The Haskell Programmer's Guide to the IO Monad

— Don't Panic —

Stefan Klinger

University of Twente, the Netherlands · EWI, Database Group

CTIT Technical Report

Stefan Klinger.
The Haskell Programmer's Guide to the IO Monad — Don't Panic.

Order-address:
Centre for Telematics and Information Technology
University of Twente
P. O. Box 217
7500 AL Enschede
the Netherlands
`mailto:a.m.annink-tanke@utwente.nl`

PDF available at `http://stefan-klinger.de`.

Stefan Klinger. The Haskell Programmer's Guide to the IO Monad — Don't Panic. Technical report, December 2005, no. 05-54, 33 pp., Centre for Telematics and Information Technology (CTIT), ISSN 1381-3625.

Version of 2005-Dec-15 15:39:54 .

Preface

Now, that you have started with Haskell, have you written a program doing IO yet, like reading a file or writing on the terminal? Then you have used the *IO monad* — but do you understand how it works?

The standard explanation is, that the IO monad hides the non-functional *IO actions* —which do have side effects— from the functional world of Haskell. It prevents pollution of the functional programming style with side effects.

However, since most beginning Haskell programmers (i.e., everyone I know and including me) lack knowledge about category theory, they have no clue about what a monad really is. Nor how this "hiding" works, apart from having IO actions disappearing beyond the borders of our knowledge.

This report scratches the surface of category theory, an abstract branch of algebra, just deep enough to find the monad structure. On the way we discuss the relations to the purely functional programming language Haskell. Finally it should become clear how the IO monad keeps Haskell pure.

We do not explain how to use the IO monad, nor discuss all the functions available to the programmer. But we do talk about the theory behind it.

Intended audience Haskell programmers that stumbled across the IO monad, and now want to look under the hood. Haskell experience and the ability to read math formulae are mandatory.

Many thanks to Sander Evers, Maarten Fokkinga, and Maurice van Keulen for reviewing, a lot of discussions, helpful insights and suggestions.

Contents

Chapter 1

Introduction

1.1 Motivation

Imagine you want to write a somewhat more sophisticated "hello world" program in Haskell, which asks for the user's name and then addresses him personally.

You probably end up with something like

```
main :: IO ()
main = do putStr "What's your name?\n> "
          x <- getLine
          putStr . concat $ [ "Hello ", x, ".\n" ]
```

So what exactly does the "`do`" mean? Is "`<-`" variable assignment? And what does the type "`IO ()`" mean?

Having accepted the presence of "`IO ()`" in the type signatures of basic IO functions (like, e.g., `putStr`, `getChar`), a lot of novice Haskell programmers try to get rid of it as soon as possible:

> *"I'll just wrap the* `getLine` *in another function which only returns what I really need, maybe convert the users' input to an* `Int` *and return that."*

However, this is not possible since it would violate the *referential transparency* Haskell enforces for all functions. Referential transparency guarantees that a function, given the same parameters, always returns the same result.

How can IO be done at all, if a function is referential transparent and must not have side effects? Try to write down the signatures of such a function that reads a character from the keyboard, and a function that writes a character to the screen, both without using the IO monad.

The intention of this guide is to show how the Monad helps in doing IO in a purely functional programming language, by illuminating the mathematical background of this structure.

1.2 Related work

This guide tries to fit exactly between the available theoretical literature about category theory on the one side (e.g., [1], [2]) and literature about how to program with monads ([3], [4]) on the other.

However, the sheer amount of available literature on both sides of this report dooms any approach to offer a complete list of related work to failure.

Chapter 2

Notation

I tried to keep the notation as readable as possible, yet unambiguous. The meaning should be clear immediately, and I hope that predicate logic people excuse some lack of purity.

1. Quite often proofs are interspersed with explanatory text, e.g., talking about integers we might note

$$\begin{aligned} & a+b \\ = \quad & [\ \text{operator '+' is commutative} \\ & b+a \ . \end{aligned}$$

2. Function application is always noted in juxtaposition —i.e., the operand is just written behind the function— to avoid a Lisp-like amount of parenthesis, i.e.

$$\begin{aligned} & fx \\ \equiv \quad & [\ \text{by definition} \\ & f(x) \ . \end{aligned}$$

3. Quantifiers have higher precedence than *and* and *or* (symbols $\wedge, \vee$), but lower than *implication* and *equivalence* ($\Rightarrow, \Leftrightarrow$).

   A quantifier "binds" all the free variables in the following ';'-separated list of predicates that are neither given in the context of the formula (constants), nor bound by an earlier quantifier. I.e.,

$$\begin{aligned} & \forall\, m, n \in \mathbb{N};\ m < n \quad \exists\, r \in \mathbb{R} \quad m < r < n \\ \equiv \quad & [\ \text{next line is predicate logic} \\ & \forall m \forall n \big(m \in \mathbb{N} \wedge n \in \mathbb{N} \wedge m < n \ \Rightarrow\ \exists r (r \in \mathbb{R} \wedge m < r < n)\big) \ . \end{aligned}$$

4. Sometimes the exercise sign (✎) occurs, requesting the reader to verify something, or to play with the Haskell code given.

5. A λ-expression binding the free occurrences of y in expression e is written

$$(y \mapsto e) \ .$$

   As a Haskell programmer, you should be familiar with λ-expressions.

6. Unless stated otherwise, most programming code is Haskell-*pseudo*code. In contradiction to this, Haskell-code that should be tried out by the user is referred to with a ✎-sign.

Chapter 3

Categories

3.1 Categories in theory

Introductory example Let us start with a quite concrete example of a category: Sets (later called objects) together with all the total functions on them (called morphisms):

- It is common practice to write $f : A \to B$ to denote that a function f maps from set A to set B. This is called the *type* of the function.
- Also, we can compose two functions f and g, if the target set of the former equals the source set of the latter, i.e., if $f : A \to B$ and $g : B \to C$ for some sets A, B, and C. The composition is commonly denoted by $g \circ f$.
- For each set A, there is an identity function $\mathrm{id}_A : A \to A$.

These are the properties of a category. While most things in this guide can be applied to the category of sets, Definition 3.1.1 is more precise and general. In particular, category theory is not restricted to sets and total functions, i.e., one can not assume that an object has elements (like sets do), or that an element is mapped to another element.

3.1.1 Definition A **category** $\mathcal{C} = (\mathcal{O}_\mathcal{C}, \mathcal{M}_\mathcal{C}, \mathcal{T}_\mathcal{C}, \circ_\mathcal{C}, \mathrm{Id}_\mathcal{C})$ is a structure consisting of *morphisms* $\mathcal{M}_\mathcal{C}$, *objects* $\mathcal{O}_\mathcal{C}$, a *composition* $\circ_\mathcal{C}$ of morphisms, and *type information* $\mathcal{T}_\mathcal{C}$ of the morphisms, which obey to the following constraints.

Note, that we often omit the subscription with $\mathcal{C}$ if the category is clear from the context.

1. We assume the collection of **objects** $\mathcal{O}$ to be a set. Quite often the objects themselves also are sets, however, category theory makes *no* assumption about that, and provides no means to explore their structure.
2. The collection of **morphisms** $\mathcal{M}$ is also assumed to be a set in this guide.
3. The ternary relation $\mathcal{T} \subseteq \mathcal{M} \times \mathcal{O} \times \mathcal{O}$ is called the **type information** of $\mathcal{C}$. We want every morphism to have a type, so a category requires
$$\forall\, f \in \mathcal{M} \quad \exists\, A, B \in \mathcal{O} \quad (f, A, B) \in \mathcal{T} \ .$$
We write $f : A \xrightarrow[\mathcal{C}]{} B$ for $(f, A, B) \in \mathcal{T}_\mathcal{C}$. Also, we want the types to be unique, leading to the claim
$$f : A \to B \wedge f : A' \to B' \ \Rightarrow\ A = A' \wedge B = B' \ .$$
This gives a notion of having a morphism to "map from one object to another". The uniqueness entitles us to give names to the objects involved in a morphism. For $f : A \to B$ we call $\mathrm{src}\, f := A$ the **source**, and $\mathrm{tgt}\, f := B$ the **target** of f.
4. The *partial* function $\circ : \mathcal{M} \times \mathcal{M} \to \mathcal{M}$, written as a binary infix operator, is called the **composition** of $\mathcal{C}$. Alternative notations are $f ; g := gf := g \circ f$.

Morphisms can be composed whenever the target of the first equals the source of the second. Then the resulting morphism maps from the source of the first to the target of the second:

$$f : A \to B \wedge g : B \to C \;\Rightarrow\; g \circ f : A \to C$$

In the following, the notation $g \circ f$ implies these type constraints to be fulfilled.

Composition is associative, i.e. $f \circ (g \circ h) = (f \circ g) \circ h$.

5. Each object $A \in \mathcal{O}$ has associated a unique **identity morphism** id_A. This is denoted by defining the function

$$\begin{array}{lcccl} \mathrm{Id} & : & \mathcal{O} & \longrightarrow & \mathcal{M} \\ & & A & \longmapsto & \mathrm{id}_A \;, \end{array}$$

which automatically implies uniqueness.

The typing of the identity morphisms adheres to

$$\forall\, A \in \mathcal{O} \quad \mathrm{id}_A : A \to A \;.$$

To deserve their name, the identity morphisms are —depending on the types— left or right neutral with respect to composition, i.e.,

$$f \circ \mathrm{id}_{\mathrm{src}\, f} = f = \mathrm{id}_{\mathrm{tgt}\, f} \circ f \;.$$

Another example Every set A with a partial order $(\leq)$ on it yields a category, say $\mathcal{Q}$. To see this, call the elements of A objects, and the relation morphisms:

$$\mathcal{O}_{\mathcal{Q}} := A \;,$$
$$\mathcal{M}_{\mathcal{Q}} := \{(x, y) \mid x, y \in A \wedge x \leq y\} \;.$$

Now (✎ as an exercise) define the type information $\mathcal{T}_{\mathcal{Q}}$, the composition $\circ_{\mathcal{Q}}$ and the identities $\mathrm{Id}_{\mathcal{Q}}$, so that $\mathcal{Q}$ is a category indeed. Note, that this example does not assume the objects to have any members.

More examples and a broader introduction to category theory (however, without discussing the monad structure) can be found in [1].

3.2 Spot a category in Haskell

This guide looks at one particular category that can be recognised in the Haskell programming language. There might be others of more or less interest, but for the purpose of explaining Haskell's Monad structure this narrow perspective is sufficient. For a more thorough discussion about functional programming languages and category theory you might read [2].

With the last section in mind, where would you look for "the obvious" category? It is not required that it models the whole Haskell language, instead it is enough to point at the things in Haskell that behave like the objects and morphisms of a category.

We call our Haskell category $\mathcal{H}$ and use Haskell's types —primitive as well as constructed— as the objects $\mathcal{O}_{\mathcal{H}}$ of the category. Then, unary Haskell functions correspond to the morphisms $\mathcal{M}_{\mathcal{H}}$, with function signatures of unary functions corresponding to the type information $\mathcal{T}_{\mathcal{H}}$.

`f :: A -> B` corresponds to $f : A \xrightarrow[\mathcal{H}]{} B$

Haskell's function composition '`.`' corresponds to the composition of morphisms $\circ_{\mathcal{H}}$. The identity in Haskell is typed

```
id :: forall a. a -> a
```

corresponding to

$$\forall\, A \in \mathcal{O}_{\mathcal{H}} \quad \mathrm{id}_A : A \to A \;.$$

Note that we do not talk about n-ary functions for $n \neq 1$. You can consider a function like $(+)$ to map a number to a function that adds this number, a technique called **currying** which is widely used by Haskell programmers. Within the context of this guide, we do not treat the resulting function as a morphism, but as an object.

Chapter 4

Functors

4.1 Functors in theory

One can define mappings between categories. If they "behave well", i.e., preserve the structural properties of being an object, morphism, identity, the types and composition, they are called *functors*.

4.1.1 Definition Let $\mathcal{A}, \mathcal{B}$ be categories. Then two mappings

$$F_{\mathcal{O}} : \mathcal{O}_{\mathcal{A}} \to \mathcal{O}_{\mathcal{B}} \quad \text{and} \quad F_{\mathcal{M}} : \mathcal{M}_{\mathcal{A}} \to \mathcal{M}_{\mathcal{B}}$$

together form a **functor** F from $\mathcal{A}$ to $\mathcal{B}$, written $F : \mathcal{A} \to \mathcal{B}$, iff

1. they preserve type information, i.e.,
$$\forall\ f : A \underset{\mathcal{A}}{\longrightarrow} B \quad F_{\mathcal{M}} f : F_{\mathcal{O}} A \underset{\mathcal{B}}{\longrightarrow} F_{\mathcal{O}} B \ ,$$
2. $F_{\mathcal{M}}$ maps identities to identities, i.e.,
$$\forall\ A \in \mathcal{O}_{\mathcal{A}} \quad F_{\mathcal{M}}\, \mathrm{id}_A = \mathrm{id}_{F_{\mathcal{O}} A} \ ,$$
3. and application of $F_{\mathcal{M}}$ distributes under composition of morphisms, i.e.,
$$\forall\ f : A \underset{\mathcal{A}}{\longrightarrow} B\, ;\ g : B \underset{\mathcal{A}}{\longrightarrow} C \quad F_{\mathcal{M}}(g \circ_{\mathcal{A}} f) = F_{\mathcal{M}} g \circ_{\mathcal{B}} F_{\mathcal{M}} f \ .$$

4.1.2 Notation Unless it is required to refer to only one of the mappings, the subscripts $_{\mathcal{M}}$ and $_{\mathcal{O}}$ are usually omitted. It is clear from the context whether an object or a morphism is mapped by the functor.

4.1.3 Definition Let $\mathcal{A}$ be a category. A functor $F : \mathcal{A} \to \mathcal{A}$ is called **endofunctor**.

It is easy to define a "identity" on a category, by simply combining the identities on objects and morphisms the same way we just have combined the functors. Also, applying one functor after another —where the target category of the first must be the source category of the second— looks like composition:

4.1.4 Definition Let $\mathcal{A}, \mathcal{B}, \mathcal{C}$ be categories, $F : \mathcal{A} \to \mathcal{B}$ and $G : \mathcal{B} \to \mathcal{C}$. We define **identities**

$$I_{\mathcal{A}} := \mathrm{id}_{\mathcal{O}_{\mathcal{A}} \uplus \mathcal{M}_{\mathcal{A}}}$$

and **functor composition** GF with

$$\forall\ A \in \mathcal{O} \quad (GF)A := G(FA)$$
$$\forall\ f \in \mathcal{M} \quad (GF)f := G(Ff) \ .$$

Due to this definition, we can write GFA and GFf without ambiguity. Multiple application of the same mapping is often noted with a superscript, i.e., we define $F^2 := FF$ for any functor F.

4.1.5 Lemma The identity and composition just defined yield functors again: In the situation of Definition 4.1.4,

$$I_{\mathcal{A}} : \mathcal{A} \to \mathcal{A} \ , \qquad GF : \mathcal{A} \to \mathcal{C} \ , \qquad I_{\mathcal{B}}F = F = FI_{\mathcal{A}} \ .$$

4.1.6 Proof is easy (✎). □

With that in mind, one can even consider categories to be the objects, and functors to be the morphisms of some higher level category (a category of categories). Although we do not follow this path, we do use composition of functors and identity functors in the sequel. However, we use the suggestive notation GF instead of $G \circ F$ since we do not discuss category theory by its own means.

4.2 Functors implement structure

The objects of a category may not have any structure, the structure may not be known to us, or the structure may not be suitable for our programming task, just as characters without additional structure are not suitable to represent a string. Functors are the tool to add some structure to an object.

A functor L may implement the List structure by mapping the object "integers" to the object "list of integers", and a function f that operates on integers, to a function Lf which applies f to each element of the list, returning the list of results. The same —suitably defined— functor L would map any object in the category to the corresponding list object, like the object "characters" to the object "strings", and booleans to bit-vectors.

In this example, a (not *the*) suitable category $\mathcal{S}$ for $L : \mathcal{S} \to \mathcal{S}$ would be the category where the objects are all sets, and the morphisms are all total functions between sets — that is our introductory example. Note, that the structure of the integers themselves is not affected by the functor application, hence we can say that the structure is added *on the outside* of the object. The addition of structure by application of a functor is often referred to using the term **lifting**, like in "*the integers are lifted to lists of integers*" or "*the lifted function now operates on lists.*"

This vague explanation of how a functor describes a structure is refined during the remainder of this guide. But already at this point, you can try to think about endofunctors that manifest structures like pairs, n-tuples, sets, and so on (✎ define some of these). Obviously, functor composition implements the nesting of structures, leading to structures like "list of pairs", "pair of lists", etc.

4.3 Functors in Haskell

Following our idea of a Haskell category $\mathcal{H}$, we can define an endofunctor $F : \mathcal{H} \to \mathcal{H}$ by giving a unary type constructor `F`, and a function `fmap`, constituting $F_{\mathcal{O}}$ and $F_{\mathcal{M}}$ respectively.

The type constructor `F` is used to construct new types from existing ones. This action corresponds to mapping objects to objects in the $\mathcal{H}$ category. The definition of `F` shows how a functor implements the structure of the constructed type. The function `fmap` lifts each function with a signature $f : A \to B$ to a function with signature $Ff : FA \to FB$.

Hence, functor application on an object is a *type level* operation in Haskell, while functor application on a morphism is a *value level* operation.

Haskell comes with the definition of a class

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

which allows overloading of `fmap` for each functor.

✎ Working examples are the *List* or *Maybe* structures (see Lemma 4.3.1). Both are members of the functor class, so you can enter the following lines into your interactive Haskell interpreter. The `:t` prefix causes *Hugs* and *GHCi* to print the type information of the following expression. This might not work in every environment. Understand the results returned by the Haskell interpreter.

```
--loading 'Char' seems to be required in GHCi, but not in Hugs
:m Char

:t fmap ord
fmap ord ""
fmap ord "lambda"
:t fmap chr
fmap chr Nothing
fmap chr (Just 42)
```

The examples illustrate how the functions `ord` and `chr` (from Haskell's Char module) are lifted to work on Lists and Maybes instead of just atomic values.

> You might stumble across the question what data constructors like `Just`, `Nothing`, `(:)` and `[]` actually *are*. At this point we are leaving category theory, and look *into* the objects which indeed turn out to have set properties in Haskell.
>
> A sound discussion of Haskell's type system is far beyond the scope of this little guide, so just imagine the data constructors to be some **markers** or **tags** to describe a member of an object. I.e., `Just "foo"` describes a member of the `Maybe String` object, as does the polymorphic `Nothing`.

The mapping function `fmap` differs from functor to functor. For the Maybe structure, it can be written as

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

while `fmap` for the List structure can be written as

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

Note, however, that having a type being a member of the `Functor` class does not imply to have a functor at all. Haskell does not check validity of the functor properties, so we have to do it by hand:

1. $f : A \to B \;\Rightarrow\; Ff : FA \to FB$ is fulfilled, since being a member of the Functor class implies that a function `fmap` with the according signature is defined.
2. $\forall\, A \in \mathcal{O} \quad F\,\mathrm{id}_A = \mathrm{id}_{FA}$ translates to

   ```
   fmap id == id
   ```

   which must be checked for each type one adds to the class. Note, that Haskell overloads the `id` function: The left occurrence corresponds to id_A, while the right one corresponds to id_{FA}.
3. $F(g \circ f) = Fg \circ Ff$ translates to

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

   which has to be checked, generalising over all functions `g` and `f` of appropriate type.

One should never add a type constructor to the `Functor` class that does not obey these laws, since this would be misleading for people reading the produced code.

4.3.1 Lemma Maybe and List are both functors.

I.e., they are not only members of the class, but also behave as expected in theory. The proof is easy (✎), and when ever you want to add a type to the functor class, you have to perform this kind of proof. So *do this as an exercise* before reading ahead.

4.3.2 Proof The proof of Lemma 4.3.1 strictly follows the structure of the Haskell types, i.e., we differentiate according to the data constructors used.

For the Maybe structure, the data constructors are `Just` and `Nothing`. So we prove the second functor property, $F\,\mathrm{id}_A = \mathrm{id}_{FA}$, by

```
fmap id Nothing
  == Nothing
  == id Nothing

fmap id (Just y)
  == Just (id y)
  == Just y
  == id (Just y) ,
```

and the third property, $F(g \circ f) = Fg \circ Ff$, by

```
(fmap g . fmap h) Nothing
  == fmap g (fmap h Nothing)
  == fmap g Nothing
  == Nothing
  == fmap (g . h) Nothing ,

(fmap g . fmap h) (Just y)
  == fmap g (fmap h (Just y))
  == fmap g (Just (h y))
  == Just (g (h y))
  == Just ((g . h) y)
  == fmap (g . h) (Just y)  .
```

Note, that List is —in contrast to Maybe— a recursive structure. This can be observed in the according definition of `fmap` above, and it urges us to use induction in the proof: The second property, $F\,\mathrm{id}_A = \mathrm{id}_{FA}$, is shown by

```
fmap id []
  == []
  == id []

fmap id (x:xs)
  == (id x):(fmap id xs)
  == (id x):(id xs)  --here we use induction
  == x:xs
  == id (x:xs) ,
```

and the third property, $F(g \circ f) = Fg \circ Ff$ can be observed in

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

fmap (g . f) (x:xs)
  == ((g . f) x):(fmap (g . f) xs)
  == ((g . f) x):((fmap g . fmap f) xs)  --induction
  == (g (f x)):(fmap g (fmap f xs))
  == fmap g ((f x):(fmap f xs))
  == fmap g (fmap f xs)
  == (fmap g . fmap f) xs  .
```

□

Chapter 5

Natural Transformations

5.1 Natural transformations in theory

A natural transformation is intended to transform from one structure to another, without effecting, or being influenced by, the objects in the structure.

Imagine two functors F and G —imposing two structures— between the same categories $\mathcal{A}$ and $\mathcal{B}$. Then a natural transformation is a collection of morphisms in the target category $\mathcal{B}$ of both functors, always mapping from an object FA to an object GA. In fact, for each object A in the source category there is one such morphism in the target category:

5.1.1 Definition Let $\mathcal{A}, \mathcal{B}$ be categories, $F, G : \mathcal{A} \to \mathcal{B}$. Then, a function

$$\begin{array}{cccc} \eta & : & \mathcal{O}_{\mathcal{A}} & \longrightarrow & \mathcal{M}_{\mathcal{B}} \\ & & A & \longmapsto & \eta_A \end{array}$$

is called a **transformation** from F to G, iff

$$\forall\, A \in \mathcal{O}_{\mathcal{A}} \quad \eta_A : FA \underset{\mathcal{B}}{\longrightarrow} GA \ ,$$

and it is called a **natural** transformation, denoted $\eta : F \stackrel{\cdot}{\longrightarrow} G$, iff

$$\forall\, f : A \underset{\mathcal{A}}{\longrightarrow} B \quad \eta_B \circ_{\mathcal{B}} Ff = Gf \circ_{\mathcal{B}} \eta_A \ .$$

$$\begin{tikzcd}
FA \arrow[rr, "\eta_A"] \arrow[ddd, "Ff"'] & & GA \arrow[ddd, "Gf"] \\
& A \arrow[ul, mapsto, "F"] \arrow[ur, mapsto, "G"'] \arrow[u, mapsto, "\eta"'] \arrow[d, "f"] & \\
& B \arrow[dl, mapsto, "F"'] \arrow[dr, mapsto, "G"] \arrow[d, mapsto, "\eta"] & \\
FB \arrow[rr, "\eta_B"'] & & GB
\end{tikzcd}$$

The definition says, that a natural transformation *transforms* from a structure F to a structure G, without altering the behaviour of morphisms on objects. I.e., it does not play a role whether a morphism f is lifted into the one or the other structure, when composed with the transformation.
In the drawing on the left, this means that the outer square *commutes*, i.e., all directed paths with common source and target are equal.

5.2 Natural transformations in Haskell

First note, that a unary function polymorphic in the same type on source and target side, in fact is a transformation. For example, Haskell's `Just` is typed

```
Just :: forall a. a -> Maybe a   .
```

If we imagine this to be a mapping `Just` $: \mathcal{O}_{\mathcal{H}} \to \mathcal{M}_{\mathcal{H}}$, the application on an object $A \in \mathcal{O}_{\mathcal{H}}$ means binding the type variable `a` to some Haskell type `A`. That is, `Just` maps an object A to a morphism of type $A \to$ `Maybe` A.

To spot a transformation here, we still miss a functor on the source side of `Just`'s type. This is overcome by adding the identity functor $I_{\mathcal{H}}$, which leads to the transformation

$$\texttt{Just} : I_{\mathcal{H}} \to \texttt{Maybe} \ .$$

5.2.1 Lemma `Just` $: I_{\mathcal{H}} \stackrel{\cdot}{\longrightarrow}$ `Maybe`.

5.2.2 Proof Let `f::A->B` be an arbitrary Haskell function. Then we have to prove, that

```
Just . I_H f == fmap f . Just
```

where the left `Just` refers to η_B, and the right one refers to η_A. In that line, we recognise the definition of the `fmap` for Maybe (just drop the $I_{\mathcal{H}}$). □

Another example, this time employing two non-trivial functors, are the `maybeToList` and `listToMaybe` functions. Their definitions read

```
maybeToList :: forall a. Maybe a -> [a]
maybeToList Nothing = []
maybeToList (Just x) = [x]

listToMaybe :: forall a. [a] -> Maybe a
listToMaybe [] = Nothing
listToMaybe (x:xs) = Just x .
```

Note, that the *loss of information* imposed by applying `listToMaybe` on a list with more than one element does not contradict naturality, i.e., *forgetting* data is not *altering* data.

We do not go through the proof of their naturality here (✎ but, of course, you can do this on your own). Instead, we discuss some more interesting examples in the next chapter (see Lemma 6.2.1).

5.3 Composing transformations and functors

To describe the Monad structure later on, we need to compose natural transformations with functors and with other natural transformations. This is discussed in the remainder of this chapter.

Natural transformations and functors Just as in Definition 4.1.4, we can define a composition between a natural transformation and a functor. Therefore, an object is first lifted by the functor and then mapped to a morphism by the transformation, or it is first mapped to a morphism which is then lifted:

5.3.1 Definition Let $\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}$ be categories, $E : \mathcal{A} \to \mathcal{B}$, $F, G : \mathcal{B} \to \mathcal{C}$, and $H : \mathcal{C} \to \mathcal{D}$. Then, for a natural transformation $\eta : F \stackrel{\cdot}{\longrightarrow} G$, we define the transformations ηE and $H\eta$ with

$$(\eta E)A := \eta_{EA} \quad \text{and} \quad (H\eta)B := H(\eta_B) \ ,$$

where A varies over $\mathcal{O}_{\mathcal{A}}$, and B varies over $\mathcal{O}_{\mathcal{B}}$.

Again, due to the above definition, we can write ηEA and $H\eta B$ (for A and B objects in the respective category) without ambiguity. The following pictures show the situation:

$$\begin{tikzcd}
\mathcal{O}_{\mathcal{A}} \arrow[r, "E"] \arrow[dr, "\eta E"'] & \mathcal{O}_{\mathcal{B}} \arrow[d, "\eta"] \\
& \mathcal{M}_{\mathcal{C}}
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathcal{O}_{\mathcal{B}} \arrow[d, "\eta"] \arrow[dr, "H\eta"] & \\
\mathcal{M}_{\mathcal{C}} \arrow[r, "H"'] & \mathcal{M}_{\mathcal{D}}
\end{tikzcd}$$

5.3.2 Lemma In the situation of Definition 5.3.1,

$$H\eta : HF \stackrel{\cdot}{\longrightarrow} HG \quad \text{and} \quad \eta E : FE \stackrel{\cdot}{\longrightarrow} GE$$

hold. In (other) words, $H\eta$ and ηE are both natural transformations.

5.3.3 Proof of Lemma 5.3.2

▷ **Part I**

$$
\begin{array}{ll}
 & \eta : F \overset{\cdot}{\longrightarrow} G \ , \\
\Rightarrow & \quad [\ \text{definition of naturality} \\
 & \forall\, f : A \xrightarrow[\mathcal{B}]{} B \quad \eta B \circ Ff = Gf \circ \eta A \\
\Rightarrow & \\
 & \forall\, f : A \xrightarrow[\mathcal{B}]{} B \quad H(\eta B \circ Ff) = H(Gf \circ \eta A) \\
\Rightarrow & \quad [\ \text{functor property} \\
 & \forall\, f : A \xrightarrow[\mathcal{B}]{} B \quad H(\eta B) \circ H(Ff) = H(Gf) \circ H(\eta A) \\
\Rightarrow & \quad [\ \text{Definition 5.3.1} \\
 & \forall\, f : A \xrightarrow[\mathcal{B}]{} B \quad (H\eta)B \circ (HF)f = (HG)f \circ (H\eta)A \\
\Rightarrow & \quad [\ \text{definition of naturality} \\
 & H\eta : HF \overset{\cdot}{\longrightarrow} HG \ ,
\end{array}
$$

▷ **Part II**

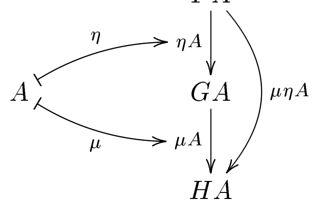
$$
\begin{array}{ll}
 & \eta : F \overset{\cdot}{\longrightarrow} G \\
\Rightarrow & \quad [\ \text{definition of naturality} \\
 & \forall\, f : A \xrightarrow[\mathcal{B}]{} B \quad \eta B \circ Ff = Gf \circ \eta A \\
\Rightarrow & \quad [\ \text{choose } f \text{ from category } \mathcal{A} \\
 & \forall\, f : A \xrightarrow[\mathcal{A}]{} B \quad \eta(EB) \circ F(Ef) = G(Ef) \circ \eta(EA) \\
\Rightarrow & \quad [\ \text{Definition 5.3.1} \\
 & \forall\, f : A \xrightarrow[\mathcal{A}]{} B \quad (\eta E)B \circ (FE)f = (GE)f \circ (\eta E)A \\
\Rightarrow & \quad [\ \text{definition of naturality} \\
 & \eta E : FE \overset{\cdot}{\longrightarrow} GH \ .
\end{array}
$$

□

Composing natural transformations Even natural transformations can be composed with each other. Since, however, transformations map objects to morphisms —instead of, e.g., objects to objects— they can not be simply applied one after another. Instead, composition is defined *component wise*, also called **vertical composition**.

5.3.4 Definition Let $\mathcal{A}, \mathcal{B}$ be categories, $F, G, H : \mathcal{A} \to \mathcal{B}$ and $\eta : F \overset{\cdot}{\longrightarrow} G$, $\mu : G \overset{\cdot}{\longrightarrow} H$. Then we define the transformations

$$
\begin{array}{rcccc}
\mathrm{id}_F & : & \mathcal{O}_{\mathcal{A}} & \longrightarrow & \mathcal{M}_{\mathcal{B}} \\
 & & A & \longmapsto & \mathrm{id}_{FA}
\end{array}
\quad \text{and} \quad
\begin{array}{rcccc}
\mu\eta & : & \mathcal{O}_{\mathcal{A}} & \longrightarrow & \mathcal{M}_{\mathcal{B}} \\
 & & A & \longmapsto & \mu A \circ \eta A \ .
\end{array}
$$



The picture on the left (with $A \in \mathcal{O}_{\mathcal{A}}$) clarifies why this composition is called *vertical*.
All compositions defined before Definition 5.3.4 are called **horizontal**. ✎ Why?

5.3.5 Lemma In the situation of Definition 5.3.4,

$$\begin{aligned}
&\mathrm{id}_F : F \overset{\cdot}{\longrightarrow} F \ , \\
&\mu\eta : F \overset{\cdot}{\longrightarrow} H \ , \\
&\mathrm{id}_G\,\eta = \eta = \eta\,\mathrm{id}_F
\end{aligned}$$

hold.

5.3.6 Proof of Lemma 5.3.5.
The naturality of id_F is trivial (✎). For the naturality of $\mu\eta$ consider any morphism $f : A \underset{\mathcal{A}}{\longrightarrow} B$. Then,

$$\begin{aligned}
& \mu : G \overset{\cdot}{\longrightarrow} H \quad \wedge \quad \eta : F \overset{\cdot}{\longrightarrow} G \\
\Rightarrow \quad & \\
& Hf \circ \mu A = \mu B \circ Gf \quad \wedge \quad Gf \circ \eta A = \eta B \circ Ff \\
\Rightarrow \quad & \qquad [\text{ apply } \eta A \text{ on the left, } \mu B \text{ on the right} \\
& (Hf \circ \mu A) \circ \eta A = (\mu B \circ Gf) \circ \eta A \quad \wedge \quad \mu B \circ (Gf \circ \eta A) = \mu B \circ (\eta B \circ Ff) \\
\Rightarrow \quad & \qquad [\text{ the two middle terms are equal} \\
& Hf \circ (\mu A \circ \eta A) = (\mu B \circ \eta B) \circ Ff \\
\Rightarrow \quad & \qquad [\text{ Definition 5.3.4} \\
& Hf \circ \mu\eta A = \mu\eta B \circ Ff \\
\Rightarrow \quad & \\
& \mu\eta : F \overset{\cdot}{\longrightarrow} H
\end{aligned}$$

□

Chapter 6

Monads

6.1 Monads in theory

6.1.1 Definition Let $\mathcal{C}$ be a category, and $F : \mathcal{C} \to \mathcal{C}$. Consider two natural transformations $\eta : I_{\mathcal{C}} \stackrel{\cdot}{\longrightarrow} F$ and $\mu : F^2 \stackrel{\cdot}{\longrightarrow} F$.

The triple (F, η, μ) is called a **monad**, iff

$$\mu(F\mu) = \mu(\mu F) \quad \text{and} \quad \mu(F\eta) = \mathrm{id}_F = \mu(\eta F) \ .$$

(Mind, that the parenthesis group composition of natural transformations and functors. They do not refer to function application. This is obvious due to the types of the expressions in question.)

The transformations η and μ are somewhat contrary: While η adds one level of structure (i.e., functor application), μ removes one. Note, however, that η transforms to F, while μ transforms from F^2. This is due to the fact that claiming the existence of a transformation from a functor F to the identity $I_{\mathcal{C}}$ would be too restrictive:

Consider the set category $\mathcal{S}$ and the list endofunctor L. Any transformation ϵ from L to $I_{\mathcal{S}}$ has to map every object A to a morphism ϵ_A, which in turn maps the empty list to some element in A, i.e.,

$$\begin{array}{ll} & \epsilon : L \to I_{\mathcal{S}} \\ \Rightarrow & \\ & \forall\ A \in \mathcal{O}_{\mathcal{S}} \quad \epsilon_A : LA \to A \\ \Rightarrow & \\ & \forall\ A \in \mathcal{O}_{\mathcal{S}} \quad \exists\ c \in A \quad \epsilon_A[\,] = c \ , \end{array}$$

where $[\,]$ denotes the empty list. This, however, implies

$$\forall\ A \in \mathcal{O}_{\mathcal{S}} \quad A \neq \emptyset \ .$$

The following drawings are intended to clarify Definition 6.1.1: The first equation of the definition is equivalent to

$$\forall\ A \in \mathcal{O} \quad \mu A \circ F\mu A = \mu A \circ \mu F A \ .$$

So what are $F\mu A$ and $\mu F A$? You can find them at the top of these drawings:

$$\begin{tikzcd} F^3A \arrow[rr, "F\mu A"] & {} & F^2A \\ F^2A \arrow[rr, "\mu A"] & {} \arrow[u, mapsto, "F"] & FA \\ & A \arrow[u, mapsto, "\mu"] & \end{tikzcd} \qquad \begin{tikzcd} F^3A \arrow[rr, "\mu F A"] & {} & F^2A \\ & FA \arrow[u, mapsto, "\mu"] & \\ & A \arrow[u, mapsto, "F"] & \end{tikzcd}$$

Since application of μA unnests a nested structure by one level, $F\mu A$ pushes application of this unnesting one level into the nested structure. We need at least two levels of nesting to apply μA.

So we need at least three levels of nesting to push the application of μA one level in. This justifies the type of $F\mu A$.

The morphism μFA, however, applies the reduction on the outer level. The FA only assures that there is one more level on the inside which, however, is not touched.

Hence, $F\mu A$ and μFA depict the two possibilities to flatten a three-level nested structure into a two-level nested structure through application of a mapping that flattens a two-level nested structure into an one-level nested structure.

The statement $\mu A \circ F\mu A = \mu A \circ \mu FA$ says, that after another step of unnesting (i.e., μA), it is irrelevant which of the two inner structure levels has been removed by prior unnesting (i.e., $F\mu A$ or μFA).

Let us have a look at the second equation as well. It is equivalent to

$$\forall\ A \in \mathcal{O}_\mathcal{C} \quad \mu A \circ F\eta A = \mathrm{id}_{FA} = \mu A \circ \eta FA \ .$$

Again, we examine $F\eta A$ and ηFA, which are at the top of the drawings.

$$\begin{tikzcd}
FA \arrow[rr, "F\eta A"] & {} & F^2A \\
A \arrow[rr, "\eta A"] & {} \arrow[u, maps to, "F"] & FA \\
& A \arrow[u, maps to, "\eta"] &
\end{tikzcd}
\qquad
\begin{tikzcd}
FA \arrow[rr, "\eta FA"] & {} & F^2A \\
& FA \arrow[u, maps to, "\eta"] & \\
& A \arrow[u, maps to, "F"] &
\end{tikzcd}$$

Since ηA adds one level of nesting to A, the morphism $F\eta A$ imposes this lifting inside a flat structure (FA), yielding a nested structure F^2A.

Also, ηFA adds one level of nesting, however on the outside of a flat structure.

I.e., similar to the situation above, $F\eta A$ and ηFA reflect the two ways to add one level of nesting to a flat structure: It can be done by lifting the whole structure, or by lifting the things within the structure.

So $\mu A \circ F\eta A = \mathrm{id}_{FA} = \mu A \circ \eta FA$ states, just as for the first equation, that after unnesting (i.e., μA) it is not relevant whether additional structure has been added on the outside (ηFA) or the inside ($F\eta A$) of a flat structure FA. Additionally, it says that nesting followed by unnesting, is the identity.

6.2 Monads in Haskell

The list structure (we use L to denote the according functor) provided by the Haskell language is probably the most intuitive example to explain a monad. That it is a monad indeed is stated in Lemma 6.2.1.

Imagine a list of lists of lists of natural numbers (i.e., $L^3\mathbb{N}$). With a transformation like list concatenation ($\mu : L^2 \to L$) we can choose between two alternatives of flattening, considering L^3 to be a "list of (list of lists)" or a "(list of lists) of lists":

μLA applies the *concatenation of a list of lists* to the outermost *list of lists*. Here, A refers to the object $L\mathbb{N}$ corresponding to the third-level list structure, which is the topmost level in the nested structure that is not effected by the transformation.

Application of μLA returns a new list, containing all the unchanged innermost lists. You can try (✎)

```
concat [   [ [1,2], [3,4] ]
       ,   [ [5,6], [7,8] ]
       ]
```

in your Haskell interpreter.

$L\mu A$ lifts the *concatenation of a list of lists* into the outermost list, applying it to each of the second-level *lists of lists*. Here, A refers to the object $\mathbb{N}$, corresponding to the members of the third level list structure. $\mathbb{N}$ is the topmost level in the nested structure that is not effected by the transformation.

Application of $L\mu A$ returns the outermost list with its elements replaced by the results of applying μA on them. You can observe this using (✎) the Haskell code

```
fmap concat [  [ [1,2], [3,4] ]
            ,  [ [5,6], [7,8] ]
            ]
```

with your interpreter.

The monad property $\mu(F\mu) = \mu(\mu F)$ can be observed by applying `concat` to the results of the two expressions given above. Both yield the same result.

Now, consider the transformation $\eta : I_\mathcal{C} \rightarrow L$, which, parametrised with an object A, maps a member from A to the singleton list in LA containing that member. Again, we face two alternatives:

ηLA applies the *list making* to the whole input list, returning a list which simply contains the original input list as sole member. Feed (✎) the code

```
(\x -> [x]) [1,2,3]
```

to Haskell.

$L\eta A$ applies the *list making* to each element in the input list, returning the input list with its members replaced by the according singleton lists. You can type (✎)

```
fmap (\x -> [x]) [1,2,3]
```

to verify this.

The monad property $\mu(L\eta) = \mathrm{id}_L = \mu(\eta L)$ can be observed (✎) by applying `concat` to the results of the two expressions given above. Both yield the same result.

6.2.1 Lemma Haskell's List structure L, together with concatenation μ and the creation of singleton lists η, forms a monad (L, μ, η).

6.2.2 Proof We already know that L is a functor (Lemma 4.3.1). The remaining work to do is: Prove the naturality of μ and η, and show that the monad property holds.

We use the Haskell notation for lists, i.e., $[\,]$ denotes the empty list, and $x : x'$ denotes the list x' prepended with the element x. The definition of `fmap` yields

$$(Lf)[\,] = [\,]$$
$$(Lf)(x : x') = x : (Lf)x' \ .$$

Note, that `concat` is defined in terms of `foldr` and a binary concatenation operator $(+\!\!+)$ we consider primitive:

```
concat = foldr (++) [] .
```

From this, we conclude $\mu_B(x : x') = x +\!\!+ \mu_B x'$ and $\mu_B[\,] = [\,]$.

▷ **Part I** To prove that μ is a natural transformation, we have to show that

$$\forall\, f : A \rightarrow B \quad \mu B \circ L^2 f = Lf \circ \mu A$$

holds. We use induction on the structure of the list x.
Let $x = [\,] \in L^2 A$. Then, we observe that

$$(\mu B \circ L^2 f)[\,] = (Lf \circ \mu A)[\,]$$

holds by looking at the definitions of `fmap` (L) and `concat` $(\mu_A$ and $\mu_B)$.

Let $x = y : y'$, where $y \in LA$ and $y' \in L^2A$.

$(\mu_B \circ L^2 f)x$
$=$
$\mu_B((L^2 f)(y : y'))$
$=$ [definition of `fmap`
$\mu_B((Lf)y : (L^2 f)y')$
$=$ [definition of `concat`
$(Lf)y +\!\!+ \mu_B((L^2 f)y')$
$=$ [induction: $(\mu_B \circ L^2 f)y' = (Lf \circ \mu_A)y'$
$(Lf)y +\!\!+ (Lf)(\mu_A y')$
$=$
$(Lf)(y +\!\!+ \mu_A y')$
$=$ [definition `concat`
$(Lf)(\mu_A(y : y'))$
$=$
$(Lf \circ \mu_A)x$

▷ **Part II** To prove that η is a natural transformation, we have to show that

$$\forall\, f : A \to B \quad \eta B \circ f = Lf \circ \eta A$$

holds. The proof reads

$(\eta_B \circ f)x$
$=$
$\eta_B(fx)$
$=$
$[fx]$
$=$
$(Lf)[x]$
$=$
$(Lf)(\eta_A x)$
$=$
$(Lf \circ \eta_A)x$.

▷ **Part III** Now we show that the monad properties for lists hold.
Again, we use induction on the structure of a list x.

▷ **Part III.a** Proof that $\mu(L\mu) = \mu(\mu L)$.
Let $x = [\,] \in F^3A$.

$(\mu_A \circ L\mu_A)[\,]$
$=$
$\mu_A((L\mu_A)[\,])$
$=$
$\mu_A[\,]$
$=$
$\mu_A(\mu_{LA}[\,])$
$=$
$(\mu_A \circ \mu_{LA})[\,]$.

Let $x = y : y'$, where $y \in L^2A$ and $y' \in L^3A$.

$(\mu_A \circ L\mu_A)x$
$=$
$\mu_A((L\mu_A)(y : y'))$
$=$ [definition of `fmap`
$\mu_A(\mu_A y : (L\mu_A)y')$
$=$ [definition of `concat`
$\mu_A y +\!\!+ \mu_A((L\mu_A)y')$
$=$ [induction
$\mu_A y +\!\!+ \mu_A(\mu_{LA} y')$
$=$ [definition of `concat`

$$
\begin{aligned}
= \;& \mu_A(y +\!\!+ \mu_{LA} y') \\
= \;& \mu_A(\mu_{LA}(y : y')) \\
& (\mu_A \circ \mu_{LA})x \;\; .
\end{aligned}
$$

▷ **Part III.b** Proof that $\mu(L\eta) = \mathrm{id}_L = \mu(\eta L)$.
Let $x = [\,] \in LA$.

$$
\begin{aligned}
& (\mu_A \circ L\eta_A)[\,] \\
= \;& \mu_A((L\eta_A)[\,]) \\
= \;& \quad [\text{ definition of } \texttt{fmap} \\
& \mu_A[\,] \\
= \;& \quad [\text{ definition of } \texttt{concat} \\
& [\,] \\
= \;& \quad [\text{ definition of } \texttt{concat} \\
& \mu_A[[\,]] \\
= \;& \mu_A(\eta_{LA}[\,]) \\
= \;& (\mu_A \circ \eta_{LA})[\,]
\end{aligned}
$$

Let $x = y : y'$, where $y \in A$ and $y' \in LA$.

$$
\begin{aligned}
& (\mu_A \circ L\eta_A)x \\
= \;& \mu_A((L\eta_A)(y : y')) \\
= \;& \quad [\text{ definition of } \texttt{fmap} \\
& \mu_A(\eta_A y : (L\eta_A)y') \\
= \;& \quad [\text{ definition of } \texttt{concat} \\
& \eta_A y +\!\!+ \mu_A((L\eta_A)y') \\
= \;& \quad [\text{ induction: } \mu_A \circ L\eta_A = \mathrm{id}_{LA} \\
& \eta_A y +\!\!+ y' \\
= \;& y : y' \\
= \;& \mu_A[y : y'] \\
= \;& \mu_A(\eta_{LA}(y : y')) \\
= \;& (\mu_A \circ \eta_{LA})x \;\; .
\end{aligned}
$$

□

6.3 An alternative definition of the monad

While the previous definition of the monad structure is quite intuitive, there is another one and, in fact, that is the one Haskell uses for its `Monad` class.

First, we use the monad as introduced in Definition 6.1.1, to define a *bind* operator, and we prove *the three monad laws* to hold for this definition. Then, we show that the three monad laws alone imply all the properties required to form a monad, hence yield an alternative definition.

6.3.1 Definition Given (F, η, μ), we define the binary infix operator **bind** by

$$
\begin{array}{ccccccc}
\gg\!= & : & FA & \times & (A \to FB) & \longrightarrow & FB \\
 & & x & , & f & \longmapsto & (\mu_B \circ Ff)x \;\; .
\end{array}
$$

Note, that we assume x to be a member of FA, which restricts us to categories which support this — like, e.g., the category of sets, or $\mathcal{H}$. In literature not related to the Haskell language a point-free variant of the bind operator, called *Kleisli star*, is used:

6.3.2 Definition The unary postfix operator **Kleisli star**, a point-free version of the bind operator, is defined by

$$\begin{array}{cccc} {}^* & : & (A \to FB) & \longrightarrow & (FA \to FB) \\ & & f & \longmapsto & \mu_B \circ Ff \ . \end{array}$$

Obviously $\forall\, x \in FA\, ;\ f : A \to FB \quad x \gg\!= f = f^* x$.

In fact, the bind operator is a special case of the Kleisli star, restricted to categories where the objects have members. Ignoring the lack of elegance and generality, we stick to the bind notation in the following, since this matches the Haskell notation. However, it is easy (✎) to reformulate the following statements and proofs to the more general Kleisli notation.

The second argument of $\gg\!=$ is a function f, which adds some structure F to what it returns. Intuitively, as defined above, the bind operator lifts the passed function f, applies it to the object $x \in FA$, and then applies μ to remove one level of structure nesting.

The bind operator is used to formulate *the three monad laws*. We give them in the form of a lemma and prove their correctness using the monad properties given in Definition 6.1.1.

6.3.3 Lemma The three monad laws are:

1. η_A resembles a left identity with respect to the bind operator:
   $\forall\, f : A \to FB\, ;\ x \in A \quad \eta_A x \gg\!= f = fx$.
2. η_A is a right identity with respect to the bind operator:
   $\forall\, x \in FA \quad x \gg\!= \eta_A = x$
3. The bind operator is quite close to being associative:
   $\forall\, x \in FA\, ;\ f : A \to FB\, ;\ g : B \to FC$
   $(x \gg\!= f) \gg\!= g = x \gg\!=(y \mapsto fy \gg\!= g)$

Just as a hint for the exercise: Using Kleisli notation, the three monad laws read

1. $\forall\, f : A \to FB \quad f^* \circ \eta_A = f$
2. $\forall\, A \in \mathcal{O} \quad \eta_A^* = \mathrm{id}_{FA}$
3. $\forall\, f : A \to FB\, ;\ g : B \to FC \quad g^* \circ f^* = (g^* \circ f)^*$

6.3.4 Proof of Lemma 6.3.3.

▹ **Part I** Let $f : A \to FB$, and $x \in A$.

$$\begin{array}{cl}
& \eta_A x \gg\!= f \\
= & \quad [\ \text{bind operator, Definition 6.3.1} \\
& (\mu_B \circ Ff)(\eta_A x) \\
= & \\
& (\mu_B \circ Ff \circ \eta_A)x \\
= & \quad [\ \text{naturality of } \eta \text{ means } \eta_{FB} \circ If = Ff \circ \eta_A \\
& (\mu_B \circ \eta_{FB} \circ If)x \\
= & \quad [\ \text{using the monad property } \mu(\eta F) = \mathrm{id}_F \\
& (\mathrm{id}_{FB} \circ f)x \\
= & \\
& fx
\end{array}$$

▹ **Part II** Let $x \in FA$.

$$\begin{array}{cl}
& x \gg\!= \eta_A \\
= & \quad [\ \text{using Definition 6.3.1 of the bind operator} \\
& (\mu_A \circ F\eta_A)x \\
= & \quad [\ \text{monad property } \mu(F\eta) = \mathrm{id}_F
\end{array}$$

$\mathrm{id}_{FA}\, x$
$=$
x

▷ **Part III** Let $x \in FA$, $f : A \to FB$, and $g : B \to FC$.

$(x \gg\!= f) \gg\!= g$
$=$ [using Definition 6.3.1 of the bind operator
$((\mu_B \circ Ff)x) \gg\!= g$
$=$ [dito
$(\mu_C \circ Fg)((\mu_B \circ Ff)x)$
$=$
$(\mu_C \circ Fg \circ \mu_B \circ Ff)x$
$=$ [naturality of μ means $\mu_{FC} \circ F^2 g = Fg \circ \mu_B$
$(\mu_C \circ \mu_{FC} \circ F^2 g \circ Ff)x$
$=$ [using the monad property $\mu(\mu F) = \mu(F\mu)$
$(\mu_C \circ F\mu_C \circ F^2 g \circ Ff)x$
$=$ [F is a functor, so it distributes under composition.
$(\mu_C \circ F(\mu_C \circ Fg \circ f))x$
$=$ [the bind operator again
$x \gg\!= \mu_C \circ Fg \circ f$
$=$ [build a λ-expression
$x \gg\!=(y \mapsto (\mu_C \circ Fg \circ f)y)$
$=$
$x \gg\!=(y \mapsto (\mu_C \circ Fg)(fy))$
$=$ [the bind operator again
$x \gg\!=(y \mapsto fy \gg\!= g)$

□

Note, that the three monad laws do not make use of μ or $F_{\mathcal{M}}$. In the following, it turns out that defining (F, η, μ) is equivalent to defining $(F_{\mathcal{O}}, \eta, \gg\!=)$, i.e., each of these tuples can be defined in terms of the other: The one direction is obvious using Definition 6.3.1, the reverse is given in Definition 6.3.6. Moreover, Theorem 6.3.5 states that the three monad laws are another way to formulate the previously given monad properties (Definition 6.1.1).

6.3.5 Theorem Let $\mathcal{C}$ be a category, and $F_{\mathcal{O}} : \mathcal{O} \to \mathcal{O}$ an arbitrary object mapping. Consider a function (like a transformation)

$$\begin{array}{rccl} \eta & : & \mathcal{O} & \longrightarrow & \mathcal{M} \\ & & A & \longmapsto & \eta_A \end{array}$$

with $\eta_A : A \underset{\mathcal{C}}{\longrightarrow} FA$, and an operator

$$\gg\!= \, : F_{\mathcal{O}}A \times (A \to F_{\mathcal{O}}B) \longrightarrow F_{\mathcal{O}}B \ .$$

Then, the triple $(F_{\mathcal{O}}, \eta, \gg\!=)$ forms a monad, iff the three monad laws (as given in Lemma 6.3.3) hold.

Note, that η is not required to be a natural transformation. This turns out to be a result of the three monad laws.

6.3.6 Definition In the situation of Theorem 6.3.5, we can define μ and $F_{\mathcal{M}}$ for all $A \in \mathcal{O}_{\mathcal{C}}$ and all $f : A \underset{\mathcal{C}}{\longrightarrow} B$ by

$$\begin{array}{rccl} \mu_A & : & F_{\mathcal{O}}^2 A & \longrightarrow & F_{\mathcal{O}}A \\ & & x & \longmapsto & x \gg\!= \mathrm{id}_{F_{\mathcal{O}}A} \end{array}$$

and

$$\begin{array}{rcccl} F_{\mathcal{M}} f & : & F_{\mathcal{O}} A & \longrightarrow & F_{\mathcal{O}} B \\ & & x & \longmapsto & x \gg\!= \eta_B \circ f \ . \end{array}$$

Again, we write F for $F_{\mathcal{O}}$ as well as for $F_{\mathcal{M}}$, since they hardly can be confused. Definition 6.3.6 is used in the following, so bear in mind that we have to prove the monad properties of the (F, η, μ) tuple just defined, by using the three monad laws we claimed to hold in Theorem 6.3.5.

6.3.7 Proof of Theorem 6.3.5. Due to Lemma 6.3.3, we only need to show the functor properties of F, that η is a natural transformation, and that we can conclude the monad properties given in Definition 6.1.1 from the three monad laws given in Lemma 6.3.3.

▷ **Part I** First, we show the functor properties of F.

▷ **Part I.a** That $F_{\mathcal{M}}$ preserves type information, i.e., that

$$\forall\, f : A \to B \quad Ff : FA \to FB$$

holds, follows directly from its definition.

▷ **Part I.b** $F_{\mathcal{M}}$ preserves identities, since for all $x \in FA$

$$(F\,\mathrm{id}_A)x = x \gg\!= \eta_A \circ \mathrm{id}_A = x \gg\!= \eta_A = x = \mathrm{id}_{FA}\, x$$

holds.

▷ **Part I.c** For the distribution of $F_{\mathcal{M}}$ under composition we fix arbitrary $x \in FA$, $f : A \to B$ and $g : B \to C$. Then

$$\begin{array}{cl} & (Fg \circ Ff)x \\ = & \quad [\text{ Definition 6.3.6 of } F_{\mathcal{M}} \text{ applied twice} \\ & (x \gg\!= \eta_B \circ f) \gg\!= \eta_C \circ g \\ = & \quad [\text{ the 3rd monad law} \\ & x \gg\!=(y \mapsto (\eta_B \circ f)y \gg\!= \eta_C \circ g) \\ = & \quad [\text{ the 1st monad law} \\ & x \gg\!=(y \mapsto (\eta_C \circ g \circ f)y) \\ = & \quad [\text{ removing the } \lambda\text{-expression} \\ & x \gg\!= \eta_C \circ g \circ f \\ = & \quad [\text{ Definition 6.3.6 of } F_{\mathcal{M}} \text{ applied to } (g \circ f) \\ & (F(g \circ f))x \ . \end{array}$$

Thus, $\forall\, f : A \to B\,;\ g : B \to C \quad F(g \circ f) = Ff \circ Gf$.

Together, we conclude that F is an endofunctor in $\mathcal{C}$.

▷ **Part II** Since F is a functor, η is a transformation from $I_{\mathcal{C}}$ to F by definition. To show its naturality, choose arbitrary $x \in A$ and $f : A \to B$. Then

$$\begin{array}{cl} & (Ff \circ \eta_A)x \\ = & \\ & Ff(\eta_A x) \\ = & \quad [\text{ Definition 6.3.6 of } F_{\mathcal{M}} \\ & \eta_A x \gg\!= \eta_B \circ f \\ = & \quad [\text{ the 1st monad law} \\ & (\eta_B \circ f)x \\ = & \quad [\text{ inserting the identity functor} \\ & (\eta_B \circ If)x \end{array}$$

Thus, $\eta : I_{\mathcal{C}} \dot{\longrightarrow} F$.

▷ **Part III** To show $\mu(\eta F) = \mathrm{id}_F = \mu(F\eta)$, we prove the two equations separately for fixed arbitrary $A \in \mathcal{O}$, and $x \in FA$.

▷ **Case III.a** For the left hand side equation the proof reads

$(\mu A \circ \eta F A)x$
$=$
$\mu A((\eta F A)x)$
$=$ [Definition 6.3.6 of μ
$((\eta F A)x) \ggg \mathrm{id}_{FA}$
$=$ [Use that $(\eta A')x \ggg f = fx$ for $x \in A'$, $f : A' \to FB'$, and assign $A' := FA$, $B' := A$ and $f := \mathrm{id}_{FA}$.
$\mathrm{id}_{FA}\, x$
$=$
x .

▷ **Case III.b** For the right hand side equation we calculate

$(\mu A \circ F\eta A)x$
$=$
$(\mu A)((F\eta A)x)$
$=$ [Definition 6.3.6 of $F_{\mathcal{M}}$ with $f := \eta_A$ reads $(F\eta_A)y = y \ggg \eta_{FA} \circ \eta_A$, for $y \in FA$.
$(\mu A)(x \ggg \eta F A \circ \eta A)$
$=$ [Definition 6.3.6 of μ
$(x \ggg \eta F A \circ \eta A) \ggg \mathrm{id}_{FA}$
$=$ [the 3rd monad law
$x \ggg (y \mapsto (\eta F A \circ \eta A)y \ggg \mathrm{id}_{FA})$
$=$ [Definition 6.3.6 of μ used backwards
$x \ggg (y \mapsto \mu A((\eta F A \circ \eta A)y))$
$=$ [remove the λ-expression
$x \ggg \mu A \circ \eta F A \circ \eta A$
$=$ [Part III.a: $\mu(\eta F) = \mathrm{id}_F$
$x \ggg \eta A$
$=$ [the 2nd monad law
x

▷ **Part IV** To show $\mu(\mu F) = \mu(F\mu)$ we fix arbitrary $A \in \mathcal{O}$ and $x \in F^3 A$. Then,

$(\mu A \circ \mu F A)x$
$=$
$(\mu A)((\mu_{FA})x)$
$=$ [Definition 6.3.6 of μ, using FA instead of A there
$(\mu A)(x \ggg \mathrm{id}_{F^2 A})$
$=$ [definition of μ again
$(x \ggg \mathrm{id}_{F^2 A}) \ggg \mathrm{id}_{FA}$
$=$ [the 3rd monad law
$x \ggg (y \mapsto \mathrm{id}_{F^2 A}\, y \ggg \mathrm{id}_{FA})$
$=$
$x \ggg (y \mapsto y \ggg \mathrm{id}_{FA})$
$=$ [definition of μ
$x \ggg (y \mapsto (\mu A)y)$
$=$ [removing the λ-expression
$x \ggg \mu A$
$=$

$$
\begin{aligned}
& x \gg\!= \mathrm{id}_{FA} \circ \mu A \\
= \quad & [\ \text{Part III.a: } \mu(\eta F) = \mathrm{id}_F \\
& x \gg\!= \mu A \circ \eta F A \circ \mu A \\
= \quad & [\ \text{building a } \lambda\text{-expression} \\
& x \gg\!= (y \mapsto (\mu A)((\eta F A \circ \mu A) y)) \\
= \quad & [\ \text{definition of } \mu \\
& x \gg\!= (y \mapsto (\eta F A \circ \mu A) y \gg\!= \mathrm{id}_{FA}) \\
= \quad & [\ \text{the 3rd monad law} \\
& (x \gg\!= \eta F A \circ \mu A) \gg\!= \mathrm{id}_{FA} \\
= \quad & [\ \text{definition of } \mu \\
& (\mu A)(x \gg\!= \eta F A \circ \mu A) \\
= \quad & [\ \text{definition of } F_{\mathcal{M}}\text{, used backwards, with } f := \mu A \\
& (\mu A)((F \mu A) x) \\
= \quad & (\mu A \circ F \mu A) x
\end{aligned}
$$

□

6.4 Haskell's monad class and do-notation

Haskell provides the programmer with a class `Monad`, which uses the alternative definition given above.

```
class Monad m :: (* -> *) where
  (>>=) :: forall a b. m a -> (a -> m b) -> m b
  return :: forall a. a -> m a;
```

In fact, there are two more functions defined —namely `(>>)` and `fail`— but we do not discuss them here. The `return` function is Haskell's equivalent to the natural transformation η. The operator `>>=` simply is the bind operator. The type variable `m` is bound to a unary type constructor: the functor.

As for the `Functor` class, being a member of `Monad` alone does not assure adherence to the monad laws. This has to be checked by the programmer in advance. Luckily, Theorem 6.3.5 saves us from doing this again, if we did it before.

Definition 6.3.1 directly shows how we could make Haskell's List structure a member of the `Monad` class — if it was not already:

```
instance Monad List where
  return x = [x]
  x >>= f = (concat . fmap f) x
```

Also, we could make `Maybe` a member of the `Monad` class:

```
instance Monad Maybe where
  return = Just
  Nothing >>= f = Nothing
  (Just x) >>= f = f x
```

It is a nice —and easy— exercise, to verify the three monad laws for the Maybe monad as given above.

Membership of the `Monad` class allows usage of the *do-notation*. This is a syntactic construct provided by the Haskell compiler which resembles imperative programming.

6.4.1 Definition The **do-notation** is defined (Section 3.14 of [5]) using the following recursive set of rules (for brevity, we skip the definition of *let* clauses). To form a do-expression, the do keyword operates on a sequence of terms $t_0; \ldots; t_n$, where t_n is a Haskell expression, and the t_i with $0 \leq i < n$ are either Haskell expressions as well, or pattern matching expressions $v_i \leftarrow e_i$, where v_i are patterns introducing new variable names.

1. If the list of terms contains only one term —which then must be an expression— the do is simply removed:

   $$\text{do}\ \{\ e\ \} \quad := \quad e\ .$$

2. If the first term is a pattern matching expression $v \leftarrow e$, a λ-expression is created binding the pattern v in a do-expression containing the remaining terms. The result of the expression e is bound to the new λ-expression:

   $$\text{do}\ \{\ v \leftarrow e;\ t_1;\ \ldots;\ t_n\ \} \quad := \quad e \ggg\!= (v \mapsto \text{do}\ \{\ t_1;\ \ldots;\ t_n\ \})\ .$$

   Note, how this resembles *variable assignment* in an imperative programming language, and also justifies the name of the bind operator.

3. If there are multiple terms in the sequence, but the first one does not contain a pattern matching, we use

   $$\text{do}\ \{\ e;\ t_1;\ \ldots;\ t_n\ \} \quad := \quad e \ggg\!= (\_ \mapsto \text{do}\ \{\ t_1;\ \ldots;\ t_n\ \})\ ,$$

   which means that the value of e is not used in the remaining term.

Above, the sequence of terms is surrounded by curly braces, but it is also possible to use Haskell's two-dimensional syntax.

6.5 First step towards IO

As introduction, we build a monad that allows us to model IO operations. Assume our running program to be connected to an input and an output character stream. We encapsulate their state inside a monad structure and use imperative programming style to manipulate them. Hence, in the following example, we do not use the "real" IO monad.

The state of the input and output streams is maintained in a pair of strings

```
(String,String)
```

where the former string models the characters waiting in the input stream, and the latter represents what was written to the output stream. Hence, a value of

```
("foo","bar")
```

means, that the output `bar` has been written already, and that `foo` has not been read from the input yet.

Obviously, we can model IO by passing around a parameter containing the *state of the world* outside our Haskell program. In fact, this is what we'll do, however encapsulating the state in a monad structure.

Functions that change the *state of the world*, i.e., read or write the streams, are called *state transformations*. We use (✎) a data type

```
data ST a = ST( (String,String) -> ((String,String),a) )
```

to model them. The state transformations are equipped with a type parameter `a`. This is required, because we want the state transformations to return data, depending on the change performed to the world. E.g., reading a character from the input not only changes the input stream (by removing that character), but also *returns* that character. Hence, we refer to `a` as **return value** of a transformation.

With this, we can already define (✎) our primitive input and output routines:

```
getc :: ST Char
getc = ST( λ((i:is),os) -> ((is,os),i) )

putc :: Char -> ST ()
putc c = ST( λ(is,os) -> ((is,os++[c]),()) )
```

The `getc` function takes the first character from the input stream, and returns it in `i` — note the type parameter `Char` passed to `ST`. Clearly, `getc` transforms the input state, by removing one character from the input stream. The output stream remains unchanged.

Writing to the output stream appends the character passed to the `putc` function to the already written output `os`. We don't want to return anything, which we model by returning Haskell's unit type value `()`.

So how do we define the monad? Since we want to use do-notation, we make the state transformation an instance of the monad class

```
instance Monad ST where
...
```

The natural transformation η is used to *return* a value, i.e., ηA maps a value $x \in A$ to a state transformation which returns x without altering the *state of the world* s_0:

$$\eta_A x := s_0 \mapsto (s_0, x) \ .$$

In Haskell syntax (✎), this reads

```
return :: a -> ST a
return val = ST (λs0 -> (s0,val)) .
```

The bind operator is a bit more complex: Assuming S to be the functor that embodies the structure `ST` of a state transformation, for a morphism $f : A \to SB$ and a transformation $t_0 \in SA$ (which returns a value in A) we define

$$t_0 \gg= f := s_0 \mapsto t_1 s_1 \quad \text{where} \quad (s_1, v) = t_0 s_0 \text{ and } t_1 = f v \ .$$

To understand this, observe that the binding of a state transformation t_0 to a function $f : A \to SB$ returns a new state transformation, which is assembled as follows:

The state transformation t_0 is applied to the input state s_0 of the generated transformation, returning a new state s_1 and a return value v. Applying f to this value leads to a new state transformation t_1 which is then applied to the new state s_1.

It becomes clear now, how the return value is used: Transforming the input state s_0 using t_0 returns a value in v which is passed to f to create the new state transformation t_1.

In Haskell syntax, the bind operator reads as follows:

```
(>>=) ::  ST a -> (a -> ST b) -> ST b
(ST t0) >>= f = ST( λs0 -> let (s1,val) = t0 s0
                               (ST t1) = f val
                               in t1 s1
                  )
```

6.5.1 Theorem The triple `(ST,return,>>=)` is a monad.

6.5.2 Notation We introduce a bit of notation to increase readability of the following proof: Let $[q]_k$ address the k-th member of a tuple q if it contains at least k entries, i.e.,

$$\forall\, k, n \in \mathbb{N}\,;\ 1 \le k \le n \quad [(a_1, \dots a_n)]_k = a_k \ .$$

We use this notation to access the new state and the return value of a state transformation. The definition of the $\gg=$ operator for the `ST` structure now reads

$$\begin{aligned} & t \gg= f \\ = \ & s \mapsto (\ f[ts]_2\ )\ [ts]_1 \end{aligned}$$

where $[ts]_1$ refers to the new state, and $[ts]_2$ refers to the return value.

6.5.3 Proof that S forms a monad.

▷ **Part I** The first two parts are quite trivial. But let us get used to the $[\cdot]_k$-notation. We see the first monad law by

$$\begin{aligned}
& t \gg\!= \eta_A \\
= \; & s \mapsto (\eta_A[ts]_2)[ts]_1 \\
= \; & \quad [\text{ definition of } \eta \\
& s \mapsto (s' \mapsto (s', [ts]_2))[ts]_1 \\
= \; & s \mapsto ([ts]_1, [ts]_2) \\
= \; & s \mapsto ts \\
= \; & t \; .
\end{aligned}$$

▷ **Part II** And the second law is shown by

$$\begin{aligned}
& \eta_A x \gg\!= f \\
= \; & \quad [\text{ definition of } \gg\!= \text{ using Notation 6.5.2} \\
& s \mapsto (f[(\eta_A x)s]_2)[(\eta_A x)s]_1 \\
= \; & \quad [\text{ definition of } \eta \\
& s \mapsto (f[(s' \mapsto (s', x))s]_2)[(s' \mapsto (s', x))s]_1 \\
= \; & s \mapsto (f[(s, x)]_2)[(s, x)]_1 \\
= \; & s \mapsto (fx)s \\
= \; & fx \; .
\end{aligned}$$

▷ **Part III** The least intuitive part is probably the third monad law:

$$\begin{aligned}
& (t \gg\!= f) \gg\!= g \quad = \quad t \gg\!=(y \mapsto fy \gg\!= g) \\
\Leftarrow \; & \quad [\text{ definition of } \gg\!= \text{ applied to its right occurrence on the left hand side} \\
& s \mapsto (g[(t \gg\!= f)s]_2)[(t \gg\!= f)s]_1 \quad = \quad t \gg\!=(y \mapsto fy \gg\!= g) \\
\Leftarrow \; & \quad [\text{ definition of } \gg\!= \text{ applied to its left occurrence on the right hand side} \\
& s \mapsto (g[(t \gg\!= f)s]_2)[t \gg\!= f)s]_1 \quad = \quad s \mapsto (f[ts]_2 \gg\!= g)[ts]_1 \\
\Leftarrow \; & \quad [\text{ generalising over all } s \\
& (g[(t \gg\!= f)s]_2)[(t \gg\!= f)s]_1 \quad = \quad (f[ts]_2 \gg\!= g)[ts]_1 \\
\Leftarrow \; & \quad [\text{ definition of } \gg\!= \text{ applied on the right hand side} \\
& (g[(t \gg\!= f)s]_2)[(t \gg\!= f)s]_1 \quad = \quad (s_2 \mapsto (g[(f[ts]_2)s_2]_2)[(f[ts]_2)s_2]_1)[ts]_1 \\
\Leftarrow \; & \quad [\text{ resolving the } \lambda\text{-expression on the right hand side} \\
& (g[(t \gg\!= f)s]_2)[(t \gg\!= f)s]_1 \quad = \quad (g[(f[ts]_2)[ts]_1]_2)[(f[ts]_2)[ts]_1]_1 \\
\Leftarrow \; & \quad [\text{ observing the } (g[X]_2)[Y]_1 \text{ skeleton on both sides} \\
& (t \gg\!= f)s \quad = \quad (f[ts]_2)[ts]_1 \; .
\end{aligned}$$

This is the definition of the $\gg\!=$ operator after applying the λ-expression.

□

Now let us have a look at some programs written in the imperative programming style offered by the do-notation. To discuss the code, we use imperative style language, like "program" or "assign".

✎ **First example** The code fragment

```
t0 = getVal (do initialise
            )
```

is the shortest program simulating IO we can write so far. It does nothing, however we have to understand this skeleton, which is also used below. The auxiliary functions `getVal` and `initialise` are required in our model:

Since we only model IO operations —i.e., our program does not do real IO yet— we somehow need to get an initial state of the input and output streams. This is done by

```
initialise :: ST ()
initialise = ST(λ_ -> (("foo",""),()))
```

which sets the input stream to `"foo"`, and the output stream to be empty. This "setting the state" is performed by creating a transformation, which ignores its input (hence `λ_ -> ...`) and always returns the same state. Since initialisation shall not return a value, we return `()`.

When our program in do-notation has been executed, we access the state of the program via

```
getVal :: ST a -> ((String,String),a)
getVal (ST p) = (p undefined)
```

which passes some arbitrary state `undefined` to the transformation returned by the do-expression.

So evaluation (using $\leadsto$ for "reduces to") of t_0 reads

$$\begin{array}{ll} & t_0 \\ \leadsto & \texttt{getVal}\ \text{do}\ \{\ \texttt{initialise}\ \} \\ \leadsto & \texttt{getVal initialise} \\ \leadsto & \texttt{getVal}\ (\_ \mapsto ((\ \texttt{"foo"}\ ,\ \texttt{""}\ ), ())) \\ \leadsto & (\_ \mapsto ((\ \texttt{"foo"}\ ,\ \texttt{""}\ ), ()))\ \texttt{undefined} \\ \leadsto & ((\ \texttt{"foo"}\ ,\ \texttt{""}\ ), ()) \end{array}$$

✎ **Second example** Now we start to fill the skeleton given above. Let us just output one character:

```
t1 = getVal (do initialise
                putc ’c’
            )
```

After initialisation, this program writes `c` to the output stream. Setting $t :=$ `initialise` and $f :=$ `putc ’c’`, the evaluation of the program body (i.e., the do-notation) reads

$$\begin{array}{ll} & \text{do}\ \{\ t;\ f\ \} \\ \leadsto & t \gg= (\_ \mapsto \text{do}\ \{\ f\ \}) \\ \leadsto & t \gg= (\_ \mapsto f) \\ \leadsto & \qquad \big[\ \text{definition of} \gg= \text{using Notation 6.5.2} \\ & s \mapsto ((\_ \mapsto f)[ts]_2)[ts]_1 \\ \leadsto & s \mapsto f[ts]_1 \\ \leadsto & \qquad \big[\ t \text{ always maps to the initial state } ((\ \texttt{"foo"}\ ,\ \texttt{""}\ ),()) \\ & s \mapsto f(\ \texttt{"foo"}\ ,\ \texttt{""}\ ) \\ \leadsto & \qquad \big[\ f = \texttt{putc 'c'} \\ & s \mapsto ((\ \texttt{"foo"}\ ,\ \texttt{"c"}\ ), ()) \end{array}$$

So, the do-notation returns a transformation that transforms an arbitrary initial state s into the state (`"foo"` , `"c"`) and the return value (). The `getVal` function binds s to `undefined`, and the *state of the world* after evaluation of the program is returned.

✎ **Third example** This program prints the first input character to the output stream

```
t2 = getVal (do initialise
                x <- getc
                putc x
            )
```

Again, setting t := `initialise` , f := `getc` , g := `putc` and s_0 := ((`"foo"` , `""`), ()), the evaluation of the do-expression reads

$$
\begin{aligned}
& \text{do } \{\ t;\ x \leftarrow f;\ gx\ \} \\
\rightsquigarrow\ & t \gg\!= (\_ \mapsto f \gg\!= (x \mapsto gx)) \\
\rightsquigarrow\ & t \gg\!= (\_ \mapsto f \gg\!= g) \\
\rightsquigarrow\ & s \mapsto ((\_ \mapsto f \gg\!= g)[ts]_2)[ts]_1 \\
\rightsquigarrow\ & \qquad \big[\ \text{since the } \lambda\text{-expression ignores its argument} \\
& s \mapsto (f \gg\!= g))[ts]_1 \\
\rightsquigarrow\ & \qquad \big[\ \text{using the definition of } t \text{ which also ignores its input } s \\
& \_ \mapsto (f \gg\!= g)\ s_0 \\
\rightsquigarrow\ & \_ \mapsto (\quad s \mapsto (g[fs]_2)[fs]_1 \quad)\ s_0 \\
\rightsquigarrow\ & \_ \mapsto (g[fs_0]_2)[fs_0]_1 \\
\rightsquigarrow\ & \qquad \big[\ \text{using } fs_0 = ((\texttt{"oo"}, \texttt{""}), \texttt{'f'}) \\
& \_ \mapsto (g\ \texttt{'f'})\ (\texttt{"oo"}, \texttt{""}) \\
\rightsquigarrow\ & \_ \mapsto ((\texttt{"oo"}, \texttt{"f"}), ())\ .
\end{aligned}
$$

✎ **Fourth example** Since the do-notation works on a list of transformations, and also returns a transformation, we can nest them. This leads to the ability to model *subroutines* in an imperative style.

We define a (recursive) function that prints a string:

```
puts :: String -> ST ()
puts "" = return ()
puts (x:xs) = do putc x
                 puts xs  --recursion
```

and then use it in

```
t3 = getVal (do initialise
                getc
                x <- getc
                puts "2nd char is "  --call puts
                putc x
            )
```

It should be clear how these functions are evaluated from the examples above. Therefore we do not go through this now.

6.6 The IO monad

A closer look at the definition of the `ST` monad reveals, that the pair of strings we used to model input and output streams is not mentioned at all. In fact, the definition of `ST` is a well known method to pass around state information between function calls.

One might argue that these achievements are vile, providing nothing more than syntactic sugar that avoids augmentation of function signatures to explicitly pass a state.

However, this is not entirely true.

The big advantage is, that the monad structure may be some kind of “one-way” object. Neither Definition 6.1.1, nor the definition given in Theorem 6.3.5, require a method to get things back out of the monad. In the last section, we used a function `getVal` to reveal the innards of the monad, but none of the laws and theorems mentioned above guarantees the existence of such a function. The same applies to the function `initialise`.

Recall, that we accepted the structure of objects in a category to be unknown (Definition 3.1.1). If, for example, we define a monad with a functor M, and apply M to an object A with well known structure in our category, we end up with another object MA whose structure might be unknown.

And this is all there is about the IO monad. The Haskell programmer has no clue about its internal structure, and it is not possible to access the *state of the world*. This protects one from doing funny things, like, e.g., make a copy of the *state of the world* and transform both “versions” of the world in different ways.

You may have noticed the similarity between the functions

```
getc :: ST Char
putc :: Char -> ST ()
puts :: String -> ST ()
```

defined above, and the standard IO functions

```
getChar :: IO Char
putChar :: Char -> IO ()
putStr :: String -> IO ()
```

provided by Haskell.

The main difference is that we do not know the internal structure of the IO monad. All we know is how to pass an object in that monad (MA) to a morphism ($f : A \to MB$) using the bind operator, thereby applying f to the “*value*” of the passed object.

Final remarks

This guide explored the theoretical backgrounds of Haskell's IO monad. On the way we have seen functors, natural transformations, and finally monads. All these purely theoretic concepts appeared to have a quite practical correspondence in the Haskell programming language, as emphasized by the according examples.

It should be clear now, how the IO monad is used to pass around the *state of the world*, without allowing the programmer to access it "too much".

Recall the novice Haskell programmer mentioned in the introduction:

> *"I'll just wrap the* `getLine` *in another function which only returns what I really need, maybe convert the users' input to an* `Int` *and return that."*

Now we can explain why the `IO` "thing" will always stick to a function that is somehow involved in doing IO. But with the knowledge about the bind operator, we can also talk "monad-free" functions into working on data returned from IO.

Also, we know why `IO ()` denotes the type of a "void IO procedure", and how the variable assignment notation `x<-...` is translated into λ abstraction.

Bibliography

[1] Maarten Fokkinga. *A Gentle Introduction to Category Theory — the calculational approach*. In *Lecture Notes of the STOP 1992 Summerschool on Constructive Algorithmics*, Part I, pages 1–72. University of Utrecht, Netherlands, September 1992.

[2] Michael Barr, Charles Wells. *Category Theory Lecture Notes for ESSLLI*. In *Lecture Notes for ESSLLI'99*, 1999. `http://www.let.uu.nl/esslli/Courses/barr/barrwells.ps`.

[3] Jeff Newbern. *All About Monads*. `http://www.nomaware.com/monads/html/index.html`.

[4] Theodore Norvell. *Monads for the Working Haskell Programmer — a short tutorial*. Memorial University of Newfoundland. `http://www.engr.mun.ca/~theo/Misc/haskell_and_monads.htm`.

[5] Simon Peyton Jones, et. al. *Haskell 98 Language and Libraries — The Revised Report*. December 2002. `http://haskell.org/definition/`.