = x ∷ []

    bind : {A B : Set} → (A → List B) → List A → List B
    bind f xs = concat (map f xs)

    associativity : {A B C : Set} {f : A → List B} {g : B → List C}
                    (xs : List A) →
                    bind g (bind f xs) ≡ bind (bind g ∘ f) xs
    associativity             []       = refl
    associativity {f = f} {g} (x ∷ xs) =
      begin
        concat (map g (f x ++ concat (map f xs)))
          ≡⟨ cong concat (map-++-commute g (f x)
               (concat (map f xs))) ⟩
        concat (map g (f x) ++ map g (concat (map f xs)))
          ≡⟨ concat-++-commute (map g (f x))
               (map g (concat (map f xs))) ⟩
        concat (map g (f x)) ++ concat (map g (concat (map f xs)))
          ≡⟨ cong (_++_ (concat (map g (f x)))) (associativity xs) ⟩
        concat (map g (f x)) ++ concat (map (bind g ∘ f) xs)
      ∎
        where open Relation.Binary.PropositionalEquality.≡-Reasoning

    unity-left : {A B : Set} {f : A → List B} (x : A) →
                 bind f (return x) ≡ f x
    unity-left {f = f} x = ++-[] (f x)
```

```
unity-right : {A : Set} (xs : List A) → bind return xs ≡ xs
unity-right []       = refl
unity-right (x ∷ xs) = cong (_∷_ x) (unity-right xs)
```

6.3.3 Equivalence of Monads and Kleisli Triples in Agda

We shall not prove that monads and Kleisli triples are coextensive in **Agda**. However, the definitions of the `Monad'` and `Monad''` records include the required constructions discussed in Section 6.1.

6.4 References

The definitions of monad and Kleisli triple are based on (Mac Lane 1998, p. 137) and (Moggi 1991, p. 58), respectively, and the theorems of the correspondence between monads and Kleisli triples are based on (Manes 1976, pp. 24, 26–29; Moggi 1991, p. 61).

Terminology. Although the common term for monads is monad, alternatives include standard construction, which is the original term (Manes 1976, p. 30), algebraic theory in monoid form (Manes 1976, p. 29), and triple (Barr and Wells 2005, p. 83; Barr and Wells 2012, p. 372). The common term for Kleisli triples is Kleisli triple, but another term is algebraic theory in extension form (Manes 1976, p. 32), which is perhaps more precise but rather outdated. We choose the common terms for the sake of simplicity and for effectively distinguishing between monads and Kleisli triples.

Chapter 7

Algebras and Initial Algebras

> "Curiouser and curiouser!"
>
> —Carroll (2004, p. 23)

In this chapter we explore algebras and initial algebras over endofunctors, and their relation to algebraic data types in Haskell. As motivation, `foldr` is a standard function that encapsulates common patterns of recursion concerning lists (Hutton 1999, pp. 355–356). The `foldr` function can be defined as follows[1]:

```
foldr :: b -> (a -> b -> b) -> [a] -> b
foldr n c []     = n
foldr n c (x:xs) = c x (foldr n c xs)
```

That is, given a value `n` of type `b` and a function `c` of type `a -> b -> b`, the function `foldr n c` of type `[a] -> b` replaces `[]` with `n` and `(:)` with `c`. This amounts to saying that `[a]` and its constructors, `[]` and `(:)`, yield an algebra over an endofunctor, and, more important, that it constitutes the initial algebra over the endofunctor. As it turns out, this fact uniquely determines both the type signature and the definition of the `foldr` function.

This idea generalizes to algebraic data types in the sense that one such type is the initial algebra over an endofunctor and that this fact uniquely determines a function that encapsulates common patterns of recursion concerning that type.

[1] Note that this is not the type signature of the standard Haskell `foldr` function.

7.1 Algebras and Initial Algebras

We begin by describing algebras (over endofunctors) and algebra homomorphisms.

Definition 7.1. Let $\mathsf{F} : \mathcal{C} \to \mathcal{C}$ be an endofunctor in a category $\mathcal{C}$. An F-algebra (a, α) is an object a, called the carrier of the algebra, and a morphism $\alpha : \mathsf{F}_{\mathrm{O}}(a) \to a$.

As examples, we consider the initial object of a category, and natural numbers and lists in **Set**, which we shall describe again as examples of algebras in Haskell.

Example 7.1.1. Let $\mathcal{C}$ be a category with an initial object 0. Then $(0, \mathrm{id}_0)$ is an algebra over the identity functor (see Example 4.1.2), that is, an I-algebra. In particular, in **Set**, $(\emptyset, \mathrm{id}_\emptyset)$ is an I-algebra.

Example 7.1.2. In **Set**, the natural numbers $\mathbb{N} = \{0, 1, 2, ...\}$, along with the functions zero $: 1 \to \mathbb{N}$ and succ $: \mathbb{N} \to \mathbb{N}$, which can be joined to a function $[\mathrm{zero}, \mathrm{succ}] : 1 + \mathbb{N} \to \mathbb{N}$, as illustrated by the diagram in Figure 7.1, yield an algebra $(\mathbb{N}, [\mathrm{zero}, \mathrm{succ}])$ over an endofunctor $\mathsf{N} : \mathbf{Set} \to \mathbf{Set}$ whose object mapping assigns to each set A a set

$$\mathsf{N}_{\mathrm{O}}(A) = 1 + A,$$

and whose morphism mapping assigns to each function $f : A \to B$ a function $\mathsf{N}_{\mathrm{M}}(f) : 1 + A \to 1 + B$ such that

$$\mathsf{N}_{\mathrm{M}}(f) \circ \iota_1 = \iota_1 \quad \text{and} \quad \mathsf{N}_{\mathrm{M}}(f) \circ \iota_2 = \iota_2 \circ f,$$

that is,

$$\mathsf{N}_{\mathrm{M}}(f)(1, ()) = (1, ()) \quad \text{and} \quad \mathsf{N}_{\mathrm{M}}(f)(2, x) = (2, f(x))$$

for all $x \in A$ (see Examples 3.2.1 and 3.3.4 for initial objects and coproducts in **Set**, respectively).

Example 7.1.3. In **Set**, lists can be represented as algebras over endofunctors. For a given set A,

$$\mathrm{nil} : 1 \to \mathrm{List}(A) \quad \text{and} \quad \mathrm{cons} : A \times \mathrm{List}(A) \to \mathrm{List}(A)$$

can be joined to a function $[\mathrm{nil}, \mathrm{cons}] : 1 + A \times \mathrm{List}(A) \to \mathrm{List}(A)$, as illustrated by the diagram in Figure 7.2. In this way, $(\mathrm{List}(A), [\mathrm{nil}, \mathrm{cons}])$

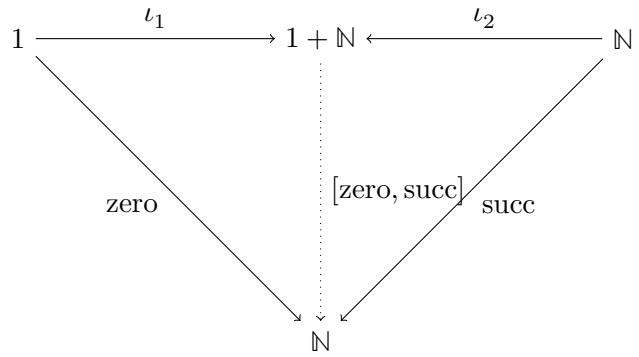


Figure 7.1: Natural numbers in **Set**.

is an algebra over an endofunctor $\mathsf{L}^A : \mathbf{Set} \to \mathbf{Set}$ whose object mapping assigns to each set B a set

$$\mathsf{L}^A_{\mathrm{O}}(B) = 1 + A \times B,$$

and whose morphism mapping assigns to each function $g : B \to C$ a function $\mathsf{L}^A_{\mathrm{M}}(g) : 1 + A \times B \to 1 + A \times C$ such that

$$\mathsf{L}^A_{\mathrm{M}}(g)(1, ()) = (1, ()) \quad \text{and} \quad \mathsf{L}^A_{\mathrm{M}}(g)(2, (x, y)) = (2, (x, g(y)))$$

for all $(x, y) \in A \times B$ (see Examples 3.2.1, 3.3.1, and 3.3.4 for initial objects, products, and coproducts in **Set**, respectively).

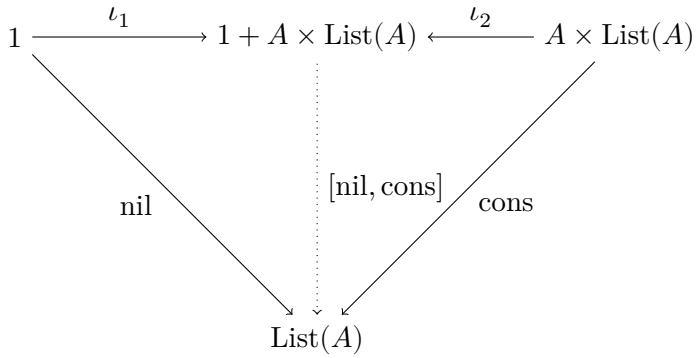


Figure 7.2: Lists in **Set**.

Definition 7.2. Let $\mathsf{F} : \mathcal{C} \to \mathcal{C}$ be an endofunctor in a category $\mathcal{C}$. If (a, α) and (b, β) are F-algebras, an F-algebra homomorphism $f : (a, \alpha) \to (b, \beta)$ is a morphism $f : a \to b$ in $\mathcal{C}$ such that

$$\beta \circ \mathsf{F}_{\mathrm{M}}(f) = f \circ \alpha, \tag{7.1}$$

that is, the diagram in Figure 7.3 is commutative. In this case, we say that $\mathrm{dom}(f) = (a, \alpha)$ and $\mathrm{cod}(f) = (b, \beta)$.

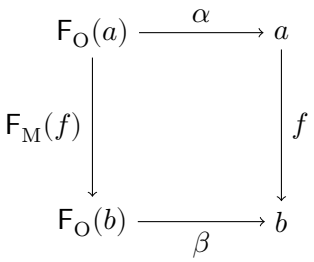


Figure 7.3: An F-algebra homomorphism.

Let us now define identity and composite algebra homomorphisms, which will allow us to construct categories of algebras and algebra homomorphisms.

Definition 7.3. Let $\mathsf{F} : \mathcal{C} \to \mathcal{C}$ be an endofunctor in a category $\mathcal{C}$. If (a, α) is an F-algebra, then its identity F-algebra homomorphism

$$\mathrm{id}_{(a,\alpha)} : (a, \alpha) \to (a, \alpha)$$

is the identity morphism $\mathrm{id}_a : a \to a$ in $\mathcal{C}$. To see that this is an F-algebra homomorphism, we prove (7.1) with $f = \mathrm{id}_{(a,\alpha)}$:

$$\begin{aligned}
&\alpha \circ \mathsf{F}_{\mathrm{M}}(\mathrm{id}_a) \\
&\quad = \quad \text{(by (4.1))} \\
&\alpha \circ \mathrm{id}_{\mathsf{F}_{\mathrm{O}}(a)} \\
&\quad = \quad \text{(by (2.2) with } f = \alpha\text{)} \\
&\mathrm{id}_a \circ \alpha
\end{aligned}$$

Definition 7.4. Let $\mathsf{F} : \mathcal{C} \to \mathcal{C}$ be an endofunctor in a category $\mathcal{C}$, and (a, α), (b, β), and (c, γ) three F-algebras. Given two F-algebra homomorphisms, their composite F-algebra homomorphism $g \circ f : (a, \alpha) \to (c, \gamma)$ is the composite morphism $g \circ f : a \to c$ in $\mathcal{C}$. To see that this is an F-algebra homomorphism, we prove (7.1) with $f = g \circ f$:

$$
\begin{aligned}
&\gamma \circ \mathsf{F}_{\mathrm{M}}(g \circ f) \\
&\quad = \quad \text{(by (4.2))} \\
&\gamma \circ \mathsf{F}_{\mathrm{M}}(g) \circ \mathsf{F}_{\mathrm{M}}(f) \\
&\quad = \quad \text{(by (7.1) with } f = g\text{)} \\
&g \circ \beta \circ \mathsf{F}_{\mathrm{M}}(f) \\
&\quad = \quad \text{(by (7.1))} \\
&g \circ f \circ \alpha
\end{aligned}
$$

Definition 7.5. Let $\mathsf{F} : \mathcal{C} \to \mathcal{C}$ be an endofunctor in a category $\mathcal{C}$. Then F-**Alg** is the category of F-algebras and F-algebra homomorphisms. Its objects are F-algebras, its morphisms are F-algebra homomorphisms, its identity morphisms are identity F-algebra homomorphisms, and its composite morphisms are composite F-algebra homomorphisms. Since (2.1) and (2.2) hold for $\mathcal{C}$, they hold for F-**Alg** too.

Having constructed categories of algebras and algebra homomorphisms, we move on to their initial objects (see Definition 3.2), that is, initial algebras.

Definition 7.6. Let $\mathsf{F} : \mathcal{C} \to \mathcal{C}$ be an endofunctor in a category $\mathcal{C}$. An F-algebra $(\mu\mathsf{F}, \mathrm{in})$ is the initial F-algebra of the category F-**Alg** if, for all F-algebras (a, α), there is a unique F-algebra homomorphism

$$(\!|\alpha|\!) : (\mu\mathsf{F}, \mathrm{in}) \to (a, \alpha),$$

that is, a morphism $(\!|\alpha|\!) : \mu\mathsf{F} \to a$ in $\mathcal{C}$ such that

$$\alpha \circ \mathsf{F}_{\mathrm{M}}((\!|\alpha|\!)) = (\!|\alpha|\!) \circ \mathrm{in}\,, \tag{7.2}$$

or, equivalently, the diagram in Figure 7.4 is commutative. Such an F-algebra homomorphism (that is, a unique F-algebra homomorphism from the initial F-algebra) is called a catamorphism.

Intuitively, the initial algebra denotes the collection of constructor functions for inductive data types. This statement is justified by a theorem, which we shall describe and prove using the following lemma.

Lemma 7.1. *Let $\mathsf{F} : \mathcal{C} \to \mathcal{C}$ be an endofunctor in a category $\mathcal{C}$. If $(\mu\mathsf{F}, \mathrm{in})$ is the initial F-algebra of the category F-**Alg**, then*

$$\mathrm{id}_{\mu\mathsf{F}} = (\!|\mathrm{in}|\!) \tag{7.3}$$

and, for all F-algebra homomorphisms $f : (a, \alpha) \to (b, \beta)$,

$$f \circ (\!|\alpha|\!) = (\!|\beta|\!). \tag{7.4}$$

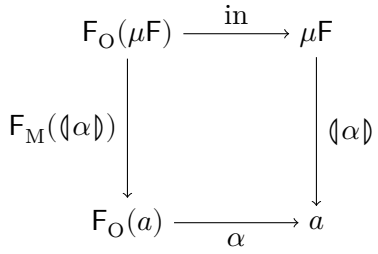


Figure 7.4: A catamorphism.

Proof. Since $(\mu\mathsf{F}, \mathrm{in})$ is initial, then $\langle\!|\mathrm{in}|\!\rangle$ and $\langle\!|\beta|\!\rangle$ are unique, which proves both equations. (For (7.4), see the diagram in Figure 7.5.)

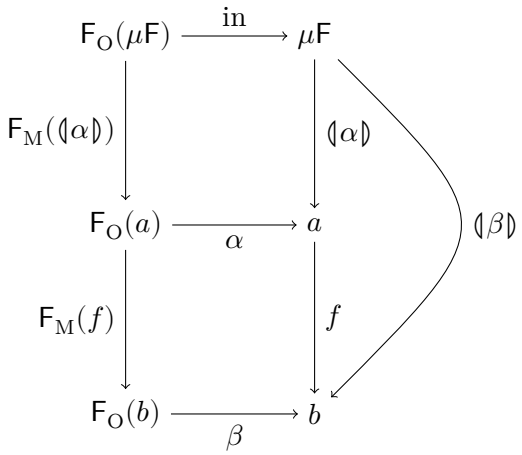


Figure 7.5: The fusion law for a catamorphism.

□

The following theorem is "the formal justification on the identification of inductive types with initial algebras" (Vene 2000, p. 17).

Theorem 7.2 (Lambek)**.** *Let* $\mathsf{F} : \mathcal{C} \to \mathcal{C}$ *be an endofunctor in a category* $\mathcal{C}$*. If* $(\mu\mathsf{F}, \mathrm{in})$ *is the initial* F*-algebra of the category* F**-Alg**, *then* in *is an isomorphism with its inverse* $\mathrm{in}^{-1} : \mu\mathsf{F} \to \mathsf{F}_{\mathrm{O}}(\mu\mathsf{F})$ *defined by*

$$\mathrm{in}^{-1} = \langle\!|\mathsf{F}_{\mathrm{M}}(\mathrm{in})|\!\rangle. \tag{7.5}$$

Proof. We prove (3.1) with $f = \mathrm{in}$. In the first place:

$$
\begin{aligned}
&\mathrm{in} \circ \mathrm{in}^{-1} \\
&\quad = \quad \text{(by (7.5))} \\
&\mathrm{in} \circ (\!|\mathsf{F}_{\mathrm{M}}(\mathrm{in})|\!) \\
&\quad = \quad \text{(by (7.4) with } f = \mathrm{in}) \\
&(\!|\mathrm{in}|\!) \\
&\quad = \quad \text{(by (7.3))} \\
&\mathrm{id}_{\mu\mathsf{F}}
\end{aligned}
$$

In the second place:

$$
\begin{aligned}
&\mathrm{in}^{-1} \circ \mathrm{in} \\
&\quad = \quad \text{(by (7.5))} \\
&(\!|\mathsf{F}_{\mathrm{M}}(\mathrm{in})|\!) \circ \mathrm{in} \\
&\quad = \quad \text{(by (7.2) with } \alpha = \mathsf{F}_{\mathrm{M}}(\mathrm{in})) \\
&\mathsf{F}_{\mathrm{M}}(\mathrm{in}) \circ \mathsf{F}_{\mathrm{M}}((\!|\mathsf{F}_{\mathrm{M}}(\mathrm{in})|\!)) \\
&\quad = \quad \text{(by (4.2) with } f = (\!|\mathsf{F}_{\mathrm{M}}(\mathrm{in})|\!) \text{ and } g = \mathrm{in}) \\
&\mathsf{F}_{\mathrm{M}}(\mathrm{in} \circ (\!|\mathsf{F}_{\mathrm{M}}(\mathrm{in})|\!)) \\
&\quad = \quad \text{(see above)} \\
&\mathsf{F}_{\mathrm{M}}(\mathrm{id}_{\mu\mathsf{F}}) \\
&\quad = \quad \text{(by (4.1) with } a = \mu\mathsf{F}) \\
&\mathrm{id}_{\mathsf{F}_{\mathrm{O}}(\mu\mathsf{F})}
\end{aligned}
$$

□

This theorem shows that the carrier of the initial algebra, $\mu\mathsf{F}$, is isomorphic to $\mathsf{F}_{\mathrm{O}}(\mu\mathsf{F})$, and, for this reason, the initial algebra is said to be "(up to isomorphism) a fixed point of the functor" (Vene 2000, p. 18), but we shall not focus on this terminology.

We have already discussed three examples of algebras (the initial object of a category, natural numbers, and lists). We describe them again as initial algebras.

Example 7.1.4. Let $\mathcal{C}$ be a category with an initial object 0. Then $(0, \mathrm{id}_0)$, the I-algebra described in Example 7.1.1, is the initial I-algebra of the category I-**Alg**. Indeed, given an I-algebra (a, α), there is a unique I-algebra

homomorphism $(\!|\alpha|\!) : (0, \mathrm{id}_0) \to (a, \alpha)$ in which the underlying morphism $(\!|\alpha|\!) : 0 \to a$ in $\mathcal{C}$ is the unique morphism given by the fact that 0 is an initial object. The uniqueness of this morphism guarantees that (7.1) holds. In particular, in **Set**, $(\emptyset, \mathrm{id}_\emptyset)$ is the initial I-algebra of the category I-**Alg**.

Example 7.1.5. In **Set**, the N-algebra $(\mathbb{N}, [\mathrm{zero}, \mathrm{succ}])$ described in Example 7.1.2 is the initial N-algebra of the category N-**Alg**. For an N-algebra $(A, [z, s])$, that is, a set A and functions $z : 1 \to A$ and $s : A \to A$, we need a unique N-algebra homomorphism $(\!|[z, s]|\!)$ or

$$\mathrm{fold}(z, s) : (\mathbb{N}, [\mathrm{zero}, \mathrm{succ}]) \to (A, [z, s]),$$

that is, a function $\mathrm{fold}(z, s) : \mathbb{N} \to A$ such that

$$\mathrm{fold}(z, s) \circ [\mathrm{zero}, \mathrm{succ}] = [z, s] \circ \mathsf{N}_{\mathrm{M}}(\mathrm{fold}(z, s)).$$

Without going into detail, this equation uniquely defines $\mathrm{fold}(z, s)$ as

$$\mathrm{fold}(z, s)(\mathrm{zero}()) = z()$$

and, for all $n \in \mathbb{N}$,

$$\mathrm{fold}(z, s)(\mathrm{succ}(n)) = s(\mathrm{fold}(z, s)(n)),$$

or, more succinctly,

$$\mathrm{fold}(z, s)(n) = s^n(z()),$$

which yields the required unique N-algebra homomorphism.

For instance, addition and multiplication of natural numbers can be defined as folds add and $\mathrm{mult} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ such that, for all $(m, n) \in \mathbb{N} \times \mathbb{N}$,

$$\mathrm{add}(m, n) = \mathrm{fold}(\lambda x.n, \mathrm{succ})(m)$$

and

$$\mathrm{mult}(m, n) = \mathrm{fold}(\mathrm{zero}, \lambda x.\, \mathrm{add}(m, x))(m),$$

respectively.

Example 7.1.6. In **Set**, for a set A, the L^A-algebra $(\mathrm{List}(A), [\mathrm{nil}, \mathrm{cons}])$ described in Example 7.1.3 is the initial algebra of the category L^A-**Alg**. Let $(B, [n, c])$ be an algebra over L^A, that is, a set A and functions $n : 1 \to B$ and $c : A \times B \to B$. Then we need a unique L^A-algebra homomorphism

$$\mathrm{fold}(n, c) : (\mathrm{List}(A), [\mathrm{nil}, \mathrm{cons}]) \to (B, [n, c]),$$

that is, a function $\mathrm{foldr}(n, c) : \mathrm{List}(A) \to B$ such that

$$\mathrm{foldr}(n, c) \circ [\mathrm{nil}, \mathrm{cons}] = [n, c] \circ \mathsf{L}^A_{\mathrm{M}}(\mathrm{foldr}(n, c)).$$

This equation uniquely defines $\mathrm{fold}(n, c)$ as

$$\mathrm{foldr}(n, c)(\mathrm{nil}()) = n()$$

and, for all $(x, xs) \in A \times \mathrm{List}(A)$,

$$\mathrm{foldr}(n, c)(\mathrm{cons}(x, xs)) = c(x, \mathrm{foldr}(n, c)(xs)),$$

which yields the required unique L^A-algebra homomorphism.

As an example, the length of a list of elements of a set A can be calculated as a fold $\mathrm{length} : \mathrm{List}(A) \to \mathbb{N}$ such that

$$\mathrm{length} = \mathrm{foldr}(\mathrm{zero}, \lambda(x, n).\, \mathrm{succ}(n)).$$

As another example, two lists of elements of a set A can be appended by a fold $\mathrm{append} : \mathrm{List}(A) \times \mathrm{List}(A) \to \mathrm{List}(A)$ such that

$$\mathrm{append}(xs, ys) = \mathrm{foldr}(\lambda x.ys, \mathrm{cons})(xs)$$

for all $(xs, ys) \in \mathrm{List}(A) \times \mathrm{List}(A)$. Finally, $\mathrm{map}(f) : \mathrm{List}(A) \to \mathrm{List}(B)$ can be defined as a fold

$$\mathrm{map}(f) = \mathrm{foldr}(\mathrm{nil}, \lambda(x, ys).\, \mathrm{cons}(f(x), ys))$$

for all functions $f : A \to B$.

7.2 Algebras and Initial Algebras in Haskell

In the previous section, we identified inductive types with initial algebras. In Haskell, such types are known as algebraic data types, which include recursive types such as natural numbers and lists. When we define an algebraic data type, its declaration introduces a new type or type constructor, and zero or more data constructors. These data specify an initial algebra over an endofunctor which can be inferred from the information at hand. Given an algebraic data type, inferring such an endofunctor and proving that its category of algebras has an initial object amounts to defining a fold function for that particular type. In fact, such a function is uniquely determined by the fact that the algebraic data type is an initial algebra.

As examples, we describe natural numbers and lists as initial algebras over endofunctors in Haskell.

Example 7.2.1. In Haskell, natural numbers can be defined as an algebraic data type, as follows:

```
data Nat = Zero | Succ Nat
```

This declaration introduces a type `Nat` of kind `*`, and constructors `Zero` and `Succ` with types

`Zero :: Nat` and `Succ :: Nat -> Nat`.

These data define an algebra over an endofunctor `N`:

```
data N a = Z | S a

instance Functor N where
  fmap _ Z     = Z
  fmap f (S x) = S (f x)
```

In detail, this algebra is given by `Nat`, `Zero`, and `Succ`. Moreover, the algebra in question is the initial algebra of the category of algebras over `N`:

```
fold :: a -> (a -> a) -> Nat -> a
fold z s Zero     = z
fold z s (Succ n) = s (fold z s n)
```

In words, given an algebra over `N`, that is, a type `a`, a value `z` of type `a`, and a function `s` of type `a -> a`, there is a unique function of type `Nat -> a`, namely `fold z s`.

For instance, given values `m` and `n` of type `Nat`, we define the addition of `m` and `n` using `fold n Succ`, that is, `fold` for the `N`-algebra specified by `Nat`, `n`, and `Succ`, as follows:

```
add :: Nat -> Nat -> Nat
add m n = fold n Succ m
```

This definition might be easier to understand if we compare it to the one yielded by using explicit recursion:

```
add Zero     n = n
add (Succ m) n = Succ (add m n)
```

As another example, given values `m` and `n` of type `Nat`, we define the multiplication of `m` and `n` using `fold Zero (add n)`, that is, `fold` for the `N`-algebra specified by `Nat`, `Zero`, and `add n`, as follows:

```
mult :: Nat -> Nat -> Nat
mult m n = fold Zero (add n) m
```

In this case, explicit recursion yields the following definition:

```
mult Zero     n = Zero
mult (Succ m) n = add n (mult m n)
```

Alternatively, we can consider `Nat` and

```
either (\() -> Zero) Succ :: Either () Nat -> Nat
```

as an algebra over `Either ()` (see Example 4.2.5). So, given a type `b` and

```
either (\() -> z) s :: Either () a -> a
```

we need a unique function `fold` such that

```
fold z s . either (\() -> Zero) Succ
```

and

```
either (\() -> z) s . fmap (fold z s)
```

are the same, but that is the above definition of `fold`.

Example 7.2.2. In Haskell, lists can be defined as an algebraic data type, as follows:

```
data List a = Nil | Cons a (List a)
```

This declaration introduces a type constructor `List` of kind `* -> *`, and constructors `Nil` and `Cons` with types

```
Nil :: List a   and   Cons :: a -> List a -> List a
```

for all types `a` of kind `*`. These data define an `L`-algebra:

```
data L a b = N | C a b

instance Functor (L a) where
  fmap _ N       = N
  fmap f (C x y) = C x (f y)
```

More precisely, given a concrete type `a`, `List a`, `Nil`, and `Cons` specify the algebra being discussed. This algebra is the initial algebra of the category of algebras over `L`:

```
foldr :: b -> (a -> b -> b) -> List a -> b
foldr n c Nil         = n
foldr n c (Cons x xs) = c x (foldr n c xs)
```

That is to say, for a concrete type `a`, given an algebra over `L`, that is, a concrete type `b`, a value `n` of type `b`, and a function `c` of type `a -> b -> b`, there is a unique function of type `List a -> b`, namely `foldr n c`.

As an example, the length of a list of values of a type `a` can be calculated using `foldr`, as follows:

```
length :: List a -> Nat
length = foldr Zero (\_ -> Succ)
```

As another example, two lists `xs` and `ys` of values of a type `a` can be appended using `fold` for the `L`-algebra specified by `List a`, `ys`, and `Cons`:

```
append :: List a -> List a -> List a
append xs ys = (foldr ys Cons) xs
```

Finally, given concrete types `a` and `b`, `map f` can be defined as a `foldr` for all functions `f` of type `a -> b`, as follows:

```
map :: (a -> b) -> List a -> List b
map f = foldr Nil (Cons . f)
```

Each of these definitions might be easier to understand if we compare them to the ones yielded by using explicit recursion:

```
length Nil         = Zero
length (Cons _ xs) = Succ (length xs)

append Nil         ys = ys
append (Cons x xs) ys = Cons x (append xs ys)

map _ Nil         = Nil
map f (Cons x xs) = Cons (f x) (map f xs)
```

7.3 References

This chapter is based on (Awodey 2010, § 10.5; Bird and de Moor 1997, § 2.6; Vene 2000, § 2.1). The definition of algebras over endofunctors, which is a simpler description of algebras over monads, is also based on (Mac Lane 1998, p. 140; Poigné 1992, pp. 595–596).

Chapter 8

Conclusions

> "What dreadful nonsense we *are* talking!"
>
> —Carroll (2004, p. 255)

> "You may call it 'nonsense' if you like, but *I've* heard nonsense, compared with which that would be as sensible as a dictionary!"
>
> —Carroll (2004, p. 173)

Our main objective with this project was to study some of the applications of category theory to functional programming, particularly in Haskell and Agda, and, more specifically, to describe and explain the concepts of category theory needed for conceptualizing and better understanding functors, polymorphism, monads, and algebraic data types, which we did in Chapters 4, 5, 6, and 7, respectively. In Chapter 2, we identified categories as the starting point for relating category theory to functional programming. In the case of algebraic data types and, more usefully, folds, we identified algebras and initial algebras over endofunctors as the required concepts for satisfying our main goal; in the case of functors, the notions of functor and endofunctor; in the case of monads, the concepts of monad and Kleisli triple; and, in the case of polymorphism or, more precisely, parametric polymorphism, natural transformations.

Obviously, we did not cover all of category theory. For instance, we did not deal with concepts such as adjoints, epimorphisms, limits, monomorphisms, and universal constructions, which were listed in the project proposal, but did not answer our purpose. Having said that, we did cover the trinity of concepts category, functor, and natural transformation, which is the foundation of all category theory (Mac Lane 1998, p. vii), and which

creates an opportunity for a deeper understanding of the subject.

Additionally, some of the applications of category theory to functional programming are not as straightforward as suggested here. For example, polymorphic functions actually correspond to lax natural transformations (Wadler 1989, p. 350), and algebraic data types in Haskell correspond to initial algebras and terminal coalgebras over endofunctors (Vene 2000, § 2), but such concepts go beyond the scope of this project. However, our use of category theory seems to be appropriate and useful, especially from the standpoint of functional programming.

Although subjective, we believe that this project provides some interesting examples of how to take advantage of category theory in functional programming and programming in general, as well as a way to become a better programmer.

Needless to say, the ideas of category theory might be difficult to understand at first. As a matter of fact, Bird and de Moor (1997, p. 25) claim that "one does not so much learn category theory as absorb it over a period of time." We claim that it is definitely worth it.

8.1 Future Work

All unanswered questions and concepts beyond the scope of this project could be considered as suggestions for future work. For instance, the questions of Haskell's and Agda's categories, the existence of initial algebras over endofunctors, and others. We describe some ideas which we find interesting and appropriate.

8.1.1 Adjoints

Category theory is based on the concepts of categories, functors, and natural transformations. Even though these ideas are important, a fundamental notion of category theory is adjoints (Marquis 2013, p. 11), which "arise everywhere" (Mac Lane 1998, p. vii). Taking into account our approach, can we study the applications of adjoints with the purpose of better understanding functional programming? Based on (Barr and Wells 2012, § 13; Elkins 2009, pp. 79–81; Pierce 1991, § 2.4; Rydeheard 1986a; Rydeheard and Burstall 1988, § 6), the answer seems to be yes. In addition, Awodey (2010, § 9) and Mac Lane (1998, § IV) seem to offer a good starting point.

8.1.2 Applicative functors

Based on (McBride and Paterson 2008), we identified and studied monoidal categories and functors in order to be able to understand applicative functors from a category-theoretical point of view. We could use our results in a future project, which seems to be a very relevant next step, particularly in the context of the "current, and very likely to succeed," Haskell 2014 `Applicative => Monad` proposal[1], which adds an `Applicative` constraint to the `Monad` type class and promotes `join` to `Monad`, which we briefly discussed in Remark 6.5.

8.1.3 Categories

In Section 2.2, we described **Hask**, the category of Haskell types and functions, in order to be able to relate category theory to functional programming. We could use the `Category` type class instead (Yorgey 2009, pp. 49–51; Elkins 2009, pp. 74–75):

```
class Category cat where
  id  :: cat a a
  (.) :: cat b c -> cat a b -> cat a c

instance Category (->) where
  id  = Prelude.id
  (.) = (Prelude..)
```

Likewise, we could study Kleisli categories (Moggi 1991, pp. 59–60) in terms of this type class, which might be a more intuitive way to justify the Kleisli triple laws. In addition, the `Category` class would lead us to `Arrow`, which is a generalization of functions (Yorgey 2009, pp. 51–57).

8.1.4 Folds

In Chapter 7, we examined catamorphisms and their relation to the `foldr` function for lists. Can we apply the results of that chapter to `foldl` and folds in general? In particular, can we use algebras and initial algebras over endofunctors for conceptualizing the `Foldable` (Yorgey 2009, pp. 44–47) and `Traversable` (McBride and Paterson 2008, § 3; Yorgey 2009, pp. 47–49) type classes?

[1] http://www.haskell.org/haskellwiki/Functor-Applicative-Monad_Proposal.

8.1.5 Monoids

As an alternative to the ideas of Section 8.1.1, a "fundamental notion of category theory is that of a monoid" (Mac Lane 1998, p. vii), as described in Example 2.1.4. In Haskell, monoids are defined by the `Monoid` type class:

```
class Monoid a where
  mempty  :: a
  mappend :: a -> a -> a
```

In connection with Remark 6.1 and Section 8.1.2, studying monoids and their relation to monads seems like a pertinent complement to this project. As basis for this study, we have identified (Mac Lane 1998, § VII; Yorgey 2009, pp. 39–44).

Bibliography

Adámek, Jiří, Horst Herrlich, and George E. Strecker (2006). *Abstract and Concrete Categories: The Joy of Cats*. Reprints in Theory and Applications of Categories 17 (cit. on p. 5). Repr. of *Abstract and Concrete Categories. The Joy of Cats*. Wiley, 1990.

The Agda Team (2014). *The Agda Wiki*. URL: http://wiki.portal.chalmers.se/agda/ (cit. on p. 1).

Awodey, Steve (2010). *Category Theory*. 2nd ed. Vol. 52. Oxford Logic Guides. Oxford University Press (cit. on pp. 4, 13, 19, 107, 110).

Barr, Michael and Charles Wells (2005). *Toposes, Triples and Theories*. Reprints in Theory and Applications of Categories 12 (cit. on pp. 5, 93). Repr. of *Toposes, Triples and Theories*. Vol. 278. Grundlehren der mathematischen Wissenschaften. Springer, 1984.

— (2012). *Category Theory for Computing Science*. Reprints in Theory and Applications of Categories 22 (cit. on pp. 5, 93, 110). Repr. of *Category Theory for Computing Science*. 3rd ed. Centre de recherches mathématiques, 1999.

Baudelaire, Charles (1857). Abel et Caïn. In: *Le Fleurs du mal*. Poulet-Malassis et de Broise (cit. on p. 1).

Bird, Richard and Oege de Moor (1997). *Algebra of Programming*. Vol. 100. Prentice Hall International Series in Computer Science. Prentice Hall (cit. on pp. 4, 58, 107, 110).

Bove, Ana and Peter Dybjer (2009). Dependent Types at Work. In: Language Engineering and Rigorous Software Development (2008). Ed. by Ana Bove et al. LNCS 5520. Springer, pp. 57–99 (cit. on pp. 5, 17).

Cardelli, Luca and Peter Wegner (1985). On Understanding Types, Data Abstraction, and Polymorphism. In: *Computing Surveys* 17.4, pp. 471–522 (cit. on p. 53).

Carroll, Lewis (2004). *Alice's Adventures in Wonderland and Through the Looking-Glass*. Barnes & Noble Classics. Barnes & Noble (cit. on pp. ii, 9, 95, 109).

Danielsson, Nils Anders et al. (Jan. 29, 2013). *The Agda standard library*. Version 0.7. URL: http://wiki.portal.chalmers.se/agda/ (visited on 04/03/2014) (cit. on pp. 6, 44).

Eilenberg, Samuel and Saunders MacLane (Oct. 1942). Group Extensions and Homology. In: *Annals of Mathematics* 43.4, pp. 757–831 (cit. on p. 4).

— (Sept. 1945). General Theory of Natural Equivalences. In: *Transactions of the American Mathematical Society* 58.2, pp. 231–294 (cit. on pp. 4, 19).

Elkins, Derek (2009). Calculating Monads with Category Theory. In: *The Monad.Reader* 13, pp. 73–91 (cit. on pp. 1, 5, 19, 58, 110, 111).

God (1769). *Bible*. King James Version (cit. on p. 7).

Goguen, Joseph A. (1991). A Categorical Manifesto. In: *Mathematical Structures in Computer Science* 1.1, pp. 49–67 (cit. on p. 1).

Hutton, Graham (1999). A Tutorial on the Universality and Expressiveness of Fold. In: *Journal of Functional Programming* 9.4, pp. 355–372 (cit. on p. 95).

Jeuring, Johan, Patrik Jansson, and Cláudio Amaral (2012). Testing Type Class Laws. In: Haskell Symposium (2012). ACM, pp. 49–60 (cit. on p. 34).

Kmett, Edward A. (2012). *The `void` package*. Version 0.5.6. URL: http://hackage.haskell.org/package/void-0.5.6 (visited on 03/13/2014) (cit. on p. 22).

— (2014). *The `semigroupoids` package*. Version 4.0.1. URL: http://hackage.haskell.org/package/semigroupoids-4.0.1 (visited on 03/29/2014) (cit. on p. 84).

Leibniz, Gottfried Wilhelm (1714). *Monadologie* (cit. on p. 59).

Lipovača, Miran (2011). *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press. URL: http://learnyouahaskell.com (cit. on pp. 5, 29, 47, 81).

Mac Lane, Saunders (1998). *Categories for the Working Mathematician*. 2nd ed. Vol. 5. Graduate Texts in Mathematics. Springer (cit. on pp. 1, 3, 4, 13, 19, 27, 29, 47, 49, 58, 61, 93, 107, 109, 110, 112).

Manes, Ernest G. (1976). *Algebraic Theories*. Vol. 26. Graduate Texts in Mathematics. Springer (cit. on pp. 64, 93).

Marlow, Simon, ed. (2010). *Haskell 2010 Language Report*. URL: http://www.haskell.org/onlinereport/haskell2010/ (cit. on p. 29).

Marquis, Jean-Pierre (2013). "Category Theory". In: *Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Summer 2013 Edition. Meta-

physics Research Lab. URL: http://plato.stanford.edu/entries/category-theory (cit. on pp. 1, 4, 32, 47, 58, 110).

McBride, Conor and Ross Paterson (2008). Applicative Programming with Effects. In: *Journal of Functional Programming* 18.1, pp. 1–13 (cit. on p. 111).

Milner, Robin (1984). A Proposal for Standard ML. In: LISP and Functional Programming (1984). ACM, pp. 184–197 (cit. on p. 1).

Moggi, Eugenio (1989). *An Abstract View of Programming Languages*. Tech. rep. University of Edinburgh (cit. on p. 60).

— (1991). Notions of Computation and Monads. In: *Information and Computation* 93.1, pp. 55–92 (cit. on pp. 60, 63, 64, 93, 111).

Norell, Ulf (2007). Towards a Practical Programming Language Based on Dependent Type Theory. PhD thesis. Chalmers University of Technology and University of Gothenburg (cit. on p. 1).

— (2009). Dependently Typed Programming in Agda. In: Advanced Functional Programming (2008). Ed. by Pieter Koopman, Rinus Plasmeijer, and Doaitse Swierstra. LNCS 5832. Springer, pp. 230–266 (cit. on pp. 5, 17).

O'Sullivan, Bryan, John Goerzen, and Don Stewart (2008). *Real World Haskell*. O'Reilly Media. URL: http://book.realworldhaskell.org (cit. on pp. 5, 59).

Peyton Jones, Simon, ed. (2003). *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press (cit. on pp. 1, 15, 30, 81).

Pierce, Benjamin C. (1991). *Basic Category Theory for Computer Scientists*. Foundations of Computing. MIT Press (cit. on pp. 1, 4, 19, 27, 110).

Pitt, David (1986). Categories. In: Category Theory and Computer Programming (1985). Ed. by David Pitt et al. LNCS 240. Springer, pp. 6–15 (cit. on p. 19).

Pitt, David et al., eds. (1986). Category Theory and Computer Programming (1985). LNCS 240. Springer.

Poigné, Axel (1992). "Basic Category Theory". In: *Handbook of Logic in Computer Science*. Ed. by S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum. Vol. 1 (Background: Mathematical Structures). Clarendon Press, pp. 413–640 (cit. on pp. 1, 4, 10, 19, 27, 47, 49, 58, 107).

Rydeheard, David E. (1986a). Adjunctions. In: Category Theory and Computer Programming (1985). Ed. by David Pitt et al. LNCS 240. Springer, pp. 51–57 (cit. on p. 110).

— (1986b). Functors and Natural Transformations. In: Category Theory and Computer Programming (1985). Ed. by David Pitt et al. LNCS 240. Springer, pp. 43–50 (cit. on p. 58).

Rydeheard, David E. and R. M. Burstall (1988). *Computational Category Theory*. Prentice Hall International Series in Computer Science. Prentice Hall (cit. on pp. 58, 110).

Saramago, José (2006). *As Pequenas Memórias*. Caminho (cit. on p. vii).

— (2008). *A Viagem do Elefante*. Caminho (cit. on p. vii).

Sturrock, Donald (2010). *Storyteller: The Authorized Biography of Roald Dahl*. Simon & Schuster (cit. on p. 117).

Turner, D. A. (1985). Miranda: A Non-strict Functional Language with Polymorphic Types. In: Functional Programming Languages and Computer Architecture (1985). Ed. by Jean-Pierre Jouannaud. LNCS 201. Springer, pp. 1–16 (cit. on p. 1).

Vene, Varmo (2000). Categorical Programming with Inductive and Coinductive Types. PhD thesis. University of Tartu (cit. on pp. 100, 101, 107, 110).

Wadler, Philip (1989). Theorems for Free! In: Functional Programming Languages and Computer Architecture (1989). ACM Press, pp. 347–359 (cit. on pp. 53, 54, 58, 110).

Weisstein, Eric W., ed. (2014). *Wolfram MathWorld*. URL: http://mathworld.wolfram.com (visited on 04/03/2014) (cit. on p. 5).

Wolfram, Stephen (2002). *A New Kind of Science*. Wolfram Media (cit. on p. 1).

Yorgey, Brent (2009). The Typeclassopedia. In: *The Monad.Reader* 13, pp. 17–68. URL: http://www.haskell.org/haskellwiki/Typeclassopedia (cit. on pp. 1, 5, 19, 33, 47, 81, 111, 112).

“Ow, fuck!”

—Sturrock (2010, p. 561)