Applied Category Theory
Monads and Haskell

Duarte Maia

June 20, 2022

Contents

1 Introduction

In the 1930s, around the same time as Alan Turing published his work on what we nowadays call Turing machines, Alonzo Church published his own formalism for computability, which is called *lambda calculus*. Around this time, it was also proven that these two notions of computability are equivalent. Today, Turing Machines serve as the 'public face' of computability: we say that a function is computable if there exists a Turing machine which computes it, for example. As far as computational models go, Turing machines are very slow and unwieldy, but they match cleanly to our mental model of what computation is: a sequence of instructions which can be followed to turn inputs into outputs.

Lambda calculus does not match our intuition as cleanly, being based on substitution of expressions in other expressions, but surprisingly turns out to be much more well behaved in practical terms. Starting with ALGOL around 1960, some programming languages have included some features originating from lambda calculus, such as first-class and anonymous functions, which will be explained later. This is especially noticeable in the programming language Haskell, whose first version was published in 1989, which at its core is simply a lambda calculus evaluator.

Lambda calculus comes in several shapes, but the kind relevant for Haskell is the so-called *typed lambda calculus*. It is typed in the sense that functions have well-defined domains and codomains, which are known at compilation time, and it is a syntatic error to compose functions whose domains and codomains do not match. In other words, Haskell is *strongly typed*, in opposition to C, in which a function which recieves a floating-point number may instead accept an integer. As a consequence, functions in Haskell can be thought of as forming a category, with the domain and codomain of every function being known at compile time and composition only being allowed between functions of compatible types. This is part of the reason why there is such a strong relation between Haskell and category theory.

Besides being strongly typed, another important feature of Haskell is *purity*. Essentially, functions in Haskell are like mathematical functions, in the sense that the same function called with the same input will always return the same output, depending on nothing other than its input and doing nothing other than returning its output. This has wide and far-reaching consequences, both in terms of performance (since there is no implicit input or output, the compiler is free to rearrange the order of computations, or even have them done in parallel) and in terms of code maintainability (when changing pre-existing code, the developer does not need to have knowledge of any other part of the system, making it easier for several developers to work in isolation), but also means that a wide array of tools programmers rely on are inviabilized. For example, simulations of physical objects are often done by defining global variables representing the x and y coordinates of the objects at play, their velocities, etc., and then defining a function which advances the global state in time, calling that function on repeat until the desired amount of time has elapsed:

```
x = 0
```

```
y = 0
vx = 0
vy = 0
t = 0
while t < 1000:
    vx, vy = force(x,y)
    x = x + vx
    y = y + vy
print("At t = 1000, the object is at " + (x,y))
```

This is impossible in Haskell, as not only can global variables never be changed, but even if they could, this could not be done inside of a function, because functions cannot modify anything, only return an output. This requires that the programmer change their perspective: in this case, instead of defining a function which *modifies* a few global variables, one instead defines a function which receives as input the state of the physical system at time t and returns the state at time $t + \mathrm{d}t$.

This is just an example of how purity requires a complete rearrangement of the way one thinks about a problem. In this case a solution was quickly found, but how does one receive input from the user? How does one write to a file? How does one implement an algorithm which relies on mutable state, e.g. marking the nodes of a graph as they are traversed? And most importantly, how does one do all of the above without making the code completely unreadable? This is a place where category theory has found surprising application, by providing powerful abstractions which pick up the slack left by the lack of global state and mutability, and at the same time giving Haskell unprecedented amounts of type safety and generality. In the same way that many seemingly unrelated phenomena were found by category theorists to be expressions of a single phenomenon (for example, adjunctions, limits or colimits), computer theorists have found abstractions which encompass many programming strategies which on the surface seem to be completely distinct.

In this essay, I hope to convince the reader that there are very real mathematical concepts underlying many tools used by in the day-to-day of a Haskell programmer. Furthermore, I also want to get across that these concepts are *useful*, allowing for cleaner and more understandable code, at least so long as the programmer is familiar with the underlying math!

2 A Crash Course in Haskell

The following examples are all valid Haskell code, which was implemented and executed in GHCi, the Glasgow Haskell Compiler interpreter, which is the *de facto* implementation of Haskell and can be downloaded here: `https://www.haskell.org/downloads/`. The reader can also try Haskell online without downloading anything on `https://tryhaskell.org/`.

(A disclaimer: If the reader tries the examples below, some of them might yield output which is different from the one shown, in particular in regards to types of functions. This is because some functions are more general than we present them as, but an

explanation of this generality must wait until we discuss typeclasses in 3.2. In any case, the reader is assured that where the output of code in this essay is wrong, it is at most a particular case of the truth.)

As mentioned in the introduction, Haskell is at its core a lambda calculus evaluator, so we begin by introducing the most basic concept to lambda calculus: the lambda expression.

2.1 Lambda expressions

A lambda expression is something similar to the notation $a \mapsto f(a)$ for defining mathematical functions without giving them a name. For example, if I am talking about the function which squares a given number, I may write it as $a \mapsto a^2$. A lambda expression is the same, but it is written using a slightly different notation: the square function would be written as $\lambda a.a^2$. The λ marks the beginning of the function, then the argument is given a name, and then an expression is written, in which the only free variable is the one in the argument.

A lambda expression can be applied as a function. For example, we could write $(\lambda a.a^2)2$; in lambda calculus, function application is written without parentheses. Whenever a lambda expression is juxtaposed to the left of another expression, it should be understood as applying the function defined by the lambda to the argument given by the expression. The above example can be written in Haskell as

```
(\a -> a^2) 2
output: 4
```

Note the changes in syntax: the lambda has been replaced by a backslash (it's like a lambda with one leg cut off), and the period has been replaced by an arrow. Also, note that the expression `(\a -> a^2)` represents a Haskell function, but it has not been given a name. For this reason, lambda expressions in Haskell are also called *anonymous functions*.

The way that this works under the hood is that the Haskell interpreter is doing something called β-reduction. Formally, given an expression fx where f is a lambda expression of the form $(\lambda a.M)$, the Haskell interpreter rewrites it as $M[a = x]$, i.e. the expression M where every instance of the variable a has been replaced by x. In the above example, `(\a -> a^2) 2` is β-reduced to `2^2`, which is then evaluated to 4.

The reason why this idea is so powerful is the following: *in Haskell and in lambda calculus, lambda expressions are first class objects.* What this means is that functions can be stored in variables and given as arguments to other functions. For example, the following lambda expression takes as an argument another lambda expression, and returns its 'square'. Mathematically, we would usually write it as $f \mapsto f \circ f$.

```
\f -> (\x -> f (f x))
```

2.2 Variables, Syntactic Sugar and Recursion

As we have seen, functions in Haskell do not need to be named, but that does not mean that they should not. If we want to use a function more than once, we do not want to rewrite its definition every time we do. Therefore, Haskell lets us name expressions so that we may reuse them later. For example,

```
f = \a -> a^2
x = 2
f x
output: 4
```

It is inconvenient to write lambda expressions whenever we want to define a function, so Haskell allows us to define functions using more convenient and slightly more standard notation

```
f a = a^2
```

though it should be understood that beneath this expression is a definition via a lambda expression. Haskell contains many of these alternative notations, in which a common but cumbersome expression is given an alternative notation, which is unfurled to its underlying meaning before being passed to the Haskell interpreter. These kind of syntactic expansion rules are collectively referred to as *syntactic sugar*.

Another piece of syntactic sugar without which writing any program would be almost impossible is recursion. As an example, consider the following implementation of the factorial function

```
factorial x = if x == 0 then 1 else x * factorial (x-1)
```

This is a valid definition in Haskell, but it is not easy to write it as a lambda expression, because the function is being called inside itself. If we were to naïvely write `factorial` as an anonymous lambda expression, we would get

```
\x -> if x == 0 then 1 else x * factorial (x-1),
```

which would require us to expand the definition of factorial inside the expression (since it is anonymous we cannot refer to it by name), obtaining a more complex expression which itself has a factorial, and so on. This tricky problem (expressing recursion in lambda calculus) has a solution via the so-called Y combinator, which is a particular lambda expression which can be used to define the factorial as above. Fortunately, as a Haskell programmer we don't need to know about it, because recursive expressions are rewritten using the Y combinator under the hood.

2.3 Types and Morphisms

As we said in the introduction, Haskell is strongly typed: every function has a domain and a codomain. This makes Haskell a category[1], which is usually called Hask, in which the types (examples: `Int`, `Char`, `String`) are the objects and the functions (lambda expressions) are the morphisms.

[1] This is not exactly true; see [3].

That said, what is the domain and codomain of the lambda expression `\a -> a^2`? We have evaluated it on the number 2, which is an integer (i.e. of type `Int`) and gotten an integer back, but we could also have applied it to the floating-point number 2.1 (of type `Float`). We'll get back to this ambiguity later, but to assuage our concerns we can add type annotations explicitly using the following syntax

```
f :: Int -> Int
f a = a^2
```

This means that f is a morphism with domain `Int` and codomain `Int`; mathematically, $f \in \mathrm{Hask}(\mathrm{Int}, \mathrm{Int})$. If we try to evaluate it on a non-integer such as 2.1, we get an error:[2]

```
f 2.1
<interactive>:1:3: error:
    • Couldn't match expected type 'Int' with actual type 'Float'
    • In the first argument of 'f', namely '2.1'
      In the expression: f 2.1
      In an equation for 'it': it = f 2.1
```

Haskell has a wide array of so-called primitive types, like `Int`, `Char` and so on, but it also allows us to make complex types out of simpler ones. We have already seen one: the humble arrow!

The sequence of characters `->` has the role usually taken by Hom or C in categorical texts: we write `a -> b` to denote what we would usually call $\mathrm{Hask}(a, b)$. This is in itself a Haskell type. In other words, *the* Hask *category has exponential objects*. This is the first taste of categorical language in Haskell.

2.4 Currying and Composition

So far, every function we have seen takes only one argument. This is also the case in lambda calculus: every lambda has exactly one variable. It also happens in, for example, the category of Sets: an arrow $A \to B$ takes as argument exactly one element of A. Of course, we emulate our functions taking multiple arguments by using cartesian products: a function f which receives two real numbers and returns another is denoted as

$$f \colon \mathbb{R} \times \mathbb{R} \to \mathbb{R}. \tag{1}$$

A similar process can be done in Haskell, so that a function that receives, say, two integers, could have type `(Int,Int) -> Int`. The comma `(,)` denotes the categorical product in Hask, and is another example of a type constructor. However, functions receiving multiple arguments are usually handled another way.

The type of a pre-existing function `f` can be found by writing `:t f` in GHCi. For example, using `f = (+)`, i.e. the predefined function which adds two numbers, we get[3]

[2] Strictly speaking, this isn't the error you'll actually get. You'll get something like `No instance for (Fractional Int) arising from the literal '2.1'`, which is Haskell trying a little harder to make 2.1 into an integer.

[3] Again, the actual output isn't exactly the one shown, as addition is defined more generally than on the integers.

```
:t (+)
output: (+) :: Int -> Int -> Int
```

What gives? This expression will be made clearer if we introduce parentheses. Haskell associates the `->` operator to the right, so the above expression should be read as

```
(+) :: Int -> (Int -> Int)
```

Aha! So what's happening here, in categorical terms, is simply that

$$(+) \in \mathrm{Hask}(\mathrm{Int}, \mathrm{Hask}(\mathrm{Int}, \mathrm{Int})). \tag{2}$$

Recall that in Sets we have an adjunction between the functors

$$F(a) = a \times b, \quad G(c) = \mathrm{Hom}(b, c). \tag{3}$$

In particular, $F \dashv G$, and the natural isomorphism given by the adjunction is defined as

$$\begin{aligned} \phi\colon \mathrm{Hom}(a \times b, c) &\to \mathrm{Hom}(a, \mathrm{Hom}(b, c)) \\ f &\mapsto (x \mapsto (y \mapsto f(x, y))). \end{aligned} \tag{4}$$

In Haskell, this isomorphism is called `curry`. We say functions in Haskell are *curried*. The nomenclature is in honor of Haskell Curry, an important logician. We can inspect the type of this function in GHCi:

```
:t curry
output: curry :: ((a, b) -> c) -> a -> b -> c
```

This matches our expectation, but remember that parentheses are implicit on the right-hand side. Note the use of the so-called *type variables*: any 'type' whose name is a lower-case letter denotes a type variable, and may be substituted by any type. Therefore, `curry` isn't a function in and of itself: it is a family of functions, parametrized by three objects in Hask: a, b and c. This is similar to how the usual isomorphism in adjunctions is parametrized by a pair of objects, as in ϕ_{xy}. In Haskell, the parametrization is implicit: the compiler keeps the definition as ambiguous as possible at every moment, and deduces what a, b and c should be from context when necessary.

Surprisingly, currying (and its inverse, `uncurry`) isn't used very often. Functions are almost always left curried, because this allows for a technique called partial application. We won't go into details, but for example expressions such as `(\a -> f 2 a)` are usually shortened to `(f 2)`.

Finally, let's talk about the most important categorical concept: function composition. It is a very surprising fact that almost no modern languages have a composition operator, but Haskell does. It is denoted with a single dot, as in `f . g`, as it is the closest a standard keyboard gets to the usual notation of composition, except perhaps for the hideous notation `f o g`. Unsurprisingly, the composition has type given by

```
:t (.)
output: (.) :: (b -> c) -> (a -> b) -> a -> c
```

Again, note the usage of type variables: the types written in lower case are a stand in for any type in Hask.

3 Elementary Categorical Concepts

3.1 A Few Limits and Colimits

We have already seen a few examples of categorical concepts cropping up in Haskell. The types and morphisms form a category called Hask (kind of, see [3]), the arrow operator `->` is the categorical exponent, the comma operator `(,)` is the categorical product, and these two functors (though we haven't realized them as functors yet) are adjoint to each other, with the bijection between hom-sets being given by `curry`.

Another common categorical construction is the coproduct. In Haskell, this is realized by using the `Either` type constructor. The coproduct of two types a and b is denoted `Either a b`, and the coproduct diagram is the following, with the inclusions being denoted by `Left` and `Right`.

$$\texttt{a} \xrightarrow{\texttt{Left}} \texttt{Either a b} \xleftarrow{\texttt{Right}} \texttt{b} \tag{5}$$

We also present the product diagram of `(a,b)`.

$$\texttt{a} \xleftarrow{\texttt{fst}} \texttt{(a,b)} \xrightarrow{\texttt{snd}} \texttt{b} \tag{6}$$

As an example, `Either` is often used for error handling: a `Right` value represents a successful computation, while `Left` represents an error. For example, we can define a safe division:

```
safediv :: Rational -> Rational -> Either String Rational
safediv x y = if y == 0 then Left "Error! Division by zero."
              else Right (x/y)

safediv 2 3
output: Right (2 % 3)

safediv 2 0
output: Left "Error! Division by zero."
```

Note the Haskell notation for fractions: `2 % 3` means the rational number $2/3$.

We may define a function on `Either a b` using *pattern matching*. In categorical terms, if i_1 and i_2 are the inclusions in the coproduct, we define $f\colon a \amalg b \to c$ by defining $f(i_1(x))$ and $f(i_2(x))$, i.e. by applying the universal property of the coproduct.

```
safeadd1 :: Either String Rational -> Either String Rational
safeadd1 (Right y) = Right (y+1)
safeadd1 (Left err) = Left ("Woah there! You have an error: " ++ err)


safeadd1 (safediv 2 3)
output: Right (5 % 3)

safeadd1 (safediv 2 0)
output: Left "Woah there! You have an error: Error! Division by zero."
```

Besides binary products and coproducts, Haskell also has empty (co)products, i.e. a final and an initial object.

The final object is the type `()`, whose only element is the (unique) zero-uple, also denoted `()`. The only function `a -> ()` is defined by `\x -> ()`.

The initial object is the type `Void`. There is no object of type `Void`, and the unique function `Void -> a` is called `absurd`.

A particularly simple approach to error handling is to use `()` as the error type, i.e. considering `Either () a`, where `Left ()` gives us the information that an error has occurred, while telling us nothing about what it was. Categorically, this corresponds to taking the coproduct with the final object, that is, 'adding one element'. Haskell actually has another type constructor, called `Maybe`, which has the same semantics.

Let `a` be a type. We define `Maybe a` as the type whose elements are either of the form `Nothing` or `Just x`, where `x :: a` (i.e. x has type a). In other words, we have the following coproduct diagram.

$$\texttt{a} \xrightarrow{\texttt{Just}} \texttt{Maybe a} \xleftarrow{\texttt{cn}} \texttt{()} \tag{7}$$

where `cn = \x -> Nothing`. A function defined on `Maybe a` is defined using the universal property, similarly to the coproduct.

```
safeadd1m :: Maybe Rational -> Maybe Rational
safeadd1m (Just x) = Just (x+1)
safeadd1m Nothing = Nothing

safeadd1m (Just 1)
output: Just (2 % 1)

safeadd1m Nothing
output: Nothing
```

3.2 Functors and Typeclasses

The type constructor `Maybe` is a particularly simple example, because it takes one type as an argument and returns another. In other words, it is a map from the objects of Hask to the objects of Hask. This suggests that it could be made into a functor: all we need is to define an adequate map of type `(a -> b) -> (Maybe a -> Maybe b)`. This map exists, and it is called `fmap`.

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

Note the use of currying. It looks like we are defining a function of two arguments (one of type `a -> b` and the other `Maybe a`), but thanks to currying, this is the same as a function of one argument which returns a function. It is possible to make a definition that maps closer to our intuitive notion of 'a function which takes a function as input and returns another function', but it requires introducing additional syntax. In the interest of keeping this essay short we will keep ourselves in a relatively minimal subset of the language.

Let's go back to the definition of `fmap`. By inspection, it is easy to deduce the functoriality conditions, i.e. `fmap (f . g) = fmap f . fmap g` and `fmap id = id` (where `id`

is the identity map). Furthermore, this agrees with the categorical definition of the functor $(1 \amalg -)$ given by taking the coproduct with the final object, so this functor in particular is well represented in Haskell. This is not an isolated occurrence, but to explain how functors are represented generally in Haskell we need to talk about *typeclasses*.

In Haskell, a typeclass corresponds to additional structure that a type or type constructor can have. For example, for a given type `a`, Haskell may or may not know how to check that two elements of `a` are equal. It certainly knows how to check that two integers `Int` are the same, but it cannot check that two functions `Int -> Int` are the same: this is an undecidable problem! Therefore, the binary function `==` will be defined on `Int` but not on `Int -> Int`.

Let us look at the type of `==`. If we check its type in GHCi using `:t`, we obtain

```
:t (==)
output: (==) :: Eq a => a -> a -> Bool
```

So what's happening here? As the reader might have already suspected, `==` is a binary function which returns a boolean value: that's what `a -> a -> Bool` means. However, we have that additional term `Eq a =>`. What that is saying is that the type signature that follows only applies when `a` is a type which satisfies the predicate `Eq`. We say that such types are *in the typeclass `Eq`*. Most predefined types are in this typeclass, so the equality operator can be used on `Int`, `Float`, `Char`, etc., but not on `Int -> Int`.

In short, to each typeclass there are a few functions which need to be defined in order for a type to be part of the typeclass: for a type `mytype` to be in `Eq`, we need to define `(==) :: mytype -> mytype -> Bool`. For it to be in `Ord` (totally ordered types) we need to define `(<=) :: mytype -> mytype -> Bool`, the 'less than or equal operator'. To tell Haskell that `Maybe Int` is in the `Eq` typeclass, we would use the notation

```
instance Eq (Maybe Int) where
    Nothing == Nothing = True
    Nothing == Just x = False
    Just x == Nothing = False
    Just x == Just y = x == y
```

to define equality. Fortunately, Haskell has built-in mechanisms to deduce that, for example, if `a` is in the `Eq` typeclass, so is `Maybe a`, so we seldom need to explicitly tell it how to define equality for a given type.

What does this have to do with functors? Well, just like how a type can be in a given typeclass, there are also typeclasses for *type constructors*. In particular, given a type constructor of kind `* -> *` (this means that it requires one type as argument and returns another type), it may or may not be in the `Functor` typeclass, which is defined as

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

In other words, for a type constructor (map from objects in Hask to objects in Hask) to be in the `Functor` typeclass, we need to define how it acts on morphisms in Hask. It is not (and can not be) enforced by the language that a given implementation of `fmap` is functorial, but it is generally understood that one should not implement such

a pathological version of `fmap` which does not satisfy them, especially as the compiler might perform optimizations which rely on these laws.

Now that we know about the `Functor` typeclass, we may list a few of its instances, most of which are familiar from category theory.

- We've already seen the `Maybe` functor, which is the coproduct with a one-element set,

- More generally, for a given type `a`, the coproduct with `a` is a functor. In other words, `Either a` is in the `Functor` typeclass. Note that this is *not* to say that `Either` is in the `Functor` typeclass: `Either` itself is, categorically speaking, a *bifunctor* in Hask, and therefore in another typeclass called `Bifunctor`. Of course, technically speaking, this is just a functor $\mathrm{Hask} \times \mathrm{Hask} \to \mathrm{Hask}$, but everything we do is in the Hask category, which is why bifunctors get a different treatment. This also means that the `Functor` typeclass could perhaps be more aptly renamed to `Endofunctor`.

  A few more words on `Either`: since we see `Either a` as a functor, the instantiation of `fmap` to `Either a b` has the type

  ```
  fmap :: (b -> c) -> (Either a b) -> (Either a c)
  ```

  In other words, `fmap f` will apply `f` to a `Right` value, and do nothing to a `Left` value. This is in keeping with the use of `Either` as an error handling tool: `fmap f` means 'if we have a valid value, apply `f`. Otherwise, preserve the error message'.

- Another important functor in category theory is the hom-functor $\mathrm{Hom}(x, -)$. In Haskell, this is denoted as `(->) x`; it should be read as partially applying the type constructor `(->)`, which takes two arguments, to `x`, becoming a type constructor that takes a type `a` and returns the type `x -> a`. Applied to morphisms, this functor has a very simple definition

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

  If one is in a particularly terse mood, they could simply write `fmap f = f .` or even `fmap = (.)`. This style of programming, in which one omits the arguments of a function as much as possible, is called *point-free style*.

  The hom-functor as we've described has a contravariant sister: the contravariant hom-functor $\mathrm{Hom}(-, x)$. This functor is denoted `Op x`, and is not a member of the `Functor` typeclass, being instead in the `Contravariant` typeclass. The function corresponding to `fmap` is called `contramap`.

  Finally, we can see `->` as a functor $\mathrm{Hask}^{\mathrm{op}} \times \mathrm{Hask} \to \mathrm{Hask}$. The nomenclature used in Haskell for such an object is a `Profunctor`. This does not completely agree with the mathematical nomenclature: in category theory, a profunctor is a functor $D^{\mathrm{op}} \times C \to \mathrm{Sets}$. However, the category of Haskell types can be approximated by the category of sets (identify a type with the set of its elements), so the nomenclature `Profunctor` makes sense.

- Let `a` be a type. Then, there exists another type, called `[a]`, whose elements are *lists of* `a`. In other words, `[]` is a map that turns a type into another type.

  Now, let `f :: a -> b`. Then, there is an obvious way to turn a list of `a` into a list of `b`: simply apply `f` to every element of the list. This operation is often referred to in programming circles as *mapping* f, and can be found in Python with the syntax `map(f, list)`, in Mathematica with the syntax `Map[f, list]` and in Haskell with the syntax `map f list`. It is easy to check that this operation is functorial, so the list functor is actually one of the most widespread examples of functors in programming. In fact, the notion of functor in Haskell actually began as a generalization of the list functor: see [4][4], where the concept of typeclass is first introduced precisely for this task. The author also uses `map` to denote application of a functor to a morphism, which was presumably changed to `fmap` to avoid name collision.

- Lots of algorithms in computer science rely on some kind of global state in order to keep track of what has already been done, what has already been visited, etc. In Haskell, this can be done by using the product `(s,a)`.

  To be more precise: let `s` be a type, which represents the global state necessary for our program. For example, where in an imperative language we would have a variable for an integer and another for a list of characters, we might set `s = (Int , [Char])`. Then, `(,) s` corresponds to the functor $(s \times -)$; `(,) s a` is a synonym for `(s,a)`. An element of type `(s,a)` can be seen as an element of `a` together with some 'background state' of type `s`. Mapping a morphism `f :: a -> b` corresponds to modifying the element of type `a` without modifying the state, yielding a function `fmap f :: (s,a) -> (s,b)`.

There are many more examples of functors in the Haskell ecosystem. In particular, most types of containers (lists, trees, sets, databases, etc) are functors, as well as type constructors which represent computations, such as `(->) x`, as well as others we will see shortly, such as `IO`, and even random number generation.

3.3 Natural Transformations and Free Theorems

Let `f` and `g` be two `Functor`s. Then, a natural transformation between `f` and `g` would be a function of type

```
eta :: f a -> g a
```

where the only free variable is `a`. We mentioned before the concept of type variables, where in the signature of a function we write lower-case letters instead of types, and the compiler will fill those letters in for whatever type makes sense at compile time. Another way to think of it is that our function is actually a function-making machine, which recieves a type as input (in this case `a`) and returns a function of appropriate

[4]This article is freely available at `https://www.cs.tufts.edu/comp/150GIT/archive/mark-jones/fpca93.pdf`.

type. This matches up with the usual notation for natural transformations η_x, where the subscript represents an object of the category, in this case a type. Sometimes, this type argument is written explicitly using the notation[5]

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

Let us look at an example. Consider the following natural transformation from `Maybe` to `[]`

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

It is easy to verify that this is indeed natural, i.e. that

```
maybeToList . (fmap f) = (fmap f) . maybeToList
```
(8)

Note that in this expression `fmap` is used for two different functors. Now, it would be reasonable to ask whether it would be possible to define functions `f a -> g a` which are not natural transformations, and the very surprising answer turns out to be *no*. This is in contrast to, for example, `fmap`, for which it is not very difficult to define instances of the `Functor` typeclass which do not satisfy the functor laws.

This has to do with something called *parametric polymorphism*. There is a very influential paper titled 'Theorems for free!' [11] by Philip Wadler, in which a method is explained for obtaining theorems about a function in typed lambda calculus from nothing but its type signature. The intuitive reason behind this very unexpected result is that in lambda calculus and in Haskell a polymorphic function (i.e. one which has more than one type at once) must be defined with the same expression for all its instantiations. For example, consider a function `f :: a -> [a]`. The expression must for `f` must be the same whether we are considering `f :: Int -> [Int]` or `f :: () -> [()]`. As a consequence, `f` cannot do anything to its argument, because it has no guarantees that anything it can do to it is valid. Therefore, there is a very limited number of functions `f` may be: it may be the constant returning the empty list, it may be the function `\x -> [x]`, it may be `\x -> [x,x]`, etc., but no more. This is an example of how the type signature of a polymorphic function can tell us a lot about the function itself, yielding the so-called theorems for free. In the case of functions `eta :: f a -> g a`, with `f` and `g` functors, the free theorem obtained from the type signature is precisely the naturality of `eta`. [8]

3.4 Hask is not Finitely Complete (Probably)

In 3.1 we discussed products and coproducts in Haskell. We showed that Hask has initial and final objects and is closed under binary products and coproducts. Consequently, Hask has all finite products and coproducts, so it makes sense to ask whether it has

[5]Don't be confused with the name `forall`. Superficially it can be understood as the universal quantifier, but from the perspective of the compiler it reads as '`eta` recieves a type `a` and returns a function of type `f a -> g a`'. The reason behind the nomenclature `forall` has to do with an important theorem in logic called the Curry-Howard isomorphism, which is far beyond the scope of this essay.

(co)equalizers, as if it did it would have all finite (co)limits. I have not found any definite source proving that the answer is negative, but this Stack Overflow answer strongly suggests that the answer is no: `https://stackoverflow.com/a/15113919/2997964`. In any case, the fact that after over 30 years of categorical language and methods being applied to the language there is still no mechanism, built in or otherwise, for constructing (co)equalizers strongly suggests that such a problem is, if solvable, very difficult.

4 A Kleisli Category: The Writer Monad

The reader might recognize the names 'monad' and 'Kleisli category' from the mathematical terminology, and indeed the concepts we are about to discuss coincide with these mathematical notions. However, since they have not yet been explained, the reader should see these as mere names; an explanation will have to wait until section 6.

4.1 The Motivation

Suppose that one is implementing a complicated algorithm and wishes to keep a log of what the algorithm is doing for debugging purposes. Then, classically, one keeps a global 'log' variable, to which one writes as the algorithm proceeds. In pseudocode:

```
log = ""
def factorial(n):
    let output = 1
    log.append("We start with 1. ")
    for i = 1 ... n:
        log.append("We multiply by " ++ i ++ ". ")
        output = output * i
        log.append("We get " ++ output ++ ". ")
    log.append("This is the final result.")
    return output

factorial(3)
output: 6
log: We start with 1. We multiply by 1. We get 1. We multiply by 2. We get 2. We
    multiply by 3. We get 6. This is the final result.
```

In Haskell, this could be dealt with by having our `factorial` function return a pair `(String, Int)`, but this solution quickly gets unwieldy when considering complex systems. For example, function composition can no longer be done via the `(.)` operator. Indeed, composition of functions with an attached log becomes a complicated endeavor: if the function `f` does some computation and returns a log of what it did, and the function `g` wants to call `f` and compose its own logs with the ones from `f`, one needs to do ugly constructions to separate the logs of `f` from the output, do the computations, and return the logs mixed with the result. This is doable but cumbersome, as the following example shows.

```
logAdd1 :: Int -> (String, Int)
logAdd1 x = ("Added 1", x+1)
```

```
logMul2 :: Int -> (String, Int)
logMul2 x = ("Multiplied by 2", 2*x)

log2nplus1 :: Int -> (String, Int)
log2nplus1 x = let (logmul2, xmul2) = logMul2 x in
               let (logadd1, result) = logAdd1 xmul2 in
               (logmul2 ++ logadd1, result)
```

Kleisli categories offer a solution to this issue.

4.2 The Solution

Let us begin by recapping the problem. We have functions which, besides computing some value, return a log of the computation. In other words, they are of type

```
f :: a -> (String, b)
g :: b -> (String, c)
```

We want to compose `f` and `g` and obtain a composite log, but clearly the expression `g . f` does not make sense. In order to deal with logging functions in a reasonable way we need to change the notion of composition, and hence the category at hand.

Definition 1. Let W be the category such that:

- The objects of W coincide with the objects of Hask,
- Let $f\colon a \twoheadrightarrow b$ mean that f is a morphism in W. We say $f\colon a \twoheadrightarrow b$ if f is a map (in Hask) of type $f\colon a \to (\mathrm{String}, b)$. In other words,

$$W(a, b) := \mathrm{Hask}(a, (\mathrm{String}, b)). \tag{9}$$

- The composition of two maps $f\colon a \twoheadrightarrow b$ and $g\colon b \twoheadrightarrow c$ is given by

```
h :: a -> (String, c)
h x = let (logg, y) = g x in
      let (logf, z) = f y in
      (logg ++ logf, z)
```

It is clear that W is a category: composition is associative because string concatenation is associative, and the identities on W are simply given by

```
idW :: a -> (String, a)
idW x = ("", x)
```

In fact, a little thought will show that the construction of W can be generalized, by replacing `String` with any type which has an associative binary operation and a null element. In other words, we need a *monoid*.

Monoids in Haskell are represented by the `Monoid` typeclass. For a type `w` to be in the `Monoid` typeclass, there must exist a constant

```
mempty :: w
```

and a binary function

```
(<>) :: w -> w -> w
```

satisfying the axioms for a monoid (associativity of `<>`, etc). Like in the case for functors, the language does not enforce the monoid axioms: it is the responsibility of the programmer, when defining a new monoid, to ensure that they hold.

That said, we can now define the so-called *Kleisli category for the* `Writer` *monad*:

Definition 2. Let `w` be a type in the `Monoid` typeclass. We define *the Kleisli category for the monad* `Writer w`, denoted W, as follows.

- The objects of W coincide with the objects of Hask,
- A morphism $f\colon a \twoheadrightarrow b$, with a and b types, is a map (in Hask) of type $f\colon a \to (w, b)$. In other words,
$$W(a, b) = \mathrm{Hask}(a, (w, b)). \tag{10}$$
- The composition of two maps $f\colon a \twoheadrightarrow b$ and $g\colon b \twoheadrightarrow c$ is given by

```
h :: a -> (w, c)
h x = let (logg, y) = g x in
      let (logf, z) = f y in
      (logg <> logf, z)
```

In Haskell, the `Writer` monad is already present in the `Control.Monad.Writer` package. Instead of `(w,a)`, for historical reasons the notation `Writer w a` is used, so a Kleisli arrow would actually be of type

```
f :: a -> Writer w b
```

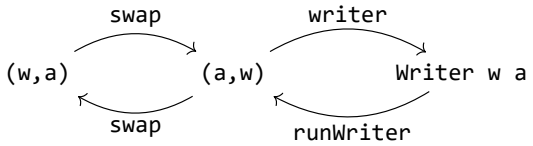
and composition in the Kleisli category is given by the so-called 'fish operator'

```
(<=<) :: (b -> Writer w c) -> (a -> Writer w b) -> a -> Writer w c
```

where `f <=< g` is implemented as the `h` above.

Until now, we have been using pairs `(w,a)` to store data and logs. However, the `Writer` monad uses instead the type `Writer w a`. This is a distinct type, but it is isomorphic to `(w,a)`, and the isomorphism is given by a function called `writer`.

Well, actually, for historical reasons, the argument of `writer` actually has the product swapped. More concretely, `writer` is a function of type `(a,w) -> Writer w a`. Obviously this is of no real consequence, as `(a,w)` is isomorphic to `(w,a)`. The following diagram shows the isomorphisms between the three types `(w,a)`, `(a,w)`, and `Writer w a`.



(11)

4.3 Examples

Factorial The following function `factorial` computes the factorial of its argument and logs the products computed on the way.

```
logProduct :: Int -> Int -> Writer String Int
logProduct x y = writer (x*y, "Compute " ++ show x ++ "*" ++ show y ++ ". ")

factorial :: Int -> Writer String Int
factorial n = if n == 0 then
                  writer (1, "0! is 1. ")
              else
                  ((logProduct n) <=< factorial) (n-1)

runWriter (factorial 4)
output: (24,"0! is 1. Compute 1*1. Compute 2*1. Compute 3*2. Compute 4*6. ")
```

Other Monoids Instead of using the `String` monoid, one could instead use `[String]`, with the operation of list concatenation. This could be useful for preventing weird formatting (such as the space at the end in the previous example) or to make it clearer where one message ends and the other begins. We present a version of the `factorial` function above using this method; we highlight in blue the significant changes.

```
logProduct :: Int -> Int -> Writer [String] Int
logProduct x y = writer (x*y,  ["Product"])

factorial :: Int -> Writer [String] Int
factorial n = if n == 0 then
                  writer (1, ["0! is 1"])
              else
                  ((logProduct n) <=< factorial) (n-1)

runWriter (factorial 4)
output: (24,["0! is 1","Product","Product","Product","Product"])
```

If for some reason we want to get our logging in reverse order, we can use the `Dual` type constructor. If `w` is a monoid, `Dual w` is essentially the same monoid, but the arguments of `<>` are swapped. The function `Dual` (not to be confused with the type constructor) is the identity `w -> Dual w`, and `getDual` is its inverse.

```
logProduct :: Int -> Int -> Writer (Dual [String]) Int
logProduct x y = writer (x*y,  Dual ["Product"])

factorial :: Int -> Writer (Dual [String]) Int
factorial n = if n == 0 then
                  writer (1, Dual ["0! is 1"])
              else
                  ((logProduct n) <=< factorial) (n-1)

runWriter (factorial 4)
output: (24,Dual {getDual = ["Product","Product","Product","Product","0! is 1"]})
```

Separation of Concerns The `writer` function does two things at the same time: it records the result of a computation, and it writes a log. However, it is possible to do each of these separately.

First observe that, as we've seen, `Writer w a` is effectively the same as `(w,a)`. In other words we can identify `Writer w` with `(,) w`, making `Writer w` a functor, and so there exists an instance of `fmap`:

```
fmap :: (a -> b) -> (Writer w a -> Writer w b)
```

We can interpret this instantiation of `fmap` in practical terms: if `f :: a -> b` then `fmap f :: Writer w a -> Writer w b` is the function which takes an `a` together with a log, and applies `f` to the `a` without modifying the log. In other words, *`fmap f` computes `f` without logging anything.*

Another (orthogonal) way to interpret the expression 'do computation without logging' is to take a value of type `a` and turn it into one of type `Writer w a`, by simply attaching an empty log to the value. This is actually the identity arrow in the Kleisli category, and it is called return:

```
return :: Monoid w => a -> Writer w a
return x = (mempty, x)
```

Note the identity:

$$\texttt{(return <=< f) = (f <=< return) = f} \tag{12}$$

Now, let us discuss how we can print to the log without doing any computation. To this effect, consider the following function

```
say :: Monoid w => (a -> w) -> a -> Writer w a
say tellfunc x = writer (x, tellfunc x)
```

This function is not part of the standard Haskell library; Haskell has its own way to log without computing, which would require introducing additional notation. In any case, with `fmap` and `say` we may rewrite our factorial function more neatly.[6]

```
logProduct :: Int -> Int -> Writer [String] Int
logProduct = curry (
                say (\res -> ["Output " ++ show res])
                <=< fmap (uncurry (*))
                . say (\p -> ["Product " ++ show p])
             )

factorial :: Int -> Writer [String] Int
factorial n = if n == 0 then
                 say (\x -> ["0! is 1"]) 1
              else
                 ((logProduct n) <=< factorial) (n-1)

runWriter (factorial 4)
output: (24,["0! is 1","Product (1,1)","Output 1","Product (2,1)","Output 2","
    Product (3,2)","Output 6","Product (4,6)","Output 24"])
```

[6] Of course, neatness is in the eye of the beholder.

Counting Operations We will now use the `Writer` monad to compare implementations of the Fibonacci sequence F_n, $n \geq 0$. To measure roughly the efficiency of such an algorithm, we will count how many times we perform sums of the form $F_{n-1} + F_n$.

We will use the `Writer` monad with the `Sum Int` monoid, which is the integers with addition (as opposed to `Product Int`, which would be the integers with multiplication). Mind the injection `Sum :: Int -> Sum Int` and its inverse `getSum :: Sum Int -> Int`).

In order to shorten the code, we define the alias `WSII` for `Writer (Sum Int) Int`. The code below is not very idiomatic, in particular the function `logAdd`: it is usually bad form to work directly with the logs, being better to let the fish operator `<=<` do the compositions itself. However, the tools that let us write `logAdd` in a more idiomatic fashion will have to wait until we talk about monads in general; see section 6 below.

```
type WSII = Writer (Sum Int) Int

logAdd :: WSII -> WSII -> WSII
logAdd x y = let (xval, xlog) = runWriter x in
             let (yval, ylog) = runWriter y in
               writer (xval + yval, xlog <> ylog <> Sum 1)

naiveFib :: Int -> WSII
naiveFib n = if n <= 1 then
                 writer (n, Sum 0)
             else
                 logAdd (naiveFib (n-1)) (naiveFib (n-2))

optFib :: Int -> WSII
optFib n = aux n 1 0 1

aux :: Int -> Int -> Int -> Int -> WSII
aux n iter prev now = if iter == n then
                       return now
                      else
                       ((aux n (iter+1) now)<=<(logAdd (return now) . return)) prev

--

runWriter (naiveFib 10)
output: (55,Sum {getSum = 88})
runWriter (naiveFib 20)
output: (6765,Sum {getSum = 10945})

runWriter (optFib 10)
output: (55,Sum {getSum = 9})
runWriter (optFib 20)
output: (6765,Sum {getSum = 19})
```

Clearly, `naiveFib` is very naïve indeed, with the number of additions growing very fast with its argument!

5 The `IO` Monad

We've talked about the purity of Haskell, how it inviabilizes a few common programming constructs, and a few ways that category theory gives us alternatives via powerful abstractions. Now let's talk about the concept that suffers the most from the introduction of purity: *input and output*.

Recall that functions in Haskell are only mathematical functions: they receive an input and they return an output. Since a Haskell program is only a composition of Haskell functions, and hence a function itself, it could do nothing upon execution other than a single computation. Let's look over some things that cannot be done under the umbrella of purity:

- Print messages to the screen (implicit output)
- Get input from the user (implicit input)
- Draw an image on screen (implicit output)
- Read/write files in disk (implicit input and output)
- Generate random numbers (unless we want the same numbers every time we run the program, we need implicit input)
- Get the time and date (implicit input)
- Execute other programs (implicit input and output)
- Internet functionality (implicit input and output)
- The list goes on...

All of these problems are partially or totally solved by the so-called `IO` monad.

5.1 The Solution

So how do we make a Haskell program, whose functions must all be pure, interact with the outside world? The built-in Haskell way to interact with the outside world is through the `IO` monad. There are many ways to interpret or explain the `IO` monad: see [2] for a few, [1] for another and [12] for an article on the difficulty of teaching `IO` and monads to novices. For concreteness and conciseness we will only look at one of these interpretations, which can be found in [2].

The idea is the following. Under the assumption that Haskell is pure without exception, a Haskell program can do nothing but compute a Haskell value, which is the same every time the program is run. The idea is to postulate the existence of a Haskell type which represents a (possibly impure) program. Let us call this hypothetical type `Program`.

Then, our Haskell program would do its computations, culminating in a value of type `Program`, and this value would be passed to an external interpreter who would then

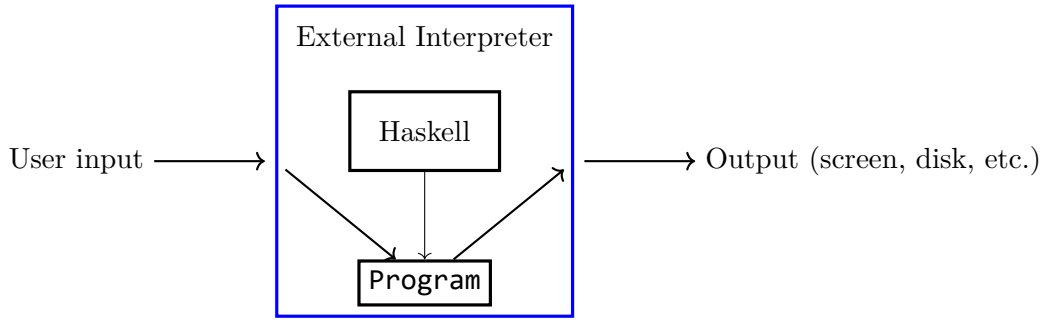


Figure 1: Visual representation of the (fictitious) external interpreter connecting to Haskell to run a `Program`

execute it. This way, with the help of this external interpreter, Haskell remains a pure engine of computation (as the returned `Program` is the same every time it is executed) but it becomes capable of interaction with the external world. See figure 1 for a visual representation of this process.

One thing that we ought to be able to do with `Program`s is compose them. For example, suppose that we have a program `askName :: Program` and another `greet :: String -> Program`, the first of which asks the user for their name, and the second of which, given a name, will print 'Hello, *name*' on the screen. It is reasonable to want to build a composite program, `askAndGreet :: Program`, which asks the user for their name and proceeds to greet them. How could we define `askAndGreet` starting from `askName` and `greet`?

Unfortunately, we can't, because the `Program` abstraction is missing one critical thing: `Program`s are unable to pass the results of their computations onwards to other `Program`s. In this case, `askName` should be able to give its output (the user input) to `greet`. Therefore, we now allow a different kind of `Program`, which besides its implicit input and output (via effects on the world) also has an explicit output, which can be passed onwards to other `Program`s. This new kind of program is called `IO`, or rather, `IO a`, with `a` a Haskell type. To recap:

- A value of type `Program` is an instruction or sequence of instructions built by Haskell, which the external interpreter executes,
- However, since this formalism does not have a built-in way to pass outcomes of some programs to others, we add a new type of program: `IO`.
- A value of type `IO a`, with `a` a Haskell type, is an instruction or sequence of instructions built by Haskell, which the external interpreter executes, which culminates in a value of type `a`.

There is but one short hop before we understand how Haskell deals with side effects: the type `Program` does not actually exist. Indeed, a program which does not culminate in any value can be identified with a program that culminates in an object of a trivial type, such as `()`. For example, the Haskell function `putStrLn` is of type `String -> IO ()`.

5.2 Working with `IO` and a few Examples

Let us look at a couple of the simplest `IO` functions in Haskell:

```
getLine :: IO String
putStrLn :: String -> IO ()
```

These types should be interpreted as: `getLine` is a program which goes off into the world and comes back to Haskell with a `String`, usually input that the user has written into the console. On the other hand, `putStrLn` isn't a program in its own right: it is a function which receives a `String` and returns a program, which in turn does some action on the outside world (printing the `String`) and returns no information.

Hello World Our first program with side effects will be the classical 'Hello world!':

```
main :: IO ()
main = putStrLn "Hello world!"
```

When compiled, this program results in an executable which, when run, prints the string 'Hello world!' to the screen. Note the use of `main`. A Haskell file will typically define many functions, so we need to tell the compiler which of them builds the `IO ()` value it is meant to execute, so it looks for a value of type `IO ()` and name `main`.

Echo and Bind Let us now make a program which receives input from the user and repeats it back to them. To get the input from the user we want the `IO String` action `getLine`, and we want to feed its output (a `String`) to the `putStrLn` function. Unfortunately, it's not as easy as writing `putStrLn . getLine`, as `getLine` returns an `IO String` and `putStrLn` requires a `String`.

A 'program composition operator' must behave as follows: given an `IO` action which culminates in a value of type `a`, and a function which turns such a value into another computation of type `IO b`, we want to write the program 'do this thing, feed the `a` into this function, and do that other thing'. The operator that does this composition is called *bind*:

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

This is the tool necessary to do the echo program:

```
main = getLine >>= putStrLn
```

Prompting and Prettying: Using Lambdas We will now make a less rude program. Instead of simply getting input and saying it right back like a petulant child, we will kindly ask the user for their name, and upon getting it with the use of `getLine`, we will return a greeting 'Hello, *name*!'.

First, we define the `IO ()` action which consists of prompting the user:

```
askName :: IO ()
askName = putStrLn "Hello! What is your name?"
```

Then, we define a function which gets the user input and returns a greeting. To this effect, we need to modify the input (turning the name into a pretty message) before printing it. To do this, we could first define an auxilliary function, which is then composed with `getLine`:

```
greet :: String -> IO ()
greet name = putStrLn ("Hello, " ++ name ++ "!")

getNameAndGreet :: IO ()
getNameAndGreet = getLine >>= greet
```

However, in order to avoid cluttering our code with functions that are only ever used once, it is sometimes better to instead use a lambda expression in place of `greet`:

```
getNameAndGreet :: IO ()
getNameAndGreet = getLine >>= (\name -> putStrLn ("Hello, " ++ name ++ "!"))
```

Finally, we compose `askName` and `getNameAndGreet`. However, we cannot use `>>=`: for that to work, we would need that `getNameAndGreet` be of type `() -> IO ()`, and not `IO ()`. The fix is simple, however: instead of writing `getNameAndGreet` on the right-hand side of `>>=` we write a function which takes one argument of type `()`, ignores it, and returns the `IO` action `getNameAndGreet`.

```
main = askName >>= (\x -> getNameAndGreet)
```

Of course, writing this is slightly silly and verbose for something as simple as 'run this program then run this one', so Haskell offers an alternative, via the `>>` operator, which can be read as 'and then':

```
(>>) :: IO a -> IO b -> IO b
a1 >> a2 = a1 >>= (\x -> a2)
```

We may write our greeting program without any auxilliary functions, to see it in all its glory:

```
main = putStrLn "Hello! What is your name?" >>
       getLine >>=
       (\name -> putStrLn ("Hello, " ++ name ++ "!"))
```

`IO` is a Functor In the previous example, we got user input and forwarded it using `>>=` to another function, which processed it and returned another `IO` action. However, sometimes it is appropriate to associate the processing to the left-hand side of `>>=` rather than to the right-hand side.

As an example, consider the following scenario: we wish to make a program that receives an integer input by the user, doubles it, and prints it on the screen. Let us take for granted that there exists a function `strToInt :: String -> Int` which reads an integer off of a string,[7] as well as a function `print :: Int -> IO ()` which prints an integer to

[7] In Haskell, there is the `read` function which does just that, but it is unsafe as it crashes the program if it is fed invalid input. There are other, safer alternatives, but it should suffice to say that they exist.

the screen.[8] Then, our program could be written as

```
main = getLine >>= (print . (\x -> 2*x) . strToInt)
```

However, suppose that we want to isolate the 'get an integer from the user' action. In other words, we want to build a Haskell object of type `IO Int`, using `getLine :: IO String` and `strToInt :: String -> Int`. We cannot simply construct a composite `strToInt . getLine`, as the types don't match, but if we could build a function `IO String -> IO Int` out of `strToInt` our problem would be solved. Interpreting this in the program framework, this function would take an `IO String`, i.e. a sequence of instructions for the external interpreter telling it how to fetch a `String` from the outside world, and add to the very end of these instructions 'when you're done getting that `String`, apply the function `strToInt` to it'.

This function exists, and can be constructed via `fmap strToInt`. Yes, this is the same `fmap` as in 3.2: `IO` is a functor! In other words, if we define a map $F\colon \mathrm{Hask} \to \mathrm{Hask}$ given by

- If a is a type, $F(a) = \mathrm{IO}\, a$,
- If $f\colon a \to b$, define $F(f)\colon F(a) \to F(b)$ as in the previous paragraph,

then F is a functor. In other words, `fmap` satisfies the functor laws.[9]

In conclusion, we are now able to define an `IO Int` action which reads an integer from the console, as

```
getInt :: IO Int
getInt = fmap strToInt getLine
```

The Monad Laws From the discussion above, we are able to conclude a very important fact relating `>>=` and `fmap` which will be very important later on: doing an `IO`-free computation before or after `>>=` is irrelevant. In other words, if `act1 = IO a`, `f :: a -> b` and `act2 = a -> IO b` then

```
act >>= (act2 . f) = (fmap f act) >>= act2
```
(13)

This is the most important of the so-called *monad laws*, which will be heavily discussed in section 6.

[8] Actually, the function `print` has type `print :: Show a => a -> IO ()`. The `Show` typeclass consists of types which may be represented as a string; to represent a value as a string one uses the function `show :: Show a => a -> String`. With it, one defines `print = putStrLn . show`.

[9] Strictly speaking, this is not necessarily true. There might be a small difference between `action :: IO a` and `fmap id action`: the former is a sequence of instructions, and the latter is the same sequence *followed by* 'and then, apply `id` to the result'. Of course, this is obviously of no consequence, except perhaps for an infinitesimal delay in execution, and it is very likely that the compiler would optimize away such a frivolous instruction anyway. Regardless of whether there exists a difference or not, there is no harm in accepting the fiction that `IO` is a functor, in much the same way that engineers do not need to correct for relativistic effects when building bridges.

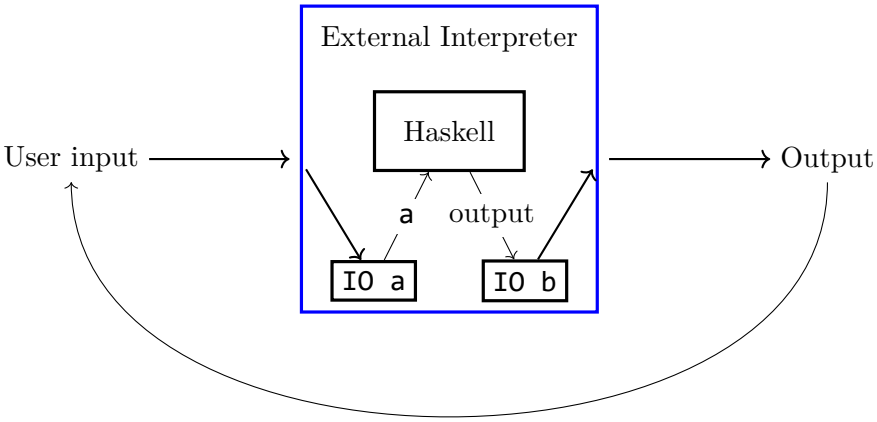


Figure 2: Visual representation of the ongoing conversation between Haskell and the (fictitious) interpreter

5.3 Conclusion

In this section, we discussed what a value of type `IO a` is (it is a sequence of instructions for the interpreter to go out into the world and get Haskell an object of type `a`), how to compose `IO` actions with functions that take their results and return other `IO` actions via the `>>=` operator, and how `IO` is a functor.

The `>>=` operator changes the `IO` metaphor slightly, because it allows for the outside world to talk to Haskell. For example, consider the program

```
main = getLine >>= (\in -> if in == "jackpot" then putStrLn "You hit the jackpot!"
    else putStrLn "...")
```

Then, when the interpreter executes this program, first it gets user input, and then it consults Haskell 'hey, I got this input, what do I do now?', and Haskell replies with another `IO` action. Consequently, figure 1 could be improved, leading to figure 2, which represents Haskell not as a box which is executed once to yield a `Program`, but rather as an engine the interpreter is in constant conversation with, executing actions, showing their results to Haskell, and consulting it about what to do next.

6 Monads

We preface this section with a joke.

Joke 1.

Novice learning Haskell: What is a Monad?

Haskell expert: Nothing special: it's just a monoid in the category of endofunctors!

6.1 Categorical Definition

Before talking about the applications to programming, we give a very brief introduction to the concept of monad. For a more in depth but still light introduction, see chapter VI of [7].

Definition 3. A monad in a category X is an endofunctor $T\colon X \to X$, together with two natural transformations

$$\eta\colon \mathrm{id}_X \Rightarrow T, \quad \mu\colon T^2 \Rightarrow T, \tag{14}$$

satisfying the equations

$$\mu \cdot T\mu = \mu \cdot \mu T, \tag{15}$$
$$\mu \cdot \eta T = \mu \cdot T\eta = \mathrm{id}_T. \tag{16}$$

Monads often, but not always, arise from adjunctions.

Proposition 1. Let $F\colon X \to Y$ and $U\colon Y \to X$ be adjoint functors, with $F \dashv U$. Then, $T = UF$ is a monad, with η the unit of the adjunction and $\mu = U\varepsilon F$, where ε is the counit of the adjunction.

Let us look at one particular example, which will be useful in the sequence. Let $F\colon \mathrm{Sets} \to \mathrm{Mon}$ be the free-monoid-generated-by functor, and $U\colon \mathrm{Mon} \to \mathrm{Sets}$ the forgetful functor. In this case, $T = UF$ is the functor which, given a set A, returns the monad freely generated by A, without the monadic structure. In other words, we can see $T(A)$ as the collection of finite sequences of elements of A.

In this case, $\eta\colon A \to T(A)$ is the function which, given an element a of A, returns the one-element sequence 'a'.

The other natural transformation, μ, is trickier. It is a function from $T(T(A))$ to $T(A)$. In other words, it receives sequences of sequences of elements of A, and returns a sequence of elements of A.[10] It is not obvious, but can be deduced by expanding the definition of μ and η, that μ consists of sequence concatenation. In other words, suppose that $s = s_1 \dots s_n$ is an element of $T(T(A))$, with

$$s_i = a_{i1} \dots a_{im_i}, \quad i = 1, \dots, n, \quad a_{ij} \in A. \tag{17}$$

Then, $\mu(s)$ is given by

$$\mu(s) = a_{11} \dots a_{1m_1} a_{21} \dots a_{2m_2} \dots a_{n1} \dots a_{nm_n}. \tag{18}$$

[10] Strictly speaking, this is not necessarily true. The result is heavily dependent on how we represent the free monad generated by a set. Here we will be assuming that the free monad generated by A is represented by the finite sequences of elements of A, with the operation given by concatenation.

6.2 A Monad in Haskell: Lists

Lists as Sequences We have seen that a simple example of a monad consists of the functor which takes a set A and returns the collection of finite sequences of elements of A. As it happens, we have already seen such an object in Haskell: the list functor![11]

The list functor List : Hask $\to$ Hask takes a type `a` and returns the type `[a]`, and it takes a function `f :: a -> b` and returns the function `fmap f :: [a] -> [b]` which applies `f` to all the elements of its input. To see it as a monad, by analogy with the example seen above we set

```
eta :: a -> [a]
eta x = [x]

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

where `concat` takes a list of lists and returns the result of their concatenation. This shows that Haskell already contains the natural transformation μ for the list functor, and it also contains η in the form of the function `return :: a -> [a]`.

Lists as Output of Nondeterministic Computations An important example of list usage is when some query has more than one possible response. For some applications, only one response is enough, but sometimes it is useful to know all possible responses to a query.

As an example, consider the following game. Start with a composite integer n. At each step, pick a divisor $1 < d < n$ of n and replace n with $d - 1$. The game halts whenever we reach a prime number n, or 1. We wish to find, for a positive integer n, the maximal number of moves until the game halts.

To a first approximation, let us implement a simple strategy to play the game: at each step of the game we pick the biggest divisor of n. Our code will execute this strategy and return the number of moves necessary to halt.

```
step :: Int -> Maybe Int
step n = let divisors = filter (\d -> n `mod` d == 0) [n-1,n-2..2] in
           fmap (\d -> d-1) (listToMaybe divisors)

stepsToHalt :: Int -> Int
stepsToHalt n = aux (step n)

aux :: Maybe Int -> Int
aux Nothing = 0
aux (Just n) = 1 + aux (step n)

stepsToHalt 73
output: 0
```

[11] This is not technically true. Since Haskell is lazy, it is possible (and oftentimes useful) to construct infinite lists. Therefore, the list functor does not correspond exactly to the monad associated to the free monoid constructor. However, it is close enough that the exposition does not suffer for it.

```
stepsToHalt 766
output: 8
```

Some explanation on the functions and notation used in the code above:

- The function `filter :: (a -> Bool) -> [a] -> [a]` takes a predicate `p` and a list `x` and returns the elements of the list which satisfy `p`. In this case the predicate is that `n `mod` d == 0`, i.e. that `d` be a divisor of `n`.
- The expression `[n-1,n-2..2]` represents the list that is suggested by the notation.
- The function `listToMaybe` takes a list and returns `Just` the first element of the list if it exists, and `Nothing` otherwise.

Now, evidently our strategy is not optimal. For example, suppose we start with $n = 48$. The current implementation goes $48 \to 23$, and so halts in one step. However, there is a better set of moves: $48 \to 15 \to 4 \to 1$.

With this in mind, let us implement a nondeterministic solver, where instead of picking a move and sticking with it, all possible moves are tried in parallel.

In preparation for this nondeterministic solver, we consider how to represent a state of the game at hand. There are two cases: either the game is still ongoing, in which case we want to store how many steps have elapsed since the start, and the value of n we currently have; or the game has already terminated, in which case we just want to store how many steps it took us to get where we are.

```
type UnhaltedState = (Int, Int)
type HaltedState = Int
type GameState = Either HaltedState UnhaltedState
```

Now, our nondeterministic step function takes an unhalted game and returns a list which contains all possible states of the game after one step; in the special case where the value of n which is input is prime or 1, it returns a singleton list, representing that the only possible next step is to halt.

```
stepND :: UnhaltedState -> [GameState]
stepND (s,n) = let divisors = filter (\d -> n `mod` d == 0) [n-1,n-2..2] in
               if divisors == [] then [Left s]
               else fmap (\d -> (Right (s+1,d-1))) divisors
```

Now we implement the function `stepsFromStateND`, which, given a state `s`, outputs all possible numbers of moves until halting:

```
stepsFromStateND :: GameState -> [Int]
stepsFromStateND (Left s) = return s
stepsFromStateND (Right st) = concat (fmap stepsFromStateND (stepND st))
```

Finally, we may extract the maximum number of moves by using the `maximum` function.

```
stepsToHaltND :: Int -> Int
stepsToHaltND n = maximum (stepsFromStateND (Right (0,n)))
```

We can see that the nondeterministic version does fare better than the deterministic version.

```
filter (\(x,y,z) -> y<z)
 (fmap (\n -> (n, stepsToHalt n, stepsToHaltND n)) [1..100])

output: [(28,1,2),(36,1,2),(40,1,2),(48,1,3),(58,2,3),(60,1,3),(64,1,3),(74,2,3)
    ,(75,2,3),(76,1,3),(82,2,3),(84,1,3),(87,2,3),(88,1,3),(96,1,3),(98,2,4)]
```

Nondeterministic Functions as Kleisli Arrows In the previous example, we used monadic structure explicitly in the function `stepsFromStateND`. Now we present another way to look at the problem.

Recall that in section 4 we introduced the idea of different function compositions. In that section, we considered functions of type `a -> Writer m b`, which took an input and returned an output together with a log, and defined the fish operator `<=<` which computes 'logged functions' by composing their logs appropriately.

We will now do a similar thing: we will redefine the concept of function composition in order to allow for composition of nondeterministic functions.

To be more precise, suppose that `f :: a -> [b]` is a nondeterministic function, which takes an `a` as input and returns possibly many (or none) values of `b` as output. Likewise, let `g :: b -> [c]`. Then, we define the nondeterministic composition of `g` and `f` as

```
(<=<) :: (b -> [c]) -> (a -> [b]) -> a -> [c]
g <=< f = \x -> concat (fmap g (f x))
```

Note the fact that we are reusing the fish operator. This is no accident: as it happens, `Writer m` is also a monad, and if we'll look at the typing of the fish operator in GHCi:

```
:t (<=<)
output: (<=<) :: Monad m => (b -> m c) -> (a -> m b) -> a -> m c
```

In other words, the fish operator is defined for pairs of functions of types `b -> m c` and `a -> m b` with `m` a monad. We have already seen the implementation for lists, and we will see that it generalizes trivially for a general monad `m`.

We can rewrite the function `stepsFromStateND` using the fish operator:

```
stepsFromStateND :: GameState -> [Int]
stepsFromStateND (Left s) = return s
stepsFromStateND (Right st) = (stepsFromStateND <=< stepND) st
```

Nondeterministic Input Let us look at a third way to see nondeterministic functions, which is often more useful for practical programming.

In practice, programmers don't always have the opportunity to write their functions as neat compositions of other functions. Sometimes dependencies are used multiple times, and it is much easier to store data in a variable and then reuse that variable multiple times than coming up with convoluted ways to carry the data across function compositions to where it needs to be. In other words, function application is useful.

This presents a problem to nondeterministic programming, because if `y :: [b]` is the (nondeterministic) output of a function `f :: a -> [b]`, and we want to feed `y` as input to another function `g :: b -> [c]`, simply writing `g y` will not do. We need some kind

of new 'nondeterministic function application' which, given nondeterministic input and a nondeterministic function, applies the function to all elements of the input.

Enter the bind operator, `>>=`. Yes, this is the same bind operator as in section 5, but now it has type

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

or, more generally,

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

For lists, it is defined by

```
(>>=) :: [a] -> (a -> [b]) -> [b]
y >>= g = concat (fmap g y)
```

and `stepsFromStateND` may be rewritten using bind:

```
stepsFromStateND :: GameState -> [Int]
stepsFromStateND (Left s) = return s
stepsFromStateND (Right st) = stepND st >>= stepsFromStateND
```

6.3 It All Ties Together (Categorically)

We have seen that the List type constructor is a monad, and from that we were able to define the fish operator `<=<` and the bind operator `>>=`. Now, we will show that the converse holds, so that, for example, `<=<` can be used to compose functions `a -> IO b` with `b -> IO c`, or that `>>=` can be used to feed a `Writer m a` into a function of type `Writer m b`. It is an interesting exercise to understand how these operations are interpreted, but for the sake of conciseness we will only show that the bind and fish operators induce different, but equivalent, ways to define monads.

Theorem 1. Let X be a category. The following data are equivalent.

- A functor $T\colon X \to X$ together with natural transformations

$$\eta\colon I \Rightarrow T, \quad \mu\colon T^2 \Rightarrow T \tag{19}$$

  such that

$$\mu \cdot \mu T = \mu \cdot T\mu, \tag{20}$$
$$\mu \cdot \eta T = \mu \cdot T\eta = \mathrm{id}_T. \tag{21}$$

- A functor $T\colon X \to X$, together with a natural transformation

$$\eta\colon I \Rightarrow T \tag{22}$$

  and an associative binary operation $(g, f) \mapsto g \odot f$, defined for $f, g \in X_1$ satisfying

$$f\colon x \to Ty, \quad g\colon y \to Tz, \tag{23}$$

with $g \odot f \colon x \to Tz$, and such that for all such f and g we have

$$f \odot \eta_x = \eta_y \odot f = f. \tag{24}$$

Furthermore, it relates to usual composition by

$$(h \odot g) \circ f = h \odot (g \circ f), \tag{25}$$
$$Th' \circ (g' \odot f') = Th' \odot (Tg' \circ f'). \tag{26}$$

whenever f, g and h compose appropriately, and likewise for f', g' and h'.

- A functor $T \colon X \to X$, together with a natural transformation

$$\eta \colon I \Rightarrow T \tag{27}$$

and an operator $\beta_{xy} \colon X(x, Ty) \to X(Tx, Ty)$ which satisfies

$$Tf = \beta_{xy}(\eta_y \circ f), \tag{28}$$
$$\beta_{xy}(f) \circ \eta_x = f, \tag{29}$$
$$\beta_{yz}(g) \circ \beta_{xy}(f) = \beta_{xz}(\beta_{yz}(g) \circ f). \tag{30}$$

Remark 1. Before going through with the proof, it is worth it to look at how these concepts are reflected in Haskell.

All three of the definitions start with a functor and a natural transformation η. In Haskell, the functor is a type constructor, e.g. `[]`, `Writer m`, `IO`, and the η is represented by the function `return :: a -> m a`.

The first definition contains an additional natural transformation, denoted μ. In Haskell, this is represented by the function `join :: m m a -> m a`.

The second definition contains the binary operator $\odot$. In Haskell, this is represented by the fish operator `<=<`.

The third definition is the one which is a little further away from its Haskell counterpart. In Haskell, β is represented by the bind operator

```
(>>=) :: m a -> (a -> m b) -> m b
```

whose typing is not the same as β. To get from one to the other, one must first invert the order of the arguments, yielding the function

```
(=<<) :: (a -> m b) -> (m a -> m b)
(=<<) = curry ((uncurry (>>=)) . swap)
```

This one maps directly to β.

The reason that this mental gymnastics is necessary is because the original typing of bind only makes sense in nice enough categories, namely those with exponentiation. The modified definition is more general, being applicable to any category.

Proof: Since all definitions provide the functor T and natural transformation $\eta \colon I \Rightarrow T$, it suffices to construct the following three pieces of data from one another: μ, $\odot$, and β.

- (Build $\odot$ from μ) Let $f\colon x \to Ty$ and $g\colon y \to Tz$. Then, set

$$g \odot f := \mu_z \circ Tg \circ f. \tag{31}$$

We need to verify that $\odot$ is associative and that η is a neutral element, as well as the relation between $\odot$ and composition.

To show associativity, we compute $(h \odot g) \odot f$, for $f\colon x \to Ty$, $g\colon y \to Tz$ and $h\colon z \to Tw$:

$$\begin{aligned}(h \odot g) \odot f &= \mu_w \circ T(h \odot g) \circ f \\ &= \mu_w \circ T\mu_w \circ TTh \circ Tg \circ f \\ &= \mu_w \circ \mu_{Tw} \circ TTh \circ Tg \circ f \quad \text{(by (20))} \\ &= \mu_w \circ Th \circ \mu_z \circ Tg \circ f \quad \text{(Naturality)} \\ &= \mu_w \circ Th \circ (g \odot f) \\ &= h \odot (g \odot f).\end{aligned} \tag{32}$$

Now, we prove that $f \odot \eta_x = f$:

$$\begin{aligned}f \odot \eta_x &= \mu_y \circ Tf \circ \eta_x \\ &= \mu_y \circ \eta_{Ty} \circ f \quad \text{(Naturality)} \\ &= \mathrm{id}_{Ty} \circ f \quad \text{(by (21))} \\ &= f.\end{aligned} \tag{33}$$

The proof that $\eta_y \odot f = f$ is similar, and so we turn to showing (25).

$$\begin{aligned}(h \odot g) \circ f &= (\mu_w \circ Th \circ g) \circ f \\ &= \mu_z \circ Th \circ (g \circ f) \\ &= h \odot (g \circ f).\end{aligned} \tag{34}$$

Finally, we prove (26).

$$\begin{aligned}Th \circ (g \odot f) &= Th \circ \mu_z \circ Tg \circ f \\ &= \mu_{Tw} \circ T^2 h \circ Tg \circ f \quad \text{(Naturality)} \\ &= Th \odot (Tg \circ f).\end{aligned} \tag{35}$$

The first part of the proof is complete.

- (Build β from $\odot$) Simply set

$$\beta_{xy}(f) = f \odot \mathrm{id}_{Tx}. \tag{36}$$

To prove (28), we compose with an appropriate identity:

$$\begin{aligned}\beta(\eta_y \circ f) &= (\eta_y \circ f) \odot \mathrm{id}_{Tx} \\ &= T\mathrm{id}_y \circ (\eta_y \circ f) \odot \mathrm{id}_{Tx} \\ &= T\mathrm{id}_y \odot (T\eta_y \circ Tf \circ \mathrm{id}_{Tx}) \quad \text{(by (26))} \\ &= T\mathrm{id}_y \odot (T\eta_y \circ Tf) \\ &= T\mathrm{id}_y \circ (\eta_y \odot Tf) \quad \text{(by (26))} \\ &= f.\end{aligned} \tag{37}$$

The proof of (29) is trivial, so we now prove (30):

$$\begin{aligned}\beta(g) \circ \beta(f) &= (g \odot \mathrm{id}_{Ty}) \circ (f \odot \mathrm{id}_{Tx}) \\ &= g \odot (f \odot \mathrm{id}_{Tx}) \quad \text{(by (25))} \\ &= (g \odot f) \odot \mathrm{id}_{Tx}) \quad \text{(Associativity)} \\ &= ((g \odot \mathrm{id}_{Ty}) \circ f) \odot \mathrm{id}_{Tx} \quad \text{(by (25))} \\ &= \beta(\beta(g) \circ f).\end{aligned} \tag{38}$$

- (Build μ from β) Define μ_x, for $x \in X_0$, by

$$\mu_x = \beta(\mathrm{id}_{Tx}). \tag{39}$$

The proofs of naturality of μ and (20) and (21) are left to the reader.

Now that we have constructed $\odot$ from μ, β from $\odot$ and μ from β, it remains to show that these constructions do not lose information. In other words, if we start from μ, build $\odot$, then β, and then a new $\overline{\mu}$, we need to show that $\mu = \overline{\mu}$, and likewise for the other two constructions. The proof of this fact is straight-forward and left to the reader. ■

Remark 2. It should be noted that the proof of theorem 1 gives us more than the statement: it actually tells us how to construct each piece of data from the others. This translates to implementation. For example, in Haskell a monad is defined in terms of the bind operator `>>=` (in the nomenclature above, β), and the fish operator `<=<` (in the notation above, $\odot$) is defined on all monads using bind, as is `join` (μ).

Remark 3. The definition in terms of T, η and β can actually be further simplified. Indeed, by (28) the definition of T on morphisms can be induced solely from the rest of the data. Therefore, if we remove the hypothesis that T is a functor and replace the hypothesis (28) by something else, we actually obtain a leaner definition of monad:

Theorem 2. Let X be a category. The following data is equivalent:

- A monad (T, η, β) on X,

- A map $T\colon X_0 \to X_0$, a collection of maps $\eta_x\colon x \to Tx$, and a collection of maps $\beta_{xy}\colon X(x,Ty) \to X(Tx,Ty)$, satisfying the so-called *monad laws*:

$$\beta(\eta) = \mathrm{id}, \tag{40}$$
$$\beta(f) \circ \eta = f, \tag{41}$$
$$\beta(g) \circ \beta(f) = \beta(\beta(g) \circ f). \tag{42}$$

Proof: Evidently a monad induces such data. On the other hand, from this data we may define T on morphisms via (28), and functoriality of the resulting T is easy to verify; note that (40) is necessary to prove that $T(\mathrm{id}) = \mathrm{id}$. ■

Theorem 2 is actually the reason behind an interesting historical accident. As it happens, functors and monads were both introduced in version 1.3 of the Haskell report (the official standard for the language), in 1996. Now, it was already known all the way back to 1989[12] that every monad was a functor, but it was only in 2015, with the so-called '`Applicative => Monad` proposal' (AMP) [6], that the language itself enforced that `fmap` be defined for every instance of `Monad`.

6.4 How Programming Inspires Math

When we introduced monads in definition 3 we remarked (proposition 1) that every adjunction induces a monad. On the other hand, it has been known since at least 1964 [5] that every monad is induced by an adjunction.

Historically, this all happened long before monads were applied to programming languages, but as it happens the work we have already done in a programming context leaves us but a short step away from this theorem.

We begin by remarking that every monad induces a category in a natural way: if we combine theorem 1 with ideas from definition 2, we conclude that, given a monad $M = (T, \eta, \odot)$ on T, the collections

$$W_0 = X_0, \quad W(x,y) = X(x,Ty) \tag{43}$$

form a category W with $\odot$ as the composition and η as the identities. This is called the *Kleisli category for the monad* M. We may define $F\colon X \to W$ and $U\colon W \to X$ as

$$\begin{aligned} F(x) &= x, & F(f) &= \eta_y \circ f, \\ U(x) &= Tx, & U(f) &= \beta(f) = f \odot \mathrm{id}_{Tx}. \end{aligned} \tag{44}$$

It is worth it to take a step back and interpret W, F and U in computational terms, and the list monad T is particularly enlightening in this respect. We saw that a function `a -> [b]` can be interpreted as a nondeterminstic function, and hence in this case *the*

[12] See [10], definition 3.2.1 (which corresponds to our definition via β) and proposition 3.2.4 (which establishes the equivalence between this definition and the classical one based on on functors). See also [5] for another proof of equivalence, way back in 1964.

Kleisli category is the category whose morphisms are nondeterministic functions, with the appropriate composition. The functor F takes a deterministic function and turns it into a 'deterministic nondeterministic function', whose list of outputs contains only one element. On the other hand, the functor U takes a nondeterministic function and turns it into a deterministic function, by seeing the nondeterministic results for what they are: lists. In breaking the nondeterministic metaphor, it must turn a function in $W(x, y)$ into a function whose output is a nondeterministic y, i.e. $U\colon W(x, y) \to X(?, Ty)$. By consistency, U must act on all objects by T, and we have already seen that β acts on $W(x, y) = X(x, Ty)$ in an appropriate way. Interpreting β computationally, U goes to a category which does not know how to perform nondeterministic computations, so U itself teaches the functions how to accept nondeterministic inputs using β.

We will now verify that F and U form an adjoint pair, which in turn originates the monad we started with. We already have the candidate for the unit of the adjunction, which is η, and so all that remains is to find the candidate for the counit, i.e. to solve the equation

$$\mu_x = U(\varepsilon_{F(x)}). \tag{45}$$

Expanding (45) using (44), we obtain

$$\mu_x = \beta(\varepsilon_x), \tag{46}$$

and so we may use equation (29) to recover ε via

$$\varepsilon_y = \beta(\varepsilon_y) \circ \eta_{Ty} = \mu_y \circ \eta_{Ty}, \tag{47}$$

and hence, by (21),

$$\varepsilon_y = \mathrm{id}_{Ty}. \tag{48}$$

It is not difficult to show that η and ε satisfy the triangular equalities, and therefore $(F, U, \eta, \varepsilon)$ forms an adjunction, whose associated monad is M. Therefore, we have just shown:

Theorem 3. Every monad is induced by an adjunction in the sense of proposition 1.

The proof that we have presented is due to Kleisli [5]. It is not the only construction; see for example VI.2 in [7] for another, due to Eilenberg-Moore.

7 Closing Remarks

The connection between Haskell and category theory is a rather interesting one, and not only for its content. First of all, it is of note that we have barely touched on the computational part of the theory behind Haskell, such as lambda calculus and type theory. Those are interesting areas in their own right, and they have a remarkable intersection with category theory, for example when it comes to theorems for free [11]. Furthermore, in this essay we have been working with 'idealized Haskell', in which all functions terminate and there are no infinite loops or errors, which is of course an

unreasonable assumption; domain theory helps us account for this, by providing better descriptions of what kind of thing can be output by a program. For example, formally, each type is inhabited by at least one element, called bottom and denoted $\bot$. This type represents a computation which does not terminate, and since Haskell is a *lazy* language (a term which we have not had the opportunity to define or explore) the notion of bottom is more nuanced than it would seem at first sight. For example, the pair $(\bot, \bot)$ is different from $(\bot, 1)$, which is different from $(\bot, 2)$, and all of these are different from $\bot$ (the undefined pair).

Another interesting thing about using mathematics to describe Haskell is that it approaches physics in a sense. This is because Haskell is a *real* programming language, and the real world is messy. Haskell is riddled with minor exceptions, concessions to practicality which break otherwise ironclad rules: the reader will have noticed that this essay is riddled with footnotes saying 'well, actually...' In this sense, any description of Haskell (short of the official documentation) will be only an approximation, and much like how Newtonian and relativistic physics both have their place as descriptions of the world to better or worse approximation, so do idealized Haskell and more complex models. In this way, the content of this essay is applied category theory in a quite literal sense, which goes to show that even the most abstract areas of mathematics may eventually find application in the real world.

As a last remark on category theory, in this essay we have gone to reasonable depth on monads in particular as a programming abstraction. No other categorical concept has found as wide application, except of course for the notion of morphism and perhaps the notion of functor, but that does not mean that other categorical concepts go unused. For a great reference, see [9], in which concepts such as comonads and coalgebras are used as a formalism for, for example, random number generation and object inspection (lenses), both of which very useful notions in practice. There is also another concept which neither we nor [9] touch on, which is the notion of *monad transformer*, which is a way to deal with the fact that composition of monads is not, in general, a monad; this is an active area of research.

References

[1] Haskell wiki: IO inside. `https://wiki.haskell.org/IO_inside`.

[2] Stack Overflow: How exactly does 'IO's `>>=` work under the hood? `https://stackoverflow.com/a/51772273/2997964`.

[3] Andrej Bauer. Hask is not a category. `http://math.andrej.com/2016/08/06/hask-is-not-a-category/`.

[4] Mark P Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of functional programming*, 5(1):1–35, 1995.

[5] Heinrich Kleisli. Every standard construction is induced by a pair of adjoint functors. *Proceedings of the American Mathematical Society*, 16(3):544–546, 1965.

[6] David Luposchainsky. Haskell 2014: Applicative => Monad proposal (AMP). https://github.com/quchen/articles/blob/master/applicative_monad.md.

[7] Saunders Mac Lane. *Categories for the working mathematician*, volume 5. Springer Science & Business Media, 2013.

[8] Bartosz Milewski. Parametricity: Money for nothing and theorems for free. https://bartoszmilewski.com/2014/09/22/parametricity-money-for-nothing-and-theorems-for-free/.

[9] Bartosz Milewski. *Category theory for programmers*. Bartosz Milewski, 2019.

[10] Eugenio Moggi. *An abstract view of programming languages*. University of Edinburgh, Department of Computer Science, Laboratory for …, 1989.

[11] Philip Wadler. Theorems for free! In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 347–359, 1989.

[12] Brent Yorgey. Abstraction, intuition, and the "monad tutorial fallacy". https://byorgey.wordpress.com/2009/01/12/abstraction-intuition-and-the-monad-tutorial-fallacy/.