The Monadnomicon

Ramiro Pastor Martin

July 29th of 2016

Contents

[1] For the definition of fundamental groupoid of a topological space, see appendix A

Chapter 1

Introduction to Categories

1.1 Category

1.1.1 Definition

Def: a **category** is a tern $\langle \mathbf{Obj}(\mathfrak{C}), \mathbf{Hom}(\mathfrak{C}), \odot \rangle$ where :

1. $Obj(\mathfrak{C})$ is a class (not necessarily a set) whose members are called **objects** of $\mathfrak{C}$. In practice one often abuses notation by denoting the class of objects of $\mathfrak{C}$ by the letter $\mathfrak{C}$ as well. In particular, the notation $X \in \mathfrak{C}$ is to be understood as "X is an object of $\mathfrak{C}$".

2. $Hom(\mathfrak{C})$ is a class (if $Obj(\mathfrak{C})$ is a class) or a set (if $Obj(\mathfrak{C})$ is a set) whose members are called **morphisms** of $\mathfrak{C}$. Each morphism f of $\mathfrak{C}$ is associated with a **departure object** X, and an **arrival object** Y, both from $Obj(\mathfrak{C})$; we write this as "f goes from X to Y" or $f : X \to Y$ or $X \xrightarrow{f} Y$.

   The (always a) set of morphisms from X to Y in the category $\mathfrak{C}$ is denoted as $\mathbf{Hom}_{\mathfrak{C}}(\mathbf{X}, \mathbf{Y})$. Also, instead of $Hom_{\mathfrak{C}}(X, X)$ we will write $Endo_{\mathfrak{C}}(X)$; its elements are called **endomorphisms** of X.

3. A composition law $\odot$ that $\forall\ X, Y, Z \in \mathfrak{C}$:

$$Hom_{\mathfrak{C}}(X, Y) \times Hom_{\mathfrak{C}}(Y, Z) \to Hom_{\mathfrak{C}}(X, Z)$$

$$(f, g) \mapsto g \odot f$$

   this is, for every $X \xrightarrow{f} Y \xrightarrow{g} Z$ there must exist a morphism $h : X \to Z$ assigned to $g \odot f$. It must verify:

   Associativity Composition of morphisms is associative. More precisely, given objects X, Y, Z, W of $\mathfrak{C}$ and morphisms $X \xrightarrow{f} Y \xrightarrow{g} Z \xrightarrow{h} W$ we require that $h \odot (g \odot f) = (h \odot g) \odot f$

   Neutral elements Every object has an "identity endomorphism". More precisely, if $X \in \mathfrak{C}$, there exists an element $id_X \in Endo_{\mathfrak{C}}(X)$ such that for every morphism $f : X \to Y$ in $Hom(\mathfrak{C})$, we have $f \odot id_X = f$, and for every morphism $g : Z \to X$ in $Hom(\mathfrak{C})$, we have $id_X \odot g = g$.

Obs: **1.** *In view of the associative axiom, whenever we have any composable sequence $f_1, \ldots, f_n$ of morphisms in a category, the expression $f_n \odot f_{n-1} \odot \ldots \odot f_2 \odot f_1$ is unambiguous.*

1.1.2 Unicity of neutral elements and examples

Prop: **1.** *For any category $\mathfrak{C}$ and any object $X \in \mathfrak{C}$, there is only one endomorphism of X satisfying the defining property of id_X. Thus one can really speak of the identity endomorphism of X.*

Proof. Given $X \in \mathfrak{C}$ suppose that there are two endomorphisms of X, $\widetilde{id_X}$ and $\widehat{id_X}$, with the property of neutral element.

This is, $\forall\ f \in Hom(A, X)$ and $\forall\ g \in Hom(X, B)$ occurs:

$$i\tilde{d}_X \odot f = f \qquad\qquad i\hat{d}_X \odot f = f$$
$$g \odot i\tilde{d}_X = g \qquad\qquad g \odot i\hat{d}_X = g$$

If we apply this to $i\hat{d}_X \odot i\tilde{d}_X$ it falls that:

$$i\hat{d}_X = i\hat{d}_X \odot i\tilde{d}_X = i\tilde{d}_X \implies i\hat{d}_X = i\tilde{d}_X$$

□

Examples: (note: $\odot$ is always the usual function composition $\circ$ unless said otherwise.)

$\mathfrak{Set}$: Obj($\mathfrak{Set}$) – the class of all sets. Hom($\mathfrak{Set}$) – functions between sets.

$\mathfrak{Grp}$: Obj($\mathfrak{Grp}$) – the class of all groups. Hom($\mathfrak{Grp}$) – group homomorphisms.

$\mathfrak{Top}$: Obj($\mathfrak{Top}$) – the class of all topological spaces. Hom($\mathfrak{Top}$) – continuous maps between topological spaces.

$\mathfrak{Vect}_{\mathbb{K}}$: Obj($\mathfrak{Vect}_{\mathbb{K}}$) – the class of all vector spaces over a given field $\mathbb{K}$. Hom($\mathfrak{Vect}_{\mathbb{K}}$) – linear maps between vector spaces.

$\mathfrak{Hask}$: Obj($\mathfrak{Hask}$) – the class of all Haskell types. Hom($\mathfrak{Hask}$) – Haskell functions. The composition law is the (.) operator.

$\leq$: any partially ordered set $\langle P, \leq \rangle$ defines a category where the objects are the elements of P, and there is a morphism (and only one) between any two objects A and B iff $A \leq B$.

This can be applied to the power set $\wp$(X) of any given set X, taking the inclusion as the partial order.

1.1.3 Isomorphisms and Automorphisms

Def: Let $\mathfrak{C}$ be a category and $X \xrightarrow{f} Y$ a morphism in $\mathfrak{C}$. We say that f is an **isomorphism**, or that f is **invertible**, if there exists a morphism $g : Y \to X$ in $\mathfrak{C}$ such that $g \odot f = id_X$ and $f \odot g = id_Y$. If this is the case, g is called an **inverse** of f, and viceversa.

Prop: **2.** *If f has an inverse, that inverse is unique (so, if f is an isomorphism, one can unambiguously denote its inverse by f^{-1}).*

Proof. given $f : X \to Y$ such that exists $g : Y \to X$ and $g* : Y \to X$ inverses of f, lets conclude that g = g*. We have:

$$g \odot f = id_X \qquad\qquad f \odot g = id_Y$$
$$g * \odot f = id_X \qquad\qquad f \odot g* = id_Y$$

beginning with the first ecuation, and composing with g* to the right, we have $g \odot f \odot g* = id_X \odot g*$ and applying the last ecuation, we get $g \odot id_Y = id_X \odot g* \implies g = g*$ □

***Prop:* 3.** *If $f : X \to Y$ and $g : Y \to Z$ are composable morphisms and are both invertible, then $g \odot f$ is also invertible, and $(g \odot f)^{-1} = f^{-1} \odot g^{-1}$.*

Proof. the existence of $(g \odot f)^{-1}$ will be given automatically as soon as we prove that $(g \odot f)^{-1} = f^{-1} \odot g^{-1}$ because we know that $f^{-1} \odot g^{-1}$ must exist.

In addition, with the already proved uniqueness of inverses (last proposition) and its consequent unambiguity in the use of the f^{-1} notation, we only need to prove that composing the function $g \odot f$ with the function $f^{-1} \odot g^{-1}$ gives us the indentity function in both X and Z. Indeed:

$$\begin{array}{llll} (g \odot f) \odot f^{-1} \odot g^{-1} = & \overset{\text{associativity}}{\longrightarrow} = & g \odot \left(f \odot f^{-1}\right) \odot g^{-1} = & id_Z \\ f^{-1} \odot g^{-1} \odot (g \odot f) = & \overset{\text{associativity}}{\longrightarrow} = & f^{-1} \odot \left(g^{-1} \odot g\right) \odot f = & id_X \end{array}$$

□

***Obs:* 2.** *the reciprocal proposition (invertible composition implies invertible factors) is not true in general. As a counterexample, in the $\mathfrak{sets}$ category take f,g : $\mathbb{N} \to \mathbb{N}$ with $f(x) = 2x$ and $g(x) = \lfloor x/2 \rfloor$, and compose $g \circ f$.*

Def: If X, Y are objects of a category $\mathfrak{C}$ such that there exists an isomorphism (i.e. invertible) $f : X \to Y$, we say that X and Y are **isomorphic**, and write $X \cong Y$ or f : X $\overset{\sim}{\to}$ Y.

***Obs:* 3.** *Usually, if an isomorphism between X and Y exists, it is non-unique. For example, there exists a bijection between two finite sets $\leftrightarrow$ both have the same number n of elements, and in that case there are exactly n! bijections between them.*

Def: If $\mathfrak{C}$ is a category and $X \in \mathfrak{C}$, the invertible endomorphisms of X are called **automorphisms**.

Def: If $\mathfrak{C}$ is a category and $X \in \mathfrak{C}$, the set of all invertible endomorphisms of X will be denoted by $\mathbf{Aut}_{\mathfrak{C}}(\mathbf{X})$.

***Prop:* 4.** *$Aut_{\mathfrak{C}}(X)$ is a group under the composition of morphisms. It is called the group of automorphisms of X.*

1.1.4 Groupoids and Subcategories[1]

Def: A **groupoid** is a category in which every morphism is invertible.

Def: Let $\mathfrak{C}$ be any category, and let $\mathfrak{C}^{\mathbf{x}}$ be the category whose class of objects is that of $\mathfrak{C}$, and where the morphisms are defined as follows. If X, Y are two objects, then $Hom_{\mathfrak{C}^x}(X,Y)$ is the set of invertible elements of $Hom_{\mathfrak{C}}(X,Y)$. The composition of morphisms in $\mathfrak{C}^x$ is defined as the composition in $\mathfrak{C}$.

***Prop:* 5.** *$\mathfrak{C}^x$ is a category and is a groupoid.*

Def: Let $\mathfrak{C}$ be a category. We say that a category $\mathfrak{D}$ is a **subcategory** of $\mathfrak{C}$ if: the class of objects of $\mathfrak{D}$ is a subclass of the class of objects of $\mathfrak{C}$, i.e. $Obj(\mathfrak{D}) \subset Obj(\mathfrak{C})$; for every pair X, Y of objects of $\mathfrak{D}$, the set $Hom_{\mathfrak{D}}(X,Y)$ is a subset of $Hom_{\mathfrak{C}}(X,Y)$; and composition of two morphisms in $\mathfrak{D}$ is the same regardless of whether it is computed in $\mathfrak{D}$ or in $\mathfrak{C}$.

We say that $\mathfrak{D}$ is a **full subcategory** of $\mathfrak{C}$ if, for every pair of objects X, Y of $\mathfrak{D}$, one has $Hom_{\mathfrak{D}}(X,Y) = Hom_{\mathfrak{C}}(X,Y)$ (this is, we only lose objects and not morphisms).

[1]For the definition of fundamental groupoid of a topological space, see appendix A

1.2 Functors and natural transformations

From now on, the symbol $\odot$ will be replaced by $\circ$

1.2.1 Definition

Def: Let $\mathfrak{C}_1$ and $\mathfrak{C}_2$ be categories. A **covariant functor** $\Phi : \mathfrak{C}_1 \to \mathfrak{C}_2$ is a rule which to every object X of $\mathfrak{C}_1$ assigns an object $\Phi(X)$ of $\mathfrak{C}_2$, and to every morphism $f : X \to Y$ in $\mathfrak{C}_1$ assigns a morphism $\Phi(f) : \Phi(X) \to \Phi(Y)$ in $\mathfrak{C}_2$ such that $\Phi(g \circ f) = \Phi(g) \circ \Phi(f)$ whenever $X \xrightarrow{f} Y \xrightarrow{g} Z$ are morphisms in $\mathfrak{C}_1$, and such that $\Phi(id_X) = id_{\Phi}(X)$ for all objects X of $\mathfrak{C}_1$.

To get the notion of a **contravariant functor** $\Psi : \mathfrak{C}_1 \to \mathfrak{C}_2$ one has to make the following changes: $\Psi(f)$ should now be a morphism from $\Psi(Y)$ to $\Psi(X)$ (i.e., Ψ"reverses the directions of all arrows"), and the first requirement in the definition of a functor has to be replaced by $\Psi(g \circ f) = \Psi(f) \circ \Psi(g)$.

Usually, the word "functor" without any adjectives refers to a covariant functor. I will also use this convention from now on.

1.2.2 Examples of functors

There are plenty:

1. For any category $\mathfrak{C}$ we have the *identity functor* $Id_{\mathfrak{C}} : \mathfrak{C} \to \mathfrak{C}$.
2. If $\mathfrak{C}$ is a category and $\mathfrak{D} \subseteq \mathfrak{C}$ is a subcategory, one has the obvious "inclusion functor "$\mathfrak{D} \hookrightarrow \mathfrak{C}$. In particular, we have inclusion functors $\mathfrak{Vect}_{\mathbb{K}} \hookrightarrow \mathfrak{Grp} \hookrightarrow \mathfrak{Set}$. These functors are usually called the "forgetful functors".
3. The *power set functor* $\mathfrak{Set} \to \mathfrak{Set}$ which maps sets to their power sets; and maps functions $f : X \to Y$ to functions $\wp(X) \to \wp(Y)$ which take inputs $U \subseteq X$ and return $f(U)$, the image of U under f, defined by $f(U) \quad = \quad \{ \quad f(u) : u \in U \quad \}$
4. The fundamental group is a functor from $\mathfrak{Top}*$, the category of *pointed topological spaces*[2], to $\mathfrak{Grp}$. More precisely, check that if $f : (X, x) \to (Y, y)$ is a morphism in $\mathfrak{Top}*$, then one can use f to define a group homomorphism $\pi_1(X, x) \to \pi_1(Y, y)$, and this yields a functor $\mathfrak{Top}* \to \mathfrak{Grp}$.

1.2.3 The dual category

Def: Let $\mathfrak{C}$ be any category. The **dual category** $\mathfrak{C}^\circ$ of $\mathfrak{C}$ is informally speaking, obtained from $\mathfrak{C}$ by "reversing all the arrows". This is:

$$Obj(\mathfrak{C})^\circ := Obj(\mathfrak{C}) \quad ; \quad Hom_{\mathfrak{C}^\circ}(X, Y) := Hom_{\mathfrak{C}}(Y, X) \quad with \quad f^\circ \circ g^\circ = (g \circ f)^\circ$$

Prop: **6.** *The rule $X \mapsto X$, $f \mapsto f^\circ$ defines a contravariant functor $\mathfrak{C} \to \mathfrak{C}^\circ$*

Prop: **7.** *If $\mathfrak{C}_1$ and $\mathfrak{C}_2$ are two categories, a covariant functor $\Phi : \mathfrak{C}_1 \to \mathfrak{C}_2$ can also be though of as a contravariant functor ${\mathfrak{C}_1}^\circ \to \mathfrak{C}_2$, or a contravariant functor $\mathfrak{C}_1 \to {\mathfrak{C}_2}^\circ$, or a covariant functor ${\mathfrak{C}_1}^\circ \to {\mathfrak{C}_2}^\circ$. The same holds if we switch "covariant" and "contravariant" throughout the last sentence.*

[2] The objects of $\mathfrak{Top}*$ are pairs (X, x) consisting of a topological space X and a point $x \in X$. A morphism $f : (X, x) \to (Y, y)$ in $\mathfrak{Top}*$ is a continuous map $f : X \to Y$ such that $f(x) = y$. Composition of morphisms is defined as the composition of maps in the usual sense.

1.2.4 Natural transformations

Def: Let $\mathfrak{C}_1$ and $\mathfrak{C}_2$ be categories, and let $\Phi, \Psi : \mathfrak{C}_1 \to \mathfrak{C}_2$ be functors. A **morphism of functors**, or a **natural transformation**, $\alpha : \Phi \to \Psi$, is a rule which to every object $X \in \mathfrak{C}_1$ assigns a morphism $\alpha_X : \Phi(X) \to \Psi(X)$ such that for any morphism $X \xrightarrow{f} Y$ in $\mathfrak{C}_1$, the following diagram commutes:

$$\begin{array}{ccc} \Phi(X) & \xrightarrow{\Phi(f)} & \Phi(Y) \\ {\scriptstyle \alpha_X}\downarrow & & \downarrow{\scriptstyle \alpha_Y} \\ \Psi(X) & \xrightarrow{\Psi(f)} & \Psi(Y) \end{array}$$

Def: We say that the collection $(\alpha_X)_{X \in \mathfrak{C}}$ is an **isomorphism (of functors)** between Φ and Ψ if each morphism α_X is invertible. In this case the collection $\left({\alpha_X}^{-1}\right)$ defines a morphism of functors from Ψ to Φ. We call this collection α^{-1}.

1.2.5 Composition of functors and natural transformations

Def: Let $\mathfrak{C}_1, \mathfrak{C}_2, \mathfrak{C}_3$ be categories and let $\Phi : \mathfrak{C}_1 \to \mathfrak{C}_2$, $\Psi : \mathfrak{C}_2 \to \mathfrak{C}_3$ be functors. The **composed functor** $\Psi \circ \Phi : \mathfrak{C}_1 \to \mathfrak{C}_3$ assigns to each object $X \in \mathfrak{C}_1$ the object $\Psi(\Phi(X)) \in \mathfrak{C}_3$; and to each morphism f of $Hom_{\mathfrak{C}_1}(X, Y)$, the morphism $\Psi(\Phi(f)) \in Hom_{\mathfrak{C}_3}(\Psi(\Phi(X)), \Psi(\Phi(Y)))$

Similarly, if $\mathfrak{C}$ and $\mathfrak{D}$ are categories, $\Phi_1, \Phi_2, \Phi_3 : \mathfrak{C} \to \mathfrak{D}$ are three functors, and $\alpha : \Phi_1 \to \Phi_2$, $\beta : \Phi_2 \to \Phi_3$ are natural transformations, invent the definition of the composition $\beta \circ \alpha : \Phi_1 \to \Phi_3$. In fact, modulo some set-theoretical issues (which should be ignored at this point), one can define the category of functors $\mathfrak{Funct}(\mathfrak{C}, \mathfrak{D})$ whose objects are functors from $\mathfrak{C}$ to $\mathfrak{D}$ and whose morphisms are natural transformations.

1.3 Commutative diagram and monad definition

The concept of monad is found deep within the theory of Categories, far beyond the point in which we are now. In fact, the full theory can be built without set theory, with its own beauties such as expressing algebraic identities as commutative diagrams. The concept of *commutative diagram* is itself basic for this approach, so it will be exposed now.

Also, i will include the definition of *monad* from *Category Theory - Steve Awodey*

Def: A diagram (such as the ones below) is **commutative** when, for each pair of vertices c and c', any two paths formed from directed edges leading from c to c' yield, by composition of labels, equal morphisms from c to c'.

A considerable part of the effectiveness of categorical methods rests on the fact that such diagrams in each situation vividly represent the actions of the arrows at hand.

Def: A **monad** on a category $\mathfrak{C}$ consists of an endofunctor $T : \mathfrak{C} \to \mathfrak{C}$, and natural transformations $\eta : 1_{\mathfrak{C}} \to T$, and $\mu : T^2 \to T$ satisfying the two commutative diagrams below, that is,

$$\mu \circ \mu_T = \mu \circ T_\mu$$

$$\mu \circ \eta_T = 1 = \mu \circ T_\eta$$

Note the formal analogy to the definition of a monoid. In fact, a monad is exactly the same thing as a *monoidal monoid* in the monoidal category $\mathfrak{C}^{\mathfrak{C}}$ with composition as the monoidal product, $G \otimes F = G \circ F$

$$\begin{tikzcd} T^3 \arrow[r, "T_\mu"] \arrow[d, "\mu_T"'] & T^2 \arrow[d, "\mu"] \\ T^2 \arrow[r, "\mu"] & T \end{tikzcd}$$

$$\mu \circ \mu_T = \mu \circ T_\mu$$

$$\begin{tikzcd} T \arrow[r, "\eta_T"] \arrow[dr, "1_T"'] & T^2 \arrow[d, "\mu"] & T \arrow[l, "T_\eta"'] \arrow[dl, "1_T"] \\ & T & \end{tikzcd}$$

$$\mu \circ \eta_T = 1_T = \mu \circ T_\eta$$

Chapter 2

Haskell Monad class

As seen in the previous chapter, monad definition in Mathematics lies beyond a long and winding path (we saw both ends, but the in-between theory was omitted); etimology doesn't help either, leading to:

Monad (n.): "Unity, arithmetical unit", 1610s, from Late Latin *monas* (genitive *monadis*), from Greek *monas* "unit", from *monos*"alone" (see *mono*). In Leibnitz's philosophy, "an ultimate unit of being" (1748). Related: *Monadic.*

So, as even more questions arise, lets sort them up:

2.1 Why? the IO monad

Beyond internally calculating values, we want our programs to interact with the world. The most common beginners' program in any language simply displays a "hello world" greeting on the screen. Here's a Haskell version:

```
Prelude> putStrLn "Hello, World!"
```

So now you should be thinking, "what is the type of the putStrLn function?" It takes a $\boxed{\text{String}}$ and gives... um... what? What do we call that? The program doesn't get something back that it can use in another function. Instead, the result involves having the computer change the screen. In other words, it does something in the world outside of the program. What type could that have? Let's see what GHCi tells us:

```
Prelude> :t putStrLn
putStrLn :: String -> IO ()
```

"IO" stands for "input and output". Wherever there is $\boxed{\text{IO}}$ in a type, interaction with the world outside the program is involved. We'll call these $\boxed{\text{IO}}$ values *actions*. The other part of the $\boxed{\text{IO}}$ type, in this case $\boxed{()}$, is the type of the return value of the action; that is, the type of what it gives back to the program (as opposed to what it does outside the program). $\boxed{()}$ (pronounced as "unit") is a type that only contains one value also called $\boxed{()}$ (effectively a tuple with zero elements). Since $\boxed{\text{putStrLn}}$ sends output to the world but doesn't return anything to the program, $\boxed{()}$ is used as a placeholder. We might read $\boxed{\text{IO ()}}$ as "action which returns $\boxed{()}$".What makes IO actually work? Lots of things happen behind the scenes to take us from $\boxed{\text{putStrLn}}$ to pixels in the screen, but we don't need to understand any of the details to write our programs. A complete Haskell program is actually a big IO action. In a compiled program, this action is called $\boxed{\text{main}}$ and has type $\boxed{\text{IO ()}}$.

From this point of view, to write a Haskell program is to combine actions and functions to form the overall action $\boxed{\text{main}}$ that will be executed when the program is run. The compiler takes care of instructing the computer on how to do this.

2.2 What?

Monads are by no means limited to input and output. Monads support a whole range of things like exceptions, state, non-determinism, continuations, coroutines, and more. In fact, thanks to the versatility of monads, none of these constructs needed to be built into Haskell as a language; instead, they are defined by the standard libraries.

2.2.1 Starring: Monad typeclass

In Haskell, the $\boxed{\text{Monad}}$ type class is used to implement monads. It is provided by the Control.Monad module and included in the Prelude. The class has the following methods:[1]

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

(>>)   :: m a -> m b -> m b
fail   :: String -> m a
```

The core methods are $\boxed{\text{return}}$ and $\boxed{\text{(>>=)}}$ (which is pronounced "bind"). Aside from return and bind, notice the two additional functions $\boxed{\text{(>>)}}$ and $\boxed{\text{fail}}$. The operator $\boxed{\text{(>>)}}$ called "then" is a mere convenience and commonly implemented as

```
m >> n   =   m >>= \_ -> n
```

$\boxed{\text{(>>)}}$ sequences two monadic actions when the second action does not involve the result of the first, which is common for monads like $\boxed{\text{IO}}$. The function $\boxed{\text{fail}}$ handles pattern match failures in $\boxed{\text{do}}$ notation. It's an unfortunate technical necessity and doesn't really have anything to do with monads. You are advised not to call $\boxed{\text{fail}}$ directly in your code.

[1]For the full definition of Monad in the Prelude, look Appendix B

2.3 Pre-example with Maybe

For a concrete example, take the $\boxed{\text{Maybe}}$ monad. The type constructor is $\boxed{\text{m = Maybe}}$, while $\boxed{\text{return}}$ and $\boxed{\text{(>>=)}}$ are defined like this:

```
return :: a -> Maybe a
return x  = Just x

(>>=)  :: Maybe a -> (a -> Maybe b) -> Maybe b
m >>= g = case m of
             Nothing -> Nothing
             Just x  -> g x
```

$\boxed{\text{Maybe}}$ is the monad, and $\boxed{\text{return}}$ brings a value into it by wrapping it with $\boxed{\text{Just}}$. As for $\boxed{\text{(>>=)}}$, it takes a $\boxed{\text{m :: Maybe a}}$ value and a $\boxed{g :: a \to \text{ Maybe b}}$ function. If $\boxed{\text{m}}$ is $\boxed{\text{Nothing}}$, there is nothing to do and the result is $\boxed{\text{Nothing}}$. Otherwise, in the $\boxed{\text{Just x}}$ case, $\boxed{\text{g}}$ is applied to $\boxed{\text{x}}$, the underlying value wrapped in $\boxed{\text{Just}}$, to give a $\boxed{\text{Maybe b}}$ result, which might be $\boxed{\text{Nothing}}$, depending on what $\boxed{\text{g}}$ does to $\boxed{\text{x}}$. To sum it all up, if there is an underlying value in $\boxed{\text{m}}$, we apply $\boxed{\text{g}}$ to it, which brings the underlying value back into the $\boxed{\text{Maybe}}$ monad.

The key first step to understand how $\boxed{\text{return}}$ and $\boxed{\text{(>>=)}}$ work is tracking which values and arguments are monadic and which ones aren't. As in so many other cases, type signatures are our guide to the process.

Motivation: Maybe

To see the usefulness of $\boxed{\text{(>>=)}}$ and the $\boxed{\text{Maybe}}$ monad, consider the following example: Imagine a family database that provides two functions

```
father :: Person -> Maybe Person
mother :: Person -> Maybe Person
```

These look up the name of someone's father or mother. In case our database is missing some information, $\boxed{\text{Maybe}}$ allows us to return a $\boxed{\text{Nothing}}$ value instead of crashing the program. Let's combine our functions to query various grandparents. For instance, the following function looks up the maternal grandfather:

```
maternalGrandfather :: Person -> Maybe Person
maternalGrandfather p =
case mother p of
    Nothing -> Nothing
    Just mom -> father mom
```

Or consider a function that checks whether both grandfathers are in the database:

```
bothGrandfathers :: Person -> Maybe (Person, Person)
bothGrandfathers p =
    case father p of
        Nothing -> Nothing
        Just dad ->
            case father dad of
                Nothing -> Nothing
                Just gf1 ->                    -- 1st grandfather
```

```
                    case mother p of
                        Nothing -> Nothing
                        Just mom ->
                            case father mom of
                                Nothing -> Nothing
                                Just gf2 ->    -- 2nd grandfather
                                    Just (gf1, gf2)
```

What a mouthful! Every single query might fail by returning `Nothing` and the whole function must fail with `Nothing` if that happens. Clearly there as to be a better way to write that instead of repeating the case of `Nothing` again and again! Indeed, that's what the `Maybe` monad is set out to do. For instance, the function retrieving the maternal grandfather has exactly the same structure as the `(>>=)` operator, so we can rewrite it as:

```
    maternalGrandfather p   =   mother p >>= father
```

With the help of lambda expressions and return, we can rewrite the two grandfathers function as well:

```
bothGrandfathers p =
    father p >>=
        (\dad -> father dad >>=
            (\gf1 -> mother p >>=
-- this line works as ‘\_ -> mother p’, but naming gf1 allows later return
                (\mom -> father mom >>=
                     (\gf2 -> return (gf1,gf2) ))))
```

While these nested lambda expressions may look confusing to you, the thing to take away here is that `(>>=)` releases us from listing all the `Nothing`s, shifting the focus back to the interesting part of the code. To be a little more precise: The result of `father p` is a monadic value (in this case, either `Just dad` or `Nothing`, depending on whether `p`'s dad is in the database). As the `father` function takes a regular (non-monadic value), the `(>>=)` feeds `p`'s dad to it as a *non-monadic* value. The result of `father dad` is then monadic again, and the process continues.

So, `(>>=)` helps us pass non-monadic values to functions without leaving a monad. In the case of the `Maybe` monad, the monadic aspect is the qualifier that we don't know with certainty whether the value will be found.

2.3.1 Notions of Computation

We've seen how `(>>=)` and `return` are very handy for removing boilerplate code that crops up when using `Maybe`. That, however, is not enough to justify why monads matter so much. We will continue our monad studies by rewriting the two-grandfathers function using `do` notation with explicit braces and semicolons. Depending on your experience with other programming languages, you may find this very suggestive:

```
    bothGrandfathers p = do {
        dad <- father p;
        gf1 <- father dad;
        mom <- mother p;
        gf2 <- father mom;
        return (gf1, gf2);
    }
```

If this looks like a code snippet of an imperative programming language to you, that's because it is. In particular, this imperative language supports *exceptions*: father and mother are functions that might fail to produce results, i.e. raise an exception, and when that happens, the whole `do`-block will fail, i.e. terminate with an exception.

In other words, the expression `father p`, which has type `Maybe Person`, is interpreted as a statement of an imperative language that returns a `Person` as result. **This is true for all monads: a value of type `M a` is interpreted as a statement of an imperative language that returns a value of type `a` as result; and the semantics of this language are determined by the monad `M`.**[2]

Under this interpretation, the bind operator `(>>=)` is simply a function version of the semicolon. Just like a `let` expression can be written as a function application,

```
let x = foo in x + 3      corresponds to      (\x -> x + 3) foo
```

an assignment and semicolon can be written as the bind operator:

```
x <- foo; return (x + 3)
corresponds to
foo >>= (\x -> return (x + 3))
```

The `return` function lifts a value `a` to `M a`, a full-fledged statement of the imperative language corresponding to the monad `M`.

Different semantics of the imperative language correspond to different monads. The following table shows the classic selection that every Haskell programmer should know. If the idea behind monads is still unclear to you, studying each of the examples in the following chapters will not only give you a well-rounded toolbox but also help you understand the common abstraction behind them.

Monad	Imperative Semantics	Found in Prelude
`Maybe`	Exception (anonymous)	Yes
`Error`	Exception (with error description)	No
`State`	Global state	No
`IO`	Input/Output	Yes
`[]` (lists)	Nondeterminism	Yes
`Reader`	Environment	No
`Writer`	Logger	No

Furthermore, these different semantics need not occur in isolation. As we will see in a few chapters, it is possible to mix and match them by using monad transformers to combine the semantics of multiple monads in a single monad.

[2]By 'semantics', we mean what the language allows you to say. In the case of `Maybe`, the semantics allow us to express failure, as statements may fail to produce a result, leading to the statements that follow it being skipped.

2.4 Who?

The first observation when studying the monad definition in the Prelude is that its a type class, just like $\boxed{\text{Eq}}$, $\boxed{\text{Ord}}$ or $\boxed{\text{Num}}$. As such, instead of *what is a monad?* we should be asking ourselves *what is TO BE monad?* - because that's how classes work and help us, enhancing types with new capabilities; for example, the $\boxed{\text{Eq}}$ and $\boxed{\text{Ord}}$ classes provide comparability between that type elements, and the $\boxed{\text{Num}}$ class allows the use of $\boxed{+}$ or $\boxed{*}$.

In fact, with a little help with the GHCi command $\boxed{\text{:kind}}$ we can already answer the question *what types can be made instance of the Monad class?*. Check it yourself!

```
Prelude> :k Bool
Bool :: *
Prelude> :k Int
Int :: *
Prelude> :k []
[] :: * -> *
Prelude> :k [Int]
[Int] :: *
Prelude> :k Maybe
Maybe :: * -> *
Prelude> :k (,,,,,)
(,,,,,) :: * -> * -> * -> * -> * -> * -> *
Prelude> :k Eq
Eq :: * -> Constraint
Prelude> :k Ord
Ord :: * -> Constraint
Prelude> :k Num
Num :: * -> Constraint
Prelude> :k Show
Show :: * -> Constraint
Prelude> :k Functor
Functor :: (* -> *) -> Constraint
Prelude> :k Monad
Monad :: (* -> *) -> Constraint

Prelude> :t Constraint

<interactive>:1:1: Not in scope: data constructor ‘Constraint’
Prelude> :k Constraint

<interactive>:1:1:
    Not in scope: type constructor or class ‘Constraint’
Prelude> :m GHC.Prim
Prelude GHC.Prim> :k Constraint
Constraint :: BOX
Prelude GHC.Prim> :k BOX
BOX :: BOX

Prelude> :m Data.Monoid
Prelude Data.Monoid> :k Monoid
Monoid :: * -> Constraint
```

Looking closely the kind of Monad, we get that **only 1-parameterized types** are allowed to be instantiated in the $\boxed{\text{Monad}}$ class. This is, types like $\boxed{\text{Maybe a}}$, $\boxed{\text{[a]}}$ or $\boxed{\text{(a)}}$; but not $\boxed{\text{Int}}$, $\boxed{\text{Bool}}$ or $\boxed{\text{Either a b}}$ (however, $\boxed{\text{Either Int a}}$ will do the trick). As soon as GHCi meets the "$\boxed{\text{instance Monad Int where}}$" line, the following error will be displayed:

```
The first argument of ‘Monad’ should have kind ‘* -> *’,
 but ‘Int’ has kind ‘*’
 In the instance declaration for ‘Monad Int’
```

You don't program with kinds: the compiler infers them for itself. But if you get parameterized types wrong then the compiler will report a kind error.

2.5 How?

2.5.1 The Rules

In Haskell, every instance of the `Monad` type class (and thus all implementations of bind `(>>=)` and `return`) must obey the following three laws:

```
m >>= return      =   m                         -- right unit
return x >>= f    =   f x                       -- left unit

(m >>= f) >>= g   =   m >>= (\x -> f x >>= g)   -- associativity
```

The behavior of `return` is specified by the left and right unit laws. They state that `return` doesn't perform any computation, it just collects values.

The law of associativity makes sure that (like the semicolon) the bind operator `(>>=)` only cares about the order of computations, not about their nesting. The associativity of the *then* operator `(>>)` is a special case:

```
(m >> n) >> o  =  m >> (n >> o)
```

2.5.2 Monadic composition

It is easier to picture the associativity of bind by recasting the law as

```
(f >=> g) >=> h  =  f >=> (g >=> h)
```

where `(>=>)` is the **monad composition operator**, a close analogue of the function composition operator `(.)`, only with flipped arguments. It is defined as:

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

We can also flip monad composition to go the other direction using `(<=<)`.

2.5.3 Alternative definitions

Monads originally come from a branch of mathematics called Category Theory. Fortunately, it is entirely unnecessary to understand category theory in order to understand and use monads in Haskell. The definition of monads in Category Theory actually uses a slightly different presentation. Translated into Haskell, this presentation gives an alternative yet equivalent definition of a monad which can give us some additional insight.

So far, we have defined monads in terms of `(>>=)` and `return`. The alternative definition, instead, starts with monads as functors with two additional combinators:

```
fmap   :: (a -> b) -> M a -> M b  -- functor
return :: a -> M a
join   :: M (M a) -> M a
```

(As will be discussed in the section on the functor class, a functor `M` can be thought of as container, so that `M a` " contains" values of type `a`, with a corresponding mapping function, i.e. `fmap`, that allows functions to be applied to values inside it.) Under this interpretation, the functions behave as follows:

- `fmap` applies a given function to every element in a container

- $\boxed{\text{return}}$ packages an element into a container
- $\boxed{\text{join}}$ takes a container of containers and flattens it into a single container

With these functions, the bind combinator can be defined as follows:

```
m >>= g = join (fmap g m)
```

Likewise, we could give a definition of $\boxed{\text{fmap}}$ and $\boxed{\text{join}}$ in terms of $\boxed{(>>=)}$ and $\boxed{\text{return}}$:

```
fmap f x = x >>= (return . f)
join x   = x >>= id
```

At this point we might, with good reason, conclude that all monads are by definition functors as well. That is indeed the case, both according to category theory and when programming in Haskell. A final observation is that $\boxed{\text{Control.Monad}}$ defines $\boxed{\text{liftM}}$, a function with a strangely familiar type signature...

```
liftM :: (Monad m) => (a1 -> r) -> m a1 -> m r
```

As you might suspect, $\boxed{\text{liftM}}$ is merely $\boxed{\text{fmap}}$ implemented with $\boxed{(>>=)}$ and $\boxed{\text{return}}$, just as we have done above. For a properly implemented monad with a matching $\boxed{\text{Functor}}$ (that is, any *sensible* monad) $\boxed{\text{liftM}}$ and $\boxed{\text{fmap}}$ are interchangeable.

2.5.4 Note: avoiding the prerequisites

While following the next few chapters, you will likely want to write instances of $\boxed{\text{Monad}}$ and try them out, be it to run the examples in this text or to do other experiments you might think of. However, $\boxed{\text{Applicative}}$ being a superclass of $\boxed{\text{Monad}}$ means that implementing $\boxed{\text{Monad}}$ requires providing $\boxed{\text{Functor}}$ and $\boxed{\text{Applicative}}$ instances as well. At this point of the report, that would be somewhat of an annoyance, especially given that we have not discussed $\boxed{\text{Applicative}}$ yet! As a workaround, once you have written the $\boxed{\text{Monad}}$ instance you can use the functions in $\boxed{\text{Control.Monad}}$ to fill in the $\boxed{\text{Functor}}$ and $\boxed{\text{Applicative}}$ implementations, as follows:

```
instance Functor Foo where
fmap = liftM

instance Applicative Foo where
pure = return
(<*>) = ap
```

We will find out what $\boxed{\text{pure}}$, $\boxed{(<*>)}$ and $\boxed{\text{ap}}$ are in due course.

2.6 Prerequisites: Functor and Applicative typeclasses

implementing Monad requires providing Functor and Applicative instances as well.

Functor class

Functor is a Prelude class for types which can be mapped over. It has a single method, called fmap. The class is defined as follows:

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

Some examples:

The Maybe functor

```
instance Functor Maybe where
         fmap f Nothing  = Nothing
         fmap f (Just x)   =  Just (f x)
```

The List functor

```
instance Functor [] where
         fmap = map
```

The Tree functor

```
instance Functor Tree where
         fmap f (Leaf x) = Leaf (f x)
         fmap f (Branch left right) = Branch (fmap f left) (fmap f right)
```

The functor laws When providing a new instance of Functor, you should ensure it satisfies the two functor laws. There is nothing mysterious about these laws; their role is to guarantee fmap behaves sanely and actually performs a mapping operation (as opposed to some other nonsense).[3] The laws are:

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

[3]Some examples of nonsense that the laws rule out: removing or adding elements from a list, reversing a list, changing a Just-value into a Nothing

Applicative functors

Like monads, applicative functors are functors with extra laws and operations; in fact, `Applicative` is an intermediate class between `Functor` and `Monad`. It enables the *applicative style*, a convenient way of structuring functorial computations, and also provides means to express a number of important patterns.

Note: For extra convenience, `fmap` has an infix synonym, `(<$>)`. It often helps readability, and also suggests how `fmap` can be seen as a different kind of function application.

```
Prelude> negate <$> Just 2
Just (-2)
```

As useful as it is, `fmap` isn't much help if we want to apply a function of two arguments to functorial values. For instance, how could we sum `Just 2` and `Just 3`? The brute force approach would be extracting the values from the `Maybe` wrapper. That, however, would mean having to do tedious checks for `Nothing`. Even worse: in a different `Functor` extracting the value might not even be an option (just think about IO).

We could use `fmap` to partially apply `(+)` to the first argument:

```
Prelude> :t (+) <$> Just 2
(+) <$> Just 2 :: Num a => Maybe (a -> a)
```

But now we are stuck: we have a function and a value both wrapped in `Maybe`, and no way of applying one to the other. What we would like to have is an operator with a type akin to

```
f (a -> b) -> f a -> f b
```

to apply functions in the context of a functor. That operator is called `(<*>)`, check this:

```
Prelude> (+) <$> Just 2 <*> Just 3
Just 5

Prelude> :t (<*>)
(<*>) :: Applicative f => f (a -> b) -> f a -> f b
```

`(<*>)` is one of the methods of `Applicative` the type class of *applicative functors* - functors that support function application within their contexts. Expressions such as

```
(+) <$> Just 2 <*> Just 3
```

are said to be written in *applicative style*, which is as close as we can get to regular function application while working with a functor. If you pretend for a moment the `(<$>)`, `(<*>)` and `Just` aren't there, our example looks just like `(+) 2 3`.

2.6.1 Applicative functor laws

The definition of $\boxed{\text{Applicative}}$ is:

```
class (Functor f) => Applicative f where
    pure  :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

Beyond $\boxed{(<*>)}$, the class has a second method, $\boxed{\text{pure}}$, which brings arbitrary values into the functor. As an example, let's have a look at the $\boxed{\text{Maybe}}$ instance:

```
instance Applicative Maybe where
    pure                = Just
    (Just f) <*> (Just x) = Just (f x)
    _        <*> _        = Nothing
```

It doesn't do anything surprising: $\boxed{\text{pure}}$ wraps the value with $\boxed{\text{Just}}$; $\boxed{(<*>)}$ applies the function to the value if both exists, and results in $\boxed{\text{Nothing}}$ otherwise.

Note For the lack of a better shorthand, in what follows we will use the word *morphism* to refer to the values to the left of $\boxed{(<*>)}$, which fit the type `Applicative f => f (a -> b)`; that is, the function-like things inserted into an applicative functor.

Just like $\boxed{\text{Functor}}$, $\boxed{\text{Applicative}}$ has a set of laws which reasonable instances should follow. They are:

```
pure id <*> v = v                            -- Identity
pure f <*> pure x = pure (f x)               -- Homomorphism
u <*> pure y = pure ($ y) <*> u              -- Interchange
pure (.) <*> u <*> v <*> w = u <*> (v <*> w) -- Composition
```

Those laws are a bit of a mouthful. They become easier to understand if you think of $\boxed{\text{pure}}$ as a way to inject values into the functor in a default, featureless way, so that the result is as close as possible to the plain value. Thus:

- The identity law says that applying the $\boxed{\text{pure id}}$ morphism does nothing, exactly like with the plain $\boxed{\text{id}}$ function.
- The homomorphism law says that applying a "pure" function to a "pure" value is the same than applying the function to the value in the normal way and then using $\boxed{\text{pure}}$ on the result. In a sense, that means $\boxed{\text{pure}}$ preserves function application.
- The interchange law says that applying a morphism to a "pure" value $\boxed{\text{pure y}}$ is the same as applying pure $\boxed{(\$\ y)}$ to the morphism. No surprises there - $\boxed{(\$\ y)}$ is the function that supplies $\boxed{\text{y}}$ as argument to another function.
- The composition law says that if $\boxed{(<*>)}$ is used to compose morphisms the composition is associative, like plain function composition.[4]

There is also a bonus law about the relation between $\boxed{\text{fmap}}$ and $\boxed{(<*>)}$:

[4] With plain functions, we have $\boxed{\text{h . g . f} = \text{(h . g) . f} = \text{h . (g . f)}}$ That is why we never bother to use parentheses in the middle of $\boxed{(.)}$ chains.

```
fmap f x = pure f <*> x                          -- fmap
```

Applying a "pure" function with $\boxed{(<*>)}$ is equivalent to using $\boxed{\text{fmap}}$. **This law is a consequence of the other ones, so you need not bother with proving it when writing instances of $\boxed{\textbf{Applicative}}$.**

2.7 *do* notation

Using do blocks as an alternative monad syntax was introduced with an IO example. Since the following examples all involve IO, we will refer to the computations/monadic values as *actions* (as we did in the earlier parts of the report). Of course, do works with any monad; there is nothing specific about IO in how it works.

2.7.1 Translating the *then* operator

The (>>) *(then)* operator works almost identically in do notation and in unsugared code. For example, suppose we have a chain of actions like the following one:

```
putStr "Hello" >>
putStr " " >>
putStr "world!" >>
putStr "\n"
```

We can rewrite that in do notation as follows:

```
do putStr "Hello"
   putStr " "
   putStr "world!"
   putStr "\n"
```

This sequence of instructions nearly matches that in any imperative language. In Haskell, we can chain any actions as long as all of them are in the same monad. In the context of the IO monad, the actions include writing to a file, opening a network connection, or asking the user for input.

Here's the step-by-step translation of do notation to unsugared Haskell code:

```
do action1
   action2
   action3
```

becomes

```
action1 >>
do action2
   action3
```

and so on, until the do block is empty.

2.7.2 Translating the *bind* operator

The (>>=) is a bit more difficult to translate from and to do notation. (>>=) passes a value, namely the result of an action or function, downstream in the binding sequence. do notation assigns a variable name to the passed value using the `<-`.

```
do x1 <- action1
   x2 <- action2
   action3 x1 x2
```

x1 and x2 are the results of action1 and action2. If, for instance, action1 is an IO Integer then x1 will be bound to an Integer. The stored values are passed as arguments to action3, which returns a third action. The do block is broadly equivalent to the following vanilla Haskell snippet:

```
action1 >>= \ x1 -> action2 >>= \ x2 -> action3 x1 x2
```

The second argument of $\boxed{(>>=)}$ is a function specifying what to do with the result of the action passed as first argument. Thus, chains of lambdas pass the results downstream. Remember that, without extra parentheses, a lambda extends all the way to the end of the expression. $\boxed{\text{x1}}$ is still in scope at the point we call $\boxed{\text{action3}}$. We can rewrite the chain of lambdas more legibly by using separate lines and indentation:

```
action1
  >>=
    \ x1 -> action2
      >>=
        \ x2 -> action3 x1 x2
```

That shows the scope of each lambda function clearly. To group things more like the $\boxed{\text{do}}$ notation, we could show it like this:

```
action1 >>= \ x1 ->
  action2 >>= \ x2 ->
    action3 x1 x2
```

These presentation differences are only a matter of assisting readability. Actually, the indentation isn't needed in this case. This is equally valid:

```
action1 >>= \ x1 ->
action2 >>= \ x2 ->
action3 x1 x2
```

2.7.3 The *fail* method

Above, we said the snippet with lambdas was "broadly equivalent" to the $\boxed{\text{do}}$ block. The translation is not exact because the $\boxed{\text{do}}$ notation adds special handling of pattern match failures. When placed at the left of either $\boxed{<-}$ or $\boxed{->}$, $\boxed{\text{x1}}$ and $\boxed{\text{x2}}$ are patterns being matched. Therefore, if $\boxed{\text{action1}}$ returned a $\boxed{\text{Maybe Integer}}$ we could write a $\boxed{\text{do}}$ block like this...

```
do Just x1 <- action1
   x2      <- action2
   action3 x1 x2
```

...and $\boxed{\text{x1}}$ be an $\boxed{\text{Integer}}$. In such a case, what happens if $\boxed{\text{action1}}$ returns $\boxed{\text{Nothing}}$? Ordinarily, the program would crash with a non-exhaustive patterns error, just like the one we get when calling $\boxed{\text{head}}$ on an empty list. With $\boxed{\text{do}}$ notation, however, failures are handled with the $\boxed{\text{fail}}$ method for the relevant monad. The $\boxed{\text{do}}$ block above translates to:

```
action1 >>= f
where f (Just x1) = do x2 <- action2
                       action3 x1 x2
      f _         = fail "..." -- A compiler-generated message.
```

What $\boxed{\text{fail}}$ actually does depends on the monad instance. Though it will often rethrow the pattern matching error, monads that incorporate some sort of error handling may deal with the failure in their own specific ways. For instance, $\boxed{\text{Maybe}}$ has $\boxed{\text{fail \_ = Nothing}}$; analogously, for the list monad $\boxed{\text{fail \_ = []}}$.[5]

[5]This explains why pattern matching failures in list comprehensions are silently ignored.

The fail method is an artifact of $\boxed{\text{do}}$ notation. Rather than calling $\boxed{\text{fail}}$ directly, you should rely on automatic handling of pattern match failures whenever you are sure that $\boxed{\text{fail}}$ will do something sensible for the monad you are using.

2.7.4 Example: user-interactive program

Note for non-ghci users We are going to interact with the user, so we will use $\boxed{\text{putStr}}$ and $\boxed{\text{getLine}}$ alternately. To avoid unexpected results in the output, we must disable output buffering when importing $\boxed{\text{System.IO}}$.

To do this, put $\boxed{\text{hSetBuffering stdout NoBuffering}}$ at the top of your code. To handle this otherwise, you would explicitly flush the output buffer before each interaction with the user (namely a $\boxed{\text{getLine}}$) using $\boxed{\text{hFlush stdout}}$. If you are testing this code with ghci, you don't have such problems.

Consider this simple program that asks the user for their first and last names:

```
nameDo :: IO ()
nameDo = do putStr "What is your first name? "
            first <- getLine
            putStr "And your last name? "
            last <- getLine
            let full = first ++ " " ++ last
            putStrLn ("Pleased to meet you, " ++ full ++ "!")
```

A possible translation into vanilla monadic code:

```
nameLambda :: IO ()
nameLambda = putStr "What is your first name? " >>
             getLine >>= \ first ->
             putStr "And your last name? ">>
             getLine >>= \ last ->
             let full = first ++ " " ++ last
             in putStrLn ("Pleased to meet you, " ++ full ++ "!")
```

In cases like this, where we just want to chain several actions, the imperative style of $\boxed{\text{do}}$ notation feels natural and convenient. In comparison, monadic code with explicit binds and lambdas is something of an acquired taste.

Notice that the first example above includes a $\boxed{\text{let}}$ statement in the $\boxed{\text{do}}$ block. The de-sugared version is simply a regular $\boxed{\text{let}}$ expression where the $\boxed{\text{in}}$ part is whatever follows from the $\boxed{\text{do}}$ syntax.

2.7.5 Returning values

The last statement in $\boxed{\text{do}}$ notation is the overall result of the $\boxed{\text{do}}$ block. In the previous example, the result was of the type $\boxed{\text{IO ()}}$, i.e. an empty value in the $\boxed{\text{IO}}$ monad.

Suppose that we want to rewrite the example but return an $\boxed{\text{IO String}}$ with the acquired name. All we need to do is add a $\boxed{\text{return}}$:

```
nameReturn :: IO String
nameReturn = do putStr "What is your first name? "
                first <- getLine
                putStr "And your last name? "
```

```
                    last <- getLine
                    let full = first ++ " " ++ last
                    putStrLn ("Pleased to meet you, " ++ full ++ "!")
                    return full
```

This example will “return” the full name as a string inside the `IO` monad, which can then be utilized downstream elsewhere:

```
greetAndSeeYou :: IO ()
greetAndSeeYou = do name <- nameReturn
                    putStrLn ("See you, " ++ name ++ "!")
```

Here, `nameReturn` will be run and the returned result (called “full” in the `nameReturn` function) will be assigned to the variable “name” in our new function. The greeting part of `nameReturn` will be printed to the screen because that is part of the calculation process. Then, the additional “see you” message will print as well, and the final returned value is back to being `IO ()`.

If you know imperative languages like C, you might think `return` in Haskell matches `return` elsewhere. A small variation on the example will dispel that impression:

```
nameReturnAndCarryOn = do
  putStr "What is your first name? "
  first <- getLine
  putStr "And your last name? "
  last <- getLine
  let full = first++" "++last
  putStrLn ("Pleased to meet you, "++full++"!")
  return full
  putStrLn "I am not finished yet!"
```

The string in the extra line *will* be printed out because `return` is not a final statement interrupting the flow (as it would be in C and other languages). Indeed, the type of `nameReturnAndCarryOn` is `IO ()`, - the type of the final `putStrLn` action. After the function is called, the `IO String` created by the `return full` will disappear without a trace.

2.7.6 Just sugar

As a syntactical convenience, `do` notation does not add anything essential, but it is often preferable for clarity and style. However, `do` is never used for a single action. The Haskell “Hello world” is simply:

```
main = putStrLn "Hello world!"
```

Snippets like this one are totally redundant:

```
fooRedundant = do x <- bar
                  return x
```

Thanks to the monad laws, we can and should write simply:

```
foo = bar
```

A subtle but crucial point relates to function composition: As we already know, the `greetAndSeeYou` action in the section just above could be rewritten as:

```
greetAndSeeYou :: IO ()
greetAndSeeYou =
  nameReturn >>= \ name -> putStrLn ("See you, " ++ name ++ "!")
```

While you might find the lambda a little unsightly, suppose we had a $\boxed{\text{printSeeYou}}$ function defined elsewhere:

```
printSeeYou :: String -> IO ()
printSeeYou name = putStrLn ("See you, " ++ name ++ "!")
```

Now, we can have a clean function definition with neither lambdas or $\boxed{\text{do}}$:

```
greetAndSeeYou :: IO ()
greetAndSeeYou = nameReturn >>= printSeeYou
```

Or, if we have a *non-monadic* $\boxed{\text{seeYou}}$ function:

```
seeYou :: String -> String
seeYou name = "See you, " ++ name ++ "!"
```

Then we can write:

```
-- Reminder: fmap f m == m >>= return . f == liftM f m
greetAndSeeYou :: IO ()
greetAndSeeYou = fmap seeYou nameReturn >>= putStrLn
```

Keep this last example with $\boxed{\text{fmap}}$ in mind; we will soon return to using non-monadic functions in monadic code, and $\boxed{\text{fmap}}$ will be useful there.

2.8 Additive monads (MonadPlus)

In our studies so far, we saw that the $\boxed{\text{Maybe}}$ and list monads both represent the number of results a computation can have. That is, you use $\boxed{\text{Maybe}}$ when you want to indicate that a computation can fail somehow (i.e. it can have 0 results or 1 result), and you use the list monad when you want to indicate a computation could have many valid answers ranging from 0 results to many results.

Given two computations in one of these monads, it might be interesting to amalgamate *all* valid solutions into a single result. For example, within the list monad, we can concatenate two lists of valid solutions.

2.8.1 *MonadPlus* definition

$\boxed{\text{MonadPlus}}$ defines two methods. $\boxed{\text{mzero}}$ is the monadic value standing for zero results; while $\boxed{\text{mplus}}$ is a binary function which combines two computations.

```
class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

Here are the two instance declarations for $\boxed{\text{Maybe}}$ and the list monad:

```
instance MonadPlus [] where
  mzero = []
  mplus = (++)


instance MonadPlus Maybe where
  mzero                   = Nothing
  Nothing `mplus` Nothing = Nothing -- 0 solutions + 0 solutions = 0 solutions
  Just x  `mplus` Nothing = Just x  -- 1 solution  + 0 solutions = 1 solution
  Nothing `mplus` Just x  = Just x  -- 0 solutions + 1 solution  = 1 solution
  Just x  `mplus` Just y  = Just x  -- 1 solution  + 1 solution  = 2 solutions,
                                    -- but Maybe can only have up to one solution,
                                    -- so we disregard the second one.
```

Also, if you import $\boxed{\text{Control.Monad.Error}}$, then $\boxed{\text{(Either e)}}$ becomes an instance:

```
instance (Error e) => MonadPlus (Either e) where
  mzero            = Left noMsg
  Left _  `mplus` n = n
  Right x `mplus` _ = Right x
```

Like $\boxed{\text{Maybe}}$, $\boxed{\text{(Either e)}}$ represents computations that can fail. Unlike $\boxed{\text{Maybe}}$, $\boxed{\text{(Either e)}}$ allows the failing computations to include an error "message" (which is usually a $\boxed{\text{String}}$). Typically, $\boxed{\text{Left s}}$ means a failed computation carrying an error message $\boxed{\text{s}}$, and $\boxed{\text{Right x}}$ means a successful computation with result $\boxed{\text{x}}$.

2.8.2 Example: parallel parsing

Traditional input parsing involves functions which consume an input one character at a time. That is, a parsing function takes an input string and chops off (i.e. 'consumes') characters from the front if they satisfy certain criteria. For example, you could write a function which consumes one uppercase character. If the characters on the front of the string don't satisfy the given criteria, the parser has *failed*; so such functions are candidates for Maybe.

Let's use mplus to run two parsers *in parallel*. That is, we use the result of the first one if it succeeds, and otherwise, we use the result of the second. If both fail, then our whole parser returns Nothing.

In the example below, we consume a digit in the input and return the digit that was parsed.

```
digit :: Int -> String -> Maybe Int
digit i s | i > 9 || i < 0 = Nothing
          | otherwise      = do
  let (c:_) = s
  if [c] == show i then Just i else Nothing
```

Our guards assure that the Int we are checking for is a single digit. Otherwise, we are just checking that the first character of our String matches the digit we are checking for. If it passes, we return the digit wrapped in a Just. The do-block assures that any failed pattern match will result in returning Nothing.

We can use our digit function with mplus to parse Strings of binary digits:

```
binChar :: String -> Maybe Int
binChar s = digit 0 s `mplus` digit 1 s
```

Parser libraries often make use of MonadPlus in this way. If you are curious, check the (+++) operator in Text.ParserCombinators.ReadP, or (<|>) in Text.ParserCombinators.Parsec.Prim.

2.8.3 The MonadPlus laws

Instances of MonadPlus are required to fulfill several rules, just as instances of Monad are required to fulfill the three monad laws. Unfortunately, the MonadPlus laws aren't fully agreed on. The most common approach says that mzero and mplus form a *monoid*. By that, we mean:

```
-- mzero is a neutral element
mzero `mplus` m  =  m
m `mplus` mzero  =  m
-- mplus is associative
-- (but not all instances obey this law because it makes some infinite structures impossible)
m `mplus` (n `mplus` o)  =  (m `mplus` n) `mplus` o
```

The Haddock documentation for Control.Monad quotes additional laws:

```
mzero >>= f  =  mzero
m >> mzero   =  mzero
```

And the HaskellWiki page cites another (with controversy):

```
(m `mplus` n) >>= k   =   (m >>= k) `mplus` (n >>= k)
```

There are even more sets of laws available. Sometimes monads like IO are used as a MonadPlus. Consult All About Monads and the Haskell Wiki page on MonadPlus for more information about such issues.

2.8.4 Useful functions

Beyond the basic mplus and mzero, there are two other general-purpose functions involving MonadPlus:

msum

A common task when working with MonadPlus: take a list of monadic values, e.g. [Maybe a] or [[a]], and fold it down with mplus. The function msum fulfills this role:

```
msum :: MonadPlus m => [m a] -> m a
msum = foldr mplus mzero
```

In a sense, msum generalizes the list-specific concat operation. Indeed, the two are equivalent when working on lists. For Maybe, msum finds the first Just x in the list and returns Nothing if there aren't any.

guard

When discussing the list monad we note how similar it is to list comprehensions, but we didn't discuss how to mirror list comprehension filtering. The guard function allows us to do exactly that.

Consider the following comprehension which retrieves all pythagorean triples (i.e. trios of integer numbers which work as the lengths of the sides for a right triangle). First we'll examine the brute-force approach. We'll use a boolean condition for filtering; namely, Pythagoras' theorem:

```
pythags = [ (x, y, z) | z <- [1..], x <- [1..z], y <- [x..z], x^2 + y^2 == z^2 ]
```

The translation of the comprehension above to the list monad is:

```
pythags = do
  z <- [1..]
  x <- [1..z]
  y <- [x..z]
  guard (x^2 + y^2 == z^2)
  return (x, y, z)
```

The guard function works like this:

```
guard :: MonadPlus m => Bool -> m ()
guard True  = return ()
guard _ = mzero
```

Concretely, guard will reduce a do-block to mzero if its predicate is False. Given the first law stated in the 'MonadPlus laws' section above, an mzero on the left-hand side of a (>>=) operation will produce mzero again. As do-blocks are decomposed to lots of expressions joined up by (>>=), an mzero at any point will cause the entire do-block to become mzero.

To further illustrate, we will examine guard in the special case of the list monad, extending on the pythags function above. First, here is guard defined for the list monad:

```
guard :: Bool -> [()]
guard True  = [()]
guard _ = []
```

Basically, guard *blocks off* a route. In pythags, we want to block off all the routes (or combinations of x, y and z) where x^2 + y^2 == z^2 is False. Let's look at the expansion of the above do-block to see how it works:

```
pythags =
  [1..] >>= \z ->
  [1..z] >>= \x ->
  [x..z] >>= \y ->
  guard (x^2 + y^2 == z^2) >>= \_ ->
  return (x, y, z)
```

Replacing `(>>=)` and `return` with their definitions for the list monad (and using some let-bindings to keep it readable), we obtain:

```
pythags =
 let ret x y z = [(x, y, z)]
     gd  z x y = concatMap (\_ -> ret x y z) (guard $ x^2 + y^2 == z^2)
     doY z x   = concatMap (gd  z x) [x..z]
     doX z     = concatMap (doY z  ) [1..z]
     doZ       = concatMap (doX    ) [1..]
 in doZ
```

Remember that `guard` returns the empty list in the case of its argument being `False`. Mapping across the empty list produces the empty list, no matter what function you pass in. So the empty list produced by the call to `guard` in the binding of `gd` will cause `gd` to be the empty list, and therefore `ret` to be the empty list.

To understand why this matters, think about list-computations as a tree. With our Pythagorean triple algorithm, we need a branch starting from the top for every choice of `z`, then a branch from each of these branches for every value of `x`, then from each of these, a branch for every value of `y`. So the tree looks like this:

```
start
   |____________________________________________ ...
   |                     |                    |
x  1                     2                    3
   |_______________ ...  |_______________ ... |_______________ ...
   |      |      |       |      |      |      |      |      |
y  1      2      3       2      3      4      3      4      5
   |___...|___...|___... |___...|___...|___...|___...|___...|___...
   | | |  | | |  | | |   | | |  | | |  | | |  | | |  | | |  | | |
z  1 2 3  2 3 4  3 4 5   2 3 4  3 4 5  4 5 6  3 4 5  4 5 6  5 6 7
```

Each combination of x, y and z represents a route through the tree. Once all the functions have been applied, each branch is concatenated together, starting from the bottom. Any route where our predicate doesn't hold evaluates to an empty list, and so has no impact on this concat operation.

2.8.5 Relationship with monoids

When discussing the MonadPlus laws, we alluded to the mathematical concept of monoids. It turns out that there is a `Monoid` class in Haskell, defined in `Data.Monoid`. A fuller presentation of is given in an appendix. For now, a minimal definition of `Monoid` implements two methods; namely, a neutral element (or 'zero') and an associative binary operation (or 'plus').

```
class Monoid m where
  mempty  :: m
  mappend :: m -> m -> m
```

For example, lists form a simple monoid:

```haskell
instance Monoid [a] where
  mempty  = []
  mappend = (++)
```

Sounds familiar, doesn't it? In spite of the uncanny resemblance to MonadPlus, there is a subtle yet key difference. Note the usage of [a] instead of [ ] in the instance declaration. Monoids are not necessarily "containers" of anything or parametrically polymorphic. For instance, the integer numbers on form a monoid under addition with 0 as neutral element.

In any case, MonadPlus instances look very similar to monoids, as both feature concepts of zero and plus. Indeed, we could even make MonadPlus a subclass of Monoid if it were worth the trouble:

```haskell
instance MonadPlus m => Monoid (m a) where
  mempty  = mzero
  mappend = mplus
```

Note Due to the "free" type variable a in the instance definition, the snippet above is not valid Haskell 98. If you want to test it, you will have to enable the GHC *language extension* FlexibleInstances:

- If you are testing with GHCi, start it with the command line option `-XFlexibleInstances` or interactively type :set -XFlexibleInstances.
- Alternatively, if you are running a compiled program, add `{-# LANGUAGE FlexibleInstances #-}` to the top of your source file.

Again, Monoids and MonadPlus work at different levels. As noted before, there is no requirement for monoids to be parameterized in relation to "contained" or related type. More formally, monoids have kind $*$, but instances of MonadPlus (which are monads) have kind $* -> *$.

2.9 Monad transformers

We have seen how monads can help handling $\boxed{\text{IO}}$ actions, $\boxed{\text{Maybe}}$, lists, and state. With monads providing a common way to use such useful general-purpose tools, a natural thing we might want to do is using the capabilities of *several* monads at once. For instance, a function could use both I/O and $\boxed{\text{Maybe}}$ exception handling. While a type like $\boxed{\text{IO (Maybe a)}}$ would work just fine, it would force us to do pattern matching within $\boxed{\text{IO}}$ do-blocks to extract values, something that the $\boxed{\text{Maybe}}$ monad was meant to spare us from.

Enter **monad transformers**: special types that allow us to roll two monads into a single one that shares the behavior of both.

2.9.1 Passphrase validation

Consider a real-life problem for IT staff worldwide: getting users to create strong passphrases. One approach: force the user to enter a minimum length with various irritating requirements (such as at least one capital letter, one number, one non-alphanumeric character, etc.)

Here's a Haskell function to acquire a passphrase from a user:

```
getPassphrase :: IO (Maybe String)
getPassphrase = do s <- getLine
                   if isValid s then return $ Just s
                                else return Nothing

-- The validation test could be anything we want it to be.
isValid :: String -> Bool
isValid s = length s >= 8
            && any isAlpha s
            && any isNumber s
            && any isPunctuation s
```

First and foremost, $\boxed{\text{getPassphrase}}$ is an $\boxed{\text{IO}}$ action, as it needs to get input from the user. We also use $\boxed{\text{Maybe}}$, as we intend to return $\boxed{\text{Nothing}}$ in case the password does not pass the $\boxed{\text{isValid}}$. Note, however, that we aren't actually using $\boxed{\text{Maybe}}$ as a monad here: the $\boxed{\text{do}}$ block is in the $\boxed{\text{IO}}$ monad, and we just happen to $\boxed{\text{return}}$ a $\boxed{\text{Maybe}}$ value into it.

Monad transformers not only make it easier to write $\boxed{\text{getPassphrase}}$ but also simplify all the code instances. Our passphrase acquisition program could continue like this:

```
askPassphrase :: IO ()
askPassphrase = do putStrLn "Insert your new passphrase:"
                   maybe_value <- getPassphrase
                   if isJust maybe_value
                     then do putStrLn "Storing in database..." -- do stuff
                     else putStrLn "Passphrase invalid."
```

The code uses one line to generate the $\boxed{\text{maybe\_value}}$ variable followed by further validation of the passphrase.With monad transformers, we will be able to extract the passphrase in one go - without any pattern matching or equivalent bureaucracy like $\boxed{\text{isJust}}$. The gains for our simple example might seem small but will scale up for more complex situations.

2.9.2 A simple monad transformer: MaybeT

To simplify `getPassphrase` and all the code that uses it, we will define a *monad transformer* that gives the `IO` monad some characteristics of the `Maybe` monad; we will call it `MaybeT`. That follows a convention where monad transformers have a "`T`" appended to the name of the monad whose characteristics they provide.

`MaybeT` is a wrapper around `m (Maybe a)`, where `m` can be any monad (`IO` in our example):

```
newtype MaybeT m a = MaybeT { runMaybeT :: m (Maybe a) }
```

This data type definition specifies a `MaybeT` type constructor, parameterized over `m`, with a term constructor, also called `MaybeT`, and a convenient accessor function `runMaybeT`, with which we can access the underlying representation.

The whole point of monad transformers is that *they are monads themselves*; and so we need to make `MaybeT m` an instance of the `Monad` class:

```
instance Monad m => Monad (MaybeT m) where
    return  = MaybeT . return . Just
```

It would also have been possible (though arguably less readable) to write `return = MaybeT . return . return`.

As in all monads, the bind operator is the heart of the transformer.

```
-- The signature of (>>=), specialized to MaybeT m
(>>=) :: MaybeT m a -> (a -> MaybeT m b) -> MaybeT m b

x >>= f = MaybeT $ do maybe_value <- runMaybeT x
                      case maybe_value of
                           Nothing    -> return Nothing
                           Just value -> runMaybeT $ f value
```

Starting from the first line of the `do` block:

1. First, the `runMaybeT` accessor unwraps `x` into an `m (Maybe a)` computation. That shows us that the whole `do` block is in `m`.

2. Still in the first line, `<-` extracts a `Maybe a` value from the unwrapped computation.

3. The `case` statement tests `maybe_value`:

   - With `Nothing`, we return `Nothing` into `m`;

   - With `Just`, we apply `f` to the `value` from the `Just`. Since `f` has `MaybeT m b` as result type, we need an extra `runMaybeT` to put the result back into the `m` monad.

4. Finally, the `do` block as a whole has `m (Maybe b)` type; so it is wrapped with the `MaybeT` constructor.

It may look a bit complicated; but aside from the copious amounts of wrapping and unwrapping, the implementation does the same as the familiar bind operator of `Maybe`:

```
-- (>>=) for the Maybe monad
maybe_value >>= f = case maybe_value of
                        Nothing -> Nothing
                        Just value -> f value
```

Why use the MaybeT constructor before the do block while we have the accessor runMaybeT within do? Well, the do block must be in the m monad, not in MaybeT m (which lacks a defined bind operator at this point).

Note The chained functions in the definition of return suggest a metaphor, which you may find either useful or confusing. Consider the combined monad as a *sandwich*. This metaphor might suggest three layers of monads in action, but there are only two really: the inner monad and the combined monad (there are no binds or returns done in the base monad; it only appears as part of the implementation of the transformer). If you like this metaphor at all, think of the transformer and the base monad as two parts of the same thing - the *bread* - which wraps the inner monad.

Technically, this is all we need; however, it is convenient to make MaybeT an instance of a few other classes:

```
instance Monad m => MonadPlus (MaybeT m) where
    mzero     = MaybeT $ return Nothing
    mplus x y = MaybeT $ do maybe_value <- runMaybeT x
                            case maybe_value of
                                 Nothing    -> runMaybeT y
                                 Just _     -> return maybe_value

instance MonadTrans MaybeT where
    lift = MaybeT . (liftM Just)
```

MonadTrans implements the lift function, so we can take functions from the m monad and bring them into the MaybeT m monad in order to use them in do blocks. As for MonadPlus, since Maybe is an instance of that class it makes sense to make the MaybeT an instance too.

Application to the passphrase example

With all this done, here is what the previous example of passphrase management looks like:

```
askPassword :: MaybeT IO ()
askPassword = do lift $ putStrLn "Insert your new password:"
                 value <- msum $ repeat getValidPassphrase
                 lift $ putStrLn "Storing in database..."
```

The code is now simpler, especially in the user function askPassphrase. Most importantly, we do not have to manually check whether the result is Nothing or Just: the bind operator takes care of that for us.

Note how we use lift to bring the functions getLine and putStrLn into the MaybeT IO monad. Also, since MaybeT IO is an instance of MonadPlus, checking for passphrase validity can be taken care of by a guard statement, which will return mzero (i.e. IO Nothing) in case of a bad passphrase.Incidentally, with the help of MonadPlus it also becomes very easy to ask the user *ad infinitum* for a valid passphrase:

```
askPassword :: MaybeT IO ()
askPassword = do lift $ putStrLn "Insert your new password:"
                 value <- msum $ repeat getValidPassphrase
                 lift $ putStrLn "Storing in database..."
```

2.9.3 A plethora of transformers

The `transformers` package provides modules with transformers for many common monads (`MaybeT`, for instance, can be found in `Control.Monad.Trans.Maybe`). These are defined consistently with their non-transformer versions; that is, the implementation is basically the same except with the extra wrapping and unwrapping needed to thread the other monad. From this point on, we will use **base monad** to refer to the non-transformer monad (e.g. Maybe in MaybeT) on which a transformer is based and **inner monad** to refer to the other monad (e.g. IO in MaybeT IO) on which the transformer is applied.

To pick an arbitrary example, `ReaderT Env IO String` is a computation which involves reading values from some environment of type `Env` (the semantics of `Reader`, the base monad) and performing some `IO` in order to give a value of type `String`. Since the `(>>=)` operator and `return` for the transformer mirror the semantics of the base monad, a `do` block of type `ReaderT Env IO String` will, from the outside, look a lot like a `do` block of the `Reader` monad, except that `IO` actions become trivial to embed by using `lift`.

Type juggling

We have seen that the type constructor for `MaybeT` is a wrapper for a Maybe value in the inner monad. So, the corresponding accessor `runMaybeT` gives us a value of type `m (Maybe a)` - i.e. a value of the base monad returned in the inner monad. Similarly, for the `ListT` and `ExceptT` transformers, which are built around lists and `Either` respectively:

```
runListT :: ListT m a -> m [a]
```

and

```
runExceptT :: ExceptT e m a -> m (Either e a)
```

Not all transformers are related to their base monads in this way, however. Unlike the base monads in the two examples above, the `Writer`, `Reader`, `State`, and `Cont` monads have neither multiple constructors nor constructors with multiple arguments. For that reason, they have `run...` functions which act as simple unwrappers, analogous to the `run...T` of the transformer versions. The table below shows the result types of the `run...` and `run...T` functions in each case, which may be thought of as the types wrapped by the base and transformed monads respectively.[6]

Base Monad	**Transformer**	Original Type “wrapped” by base	Combined Type “wrapped” by transformer
Writer	WriterT	`(a, w)`	`m (a, w)`
Reader	ReaderT	`r -> a`	`r -> m a`
State	StateT	`s -> (a, s)`	`s -> m (a, s)`
Cont	ContT	`(a -> r) -> r`	`(a -> m r) -> m r`

[6]The wrapping interpretation is only literally true for versions of the `mtl` package older than 2.0.0.0 .

Notice that the base monad is absent in the combined types. Without interesting constructors (of the sort for $\boxed{\text{Maybe}}$ or lists), there is no reason to retain the base monad type after unwrapping the transformed monad. It is also worth noting that in the latter three cases we have function types being wrapped. $\boxed{\text{StateT}}$, for instance, turns state-transforming functions of the form $\boxed{s \to (a, s)}$ into state-transforming functions of the form $\boxed{s \to \text{m (a, s)}}$; only the result type of the wrapped function goes into the inner monad. $\boxed{\text{ReaderT}}$ is analogous. $\boxed{\text{ContT}}$ is different because of the semantics of $\boxed{\text{Cont}}$ (the *continuation* monad): the result types of both the wrapped function and its function argument must be the same, and so the transformer puts both into the inner monad. In general, there is no magic formula to create a transformer version of a monad; the form of each transformer depends on what makes sense in the context of its non-transformer type.

2.9.4 Lifting

We will now have a more detailed look at the $\boxed{\text{lift}}$ function, which is critical in day-to-day use of monad transformers. The first thing to clarify is the name "lift". One function with a similar name that we already know is $\boxed{\text{liftM}}$. As we already know, it is a monad-specific version of $\boxed{\text{fmap}}$:

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

$\boxed{\text{liftM}}$ applies a function $\boxed{(a- > b)}$ to a value within a monad $\boxed{\text{m}}$. We can also look at it as a function of just one argument:

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

$\boxed{\text{liftM}}$ converts a plain function into one that acts within $\boxed{\text{m}}$. By "lifting", we refer to bringing something into something else – in this case, a function into a monad.

$\boxed{\text{liftM}}$ allows us to apply a plain function to a monadic value without needing do-blocks or other such tricks:

```
.-----------------------------------------------------------------.
|                              |                                  |
|         do notation          |             liftM                |
|------------------------------|----------------------------------|
|     do x <- monadicValue     |      liftM f monadicValue        |
|        return (f x)          |                                  |
|______________________________|__________________________________|
```

The $\boxed{\text{lift}}$ function plays an analogous role when working with monad transformers. It brings (or, to use another common word for that, *promotes*) inner monad computations to the combined monad. By doing so, it allows us to easily insert inner monad computations as part of a larger computation in the combined monad.

$\boxed{\text{lift}}$ is the single method of the $\boxed{\text{MonadTrans}}$ class, found in $\boxed{\text{Control.Monad.Trans.Class}}$. All monad transformers are instances of $\boxed{\text{MonadTrans}}$, and so $\boxed{\text{lift}}$ is available for them all.

```
class MonadTrans t where
    lift :: (Monad m) => m a -> t m a
```

There is a variant of $\boxed{\text{lift}}$ specific to $\boxed{\text{IO}}$ operations, called $\boxed{\text{liftIO}}$, which is the single method of the $\boxed{\text{MonadIO}}$ class in $\boxed{\text{Control.Monad.IO.Class}}$.

```
class (Monad m) => MonadIO m where
   liftIO :: IO a -> m a
```

liftIO can be convenient when multiple transformers are stacked into a single combined monad. In such cases, IO is always the innermost monad, and so we typically need more than one lift to bring IO values to the top of the stack. liftIO is defined for the instances in a way that allows us to bring an IO value from any depth while writing the function a single time.

Implementing lift

Implementing lift is usually pretty straightforward. Consider the MaybeT transformer:

```
instance MonadTrans MaybeT where
    lift m = MaybeT (liftM Just m)
```

We begin with a monadic value of the inner monad. With liftM (fmap would have worked just as fine), we slip the base monad (through the Just constructor) underneath, so that we go from m a to m (Maybe a)). Finally, we use the MaybeT constructor to wrap up the monadic sandwich. Note that the liftM here works in the inner monad, just like the do-block wrapped by MaybeT in the implementation of (>>=) we saw early on was in the inner monad.

2.9.5 Implementing transformers

The State transformer

As an additional example, we will now have a detailed look at the implementation of StateT. You might want to review the appendix on the State monad before continuing.

Just as the State monad might have been built upon the definition

```
newtype State s a = State { runState :: (s -> (a,s)) }
```

, the StateT transformer is built upon the definition:

```
newtype StateT s m a = StateT { runStateT :: (s -> m (a,s)) }
```

StateT s m will have the following Monad instance, here shown alongside the one for the base state monad:

```
 .-----------------------------------------------------------------------------------------.
|                                         :                                                 |
|                State                    :                   StateT                        |
|-----------------------------------------:-------------------------------------------------|
|                                         :                                                 |
| newtype State s a =                     : newtype StateT s m a =                          |
|  State { runState :: (s -> (a,s)) }     :  StateT { runStateT :: (s -> m (a,s)) }         |
|                                         :                                                 |
| instance Monad (State s) where          : instance (Monad m) => Monad (StateT s m) where  |
|  return a        = State $ \s -> (a,s)  :  return a         = StateT $ \s -> return (a,s) |
|  (State x) >>= f = State $ \s ->        :  (StateT x) >>= f = StateT $ \s -> do           |
|   let (v,s') = x s                      :   (v,s') <- x s         -- get new value and state |
|   in runState (f v) s'                  :   runStateT (f v) s'    -- pass them to f        |
|                                         :                                                 |
|_________________________________________|_________________________________________________|
```

Our definition of return makes use of the return function of the inner monad. (>>=) uses a do-block to perform a computation in the inner monad.

Note Incidentally, we can now finally explain why, in the appendix about `State`, there is a `state` function instead of a `State` constructor. In the `transformers` and `mtl` packages, `State s` is implemented as a type synonym for `StateT s Identity`, with `Identity` being the dummy monad introduced in an exercise of the previous section. The resulting monad is equivalent to the one defined using `newtype` that we have used up to now.

If the combined monads `StateT s m` are to be used as state monads, we will certainly want the all-important `get` and `put` operations. Here, we will show definitions in the style of the `mtl` package. In addition to the monad transformers themselves, `mtl` provides type classes for the essential operations of common monads. For instance, the `MonadState` class, found in `Control.Monad.State`, has `get` and `put` as methods:

```
instance (Monad m) => MonadState s (StateT s m) where
  get   = StateT $ \s -> return (s,s)
  put s = StateT $ \_ -> return ((),s)
```

Note The first line should be read as: "For any type `s` and any instance of `Monad m`; `s` and `StateT s m` together form an instance of `MonadState`". `s` and `m` correspond to the state and the inner monad, respectively. `s` is an independent part of the instance specification so that the methods can refer to it - for instance, the type of `put` is `s → StateT s ()`.

There are `MonadState` instances for state monads wrapped by other transformers, such as

```
MonadState s m => MonadState s (MaybeT m)
```

They bring us extra convenience by making it unnecessary to lift uses of `get` and `put` explicitly, as the `MonadState` instance for the combined monads handles the lifting for us.

It can also be useful to lift instances that might be available for the inner monad to the combined monad. For instance, all combined monads in which `StateT` is used with an instance of `MonadPlus` can be made instances of `MonadPlus`:

```
instance (MonadPlus m) => MonadPlus (StateT s m) where
  mzero = StateT $ \_ -> mzero
  (StateT x1) `mplus` (StateT x2) = StateT $ \s -> (x1 s) `mplus` (x2 s)
```

The implementations of `mzero` and `mplus` do the obvious thing; that is, delegating the actual work to the instance of the inner monad.

Lest we forget, the monad transformer must have a `MonadTrans` instance, so that we can use `lift`:

```
instance MonadTrans (StateT s) where
  lift c = StateT $ \s -> c >>= (\x -> return (x,s))
```

The `lift` function creates a `StateT` state transformation function that binds the computation in the inner monad to a function that packages the result with the input state. If, for instance, we apply `StateT` to the `List` monad, a function that returns a list (i.e., a computation in the List monad) can be lifted into `StateT s []` where it becomes a function that returns a `StateT s→ [(a,s)])`. I.e. the lifted computation produces *multiple* (value,state) pairs from its input state. This "forks" the computation in StateT, creating a different branch of the computation for each value in the list returned by the lifted function. Of course, applying `StateT` to a different monad will produce different semantics for the `lift` function.

Chapter 3

Last Steps

3.1 Revisiting the *Applicative* class

A more-in-depth look at the `Applicative` class. The first subsection is just the same text as in chapter 2.

3.1.1 *Applicative* recap

The definition of `Applicative` is:

```
class (Functor f) => Applicative f where
    pure  :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

Beyond `(<*>)`, the class has a second method, `pure`, which brings arbitrary values into the functor. As an example, let's have a look at the `Maybe` instance:

```
instance Applicative Maybe where
    pure                  = Just
    (Just f) <*> (Just x) = Just (f x)
    _        <*> _        = Nothing
```

It doesn't do anything surprising: `pure` wraps the value with `Just`; `(<*>)` applies the function to the value if both exists, and results in `Nothing` otherwise.

Note For the lack of a better shorthand, in what follows we will use the word *morphism* to refer to the values to the left of `(<*>)`, which fit the type `Applicative f => f (a -> b)`; that is, the function-like things inserted into an applicative functor.

Just like `Functor`, `Applicative` has a set of laws which reasonable instances should follow. They are:

```
pure id <*> v = v                            -- Identity
pure f <*> pure x = pure (f x)               -- Homomorphism
u <*> pure y = pure ($ y) <*> u              -- Interchange
pure (.) <*> u <*> v <*> w = u <*> (v <*> w) -- Composition
```

Those laws are a bit of a mouthful. They become easier to understand if you think of `pure` as a way to inject values into the functor in a default, featureless way, so that the result is as close as possible to the plain value. Thus:

- The identity law says that applying the $\boxed{\text{pure id}}$ morphism does nothing, exactly like with the plain $\boxed{\text{id}}$ function.
- The homomorphism law says that applying a "pure" function to a "pure" value is the same than applying the function to the value in the normal way and then using $\boxed{\text{pure}}$ on the result. In a sense, that means $\boxed{\text{pure}}$ preserves function application.
- The interchange law says that applying a morphism to a "pure" value $\boxed{\text{pure y}}$ is the same as applying pure $\boxed{(\$\ \text{y})}$ to the morphism. No surprises there - $\boxed{(\$\ \text{y})}$ is the function that supplies $\boxed{\text{y}}$ as argument to another function.
- The composition law says that if $\boxed{(<*>)}$ is used to compose morphisms the composition is associative, like plain function composition.[1]

There is also a bonus law about the relation between $\boxed{\text{fmap}}$ and $\boxed{(<*>)}$:

```
fmap f x = pure f <*> x                         -- fmap
```

Applying a "pure" function with $\boxed{(<*>)}$ is equivalent to using $\boxed{\text{fmap}}$. **This law is a consequence of the other ones, so you need not bother with proving it when writing instances of $\boxed{\textbf{Applicative}}$.**

3.1.2 Deja vu

Does $\boxed{\text{pure}}$ remind you of anything?

```
pure :: Applicative f => a -> f a
```

The only difference between that and...

```
return :: Monad m => a -> m a
```

... is the class constraint. $\boxed{\text{pure}}$ and $\boxed{\text{return}}$ serve the same purpose; that is, bringing values into functors. The uncanny resemblances do not stop here. In the appendix about $\boxed{\text{State}}$ we mention a function called $\boxed{\text{ap}}$...

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

... which could be used to make functions with many arguments less painful to handle in monadic code:

```
allTypes :: GeneratorState (Int, Float, Char, Integer, Double, Bool, Int)
allTypes = liftM (,,,,,,) getRandom
                     `ap` getRandom
                     `ap` getRandom
                     `ap` getRandom
                     `ap` getRandom
                     `ap` getRandom
                     `ap` getRandom
```

$\boxed{\text{ap}}$ looks a lot like $\boxed{(<*>)}$.

Those, of course, are not coincidences. $\boxed{\text{Monad}}$ inherits from $\boxed{\text{Applicative}}$...

[1] With plain functions, we have $\boxed{h\ .\ g\ .\ f = (h\ .\ g)\ .\ f = h\ .\ (g\ .\ f)}$ That is why we never bother to use parentheses in the middle of $\boxed{(.)}$ chains.

```
Prelude> :info Monad
class Applicative m => Monad (m :: * -> *) where
--etc.
```

... because `return` and `(>>=)` are enough to implement `pure` and `(<*>)`[2]

```
pure = return
(<*>) = ap

ap u v = do
    f <- u
    x <- v
    return (f x)
```

Several other monadic functions have more general applicative versions. Here are a few of them:

Monadic	**Applicative**	Module (where to find the applicative version)
`(>>)`	`(*>)`	Prelude (GHC 7.10+); Control.Applicative
`liftM2`	`liftA2`	Control.Applicative
`mapM`	`traverse`	Prelude (GHC 7.10+); Data.Traversable
`sequence`	`sequenceA`	Data.Traversable
`forM_`	`for_`	Data.Foldable

3.1.3 *ZipList*

Lists are applicative functors as well. Specialised to lists, the type of `(<*>)` becomes...

```
[a -> b] -> [a] -> [b]
```

... and so `(<*>)` applies a list of functions to another list. But exactly how is that done?

The standard instance of `Applicative` for lists, which follows from the `Monad` instance, applies every function to every element, like an explosive version of `map`.

```
Prelude> [(2*),(5*),(9*)] <*> [1,4,7]
[2,8,14,5,20,35,9,36,63]
```

Interestingly, there is another reasonable way of applying a list of functions. Instead of using every combination of functions and values, we can match each function with the value in the corresponding position in the other list. A Prelude function which can be used for that is `zipWith`:

```
Prelude> :t zipWith
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
Prelude> zipWith ($) [(2*),(5*),(9*)] [1,4,7]
[2,20,63]
```

When there are two useful possible instances for a single type, the dilemma is averted by creating a `newtype` which implements one of them. In this case, we have `ZipList`, which lives in `Control.Applicative`:

[2] And if the `Monad` instance follows the monad laws, the resulting `pure` and `(<*>)` will automatically follow the applicative laws.

```
newtype ZipList a = ZipList { getZipList :: [a] }
```

We have already seen what $\boxed{(<*>)}$ should be for zip-lists; all that is needed is to add the $\boxed{\text{newtype}}$ wrappers:

```
instance Applicative ZipList where
    (ZipList fs) <*> (ZipList xs) = ZipList (zipWith ($) fs xs)
    pure x                        = undefined -- TODO
```

As for $\boxed{\text{pure}}$, it is tempting to use $\boxed{\text{pure x = ZipList [x]}}$, following the standard list instance. We can't do that, however, as it violates the applicative laws. According to the identity law:

```
pure id <*> v = v
```

Substituting $\boxed{(<*>)}$ and the suggested $\boxed{\text{pure}}$, we get:

```
ZipList [id] <*> ZipList xs = ZipList xs
ZipList (zipWith ($) [id] xs) = ZipList xs
```

Now, suppose $\boxed{\text{xs}}$ is the infinite list $\boxed{[1..]}$:

```
ZipList (zipWith ($) [id] [1..]) = ZipList [1..]
ZipList [1] = ZipList [1..]
[1] = [1..] -- Obviously false!
```

The problem is that $\boxed{\text{zipWith}}$ produces lists whose length is that of the shortest list passed as argument, and so `(ZipList [id] <*>)` will cut off all elements of the other zip-list after the first. The only way to ensure `zipWith ($) fs` never removes elements is making $\boxed{\text{fs}}$ infinite. The correct $\boxed{\text{pure}}$ follows from that:

```
instance Applicative ZipList where
    (ZipList fs) <*> (ZipList xs) = ZipList (zipWith ($) fs xs)
    pure x                        = ZipList (repeat x)
```

The $\boxed{\text{ZipList}}$ applicative instance offers an alternative to all the zipN and zipWithN functions in $\boxed{\text{Data.List}}$ which can be extended to any number of arguments:

```
>>> import Control.Applicative
>>> ZipList [(2*),(5*),(9*)] <*> ZipList [1,4,7]
ZipList {getZipList = [2,20,63]}
>>> (,,) <$> ZipList [1,4,9] <*> ZipList [2,8,1] <*> ZipList [0,0,9]
ZipList {getZipList = [(1,2,0),(4,8,0),(9,1,9)]}
>>> liftA3 (,,) (ZipList [1,4,9]) (ZipList [2,8,1]) (ZipList [0,0,9])
ZipList {getZipList = [(1,2,0),(4,8,0),(9,1,9)]}
```

3.1.4 Sequencing of effects

As we have just seen, the standard $\boxed{\text{Applicative}}$ instance for lists applies every function in one list to every element of the other. That, however, does not specify $\boxed{(<*>)}$ unambiguously. To see why, try to guess what is the result of

```
[(2*),(3*)]<*>[4,5]
```

without looking at the example above or the answer just below.

```
Prelude> [(2*),(3*)] <*> [4,5]
```

```
--- ...
```

```
[8,10,12,15]
```

Unless you were paying very close attention or had already analysed the implementation of `(<*>)`, the odds of getting it right were about even. The other possibility would be `[8,12,10,15]`. The difference is that for the first (and correct) answer the result is obtained by taking the skeleton of the first list and replacing each element by all possible combinations with elements of the second list, while for the other possibility the starting point is the second list.

In more general terms, the difference between is one of *sequencing of effects*. Here, by effects we mean the functorial context, as opposed to the values within the functor (some examples: the skeleton of a list, actions performed in the real world in `IO`, the existence of a value in `Maybe`). The existence of two legal implementations of `(<*>)` for lists which only differ in the sequencing of events indicates that `[]` is a non-commutative applicative functor. A *commutative* applicative functor, by contrast, leaves no margin for ambiguity in that respect. More formally, a commutative applicative functor is one for which the following holds:

```
liftA2 f u v = liftA2 (flip f) v u -- Commutativity
```

Or, equivalently,

```
f <$> u <*> v = flip f <$> v <*> u
```

By the way, if you hear about *commutative monads* in Haskell, the concept involved is the same, only specialised to `Monad`.

Commutativity (or the lack thereof) affects other functions which are derived from `(<*>)` as well. `(*>)` is a clear example:

```
(*>) :: Applicative f => f a -> f b -> f b
```

`(*>)` combines effects while preserving only the values of its second argument. For monads, it is equivalent to `(>>)`. Here is a demonstration of it using `Maybe`, which is commutative:

```
Prelude> Just 2 *> Just 3
Just 3
Prelude> Just 3 *> Just 2
Just 2
Prelude> Just 2 *> Nothing
Nothing
Prelude> Nothing *> Just 2
Nothing
```

Swapping the arguments does not affect the effects (that is, the being and nothingness of wrapped values). For `IO`, however, swapping the arguments does reorder the effects:

```
Prelude> (print "foo" *> pure 2) *> (print "bar" *> pure 3)
"foo"
"bar"
3
```

```
Prelude> (print "bar" *> pure 3) *> (print "foo" *> pure 2)
"bar"
"foo"
2
```

The convention in Haskell is to always implement (<*>) and other applicative operators using left-to-right sequencing. Even though this convention helps reducing confusion, it also means appearances sometimes are misleading. For instance, the (<*) function is *not* `flip (*>)`, as it sequences effects from left to right just like (*>):

```
Prelude> (print "foo" *> pure 2) <* (print "bar" *> pure 3)
"foo"
"bar"
2
```

For the same reason, `(<**>) :: Applicative f => f a -> f (a -> b) -> f b` from Control.Applicative is not `flip (<*>)`. That means it provides a way of inverting the sequencing:

```
>>> [(2*),(3*)] <*> [4,5]
[8,10,12,15]
>>> [4,5] <**> [(2*),(3*)]
[8,12,10,15]
```

An alternative is the Control.Applicative.Backwards module from transformers, which offers a newtype for flipping the order of effects:

```
newtype Backwards f a = Backwards { forwards :: f a }
```

```
>>> Backwards [(2*),(3*)] <*> Backwards [4,5]
Backwards [8,12,10,15]
```

3.1.5 A sliding scale of power

Functor, Applicative, Monad. Three closely related functor type classes; three of the most important classes in Haskell. Though we have seen many examples of Functor and Monad in use, and a few of Applicative, we have not compared them head to head yet. If we ignore pure/return for a moment, the characteristic methods of the three classes are:

```
fmap :: Functor f => (a -> b) -> f a -> f b
(<*>) :: Applicative f => f (a -> b) -> f a -> f b
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

While those look like disparate types, we can change the picture with a few aesthetic adjustments. Let's replace fmap by its infix synonym, (<$>); (>>=) by its flipped version, (=<<); and tidy up the signatures a bit:

```
(<$>) :: Functor t     =>   (a -> b) -> (t a -> t b)
(<*>) :: Applicative t => t (a -> b) -> (t a -> t b)
(=<<) :: Monad t       => (a -> t b) -> (t a -> t b)
```

Suddenly, the similarities are striking. fmap, (<*>) and (=<<) are all mapping functions over Functors.[3] The differences between them are in what is being mapped over in each case:

- fmap maps arbitrary functions over functors.
- (<*>) maps `t (a -> b)` morphisms over (applicative) functors.
- (=<<) maps `a -> t b` functions over (monadic) functors.

The day-to-day differences in uses of Functor, Applicative and Monad follow from what the types of those three mapping functions allow you to do. As you move from fmap to (<*>) and then to (>>=), you gain in power, versatility and control, at the cost of guarantees about the results. We will now slide along this scale. While doing so, we will use the contrasting terms *values* and *context* to refer to plain values within a functor and to the whatever surrounds them, respectively.

The type of fmap ensures that it is impossible to use it to change the context, no matter which function it is given. In

```
(a -> b) -> t a -> t b
```

, the `(a -> b)` function has nothing to do with the t context of the t a functorial value, and so applying it cannot affect the context. For that reason, if you do fmap f xs on some list xs the number of elements of the list will never change.

```
Prelude> fmap (2*) [2,5,6]
[4,10,12]
```

That can be taken as a safety guarantee or as an unfortunate restriction, depending on what you intend. In any case, (<*>) is clearly able to change the context:

```
Prelude> [(2*),(3*)] <*> [2,5,6]
[4,10,12,6,15,18]
```

The `t (a -> b)` morphism carries a context of its own, which is combined with that of the t a functorial value. (<*>), however, is subject to a more subtle restriction. While `t (a -> b)` morphisms carry context, within them there are plain `(a -> b)` functions, which are still unable to modify the context. That means the changes to the context (<*>) performs are fully determined by the context of its arguments, and the values have no influence over the resulting context.

```
Prelude> (print "foo" *> pure (2*)) <*> (print "bar" *> pure 3)
"foo"
"bar"
6
Prelude> (print "foo" *> pure 2) *> (print "bar" *> pure 3)
"foo"
"bar"
3
Prelude> (print "foo" *> pure undefined) *> (print "bar" *> pure 3)
"foo"
"bar"
3
```

[3] It is not just a question of type signatures resembling each other: the similarity has theoretical ballast. One aspect of the connection is that it is no coincidence that all three type classes have identity and composition laws.

Thus with list $\boxed{(<*>)}$ you know that the length of the resulting list will be the product of the lengths of the original lists, with $\boxed{\text{IO}}\boxed{(<*>)}$ you know that all real world effect will happen as long as the evaluation terminates, and so forth.

With $\boxed{\text{Monad}}$, however, we are in a very different game. $\boxed{(>>=)}$ takes an `a -> t b` function, and so it is able to create context from values. That means a lot of flexibility:

```
Prelude> [1,2,5] >>= \x -> replicate x x
[1,2,2,5,5,5,5,5]
Prelude> [0,0,0] >>= \x -> replicate x x
[]
Prelude> return 3 >>= \x -> print $ if x < 10 then ‘Too small’ else ‘OK’
‘Too small’
Prelude> return 42 >>= \x -> print $ if x < 10 then ‘Too small’ else ‘OK’
‘OK’
```

Taking advantage of the extra flexibility, however, might mean having less guarantees about, for instance, whether your functions are able to unexpectedly erase parts of a data structure for pathological inputs, or whether the control flow in your application remains intelligible. In some situations there might be performance implications as well, as the complex data dependencies monadic code makes possible might prevent useful refactorings and optimisations.

All in all, it is a good idea to only use as much power as needed for the task at hand. If you do need the extra capabilities of Monad, go right ahead; however, it is often worth it to check whether $\boxed{\text{Applicative}}$ or $\boxed{\text{Functor}}$ are sufficient.

3.1.6 The monoidal presentation

Back in last chapter, we saw how the $\boxed{\text{Monad}}$ class can be specified using either $\boxed{(>=>)}$ or join instead of $\boxed{(>>=)}$. In a similar way, $\boxed{\text{Applicative}}$ also has an alternative presentation, which might be implemented through the following type class:

```
class Functor f => Monoidal f where
    unit  :: f ()
    (*&*) :: f a -> f b -> f (a,b)
```

There are deep theoretical reasons behind the name “monoidal”.[4] In any case, we can informally say that it does look a lot like a monoid: $\boxed{\text{unit}}$ provides a default functorial value whose context wraps nothing of interest, and $\boxed{(*\&*)}$ combines functorial values by pairing values and combining effects. The $\boxed{\text{Monoidal}}$ formulation provides a clearer view of how $\boxed{\text{Applicative}}$ manipulates functorial contexts. Naturally, $\boxed{\text{unit}}$ and $\boxed{(*\&*)}$ can be used to define $\boxed{\text{pure}}$ and $\boxed{(<*>)}$, and vice-versa.

The Applicative laws are equivalent to the following set of laws, stated in terms of $\boxed{\text{Monoidal}}$:

```
fmap snd $ unit *&* v = v                 -- Left identity
fmap fst $ u *&* unit = u                 -- Right identity
fmap asl $ u *&* (v *&* w) = (u *&* v) *&* w -- Associativity
-- asl (x, (y, z)) = ((x, y), z)
```

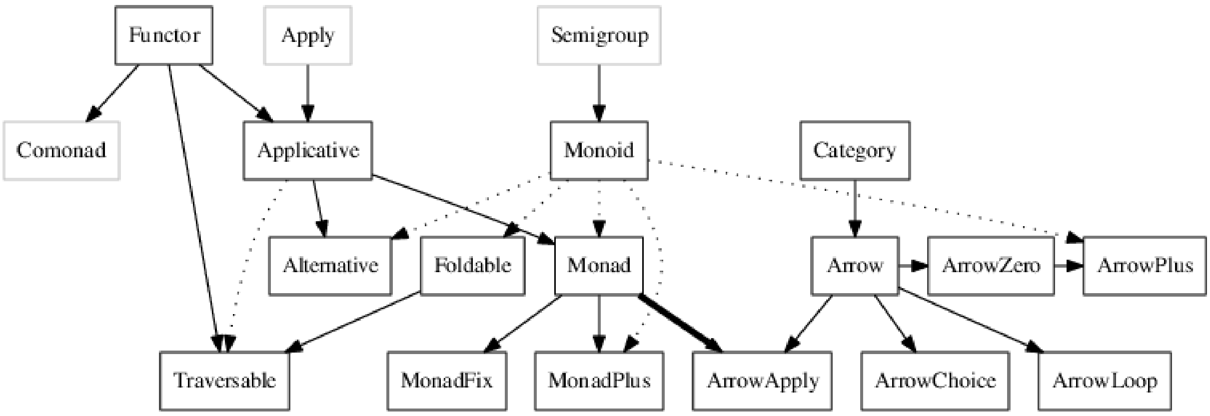
The functions to the left of the $\boxed{(\$)}$ are just boilerplate to convert between equivalent types, such as $\boxed{\text{b}}$ and $\boxed{((), \text{b})}$. If you ignore them, the laws are a lot less opaque than in the usual $\boxed{\text{Applicative}}$

[4]For extra details, follow the leads from the corresponding section of the Typeclasseopedia (https://wiki.haskell.org/Typeclassopedia#Alternative_formulation) and the blog post by Edward Z. Yang which inspired it. (http://blog.ezyang.com/2012/08/applicative-functors/)

formulation. By the way, just like for $\boxed{\text{Applicative}}$ there is a bonus law, which is guaranteed to hold in Haskell:

```
fmap (g *** h) (u *&* v) = fmap g u *&* fmap h v -- Naturality
-- g *** h = \(x, y) -> (g x, h y)
```

3.1.7 Class heritage



3.2 Still for the curious: The Hask Category

In this section, we will dive in the **Hask** category, identifying some Haskell funcstions with their mathematical equivalent in Category Theory. Recalling from the first chapter:

$\mathfrak{Hask}$: Obj($\mathfrak{Hask}$) – the class of all Haskell types. Hom($\mathfrak{Hask}$) – Haskell functions. The composition law is the (.) operator.

3.2.1 Checking that Hask is a category

We can check the first and second law easily: we know $\boxed{\text{(.)}}$ is an associative function, and clearly, for any $\boxed{\text{f}}$ and $\boxed{\text{g}}$, $\boxed{\text{f . g}}$ is another function.

In Hask, the identity morphism is $\boxed{\text{id}}$, and we have trivially:

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

This isn't an exact translation of the law above, though; we're missing subscripts. The function $\boxed{\text{id}}$ in Haskell is *polymorphic*– it can take many different types for its domain and range, or, in category-speak, can have many different source and target objects. But morphisms in category theory are by definition *monomorphic*– each morphism has one specific source object and one specific target object. A polymorphic Haskell function can be made monomorphic by specifying its type (*instantiating* with a monomorphic type), so it would be more precise if we said that the identity morphism from **Hask** on a type $\boxed{\text{A}}$ is `(id :: A -> A)`. With this in mind, the above law would be rewritten as:

```
(id :: B -> B) . f = f . (id :: A -> A) = f
```

However, for simplicity, we will ignore this distinction when the meaning is clear.

Actually, there is a subtlety here: because $\boxed{\text{(.)}}$ is a lazy function, if $\boxed{\text{f}}$ is $\boxed{\text{undefined}}$, we have that `id . f = \_ -> ⊥` . Now, while this may seem equivalent to $\perp$ for all intents and purposes, you can actually tell them apart using the strictifying function $\boxed{\text{seq}}$, meaning that the last category law is broken. We can define a new strict composition function,

```
f .! g = ((.) $! f) $! g
```

that makes **Hask** a category. We proceed by using the normal $\boxed{\text{(.)}}$, though, and attribute any discrepancies to the fact that $\boxed{\text{seq}}$ breaks an awful lot of the nice language properties anyway.

3.2.2 Functors on Hask

The Functor typeclass you have probably seen in Haskell does in fact tie in with the categorical notion of a functor. Remember that a functor has two parts: it maps objects in one category to objects in another and morphisms in the first category to morphisms in the second. Functors in Haskell are from **Hask** to *func*, where *func* is the subcategory of **Hask** defined on just that functor's types. E.g. the list functor goes from **Hask** to **Lst**, where **Lst** is the category containing only *list types*, that is, $\boxed{\text{[T]}}$ for any type $\boxed{\text{T}}$. The morphisms in **Lst** are functions defined on list types, that is, functions `[T] -> [U]` for types $\boxed{\text{T}}$, $\boxed{\text{U}}$. How does this tie into the Haskell typeclass $\boxed{\text{Functor}}$? Recall its definition:

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

Let's have a sample instance, too:

```
instance Functor Maybe where
  fmap f (Just x) = Just (f x)
  fmap _ Nothing  = Nothing
```

Here's the key part: the *type constructor* $\boxed{\text{Maybe}}$ takes any type $\boxed{\text{T}}$ to a new type, $\boxed{\text{Maybe T}}$. Also, $\boxed{\text{fmap}}$ restricted to Maybe types takes a function `a -> b` to a function `Maybe a -> Maybe b`. But that's it! We've defined two parts, something that takes objects in **Hask** to objects in another category (that of Maybe types and functions defined on Maybe types), and something that takes morphisms in **Hask** to morphisms in this category. So Maybe is a functor.

A useful intuition regarding Haskell functors is that they represent types that can be mapped over. This could be a list or a Maybe, but also more complicated structures like trees. A function that does some mapping could be written using $\boxed{\text{fmap}}$, then any functor structure could be passed into this function. E.g. you could write a generic function that covers all of $\boxed{\text{Data.List.map}}$, $\boxed{\text{Data.Map.map}}$, $\boxed{\text{Data.Array.IArray.amap}}$, and so on.

What about the functor axioms? The polymorphic function $\boxed{\text{id}}$ takes the place of id_A for any $\boxed{\text{A}}$, so the first law states:

```
fmap id = id
```

With our above intuition in mind, this states that mapping over a structure doing nothing to each element is equivalent to doing nothing overall. Secondly, morphism composition is just $\boxed{(.)}$, so

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

This second law is very useful in practice. Picturing the functor as a list or similar container, the right-hand side is a two-pass algorithm: we map over the structure, performing $\boxed{\text{g}}$, then map over it again, performing $\boxed{\text{f}}$. The functor axioms guarantee we can transform this into a single-pass algorithm that performs $\boxed{\text{f . g}}$. This is a process known as *fusion*.

Translating categorical concepts into Haskell

Functors provide a good example of how category theory gets translated into Haskell. The key points to remember are that:

- We work in the category **Hask** and its subcategories.
- Objects are types.
- Morphisms are functions. Things that take a type and return another type are type constructors.
- Things that take a function and return another function are higher-order functions.
- Typeclasses, along with the polymorphism they provide, make a nice way of capturing the fact that in category theory things are often defined over a number of objects at once.

3.2.3 Monads

Monads are obviously an extremely important concept in Haskell, and in fact they originally came from category theory.[5] A monad is a special type of functor, from a category to that same category, that supports some additional structure. So, down to definitions. A monad is a functor $M : C \to C$, along with two morphisms for every object X in C:

[5]Experienced category theorists will notice that we're simplifying things a bit here; instead of presenting *unit* and *join* as natural transformations, we treat them explicitly as morphisms, and require naturality as extra axioms

- $unit_X{}^M : X \to M(X)$
- $join_X{}^M M(M(X)) \to M(X)$

When the monad under discussion is obvious, we'll leave out the ^{M} superscript for these functions and just talk about $unit_X$ and $join_X$ for some X.

Translating

Let's see how this translates to the Haskell typeclass Monad, then.

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

The class constraint of $\boxed{\text{Functor m}}$ ensures that we already have the functor structure: a mapping of objects and of morphisms. $\boxed{\text{return}}$ is the (polymorphic) analogue to $unit_X$ for any X. But we have a problem.

Although $\boxed{\text{return}}$'s type looks quite similar to that of *unit*; the other function, $\boxed{\text{(>>=)}}$, often called *bind*, bears no resemblance to *join*. There is however another monad function,

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

that looks quite similar. Indeed, we can recover $\boxed{\text{join}}$ and $\boxed{\text{(>>=)}}$ from each other:

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

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

So specifying a monad's $\boxed{\text{return}}$, $\boxed{\text{fmap}}$, and $\boxed{\text{join}}$ is equivalent to specifying its $\boxed{\text{return}}\boxed{\text{fmap}}$, and $\boxed{\text{(>>=)}}$. It just turns out that the normal way of defining a monad in category theory is to give $\boxed{\text{unit}}$ and $\boxed{\text{join}}$, whereas Haskell programmers like to give $\boxed{\text{return}}$ and $\boxed{\text{bind.}}$[6] Often, the categorical way makes more sense. Any time you have some kind of structure M and a natural way of taking any object X into $M(X)$, as well as a way of taking $M(M(X))$ into $M(X)$, you probably have a monad. We can see this in the following example section.

Example: the powerset functor is also a monad

The power set functor $P : \mathbf{Set} \to \mathbf{Set}$ forms a monad. For any set S you have a $unit_S(x) = \{x\}$, mapping elements to their singleton set. Note that each of these singleton sets are trivially a subset of S, so $unit_S$ returns elements of the powerset of S, as is required. Also, you can define a function $join_S$ as follows: we receive an input $L \in \mathcal{P}(\mathcal{P}(S))$. This is:

- A member of the powerset of the powerset of S.

---

alongside the standard monad laws (laws 3 and 4). The reasoning is simplicity; we are not trying to teach category theory as a whole, simply give a categorical background to some of the structures in Haskell. You may also notice that we are giving these morphisms names suggestive of their Haskell analogues, because the names η and μ don't provide much intuition.

[6] This is perhaps due to the fact that Haskell programmers like to think of monads as a way of sequencing computations with a common feature, whereas in category theory the container aspect of the various structures is emphasised. $\boxed{\text{join}}$ pertains naturally to containers (squashing two layers of a container down into one), but $\boxed{\text{(>>=)}}$ is the natural sequencing operation (do something, feeding its results into something else).

- So a member of the set of all subsets of the set of all subsets of S.
- So a set of subsets of S.

We then return the union of these subsets, giving another subset of S. Symbolically,

$$join_S(L) = \bigcup L$$

Hence $\mathcal{P}$ is a monad.[7]

In fact, $\mathcal{P}$ is almost equivalent to the list monad; with the exception that we're talking lists instead of sets, they're almost the same. Compare:

Power set functor on **Set**, given a set S and a morphism $f : A \to B$

Function type	**Definition**
$P(f) : \mathcal{P}(A) \to \mathcal{P}(B)$	$(P(f))(S) = \{f(a) : a \in S\}$
$unit_S : S \to \mathcal{P}(S)$	$unit_S(x) = \{x\}$
$join_S : \mathcal{P}(\mathcal{P}(S)) \to \mathcal{P}(S)$	$join_S(L) = \bigcup L$

List monad from Haskell, given a type $\boxed{\text{T}}$ and a function `f :: A -> B`

Function type	**Definition**
`fmap f :: [A] -> [B]`	`fmap f xs = [ f a | a <- xs ]`
`return :: T -> [T]`	`return x = [x]`
`join :: [[T]] -> [T]`	`join xs = concat xs`

3.2.4 The monad laws and their importance

Just as functors had to obey certain axioms in order to be called functors, monads have a few of their own. We'll first list them, then translate to Haskell, then see why they're important.

Given a monad $M : C \to C$ and a morphism $f : A \to B$ for $A, B \in C$,

1. $join \circ M(join) = join \circ join$
2. $join \circ M(unit) = join \circ unit = id$
3. $unit \circ f = M(f) \circ unit$
4. $join \circ M(M(f) = M(f) \circ join$

By now, the Haskell translations should be hopefully self-explanatory:

1. $\boxed{\text{join . fmap join} = \text{join . join}}$
2. $\boxed{\text{join . fmap return} = \text{join . return} = \text{id}}$

[7]If you can prove that certain laws hold, which we'll explore with lists in the next subsection.

3. $\boxed{\text{return . f = fmap f . return}}$

4. $\boxed{\text{join . fmap (fmap f) = fmap f . join}}$

(Remember that $\boxed{\text{fmap}}$ is the part of a functor that acts on morphisms.) These laws seem a bit impenetrable at first, though. What on earth do these laws mean, and why should they be true for monads? Let's explore the laws.

The first law

$\boxed{\text{join . fmap join = join . join}}$

In order to understand this law, we'll first use the example of lists. The first law mentions two functions, $\boxed{\text{join . fmap join}}$ (the left-hand side) and $\boxed{\text{join . join}}$ (the right-hand side). What will the types of these functions be? Remembering that $\boxed{\text{join}}$'s type is `[[a]] -> [a]` (we're talking just about lists for now), the types are both `[[[a]]] -> [a]` (the fact that they're the same is handy; after all, we're trying to show they're completely the same function!). So we have a list of lists of lists. The left-hand side, then, performs $\boxed{\text{fmap join}}$ on this 3-layered list, then uses $\boxed{\text{join}}$ on the result. $\boxed{\text{fmap}}$ is just the familiar $\boxed{\text{map}}$ for lists, so we first map across each of the list of lists inside the top-level list, concatenating them down into a list each. Now we have a list of lists, which we then run through $\boxed{\text{join}}$. In summary, we 'enter' the top level, collapse the second and third levels down, then collapse this new level with the top level.

What about the right-hand side? We first run $\boxed{\text{join}}$ on our list of list of lists. Although this is three layers, and you normally apply a two-layered list to $\boxed{\text{join}}$, this will still work, because a $\boxed{\text{[[[a]]]}}$ is just $\boxed{\text{[[b]]}}$, where $\boxed{\text{b = [a]}}$, so in a sense, a three-layered list is just a two layered list, but rather than the last layer being 'flat', it is composed of another list. So if we apply our list of lists (of lists) to $\boxed{\text{join}}$, it will flatten those outer two layers into one. As the second layer wasn't flat but instead contained a third layer, we will still end up with a list of lists, which the other join flattens. Summing up, the left-hand side will flatten the inner two layers into a new layer, then flatten this with the outermost layer. The right-hand side will flatten the outer two layers, then flatten this with the innermost layer. These two operations should be equivalent. It's sort of like a law of associativity for $\boxed{\text{join}}$.

$\boxed{\text{Maybe}}$ is also a monad, with

```
return :: a -> Maybe a
return x = Just x

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

So if we had a *three*-layered Maybe (i.e., it could be $\boxed{\text{Nothing}}$, $\boxed{\text{Just Nothing}}$, $\boxed{\text{Just (Just Nothing)}}$ or $\boxed{\text{Just (Just (Just x)))}}$, the first law says that collapsing the inner two layers first, then that with the outer layer is exactly the same as collapsing the outer layers first, then that with the innermost layer.

The second law

$\boxed{\text{join . fmap return = join . return = id}}$

What about the second law, then? Again, we'll start with the example of lists. Both functions mentioned in the second law are functions `[a] -> [a]`. The left-hand side expresses a function that maps over the list, turning each element $\boxed{\text{x}}$ into its singleton list $\boxed{\text{[x]}}$, so that at the end we're left with a list of singleton lists. This two-layered list is flattened down into a single-layer list again using the $\boxed{\text{join}}$. The right hand side, however, takes the entire list $\boxed{\text{[x, y, z, ...]}}$, turns it into the singleton list of lists $\boxed{\text{[[x, y, z, ...]]}}$, then flattens the two layers down into one again. This law is less obvious to state quickly, but it essentially says that applying $\boxed{\text{return}}$ to a monadic value, then $\boxed{\text{join}}$ing the result should have the same effect whether you perform the $\boxed{\text{return}}$ from inside the top layer or from outside it.

The third and fourth laws

$\boxed{\text{return . f = fmap f . return}}$

$\boxed{\text{join . fmap (fmap f) = fmap f . join}}$

The last two laws express more self evident fact about how we expect monads to behave. The easiest way to see how they are true is to expand them to use the expanded form:

```
1.   \x -> return (f x) = \x -> fmap f (return x)

2.   \x -> join (fmap (fmap f) x) = \x -> fmap f (join x)
```

Application to do-blocks

Well, we have intuitive statements about the laws that a monad must support, but why is that important? The answer becomes obvious when we consider do-blocks. Recall that a do-block is just syntactic sugar for a combination of statements involving $\boxed{\text{(>>=)}}$ as witnessed by the usual translation:

```
do { x }                 -->  x
do { let { y = v }; x }  -->  let y = v in do { x }
do { v <- y; x }         -->  y >>= \v -> do { x }
do { y; x }              -->  y >>= \_ -> do { x }
```

Also notice that we can prove what are normally quoted as the monad laws using $\boxed{\text{return}}$ and $\boxed{\text{(>>=)}}$ from our above laws (the proofs are a little heavy in some cases, feel free to skip them if you want to):

1. `return x >>= f = f x   -- First Law`

   Proof:

   ```
     return x >>= f
   = join (fmap f (return x)) -- By the definition of (>>=)
   = join (return (f x))      -- By law 3
   = (join . return) (f x)
   = id (f x)                 -- By law 2
   = f x
   ```

2. `m >>= return = m   -- Second Law`

   Proof:

```
 m >>= return
= join (fmap return m)    -- By the definition of (>>=)
= (join . fmap return) m
= id m                    -- By law 2
= m
```

3. `(m >>= f) >>= g = m >>= (\x -> f x >>= g)`

Proof (recall that $\boxed{\text{fmap f . fmap g = fmap (f . g)}}$):

```
  (m >>= f) >>= g
= (join (fmap f m)) >>= g                      -- By the definition of (>>=)
= join (fmap g (join (fmap f m)))              -- By the definition of (>>=)
= (join . fmap g) (join (fmap f m))
= (join . fmap g . join) (fmap f m)
= (join . join . fmap (fmap g)) (fmap f m)     -- By law 4
= (join . join . fmap (fmap g) . fmap f) m
= (join . join . fmap (fmap g . f)) m          -- By the distributive law of functors
= (join . join . fmap (\x -> fmap g (f x))) m
= (join . fmap join . fmap (\x -> fmap g (f x))) m -- By law 1
= (join . fmap (join . (\x -> fmap g (f x)))) m    -- By the distributive law of functors
= (join . fmap (\x -> join (fmap g (f x)))) m
= (join . fmap (\x -> f x >>= g)) m            -- By the definition of (>>=)
= join (fmap (\x -> f x >>= g) m)
= m >>= (\x -> f x >>= g)                      -- By the definition of (>>=)
```

These new monad laws, using $\boxed{\text{return}}$ and $\boxed{(>>=)}$, can be translated into do-block notation.

Points-free style	Do-block style
`return x >>= f = f x`	`do { v <- return x; f v } = do { f x }`
`m >>= return = m`	`do { v <- m; return v } = do { m }`

`(m >>= f) >>= g = m >>= (\x -> f x >>= g)`	`do { y <- do { x <- m; f x }; g y}`
	`=`
	`do { x <- m; y <- f x; g y }`

The monad laws are now common-sense statements about how do-blocks should function. If one of these laws were invalidated, users would become confused, as you couldn't be able to manipulate things within the do-blocks as would be expected. The monad laws are, in essence, usability guidelines.

Appendix A

Appendix: The fundamental groupoid

NOT directly RELATED TO MONAD

Let X be a topological space. We define a category $\Pi(X)$ (which will turn out to be a groupoid) as follows.

- The objects of $\Pi(X)$ are the points of X.

In order to define morphisms in $\Pi(X)$ we need to recall the notion of *homotopy of paths*. Suppose $x, y \in X$ and $\gamma_0, \gamma_1 : [0,1] \to X$ are continous maps (where the closed interval [0,1] is equipped with the usual topology) such that $\gamma_0(0) = x = \gamma_1(0)$ and $\gamma_0(1) = y = \gamma_1(1)$. (One can say that γ_0 and γ_1 are (continous) paths from x to y).

We say that γ_0 and γ_1 are **homotopic** if there exists a continous map $H : [0,1] \times [0,1] \to X$, called a **homotopy** between γ_0 and γ_1, such that $H(t,0) = \gamma_0(t)$ and $H(t,1) = \gamma_1(t)$, $\forall t \in [0,1]$, and also $H(0,s) = x$ and $H(1,s) = y$, $\forall s \in [0,1]$. Observe that these conditions can be rephrased as follows. If we define $H_s(t) = H(t,s)$, then, for every fixed $s \in [0,1]$, H_s should be a (continous) path from x to y, and, moreover, one should have $H_0 = \gamma_0$ and $H_1 = \gamma_1$.

Check that being homotopic is an equivalence relation on continuous paths from x to y. This allows us to define, for every pair $x, y \in X$, the set of equivalence classes of continuous paths from x to y modulo homotopy.

- For two objects *x, y* of $\Pi(X)$, we define $Hom_{\Pi(X)}(x,y)$ to be the set of homotopy classes of continuous paths from x to y.

Finally, we need to define composition of paths. If $\alpha : [0,1] \to X$ and $\beta : [0,1] \to Y$ are continuous maps such that $\alpha(1) = \beta(0)$, we can define a continuous map

$$\beta \cdot \alpha : [0,1] \to X; \qquad t \mapsto \begin{cases} \alpha(2t), & \text{if } 0 \le t \le \frac{1}{2} \\ \beta(2t-1), & \text{if } \frac{1}{2} \le t \le 1 \end{cases}$$

Check that composition of paths respects homotopy. In other words, if α_0 , α_1 are two homotopic paths from $x \to y$, and β_0 , β_1 are two homotopic paths from $y \to z$, then the paths $\beta_0 \cdot \alpha_0$ and $\beta_1 \cdot \alpha_0$ from $x \to z$ are also homotopic.

- This allows us to define composition of morphisms in $\Pi(X)$ unambiguously: if $x, y, z \in X$, then to define the composition map

$$Hom(y,z) \times Hom(x,y) \to Hom(x,z)$$

let us pick equivalence classes $f \in Hom(x,y)$, $g \in Hom(y,z)$ and representatives $\alpha : [0,1] \to X$, $\beta : [0,1] \to X$ of f and g, respectively. Then we define $g \circ f$ to be the equivalence class of $\beta \cdot \alpha$.

***Prop:* 8.** *Verify the following statements:*

1. *Composition of homotopy classes of continuous paths is associative. (There is something to think about, because composition of continuous paths, before passing to homotopy classes, is NOT associative!)*

2. *The definitions above turn* $\Pi(X)$ *into a category.*

3. $\Pi(X)$ *is in fact a groupoid. Indeed, if* $f : x \to y$ *is a morphism in* $\Pi(X)$ *and* $\alpha : [0,1] \to X$ *is a representative of* f*, as before, check that the equivalence class of the path* $\alpha^{-1} : [0,1] \to X$ *defined by* $\alpha^{-1}(t) = \alpha(1-t)$ *is an inverse of* f*.*

Def: One calls $\Pi(X)$ the **fundamental groupoid** of the topological space X. If we fix a point $x \in X$, then, in particular, all morphisms from x to x in $\Pi(X)$ form a group which we denote $Aut_{\Pi(X)}(x)$. In topology it has a diferent name: the *fundamental group of* X *at the point* x is defined to be

$$\pi_1(X,x) := Aut_{\Pi(X)}(x)$$

Check that this definition of the fundamental group is equivalent to the other one(s) you have seen. The proof will be essentially tautological. The definition of the fundamental groupoid is no more complicated than the definition of the fundamental group; however, for many purposes it is much more convenient to think in terms of the fundamental groupoid rather than the fundamental group.

For example, the definition of the fundamental groupoid is completely canonical, while the definition of the fundamental group depends on the choice of a base point. In particular, if X has several connected components, then $\Pi(X)$ "keeps track" of all of them, while if we choose a base point $x \in X$, then $\pi_1(X,x)$ does not know anything about the connected components of X that do not contain x. For instance, if X is the disjoint union of a circle and a line, and $x \in X$ is a point lying on the line, then $\pi_1(X,x)$ is the trivial group, while $\Pi(X)$ looks like the "disjoint union" (you can try to think how to define the disjoint union of two categories in general - this is not dificult) of the fundamental groupoid of a circle and the fundamental groupoid of a line.

Appendix B

Appendix: Full Monad documentation

The full $\boxed{\text{Monad}}$ code, as found in the Prelude documentation when searching "Monad" in Hoogle

```
{- | The 'Monad' class defines the basic operations over a /monad/,
 a concept from a branch of mathematics known as /category theory/.
 From the perspective of a Haskell programmer, however, it is best to
 think of a monad as an /abstract datatype/ of actions.
 Haskell's @do@ expressions provide a convenient syntax for writing
 monadic expressions.

 Instances of 'Monad' should satisfy the following laws:

 * @'return' a '>>=' k  =  k a@
 * @m '>>=' 'return'  =  m@
 * @m '>>=' (\x -> k x '>>=' h)  =  (m '>>=' k) '>>=' h@

 Furthermore, the 'Monad' and 'Applicative' operations should relate as follows:

 * @'pure' = 'return'@
 * @('<*>') = 'ap'@

 The above laws imply:

 * @'fmap' f xs  =  xs '>>=' 'return' . f@
 * @('>>') = ('*>')@

 and that 'pure' and ('<*>') satisfy the applicative functor laws.

 The instances of 'Monad' for lists, 'Data.Maybe.Maybe' and 'System.IO.IO'
 defined in the "Prelude" satisfy these laws.
 -}
 class Applicative m => Monad m where
     -- | Sequentially compose two actions, passing any value produced
     -- by the first as an argument to the second.
     (>>=)       :: forall a b. m a -> (a -> m b) -> m b
```

```
    -- | Sequentially compose two actions, discarding any value produced
    -- by the first, like sequencing operators (such as the semicolon)
    -- in imperative languages.
    (>>)        :: forall a b. m a -> m b -> m b
    m >> k = m >>= \_ -> k -- See Note [Recursive bindings for Applicative/Monad]
    {-# INLINE (>>) #-}

    -- | Inject a value into the monadic type.
    return      :: a -> m a
    return      = pure

    -- | Fail with a message.  This operation is not part of the
    -- mathematical definition of a monad, but is invoked on pattern-match
    -- failure in a @do@ expression.
    fail        :: String -> m a
    fail s      = error s

{- Note [Recursive bindings for Applicative/Monad]
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

The original Applicative/Monad proposal stated that after
implementation, the designated implementation of (>>) would become

  (>>) :: forall a b. m a -> m b -> m b
  (>>) = (*>)

by default. You might be inclined to change this to reflect the stated
proposal, but you really shouldn't! Why? Because people tend to define
such instances the /other/ way around: in particular, it is perfectly
legitimate to define an instance of Applicative (*>) in terms of (>>),
which would lead to an infinite loop for the default implementation of
Monad! And people do this in the wild.

This turned into a nasty bug that was tricky to track down, and rather
than eliminate it everywhere upstream, it's easier to just retain the
original default.

-}
```

Appendix C

Appendix: the Monoid type class

Not to be confused with the Monad class, the more pleasant Monoid class, with kind `* -> Constraint` , found in the $\boxed{\text{Data.Monoid}}$ module, modelizes the semigroups or monoids.

A **monoid** in Mathematics is an algebraic structure consisting of a set of objects with an operation between them, being this operation *associative* and with a *neutral element*. Phew! But what is the meaning of this? By *associative* we mean that, if you have three elements a, b and c, then $a * (b * c) = (a * b) * c$. A *neutral element* is the one that does not worth to operate with, because it does nothing! To say, e is a *neutral element* if $e*a = a*e = a$, given any object a. As an example, you may take the real numbers as objects and the ordinary multiplication as operation.

Now that you know the math basics behind the Monoid class, let's see its definition:

```
class Monoid m where
   mempty :: m
   mappend :: m -> m -> m
   mconcat :: [m] -> m
   mconcat = foldr mappend mempty
   (<>) :: m -> m -> m    -- infix synonym for mappend
```

See that **mappend** corresponds to the monoid operation and **mempty** to its neutral element. The names of the methods may seem unsuitable, but they correspond to an example of monoid: the lists with the appending `(++)` operation. Who is the neutral element here? The empty list:

```
xs ++ [] = [] ++ xs = xs
```

Some examples:

The list monoid

```
instance Monoid [a] where
        mempty  = []
        mappend = (++)
        mconcat xss = [x | xs <- xss, x <- xs]
```

The monoid of functions with range a monoid

```
instance Monoid b => Monoid (a -> b) where
        mempty _ = mempty
        mappend f g x = f x `mappend` g x
```

The Unit monoid

```
instance Monoid () where
        mempty        = ()
        _ `mappend` _ = ()
        mconcat _     = ()
```

The cartesian product of two monoids

```
instance (Monoid a, Monoid b) => Monoid (a,b) where
        mempty = (mempty, mempty)
        (a1,b1) `mappend` (a2,b2) =
                (a1 `mappend` a2, b1 `mappend` b2)
```

Lexicographical ordering

```
instance Monoid Ordering where
        mempty         = EQ
        LT `mappend` _ = LT
        EQ `mappend` y = y
        GT `mappend` _ = GT
```

Lift a semigroup into 'Maybe' forming a 'Monoid'

```
instance Monoid a => Monoid (Maybe a) where
   mempty = Nothing
   Nothing `mappend` m = m
   m `mappend` Nothing = m
   Just m1 `mappend` Just m2 = Just (m1 `mappend` m2)
```

As you can see in all the examples, the following rules are verified:

```
  (x <> y) <> z = x <> (y <> z)    -- associativity
  mempty <> x = x                  -- left identity
  x <> mempty = x                  -- right identity
```

Appendix D

Appendix: the Maybe monad

We introduced monads using $\boxed{\text{Maybe}}$ as an example. The $\boxed{\text{Maybe}}$ monad represents computations which might "go wrong" by not returning a value. For reference, here are our definitions of $\boxed{\text{return}}$ and $\boxed{(>>=)}$ for $\boxed{\text{Maybe}}$ as we saw in the main body:[1]

```
return :: a -> Maybe a
return x  = Just x

(>>=)  :: Maybe a -> (a -> Maybe b) -> Maybe b
m >>= g = case m of
             Nothing -> Nothing
             Just x  -> g x
```

D.1 Safe functions

The $\boxed{\text{Maybe}}$ datatype provides a way to make a safety wrapper around *partial functions*, that is, functions which can fail to work for a range of arguments. For example, $\boxed{\text{head}}$ and $\boxed{\text{tail}}$ only work with non-empty lists. Another typical case, which we will explore in this section, are mathematical functions like $\boxed{\text{sqrt}}$ and $\boxed{\text{log}}$; (as far as real numbers are concerned) these are only defined for non-negative arguments.

```
> log 1000
6.907755278982137
> log (-1000)
''ERROR'' -- runtime error
```

To avoid this crash, a "safe" implementation of log could be:

```
safeLog :: (Floating a, Ord a) => a -> Maybe a
safeLog x
    | x > 0     = Just (log x)
    | otherwise = Nothing

> safeLog 1000
Just 6.907755278982137
```

[1] The definitions in the actual instance in $\boxed{\text{Data.Maybe}}$ are written a little differently, but are fully equivalent to these.

```
> safeLog -1000
Nothing
```

We could write similar "safe functions" for all functions with limited domains such as division, square-root, and inverse trigonometric functions ($\boxed{\text{safeDiv}}$, $\boxed{\text{safeSqrt}}$, $\boxed{\text{safeArcSin}}$, etc. all of which would have the same *type* as $\boxed{\text{safeLog}}$ but definitions specific to their constraints)

If we wanted to combine these monadic functions, the cleanest approach is with monadic composition and point-free style:

```
safeLogSqrt = safeLog <=< safeSqrt
```

Written in this way, $\boxed{\text{safeLogSqrt}}$ resembles a lot its unsafe, non-monadic counterpart:

```
unsafeLogSqrt = log . sqrt
```

D.2 Lookup tables

A lookup table relates *keys* to *values*. You look up a value by knowing its key and using the lookup table. For example, you might have a phone book application with a lookup table where contact names are keys to corresponding phone numbers. An elementary way of implementing lookup tables in Haskell is to use a list of pairs: $\boxed{\text{[(a, b)]}}$. Here $\boxed{\text{a}}$ is the type of the keys, and $\boxed{\text{b}}$ the type of the values.[2]Here's how the phone book lookup table might look like:

```
phonebook :: [(String, String)]
phonebook = [ ('Bob',   '01788 665242'),
              ('Fred',  '01624 556442'),
              ('Alice', '01889 985333'),
              ('Jane',  '01732 187565') ]
```

The most common thing you might do with a lookup table is look up values. Everything is fine if we try to look up "Bob", "Fred", "Alice" or "Jane" in our phone book, but what if we were to look up "Zoe"? Zoe isn't in our phone book, so the lookup would fail. Hence, the Haskell function to look up a value from the table is a $\boxed{\text{Maybe}}$ computation (it is available from Prelude):

```
lookup :: Eq a => a  -- a key
       -> [(a, b)]   -- the lookup table to use
       -> Maybe b    -- the result of the lookup
```

Let us explore some of the results from lookup:

```
Prelude> lookup 'Bob' phonebook
Just '01788 665242'
Prelude> lookup 'Jane' phonebook
Just '01732 187565'
Prelude> lookup 'Zoe' phonebook
Nothing
```

Now let's expand this into using the full power of the monadic interface. Say, we're now working for the government, and once we have a phone number from our contact, we want to look up this phone number in a big, government-sized lookup table to find out the registration number of their car. This, of course, will be another $\boxed{\text{Maybe}}$-computation. But if the person we're looking for

[2] Check $\boxed{\text{Data.Map}}$ for a different, and potentially more useful, implementation.

isn't in our phone book, we certainly won't be able to look up their registration number in the governmental database.

What we need is a function that will take the results from the first computation and put it into the second lookup *only* if we get a successful value in the first lookup. Of course, our final result should be `Nothing` if we get `Nothing` from either of the lookups.

```
getRegistrationNumber :: String       -- their name
                      -> Maybe String -- their Reg.Num.
getRegistrationNumber name =
  lookup name phonebook >>=
    (\number -> lookup number governmentDatabase)
```

If we then wanted to use the result from the governmental database lookup in a third lookup (say we want to look up their registration number to see if they owe any car tax), then we could extend our `getRegistrationNumber` function:

```
getTaxOwed :: String       -- their name
           -> Maybe Double -- the amount of tax they owe
getTaxOwed name =
  lookup name phonebook >>=
    (\number -> lookup number governmentDatabase) >>=
      (\registration -> lookup registration taxDatabase)
```

Or, using the `do`-block style:

```
getTaxOwed name = do
  number       <- lookup name phonebook
  registration <- lookup number governmentDatabase
  lookup registration taxDatabase
```

Let's just pause here and think about what would happen if we got a `Nothing` anywhere. By definition, when the first argument to `(>>=)` is `Nothing`, it just returns `Nothing` while ignoring whatever function it is given. Thus, a `Nothing` at any stage in the large computation will result in a `Nothing` overall, regardless of the other functions. After the first `Nothing` hits, all `(>>=)`s will just pass it to each other, skipping the other function arguments. The technical description says that the structure of the `Maybe` monad **propagates failures**.

D.3 Open monads

Another trait of the `Maybe` monad is that it is **open**: if we have a `Just` value, we can see the contents and extract the associated values through pattern matching.

```
zeroAsDefault :: Maybe Int -> Int
zeroAsDefault mx = case mx of
    Nothing -> 0
    Just x -> x
```

This usage pattern of replacing `Nothing` with a default is captured by the `fromMaybe` function in `Data.Maybe`.

```
zeroAsDefault :: Maybe Int -> Int
zeroAsDefault mx = fromMaybe 0 mx
```

The $\boxed{\text{maybe}}$ Prelude function allows us to do it in a more general way, by supplying a function to modify the extracted value.

```
displayResult :: Maybe Int -> String
displayResult mx = maybe s1 ((s2++).show) mx
  where
    s1 = 'There was no result'
    s2 = 'The result was'
```

```
Prelude> :t maybe
maybe :: b -> (a -> b) -> Maybe a -> b
Prelude> displayResult (Just 10)
'The result was 10'
Prelude> displayResult Nothing
'There was no result'
```

This possibility makes sense for $\boxed{\text{Maybe}}$, as it allows us to recover from failures. Not all monads are open in this way; often, they are designed to hide unnecessary details. $\boxed{\text{return}}$ and $\boxed{\text{(>>=)}}$ alone do not allow us to extract the underlying value from a monadic computation, and so it is perfectly possible to make a "no-exit" monad, from which it is never possible to extract values. The most obvious example of that is the $\boxed{\text{IO}}$ monad.

D.4 Maybe and safety

We have seen how $\boxed{\text{Maybe}}$ can make code safer by providing a graceful way to deal with failure that does not involve runtime errors. Does that mean we should always use $\boxed{\text{Maybe}}$ for everything? Not really.

When you write a function, you are able to tell whether it might fail to produce a result during normal operation of the program[3], either because the functions you use might fail (as in the examples in this chapter) or because you know some of the argument or intermediate result values do not make sense (for instance, imagine a calculation that is only meaningful if its argument is less than 10). If that is the case, by all means use $\boxed{\text{Maybe}}$ to signal failure; it is far better than returning an arbitrary default value or throwing an error.

Now, adding $\boxed{\text{Maybe}}$ to a result type without a reason would only make the code more confusing and no safer. The type signature of a function with unnecessary $\boxed{\text{Maybe}}$ would tell users of the code that the function could fail when it actually can't. Of course, that is not as bad a lie as the opposite one (that is, claiming that a function will not fail when it actually can), but we really want honest code in *all* cases. Furthermore, using $\boxed{\text{Maybe}}$ forces us to propagate failure (with $\boxed{\text{fmap}}$ or monadic code) and eventually handle the failure cases (using pattern matching, the $\boxed{\text{maybe}}$ function, or $\boxed{\text{fromMaybe}}$ from $\boxed{\text{Data.Maybe}}$). If the function cannot actually fail, coding for failure is an unnecessary complication.

[3] With "normal operation" we mean to exclude failure caused by uncontrollable circumstances in the real world, such as memory exhaustion or a dog chewing the printer cable.

Appendix E

Appendix: The List monad

Lists are a fundamental part of Haskell, and we've used them extensively before getting to this chapter. The novel insight is that the list type is a monad too!

As monads, lists are used to model *nondeterministic* computations which may return an arbitrary number of results. There is a certain parallel with how $\boxed{\text{Maybe}}$ represented computations which could return zero or one value; but with lists, we can return zero, one, or many values (the number of values being reflected in the length of the list).

E.1 List instantiated as monad

The $\boxed{\text{return}}$ function for lists simply injects a value into a list:

```
return x  =  [x]
```

In other words, $\boxed{\text{return}}$ here makes a list containing one element, namely the single argument it took. The type of the *list return* is $\boxed{\text{return} :: a \to [a]}$, or, equivalently, $\boxed{\text{return} :: a \to [\,]\,a}$. The latter style of writing it makes it more obvious that we are replacing the generic type constructor (which we had called $\boxed{\text{M}}$ in Understanding monads) by the list type constructor $\boxed{[\,]}$ (which is distinct from but easy to confuse with the empty list!).

The binding operator is less trivial. We will begin by considering its type, which for the case of lists should be:

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

This is just what we'd expect: it pulls out the value from the list to give to a function that returns a new list.

The actual process here involves first mapping a given function over a given list to get back a list of lists, i.e. type $\boxed{[[\text{b}]]}$ (of course, many functions which you might use in mapping do not return lists; but, as shown in the type signature above, **monadic binding for lists only works with functions that return lists**). To get back to a regular list, we then concatenate the elements of our list of lists to get a final result of type $\boxed{[\text{b}]}$. Thus, we can define the list version of $\boxed{(>>=)}$:

```
xs >>= f = concat (map f xs)
```

The bind operator is key to understanding how different monads do their jobs, and its definition indicates the chaining strategy for working with the monad.

For the list monad, non-determinism is present because different functions may return any number of different results when mapped over lists.

Bunny invasion

```
Prelude> let generation = replicate 2
Prelude> ['bunny'] >>= generation
['bunny','bunny']
Prelude> ['bunny'] >>= generation >>= generation
['bunny','bunny','bunny','bunny']
```

In this silly example all elements are equal, but the same overall logic could be used to model radioactive decay, or chemical reactions, or any phenomena that produces a series of elements starting from a single one.

E.2 Board game example

Suppose we are modeling a turn-based board game and want to find all the possible ways the game could progress. We would need a function to calculate the list of options for the next turn, given a current board state:

```
nextConfigs :: Board -> [Board]
nextConfigs bd = undefined -- details not important
```

To figure out all the possibilities after two turns, we would again apply our function to each of the elements of our new list of board states. Our function takes a single board state and returns a list of possible new states. Thus, we can use monadic binding to map the function over each element from the list:

```
nextConfigs bd >>= nextConfigs
```

In the same fashion, we could bind the result back to the function yet again (ad infinitum) to generate the next turn's possibilities. Depending on the particular game's rules, we may reach board states that have no possible next-turns; in those cases, our function will return the empty list.

On a side note, we could translate several turns into a do block (like we did for the grandparents example in Understanding monads):

```
threeTurns :: Board -> [Board]
threeTurns bd = do
  bd1 <- nextConfigs bd
  bd2 <- nextConfigs bd1
  nextConfigs bd2
```

If the above looks too magical, keep in mind that `do` notation is syntactic sugar for `(>>=)` operations. To the right of each left-arrow, there is a function with arguments that evaluate to a list; the variable to the left of the arrow stands for the list elements. After a left-arrow assignment line, there can be later lines that call the assigned variable as an argument for a function. This later function will be performed for *each* of the elements from within the list that came from the left-arrow line's function. This per-element process corresponds to the 'map' in the definition of `(>>=)`. A resulting list of lists (one per element of the original list) will be flattened into a single list (the 'concat' in the definition of `(>>=)`).

E.3 List comprehensions

The list monad works in a way that has uncanny similarity to list comprehensions. Let's slightly modify the `do` block we just wrote for `threeTurns` so that it ends with a `return`...

```
threeTurns bd = do
  bd1 <- nextConfigs bd
  bd2 <- nextConfigs bd1
  bd3 <- nextConfigs bd2
  return bd3
```

This mirrors exactly the following list comprehension:

```
threeTurns bd =
  [ bd3 | bd1 <- nextConfigs bd, bd2 <- nextConfigs bd1, bd3 <- nextConfigs bd2 ]
```

(In a list comprehension, it is perfectly legal to use the elements drawn from one list to define the following ones, like we did here.)

The resemblance is no coincidence: list comprehensions are, behind the scenes, defined in terms of $\boxed{\text{concatMap}}$

```
concatMap f xs = concat (map f xs)
```

. That's just the list monad binding definition again! To summarize the nature of the list monad: binding for the list monad is a combination of concatenation and mapping, and so the combined function $\boxed{\text{concatMap}}$ is effectively the same as $\boxed{\text{(>>=)}}$ for lists (except for different syntactic order).

For the correspondence between list monad and list comprehension to be complete, we need a way to reproduce the filtering that list comprehensions can do. Search for Additive Monads (MonadPlus).

Appendix F

Appendix: The IO (Input/Output) monad

Haskell is a *functional* and *lazy* language. However, the real world effects of input/output operations can't be expressed through pure functions. Furthermore, in most cases I/O can't be done lazily. Since lazy computations are only performed when their values become necessary, unfettered lazy I/O would make the order of execution of the real world effects unpredictable. Haskell addresses these issues through the $\boxed{\text{IO}}$ monad.

F.1 Input/output and purity

Haskell functions are *pure*: when given the same arguments, they return the same results. Pure functions are reliable and predictable; they ease debugging and validation. Test cases can also be set up easily since we can be sure that nothing other than the arguments will influence a function's result. Being entirely contained within the program, the Haskell compiler can evaluate functions thoroughly in order to optimize the compiled code.

So, how do we manage actions like opening a network connection, writing a file, reading input from the outside world, or anything else that does something more than returning a calculated result? Well, the key is: *these actions are not functions.* The $\boxed{\text{IO}}$ monad is a means to represent actions as Haskell values, so that we can manipulate them with pure functions.

F.2 Combining functions and I/O actions

Let's combine functions with I/O to create a full program that will:

1. Ask the user to insert a string
2. Read their string
3. Use $\boxed{\text{fmap}}$ to apply a function $\boxed{\text{shout}}$ that capitalizes all the letters from the string
4. Write the resulting string

```
module Main where

import Data.Char (toUpper)
import Control.Monad
```

```
main = putStrLn "Write your string: " >> fmap shout getLine >>= putStrLn

shout = map toUpper
```

We have a full-blown program, but we didn't include any type definitions. Which parts are functions and which are IO actions or other values? We can load our program in GHCi and check the types:

```
main :: IO ()
putStrLn :: String -> IO ()
"Write your string: " :: [Char]
(>>) :: Monad m => m a -> m b -> m b
fmap :: Functor m => (a -> b) -> m a -> m b
shout :: [Char] -> [Char]
getLine :: IO String
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

Whew, that is a lot of information there. We've seen all of this before, but let's review.

main is IO (). That's not a function. Functions are of types $a \to b$. Our entire program is an IO action.

putStrLn is a function, but it results in an IO action. The "Write your string:" text is a String (remember, that's just a synonym for [Char]). It is used as an argument for putStrLn and is incorporated into the IO action that results. So, putStrLn is a function, but putStrLn x evaluates to an IO action. The () part of the IO type indicates that nothing is available to be passed on to any later functions or actions.That last part is key. We sometimes say informally that an IO action "returns" something; however, taking that too literally leads to confusion. It is clear what we mean when we talk about *functions* returning results, but IO actions are not functions. Let's skip down to getLine - an IO action that *does* provide a value. getLine is not a function that returns a String because getLine *isn't a function*. Rather, getLine is an IO action which, when evaluated, will materialize a String, which can then be passed to later functions through, for instance, fmap and (>>=).When we use getLine to get a String, the value is monadic because it is wrapped in IO functor (which happens to be a monad). We cannot pass the value directly to a function that takes plain (non-monadic, or non-functorial) values. fmap does the work of taking a non-monadic function while passing in and returning monadic values.

As we've seen already, (>>=) does the work of passing a monadic value into a function that takes a non-monadic value and returns a monadic value. It may seem inefficient for fmap to take the non-monadic result of its given function and return a monadic value only for (>>=) to then pass the underlying non-monadic value to the next function. It is precisely this sort of chaining, however, that creates the reliable sequencing that make monads so effective at integrating pure functions with IO actions.

do notation review

Given the emphasis on sequencing, the do notation can be especially appealing with the IO monad. Our program

```
putStrLn "Write your string: " >> fmap shout getLine >>= putStrLn
```

could be written as:

```
do putStrLn "Write your string: "
   string <- getLine
   putStrLn (shout string)
```

F.3 The universe as part of our program

One way of viewing the IO monad is to consider IO a as a computation which provides a value of type a while changing *the state of the world* by doing input and output. Obviously, you cannot literally set the state of the world; it is hidden from you, as the IO functor is abstract (that is, you cannot dig into it to see the underlying values; it is closed in a way opposite to that in which Maybe can be said to be open). Seen this way, IO is roughly analogous to the State monad, which we will meet shortly. With State, however, the state being changed is made of normal Haskell values, and so we can manipulate it directly with pure functions.

Understand that this idea of the universe as an object affected and affecting Haskell values through IO is only a metaphor; a loose interpretation at best. The more mundane fact is that IO simply brings some very base-level operations into the Haskell language.[1] Remember that Haskell is an abstraction, and that Haskell programs must be compiled to machine code in order to actually run. The actual workings of IO happen at a lower level of abstraction, and are wired into the very definition of the Haskell language.[2]

F.4 Pure and impure

Consider the following snippet:

```
speakTo :: (String -> String) -> IO String
  speakTo fSentence = fmap fSentence getLine

  -- Usage example.
  sayHello :: IO String
  sayHello = speakTo (\name -> "Hello, " ++ name ++ "!")
```

In most other programming languages, which do not have separate types for I/O actions, speakTo would have a type akin to:

```
speakTo :: (String -> String) -> String
```

With such a type, however, speakTo would not be a function at all! Functions produce the same results when given the same arguments; the String delivered by speakTo, however, also depends on whatever is typed at the terminal prompt. In Haskell, we avoid that pitfall by returning an IO String, which is not a String but a promise that *some* String will be delivered by carrying out certain instructions involving I/O (in this case, the I/O consists of getting a line of input from the terminal). Though the String can be different each time speakTo is evaluated, the I/O instructions are always the same.

[1] The technical term is "primitive", as in primitive operations.

[2] The same can be said about all higher-level programming languages, of course. Incidentally, Haskell's IO operations can actually be extended via the *Foreign Function Interface* (FFI) which can make calls to C libraries. As C can use inline assembly code, Haskell can indirectly engage with anything a computer can do. Still, Haskell functions manipulate such outside operations only *indirectly* as values in IO functors.

When we say Haskell is a purely functional language, we mean that all of its functions are *really* functions, which is not the case in most other languages. To be precise, Haskell expressions are always *referentially transparent*; that is, you can always replace an expression (such as `speakTo`) with its value (in this case, `\fSentence -¿ fmap fSentence getLine`) without changing the behaviour of the program. The `String` delivered by `getLine`, in contrast, is opaque; its value is not specified and can't be discovered in advance by the program. If `speakTo` had the problematic type we mentioned above, `sayHello` would be a `String`; however, replacing it by any specific string would break the program.

In spite of Haskell being purely functional, `IO` actions can be said to be *impure* because their impact on the outside world are *side effects* (as opposed to the regular effects that are entirely contained within Haskell). Programming languages that lack purity may have side-effects in many other places connected with various calculations. Purely functional languages, however, assure that *even expressions with impure values are referentially transparent.* That means we can talk about, reason about and handle impurity in a purely functional way, using purely functional machinery such as functors and monads. While `IO` actions are impure, all of the Haskell functions that manipulate them remain pure.

Functional purity, coupled to the fact that I/O shows up in types, benefit Haskell programmers in various ways. The guarantees about referential transparency increase a lot the potential for compiler optimizations. `IO` values being distinguishable through types alone make it possible to immediately tell where we are engaging with side effects or opaque values. As `IO` itself is just another functor, we maintain to the fullest extent the predictability and ease of reasoning associated with pure functions.

F.5 Functional and imperative

When we introduced monads, we said that a monadic expression can be interpreted as a statement of an imperative language. That interpretation is immediately compelling for `IO`, as the language around IO actions looks a lot like a conventional imperative language. It must be clear, however, that we are talking about an *interpretation*. We are not saying that monads or `do` notation turn Haskell into an imperative language. The point is merely that you can view and understand monadic code in terms of imperative statements. The semantics may be imperative, but the implementation of monads and `(>>=)` is still purely functional. To make this distinction clear, let's look at a little illustration:

```
int x;
 scanf("%d", &x);
 printf("%d\n", x);
```

This is a snippet of C, a typical imperative language. In it, we declare a variable `x`, read its value from user input with `scanf` and then print it with `printf`. We can, within an `IO` do block, write a Haskell snippet that performs the same function and looks quite similar:

```
x <- readLn
 print x
```

Semantically, the snippets are nearly equivalent.[3] In the C code, however, the statements directly

[3]One difference is that `x` is a mutable variable in C, and so it is possible to declare it in one statement and set its value in the next; Haskell never allows such mutability. If we wanted to imitate the C code even more closely, we

correspond to instructions to be carried out by the program. The Haskell snippet, on the other hand, is desugared to:

```
readLn >>= \x -> print x
```

The desugared version has no statements, only functions being applied. We tell the program the order of the operations indirectly as a simple consequence of *data dependencies*: when we chain monadic computations with (>>=), we get the later results by applying functions to the results of the earlier ones. It just happens that, for instance, evaluating print x leads to a string to be printed in the terminal.

When using monads, Haskell allows us to write code with imperative semantics while keeping the advantages of functional programming.

F.6 I/O in the libraries

So far the only I/O primitives we have used were putStrLn and getLine and small variations thereof. The standard libraries, however, offer many other useful functions and actions involving IO. We present some of the most important ones in the next appendix, including the basic functionality needed for reading from and writing to files.

F.7 monadic control structures

Given that monads allow us to express sequential execution of actions in a wholly general way, could we use them to implement common iterative patterns, such as loops? In this section, we will present a few of the functions from the standard libraries which allow us to do precisely that. While the examples are presented here applied to IO, keep in mind that the following ideas apply to *every* monad.

Remember, there is nothing magical about monadic values; we can manipulate them just like any other values in Haskell. Knowing that, we might think to try the following function to get five lines of user input:

```
fiveGetLines = replicate 5 getLine
```

That won't do, however (try it in GHCi!). The problem is that replicate produces, in this case, a list of actions, while we want an action which returns a list (that is, IO [String] rather than [IO String]). What we need is a *fold* to run down the list of actions, executing them and combining the results into a single list. As it happens, there is a Prelude function which does that: sequence.

```
sequence :: (Monad m) => [m a] -> m [a]
```

And so, we get the desired action with:

```
fiveGetLines = sequence $ replicate 5 getLine
```

could have used an IORef, which is a cell that contains a value which can be destructively updated. For obvious reasons, IORefs can only be used within the IO monad.

`replicate` and `sequence` form an appealing combination; so `Control.Monad` offers a `replicateM` function for repeating an action an arbitrary number of times. `Control.Monad` provides a number of other convenience functions in the same spirit - monadic zips, folds, and so on.

```
fiveGetLinesAlt = replicateM 5 getLine
```

A particularly important combination is `map` and `sequence`. Together, they allow us to make actions from a list of values, run them sequentially, and collect the results. `mapM`, a Prelude function, captures this pattern:

```
mapM :: (Monad m) => (a -> m b) -> [a] -> m [b]
```

We also have variants of the above functions with a trailing underscore in the name, such as `sequence_`, `mapM_` and `replicateM_`. These discard any final values and so are appropriate when you are only interested in performing actions. Compared with their underscore-less counterparts, these functions are like the distinction between `(>>=)` and `(>>=)`. `mapM_` for instance has the following type:

```
mapM_ :: (Monad m) => (a -> m b) -> [a] -> m ()
```

Finally, it is worth mentioning that `Control.Monad` also provides `forM` and `forM_`, which are flipped versions of `mapM` and `mapM_`. `forM_` happens to be the idiomatic Haskell counterpart to the imperative for-each loop; and the type signature suggests that neatly:

```
forM_ :: (Monad m) => [a] -> (a -> m b) -> m ()
```

Appendix G

Appendix: The IO library

Here, we'll explore the most commonly used elements of the System.IO module.

```
data IOMode  = ReadMode   | WriteMode
             | AppendMode | ReadWriteMode

openFile     :: FilePath -> IOMode -> IO Handle
hClose       :: Handle -> IO ()

hIsEOF       :: Handle -> IO Bool

hGetChar     :: Handle -> IO Char
hGetLine     :: Handle -> IO String
hGetContents :: Handle -> IO String

getChar      :: IO Char
getLine      :: IO String
getContents  :: IO String

hPutChar     :: Handle -> Char -> IO ()
hPutStr      :: Handle -> String -> IO ()
hPutStrLn    :: Handle -> String -> IO ()

putChar      :: Char -> IO ()
putStr       :: String -> IO ()
putStrLn     :: String -> IO ()

readFile     :: FilePath -> IO String
writeFile    :: FilePath -> String -> IO ()
```

Note FilePath is a *type synonym* for String. So, for instance, the readFile function takes a String (the file to read) and returns an action that, when run, produces the contents of that file.

Most of the IO functions are self-explanatory. The openFile and hClose functions open and close a file, respectively. The IOMode argument determines the mode for opening the file. hIsEOF tests for end-of file. hGetChar and hGetLine read a character or line (respectively) from a file. hGetContents reads the entire file. The getChar, getLine, and getContents variants read from standard input. $\boxed{\text{hPutChar}}$ prints a character to a file; $\boxed{\text{hPutStr}}$ prints a string; and $\boxed{\text{hPutStrLn}}$ prints a string with a newline character at the end. The variants without the $\boxed{\text{h}}$ prefix work on standard output. The $\boxed{\text{readFile}}$ and $\boxed{\text{writeFile}}$ functions read and write an entire file without having to open it first.

G.1 Bracket

The $\boxed{\text{bracket}}$ function comes from the $\boxed{\text{Control.Exception}}$ module. It helps perform actions safely.

```
bracket :: IO a -> (a -> IO b) -> (a -> IO c) -> IO c
```

Consider a function that opens a file, writes a character to it, and then closes the file. When writing such a function, one needs to be careful to ensure that, if there were an error at some point, the file is still successfully closed. The $\boxed{\text{bracket}}$ function makes this easy. It takes three arguments: The first is the action to perform at the beginning. The second is the action to perform at the end, regardless of whether there's an error or not. The third is the action to perform in the middle, which might result in an error. For instance, our character-writing function might look like:

```
writeChar :: FilePath -> Char -> IO ()
writeChar fp c =
    bracket
      (openFile fp WriteMode)
      hClose
      (\h -> hPutChar h c)
```

This will open the file, write the character, and then close the file. However, if writing the character fails, $\boxed{\text{hClose}}$ will still be executed, and the exception will be reraised afterwards. That way, you don't need to worry too much about catching the exceptions and about closing all of your handles.

G.2 A file reading program

We can write a simple program that allows a user to read and write files. The interface is admittedly poor, and it does not catch all errors (such as reading a non-existent file). Nevertheless, it should give a fairly complete example of how to use IO. Enter the following code into "FileRead.hs", and compile/run:

```
module Main
      where

  import System.IO
  import Control.Exception

  main = doLoop

  doLoop = do
    putStrLn "Enter a command rFN wFN or q to quit:"
    command <- getLine
    case command of
      'q':_ -> return ()
      'r':filename -> do putStrLn ("Reading " ++ filename)
                         doRead filename
                         doLoop
```

```
      'w':filename -> do putStrLn ("Writing " ++ filename)
                         doWrite filename
                         doLoop
      _            -> doLoop

  doRead filename =
      bracket (openFile filename ReadMode) hClose
              (\h -> do contents <- hGetContents h
                        putStrLn "The first 100 chars:"
                        putStrLn (take 100 contents))

  doWrite filename = do
    putStrLn "Enter text to go into the file:"
    contents <- getLine
    bracket (openFile filename WriteMode) hClose
            (\h -> hPutStrLn h contents)
```

What does this program do? First, it issues a short string of instructions and reads a command. It then performs a **case** switch on the command and checks first to see if the first character is a 'q'. If it is, it returns a value of unit type.

Note The `return` function is a function that takes a value of type `a` and returns an action of type `IO a`. Thus, the type of `return ()` is `IO ()`.

If the first character of the command wasn't a 'q', the program checks to see if it was an 'r' followed by some string that is bound to the variable `filename`. It then tells you that it's reading the file, does the read and runs `doLoop` again. The check for 'w' is nearly identical. Otherwise, it matches `_`, the wildcard character, and loops to `doLoop`.

The `doRead` function uses the `bracket` function to make sure there are no problems reading the file. It opens a file in `ReadMode`, reads its contents and prints the first 100 characters.

The `doWrite` function asks for some text, reads it from the keyboard, and then writes it to the specified file.

Note Both `doRead` and `doWrite` could have been made simpler by using `readFile` and `writeFile`, but they were written in the extended fashion to show how the more complex functions are used.

The program has one major problem: it will die if you try to read a file that doesn't already exist or if you specify some bad filename like `*bs^#_@`. You may think that the calls to `bracket` in `doRead` and `doWrite` should take care of this, but they don't. They only catch exceptions within the main body, not within the startup or shutdown functions (`openFile` and `hClose`, in these cases). To make this completely reliable, we would need a way to catch exceptions raised by `openFile`.

Appendix H

Appendix: The State monad (Random Number Generation)

If you have programmed in any other language before, you likely wrote some functions that "kept state". For those new to the concept, a *state* is one or more variables that are required to perform some computation but are not among the arguments of the relevant function. Object-oriented languages, like C++, suggest extensive use of state variables within objects in the form of member variables. Programs written in procedural languages, like C, typically use variables declared outside the current scope to keep track of state.

In Haskell, however, such techniques are not as straightforward to apply. They require mutable variables and imply functions will have hidden dependencies, which is at odds with Haskell's functional purity. Fortunately, in most cases it is possible to avoid such extra complications and keep track of state in a functionally pure way. We do so by passing the state information from one function to the next, thus making the hidden dependencies explicit. The `State` type is a tool crafted to make this process of threading state through functions more convenient. In this chapter, we will see how it can assist us in a typical problem involving state: generating pseudo-random numbers.

H.1 Pseudo-Random Numbers

Generating actual random numbers is far from easy. Computer programs almost always use *pseudo*-random numbers instead. They are called "pseudo" because they are not truly random. Rather, they are genererated by algorithms (the pseudo-random number generators) which take an initial state (commonly called the seed) and produce from it a sequence of numbers that have the appearance of being random.[1]

Every time a pseudo-random number is requested, state somewhere must be updated, so that the generator can be ready for producing a fresh, different random number. Sequences of pseudo-random numbers can be replicated exactly if the initial seed and the generating algorithm are known.

H.1.1 Implementation in Haskell

Producing a pseudo-random number in most programming languages is very simple: there is a function somewhere in the libraries that provides a pseudo-random value (perhaps even a truly ran-

[1] A common source of seeds is the current date and time as given by the internal clock of the computer. Assuming the clock is functioning correctly, it can provide unique seeds suitable for most day-to-day needs (as opposed to applications which demand high-quality randomness, as in cryptography or statistics)

dom one, depending on how it is implemented). Haskell has a similar one in the `System.Random` module from the `random` package:

```
GHCi> :m System.Random
GHCi> :t randomIO
randomIO :: Random a => IO a
GHCi> randomIO
-1557093684
GHCi> randomIO
1342278538
```

`randomIO` is an `IO` action. It couldn't be otherwise, as it makes use of mutable state, which is kept out of reach from our Haskell programs. Thanks to this hidden dependency, the pseudo-random values it gives back can be different every time.

H.1.2 Example: rolling dice

Suppose we are coding a game in which at some point we need an element of chance. In real-life games that is often obtained by means of dice. So, let's create a dice-throwing function. We'll use the `IO` function `randomIO`, which allows us to specify a range from which the pseudo-random values will be taken. For a 6 die, the call will be `randomIO (1,6)`.

```
import Control.Monad
import System.Random

rollDiceIO :: IO (Int, Int)
rollDiceIO = liftM2 (,) (randomRIO (1,6)) (randomRIO (1,6))
```

That function rolls two dice. Here, `liftM2` is used to make the non-monadic two-argument function `(,)` work within a monad. The `(,)` is the non-infix version of the tuple constructor. Thus, the two die rolls will be returned as a tuple in `IO`.

Getting rid of *IO*

A disadvantage of `randomIO` is that it requires us to use `IO` and store our state outside the program, where we can't control what happens to it. We would rather only use I/O when there is an unavoidable reason to interact with the outside world.

To avoid bringing `IO` into play, we can build a *local* generator. The `random` and `mkStdGen` functions in `System.Random` allow us to generate tuples containing a pseudo-random number together with an updated generator to use the next time the function is called.

```
GHCi> :m System.Random
GHCi> let generator = mkStdGen 0
  -- ‘‘0’’ is our seed
GHCi> :t generator
generator :: StdGen
GHCi> generator
1 1
GHCi> :t random
random :: (RandomGen g, Random a) => g -> (a, g)
GHCi> random generator :: (Int, StdGen)
(2092838931,1601120196 1655838864)
```

Note In $\boxed{\text{random generator :: (Int, StdGen)}}$, we use the $\boxed{::}$ to introduce a *type annotation*, which is essentially a type signature that we can put in the middle of an expression. Here, we are saying that the expression to the right, $\boxed{\text{random generator}}$ has type $\boxed{\text{(Int, StdGen)}}$. It makes sense to use a type annotation here because, as we will discuss later, $\boxed{\text{random}}$ can produce values of different types, so if we want it to give us an $\boxed{\text{Int}}$ we'd better specify it in some way.

While we managed to avoid $\boxed{\text{IO}}$, there are new problems. First and foremost, if we want to use $\boxed{\text{generator}}$ to get random numbers, the obvious definition...

```
GHCi> let randInt = fst . random $ generator :: Int
GHCi> randInt
2092838931
```

... is useless. It will always give back the same value, $\boxed{2092838931}$, as the same generator in the same state will be used every time. To solve that, we can take the second member of the tuple (that is, the new generator) and feed it to a *new* call to $\boxed{\text{random}}$:

```
GHCi> let (randInt, generator') = random generator :: (Int, StdGen)
GHCi> randInt
  -- Same value
2092838931
GHCi> random generator' :: (Int, StdGen)
  -- Using new generator' returned from ``random generator''
(-2143208520,439883729 1872071452)
```

That, of course, is clumsy and rather tedious, as we now need to deal with the fuss of carefully passing the generator around.

H.1.3 Dice without IO

We can re-do our dice throw with our new approach using the $\boxed{\text{randomR}}$ function:

```
GHCi> randomR (1,6) (mkStdGen 0)
(6, 40014 40692)
```

The resulting tuple combines the result of throwing a single die with a new generator. A simple implementation for throwing two dice is then:

```
clumsyRollDice :: (Int, Int)
clumsyRollDice = (n, m)
        where
        (n, g) = randomR (1,6) (mkStdGen 0)
        (m, _) = randomR (1,6) g
```

The implementation of $\boxed{\text{clumsyRollDice}}$ works as an one-off, but we have to manually pass the generator $\boxed{\text{g}}$ from one $\boxed{\text{where}}$ clause to the other. This approach becomes increasingly cumbersome as our programs get more complex, which means we have more values to shift around. It is also error-prone: what if we pass one of the middle generators to the wrong line in the $\boxed{\text{where}}$ clause?

What we really need is a way to automate the extraction of the second member of the tuple (i.e. the new generator) and feed it to a new call to $\boxed{\text{random}}$. This is where the $\boxed{\text{State}}$ comes into the picture.

H.2 Introducing *State*

Note In this chapter we will use the state monad provided by the module `Control.Monad.Trans.State` of the `transformers` package. By reading Haskell code in the wild, you will soon meet `Control.Monad.State`, a module of the closely related `mtl` package. The differences between these two modules need not concern us at the moment; everything we discuss here also applies to the `mtl` variant.

The Haskell type `State` describes functions that consume a state and produce both a result and an updated state, which are given back in a tuple.

The state function is wrapped by a data type definition which comes along with a `runState` accessor so that pattern matching becomes unnecessary. For our current purposes, the `State` type might be defined as:

```
newtype  State s a  =  State { runState :: s -> (a, s) }
```

Here, `s` is the type of the state, and `a` the type of the produced result. Calling the type `State` is arguably a bit of a misnomer because the wrapped value is not the state itself but a *state processor*.

newtype

Note that we defined the data type with the `newtype` keyword, rather than the usual `data`. `newtype` can be used only for types with just one constructor and just one field.

It ensures that the trivial wrapping and unwrapping of the single field is eliminated by the compiler. For that reason, simple wrapper types such as `State` are usually defined with `newtype`. Would defining a synonym with `type` be enough in such cases? Not really, because `type` does not allow us to define instances for the new data type, which is what we are about to do...

H.2.1 Where did the *State* constructor go?

When you start using `Control.Monad.Trans.State`, you will quickly notice there is no `State` constructor available. The `transformers` package implements the `State` type in a somewhat different way. The differences do not affect how we use or understand `State`; except that, instead of a `State` constructor, `Control.Monad.Trans.State` exports a `state` function,

```
state :: (s -> (a, s)) -> State s a
```

which does the same job. As for *why* the implementation is not the obvious one we presented above, we will get back to that a few chapters down the road.

H.2.2 Instantiating the monad

So far, all we have done was to wrap a function type and give it a name. There is another ingredient, however: `State` is a monad, and that gives us very handy ways of using it. Unlike the instances of `Functor` or `Monad` we have seen so far, `State` has *two* type parameters. Since the type class only allows one parametrised parameter, the last one, we have to indicate the other one, `s`, will be fixed.

```
instance Monad (State s) where
```

That means there are actually *many* different `State` monads, one for each possible type of state - `State String`, `State Int`, `State SomeLargeDataStructure`, and so forth. Naturally, we only need to write one implementation of $\boxed{\text{return}}$ and $\boxed{\text{(>>=)}}$; the methods will be able to deal with all choices of $\boxed{\text{s}}$.

The $\boxed{\text{return}}$ function is implemented as:

```
return :: a -> State s a
return x = state ( \ st -> (x, st) )
```

Giving a value ($\boxed{\text{x}}$) to $\boxed{\text{return}}$ produces a function which takes a state ($\boxed{\text{st}}$) and returns it unchanged, together with value we want to be returned. As a finishing step, the function is wrapped up with the $\boxed{\text{state}}$ function.

Binding is a bit intricate:

```
(>>=) :: State s a -> (a -> State s b) -> State s b
pr >>= k = state $ \ st ->
   let (x, st') = runState pr st
       -- Running the first processor on st.
   in runState (k x) st'
       -- Running the second processor on st'.
```

$\boxed{\text{(>>=)}}$ is given a state processor ($\boxed{\text{pr}}$) and a function ($\boxed{\text{k}}$) that is used to create another processor from thGe result of the first one. The two processors are combined into a function that takes the $\boxed{\text{initial}}$ state ($\boxed{\text{st}}$) and returns the *second* result and the *third* state (i.e. the output of the second processor). Overall, $\boxed{\text{(>>=)}}$ here allows us to run two state processors in sequence, while allowing the result of the first stage to influence what happens in the second one.

One detail in the implementation is how $\boxed{\text{runState}}$ is used to undo the $\boxed{\text{State}}$ wrapping, so that we can reach the function that will be applied to the states. The type of $\boxed{\text{runState pr}}$, for instance, is `s -> (a, s)`.

H.2.3 Setting and accessing the State

The monad instance allows us to manipulate various state processors, but you may at this point wonder where exactly the *original* state comes from in the first place. That issue is handily dealt with by the function $\boxed{\text{put}}$:

```
put newState = state $ \_ -> ((), newState)
```

Given a state (the one we want to introduce), $\boxed{\text{put}}$ generates a state processor which ignores whatever state it receives, and gives back the state we originally provided to $\boxed{\text{put}}$. Since we don't care about the result of this processor (all we want to do is to replace the state), the first element of the tuple will be $\boxed{\text{()}}$, the universal placeholder value.

As a counterpart to $\boxed{\text{put}}$, there is $\boxed{\text{get}}$:

```
get = state $ \st -> (st, st)
```

The resulting state processor gives back the state $\boxed{\text{st}}$ it is given in both as a result and as a state. That means the state will remain unchanged, and that a copy of it will be made available for us to manipulate.

H.2.4 Getting Values and State

As we have seen in the implementation of `(>>=)`, `runState` is used to unwrap the `State a b` value to get the actual state processing function, which is then applied to some initial state. Other functions which are used in similar ways are `evalState` and `execState`. Given a `State a b` and an initial state, the function `evalState` will give back only the result value of the state processing, whereas `execState` will give back just the new state.

```
evalState :: State s a -> s -> a
evalState pr st = fst (runState pr st)

execState :: State s a -> s -> s
execState pr st = snd (runState pr st)
```

H.2.5 Dice and state

Time to use the `State` monad for our dice throw examples.

```
import Control.Monad.Trans.State
import System.Random
```

We want to generate `Int` dice throw results from a pseudo-random generator of type `StdGen`. Therefore, the type of our state processors will be `State StdGen Int`, which is equivalent to $\boxed{StdGenn; \to (Int, StdGen)}$ bar the wrapping.

We can now implement a processor that, given a `StdGen` generator, produces a number between 1 and 6. Now, the type of `randomR` is:

```
-- The StdGen type we are using is an instance of RandomGen.
randomR :: (Random a, RandomGen g) => (a, a) -> g -> (a, g)
```

Doesn't it look familiar? If we assume `a` is `Int` and `g` is `StdGen` it becomes:

```
randomR (1, 6) :: StdGen -> (Int, StdGen)
```

We already have a state processing function! All that is missing is to wrap it with `state`:

```
rollDie :: State StdGen Int
rollDie = state $ randomR (1, 6)
```

For illustrative purposes, we can use `get`, `put` and do-notation to write `rollDie` in a very verbose way which displays explicitly each step of the state processing:

```
rollDie :: State StdGen Int
rollDie = do generator <- get
             let (value, newGenerator) = randomR (1,6) generator
             put newGenerator
             return value
```

Let's go through each of the steps:

1. First, we take out the pseudo-random generator from the monadic context with $\boxed{< -}$, so that we can manipulate it.

2. Then, we use the `randomR` function to produce an integer between 1 and 6 using the generator we took. We also store the new generator graciously returned by `randomR`.

3. We then set the state to be the `newGenerator` using `put`, so that any further `randomR` in the do-block, or further on in a `(>>=)` chain, will use a different pseudo-random generator.

4. Finally, we inject the result back into the `State StdGen` monad using `return`.

We can finally use our monadic die. As before, the initial generator state itself is produced by the `mkStdGen` function.

```
GHCi> evalState rollDie (mkStdGen 0)
6
```

Why have we involved monads and built such an intricate framework only to do exactly what `fst $ randomR (1,6)` already does? Well, consider the following function:

```
rollDice :: State StdGen (Int, Int)
rollDice = liftM2 (,) rollDie rollDie
```

We obtain a function producing *two* pseudo-random numbers in a tuple. Note that these are in general different:

```
GHCi> evalState rollDice (mkStdGen 666)
 (6,1)
```

Under the hood, state is being passed through `(>>=)` from one `rollDie` computation to the other. Doing that was previously very clunky using `randomR (1,6)` alone because we had to pass state manually. Now, the monad instance is taking care of that for us. Assuming we know how to use the lifting functions, constructing intricate combinations of pseudo-random numbers (tuples, lists, whatever) has suddenly become much easier.

H.3 Pseudo-random values of different types

Until now, we have used only `Int` as type of the value produced by the pseudo-random generator. However, looking at the type of `randomR` shows we are not restricted to `Int`. It can generate values of any type in the `Random` class from `System.Random`. There already are instances for `Int`, `Char`, `Integer`, `Bool`, `Double` and `Float`, so you can immediately generate any of those.

Because `State StdGen` is "agnostic" in regard to the type of the pseudo-random value it produces, we can write a similarly "agnostic" function that provides a pseudo-random value of unspecified type (as long as it is an instance of `Random`):

```
getRandom :: Random a => State StdGen a
getRandom = state random
```

Compared to `rollDie`, this function does not specify the `Int` type in its signature and uses `random` instead of `randomR`; otherwise, it is just the same. `getRandom` can be used for any instance of `Random`:

```
GHCi> evalState getRandom (mkStdGen 0) :: Bool
True
GHCi> evalState getRandom (mkStdGen 0) :: Char
''
GHCi> evalState getRandom (mkStdGen 0) :: Double
0.9872770354820595
GHCi> evalState getRandom (mkStdGen 0) :: Integer
2092838931
```

Indeed, it becomes quite easy to conjure all these at once:

```
someTypes :: State StdGen (Int, Float, Char)
someTypes = liftM3 (,,) getRandom getRandom getRandom

allTypes :: State StdGen (Int, Float, Char, Integer, Double, Bool, Int)
allTypes = liftM (,,,,,,) getRandom
                     `ap` getRandom
                     `ap` getRandom
                     `ap` getRandom
                     `ap` getRandom
                     `ap` getRandom
                     `ap` getRandom
```

For `allTypes`, since there is no `liftM7` (the standard libraries only go to `liftM5`) we have used the `ap` function from `Control.Monad` instead. `ap` fits multiple computations into an application of a multiple argument function, which here is the (lifted) 7-element-tuple constructor. To understand `ap` further, look at its signature:

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

Remember then that the type variable `a` in Haskell can be replaced by a function type as well as a regular value one, and compare to:

```
GHCi>:t liftM (,,,,,,) getRandom
liftM (,,,,,) getRandom :: (Random a1) =>
                           State StdGen (b -> c -> d -> e -> f -> g
                               -> (a1, b, c, d, e, f, g))
```

The monad `m` obviously becomes `State StdGen`, while `ap`'s first argument is a function

```
b -> c -> d -> e -> f -> g -> (a1, b, c, d, e, f, g)
```

Applying `ap` over and over (in this case 6 times), we finally get to the point where `b` is an actual value (in our case, a 7-element tuple), not another function. To sum it up, `ap` applies a function-in-a-monad to a monadic value (compare with `liftM` / `fmap`, which applies a function *not* in a monad to a monadic value).

So much for understanding the implementation. Function `allTypes` provides pseudo-random values for all default instances of `Random`; an additional `Int` is inserted at the end to prove that the generator is not the same, as the two `Int`s will be different.

```
GHCi> evalState allTypes (mkStdGen 0)
GHCi>(2092838931,9.953678e-4,'',-868192881,0.4188001483955421,False,316817438)
```

Appendix I

The System.Random library

This library deals with the common task of pseudo-random number generation.[1]

The library makes it possible to generate repeatable results, by starting with a specified initial random number generator, or to get different results on each run by using the system-initialised generator or by supplying a seed from some other source.[2]

The library is split into two layers:

- A core *random number generator* provides a supply of bits. The class RandomGen provides a common interface to such generators. The library provides one instance of RandomGen, the abstract data type StdGen. Programmers may, of course, supply their own instances of RandomGen.
- The class Random provides a way to extract values of a particular type from a random number generator. For example, the Float instance of Random allows one to generate random values of type Float.

I.1 The *RandomGen* class

The class RandomGen provides a common interface to random number generators. The most common approach is using the StdGen type, presented in the next subsection.

```
class RandomGen g where

Minimal complete definition
  next, split

Methods
  next     :: g -> (Int , g)
  split    :: g -> (g , g)
  genRange :: g -> (Int , Int)
  genRange _ = (minBound , maxBound)

Instances
  RandomGen StdGen
```

[1]This implementation uses the Portable Combined Generator of L'Ecuyerfor 32-bit computers, transliterated by Lennart Augustsson. It has a period of roughly 2.30584e18.

[2]For example, the third decimal of the internal clock

The $\boxed{\text{next}}$ operation returns an $\boxed{\text{Int}}$ that is uniformly distributed in the range returned by $\boxed{\text{genRange}}$ (including both end points), and a new generator.

The $\boxed{\text{genRange}}$ operation yields the range of values returned by the generator. The default definition spans the full range of $\boxed{\text{Int}}$.

It is required that:

- If `(a,b) = genRange g`, then `a < b`.
- $\boxed{\text{genRange}}$ always returns a pair of defined $\boxed{\text{Int}}$s.

The second condition ensures that $\boxed{\text{genRange}}$ cannot examine its argument, and hence the value it returns can be determined only by the instance of $\boxed{\text{RandomGen}}$. That in turn allows an implementation to make a single call to $\boxed{\text{genRange}}$ to establish a generator's range, without being concerned that the generator returned by (say) $\boxed{\text{next}}$ might have a different range to the generator passed to $\boxed{\text{next}}$.

The $\boxed{\text{split}}$ operation allows one to obtain two distinct random number generators. This is very useful in functional programs (for example, when passing a random number generator down to recursive calls), but very little work has been done on statistically robust implementations of $\boxed{\text{split}}$.

I.2 The type *StdGen* and the global number generator

I.2.1 *StdGen*

```
data StdGen

Instances
  Read StdGen
  Show StdGen
  RandmGen StdGen

mkStdGen :: Int -> StdGen
```

The $\boxed{\text{StdGen}}$ instance of $\boxed{\text{RandomGen}}$ has a $\boxed{\text{genRange}}$ of at least 30 bits.

The result of repeatedly using $\boxed{\text{next}}$ should be statistically robust.

The $\boxed{\text{Show}}$ and $\boxed{\text{Read}}$ instances of $\boxed{\text{StdGen}}$ provide a primitive way to save the state of a random number generator. It is required that `read (show g) == g`.

In addition, $\boxed{\text{reads}}$ may be used to map an arbitrary string (not necessarily one produced by $\boxed{\text{show}}$) onto a value of type $\boxed{\text{StdGen}}$. In general, the $\boxed{\text{Read}}$ instance of $\boxed{\text{StdGen}}$ has the following properties:

- It guarantees to succeed on any string.
- It guarantees to consume only a finite portion of the string.
- Different argument strings are likely to result in different results.

The function $\boxed{\text{mkStdGen}}$ provides an alternative way of producing an initial generator, by mapping an $\boxed{\text{Int}}$ into a generator. Again, distinct arguments should be likely to produce distinct generators.

I.2.2 The global number generator

There is a single, implicit, global random number generator of type StdGen, held in some global variable maintained by the IO monad. It is initialised automatically in some system-dependent fashion, for example, by using the time of day, or Linux's kernel random number generator. To get deterministic behaviour, use setStdGen.

```
getStdRandom :: (StdGen -> (a, StdGen)) -> IO a
```

Uses the supplied function to get a value from the current global random generator, and updates the global generator with the new generator returned by the function.

```
getStdGen :: IO StdGen
```

Gets the global random number generator.

```
setStdGen :: StdGen -> IO ()
```

Sets the global random number generator.

```
newStdGen :: IO StdGen
```

Applies split to the current global random generator, updates it with one of the results, and returns the other.

I.3 Random vaues of other types: the *Random* class

With a source of random number supply in hand, the Random class allows the programmer to extract random values of a variety of types.

```
class Random a where

Minimal complete definition
  randomR, random

Methods
  randomR   :: RandomGen g => (a, a) -> g -> (a, g)
  random    :: RandomGen g => g -> (a, g)
  randomRs  :: RandomGen g => (a, a) -> g -> [a]
  randoms   :: RandomGen g => g -> [a]
  randomRIO :: (a, a) -> IO a
  randomIO  :: IO a

Instances
  Random Bool
  Random Char
  Random Double
  Random Float
  Random Int
  ...
```

randomR takes a range *(lo,hi)* and a random number generator g, and returns a random value uniformly distributed in the closed interval *[lo,hi]*, together with a new generator. It is unspecified what happens if $lo > hi$. For continuous types there is no requirement that the values lo and hi are ever produced, but they may be, depending on the implementation and the interval.

random is the same as randomR, but using a default range determined by the type:

- For bounded types (instances of `Bounded`, such as `Char`), the range is normally the whole type.
- For fractional types, the range is normally the semi-closed interval *[0,1)*.
- For `Integer`, the range is (arbitrarily) the range of `Int`.

`randomRs` is a plural variant of `randomR`, producing an infinite list of random values instead of returning a new generator.

`randoms` is a plural variant of `random`, producing an infinite list of random values instead of returning a new generator.

`randomRIO` is a variant of `randomR` that uses the global random number generator.

`randomIO` is a variant of `random` that uses the global random number generator.

I.4 Other functions (that are not exported)

The following code is found in System.Random[3] but not exported.

I.4.1 The global number generator coding

First, some note found early in the module

```
-- The standard nhc98 implementation of Time.ClockTime does not match
-- the extended one expected in this module, so we lash-up a quick
-- replacement here.
#ifdef __NHC__
foreign import ccall "time.h time" readtime :: Ptr CTime -> IO CTime
getTime :: IO (Integer, Integer)
getTime = do CTime t <- readtime nullPtr;  return (toInteger t, 0)
#else
getTime :: IO (Integer, Integer)
getTime = do
  utc <- getCurrentTime
  let daytime = toRational $ utctDayTime utc
  return $ quotRem (numerator daytime) (denominator daytime)
#endif
```

The function `getTime` is used in:

```
mkStdRNG :: Integer -> IO StdGen
mkStdRNG o = do
    ct          <- getCPUTime
    (sec, psec) <- getTime
    return (createStdGen (sec * 12345 + psec + ct + o))
```

Which finally gives us

```
theStdGen :: IORef StdGen
theStdGen  = unsafePerformIO $ do
   rng <- mkStdRNG 0
   newIORef rng
```

[3] http://hackage.haskell.org/package/random-1.1/docs/src/System-Random.html

Appendix J

Appendix: Summary of functions

Everything has been taken from the Haskell documentation

J.1 Functor context

```
class Functor f where
  The Functor class is used for types that can be mapped over.

  Instances of Functor should satisfy the following laws:
    fmap id  ==  id
    fmap (f . g)  ==  fmap f . fmap g

  Minimal complete definition
    fmap

  Methods
    fmap :: (a -> b) -> f a -> f b
    (<$) :: a -> f b -> f a                          infixl 4

  Predefined functions  (in Data.Functor)
    (<$>) :: Functor f => (a -> b) -> f a -> f b        infixl 4
    (<$>) = fmap
    ($>) :: Functor f => f a -> b -> f b                infixl 4
    ($>) = flip (<$)
    void :: Functor f => f a -> f ()
    void x = () <$ x
```

$\boxed{(<\$>)}$ is an infix synonym for $\boxed{\text{fmap}}$.

The method $\boxed{(<\$)}$ replaces all locations in the input with the same value. The default definition is $\boxed{\text{fmap . const}}$, but this may be overriden with a more efficient version.

$\boxed{(\$>)}$ is a flipped version of $\boxed{(<\$)}$.

$\boxed{\text{void}}$ value discards or ignores the result of evaluation, such as the return value of an $\boxed{\text{System.IO.IO}}$ action.

J.2 Applicative context

The Control.Applicative module describes a structure intermediate between a functor and a monad (technically, a strong lax monoidal functor). Compared with monads, this interface lacks the full power of the binding operation (>>=), but

1. it has more instances.
2. it is sufficient for many uses, e.g. context-free parsing, or the Traversable class.
3. instances can perform analysis of computations before they are executed, and thus produce shared optimizations.

```
class Functor f => Applicative f where
  A functor with application, providing operations to
  embed pure expressions (pure), and
  sequence computations and combine their results: (<*>).

  Instances of Functor should satisfy the following laws:
    pure id <*> v = v                                 -- identity
    pure (.) <*> u <*> v <*> w = u <*> (v <*> w)      -- composition
    pure f <*> pure x = pure (f x)                    -- homomorphism
    u <*> pure y = pure ($ y) <*> u                   -- interchange

    As a consequence of these laws, the Functor instance for f will satisfy
      fmap f x = pure f <*> x

    If f is also a Monad, it should satisfy
      pure = return
      (<*>) = ap
      (which implies that pure and <*> satisfy the applicative functor laws).

  Minimal complete definition
    pure, (<*>)

  Methods
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b                   infixl 4
    (*>) :: f a -> f b -> f b                           infixl
    u *> v = pure (const id) <*> u <*> v
    (<*) :: f a -> f b -> f a                           infixl 4
    u <* v = pure const <*> u <*> v

  Utility functions
    (<**>) :: Applicative f => f a -> f (a -> b) -> f b   infixl 4
    (<**>) = liftA2 (flip ($))
    liftA  :: Applicative f => (a -> b) -> f a -> f b
    liftA f a = pure f <*> a
    liftA2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c
    liftA2 f a b = fmap f a <*> b
    liftA3 :: Applicative f => (a -> b -> c -> d) -> f a -> f b -> f c -> f d
    liftA3 f a b c = fmap f a <*> b <*> c
    when   :: (Applicative f) => Bool -> f () -> f ()
    when p s = if p then s else pure ()
```

$\boxed{(*>)}$ and $\boxed{(<*)}$ are already defined, but may be overriden with equivalent specialized implementations.

$\boxed{(<**>)}$ is a variant of $\boxed{(<*>)}$ with the arguments reversed.

$\boxed{\text{liftA}}$ lifts a function to actions. This function may be used as a value for $\boxed{\text{fmap}}$ in a $\boxed{\text{Functor}}$ instance.

$\boxed{\text{liftA2}}$ lifts a binary function to actions.

$\boxed{\text{liftA3}}$ lifts a ternary function to actions.

$\boxed{\text{when}}$ is a conditional execution of $\boxed{\text{Applicative}}$ expressions.

J.3 Monad context

```
class Applicative m => Monad m where

  Instances of Monad should satisfy the following laws:
    return a >>= k  =  k a
    m >>= return  =  m
    m >>= (x -> k x >>= h)  =  (m >>= k) >>= h

  Furthermore, the Monad and Applicative operations should relate as follows:
    pure = return
    (<*>) = ap
  The above laws imply:
    fmap f xs  =  xs >>= return . f
    (>>) = (*>)
    pure and (<*>) satisfy the applicative functor laws.

  Minimal complete definition
    (>>=)

  Methods
    (>>=)    :: m a -> (a -> m b) -> m b                     infixl 1
    (>>)     :: m a -> m b -> m b                            infixl 1
    m >> k   =   m >>= \_ -> k
    return   :: a -> m a
    return   =   pure
    fail     :: String -> m a
    fail s   =   error s

  Utility functions
    join     :: (Monad m) => m (m a) -> m a
    join x            =  x >>= id
    (=<<)    :: Monad m => (a -> m b) -> m a -> m b
    f =<< x    =    x >>= f
    sequence :: Monad m => [m a] -> m [a]
    sequence = mapM id
    mapM     :: Monad m => (a -> m b) -> [a] -> m [b]
    mapM f as = foldr k (return []) as
    where
      k a r = do { x <- f a; xs <- r; return (x:xs) }
    liftM    :: (Monad m) => (a1 -> r) -> m a1 -> m r
    liftM f m1         = do { x1 <- m1; return (f x1) }
    liftM2   :: (Monad m) => (a1 -> a2 -> r) -> m a1 -> m a2 -> m r
    liftM2 f m1 m2     = do { x1 <- m1; x2 <- m2; return (f x1 x2) }
    liftM3   :: (Monad m) => (a1 -> a2 -> a3 -> r) -> m a1 -> m a2 -> m a3 -> m r
    liftM3 f m1 m2 m3  = do { x1 <- m1; x2 <- m2; x3 <- m3; return (f x1 x2 x3) }
    ap       :: (Monad m) => m (a -> b) -> m a -> m b
    ap m1 m2           = do { x1 <- m1; x2 <- m2; return (x1 x2) }
```

J.4 Alternative context

```
class Applicative f => Alternative f where
  A monoid on applicative functors.

  If defined, some and many should be the least solutions of the equations:
    some v = (:) <$> v <*> many v
    many v = some v <|> pure []

  Minimal complete definition
    empty, (<|>)
  Methods
    empty :: f a
      The identity of <|>
    (<|>) :: f a -> f a -> f a                          infixl 3
      An associative binary operation
    some :: f a -> f [a]
      One or more
    many :: f a -> f [a]
      Zero or more.

  Utility functions
    optional :: Alternative f => f a -> f (Maybe a)
      One or none.
```

J.5 Module System.Random

```
class RandomGen g where

Minimal complete definition
  next, split

Methods
  next     :: g -> (Int , g)
  split    :: g -> (g , g)
  genRange :: g -> (Int , Int)
  genRange _ = (minBound , maxBound)

Instances
  RandomGen StdGen


data StdGen

Instances
  Read StdGen
  Show StdGen
  RandmGen StdGen

mkStdGen :: Int -> StdGen


class Random a where

Minimal complete definition
  randomR, random

Methods
  randomR   :: RandomGen g => (a, a) -> g -> (a, g)
  random    :: RandomGen g => g -> (a, g)
  randomRs  :: RandomGen g => (a, a) -> g -> [a]
  randoms   :: RandomGen g => g -> [a]
  randomRIO :: (a, a) -> IO a
  randomIO  :: IO a

Instances
  Random Bool
  Random Char
  Random Double
  Random Float
  Random Int
  ...
```

And the global random number generator

```
getStdRandom :: (StdGen -> (a, StdGen)) -> IO a
```

Uses the supplied function to get a value from the current global random generator, and updates the global generator with the new generator returned by the function.

```
getStdGen :: IO StdGen
```

Gets the global random number generator.

```
setStdGen :: StdGen -> IO ()
```

Sets the global random number generator.

```
newStdGen :: IO StdGen
```

Applies $\boxed{\text{split}}$ to the current global random generator, updates it with one of the results, and returns the other.

J.6 Module Control.Monad

The Functor, Monad and MonadPlus classes, with some useful operations on monads. Some of the information is already exposed in the previous sections.

```
class Functor f where
  already seen

class Applicative m => Monad m where
  already seen

class (Alternative m, Monad m) => MonadPlus m where
  Monads that also support choice and failure.

  Instances of MonadPlus should satisfy the following laws:
    mzero `mplus` m = m
    m `mplus` mzero = m
    associativity of mplus
    mzero >>= f     = mzero
    m >> mzero      = mzero

  Minimal complete definition
    Nothing
  Methods
    mzero :: m a
    mplus :: m a -> m a -> m a

Basic Monad functions
  mapM      :: (Traversable t, Monad m) => (a -> m b) -> t a -> m (t b)
  mapM_     :: (Foldable t, Monad m)    => (a -> m b) -> t a -> m ()
  forM      :: (Traversable t, Monad m) => t a -> (a -> m b) -> m (t b)
  forM_     :: (Foldable t, Monad m)    => t a -> (a -> m b) -> m ()
  sequence  :: (Traversable t, Monad m) => t (m a) -> m (t a)
  sequence_ :: (Foldable t, Monad m)    => t (m a) -> m ()
  (=<<)     :: Monad m   => (a -> m b) -> m a -> m b                infixr 1
  (>=>)     :: Monad m   => (a -> m b) -> (b -> m c) -> a -> m c    infixr 1
  (<=<)     :: Monad m   => (b -> m c) -> (a -> m b) -> a -> m c    infixr 1
  forever   :: Monad m   => m a -> m b
  void      :: Functor f => f a -> f ()

Generalisations of list functions
  join      :: Monad m => m (m a) -> m a
  msum      :: (Foldable t, MonadPlus m) => t (m a) -> m a
  mfilter      :: MonadPlus m => (a -> Bool) -> m a -> m a
  filterM      :: Monad m => (a -> m Bool) -> [a] -> m [a]
  mapAndUnzipM :: Monad m => (a -> m (b, c)) -> [a] -> m ([b], [c])
  zipWithM     :: Monad m => (a -> b -> m c) -> [a] -> [b] -> m [c]
  zipWithM_    :: Monad m => (a -> b -> m c) -> [a] -> [b] -> m ()
  foldM     :: (Foldable t, Monad m) => (b -> a -> m b) -> b -> t a -> m b
  foldM_    :: (Foldable t, Monad m) => (b -> a -> m b) -> b -> t a -> m ()
  replicateM   :: Monad m => Int -> m a -> m [a]
  replicateM_  :: Monad m => Int -> m a -> m ()
```

```
Conditional execution of monadic expressions
  guard     :: Alternative f => Bool -> f ()
  when      :: Applicative f => Bool -> f () -> f ()
  unless    :: Applicative f => Bool -> f () -> f ()

Monadic lifting operators
  liftM  :: Monad m => (a1 -> r) -> m a1 -> m r
  liftM2 :: Monad m => (a1 -> a2 -> r) -> m a1 -> m a2 -> m r
  liftM3 :: Monad m => (a1 -> a2 -> a3 -> r) -> m a1 -> m a2 -> m a3 -> m r
  liftM4 :: Monad m => (a1 -> a2 -> a3 -> a4 -> r) -> m a1 -> m a2 -> m a3 -> m a4 -> m r
  liftM5 :: Monad m => (a1 -> a2 -> a3 -> a4 -> a5 -> r) -> m a1 -> m a2 -> m a3 -> m a4 -> m a5 -> m
  ap :: Monad m => m (a -> b) -> m a -> m b

Strict monadic functions
  (<$!>)    :: Monad m => (a -> b) -> m a -> m b                    infixl 4
```

Naming conventions The functions in this library use the following naming conventions:

- A postfix 'M' always stands for a function in the Kleisli category: The monad type constructor $\boxed{\mathrm{m}}$ is added to function results (modulo currying) and nowhere else. So, for example,

```
filter  ::              (a ->   Bool) -> [a] ->   [a]
filterM :: (Monad m) => (a -> m Bool) -> [a] -> m [a]
```

- A postfix '_' changes the result type from $\boxed{\mathrm{m\ a}}$ to $\boxed{\mathrm{m\ ()}}$. Thus, for example:

```
sequence  :: Monad m => [m a] -> m [a]
sequence_ :: Monad m => [m a] -> m ()
```

- A prefix 'm' generalizes an existing function to a monadic form. Thus, for example:

```
sum  :: Num a       => [a]   -> a
msum :: MonadPlus m => [m a] -> m a
```

Appendix K

Exercises

These exercises have been taken from several different sources, and are not necessarily sorted by any criteria

K.1 Basic *Functor* and *Applicative* exercises

1. Define instances of $\boxed{\text{Functor}}$ for the following types:

   - A rose tree, defined as: $\boxed{\text{data Tree a} = \text{Node a [Tree a]}}$
   - $\boxed{\text{Either e}}$ for a fixed $\boxed{\text{e}}$.
   - The function type $\boxed{((\to)\ r)}$. In this case, $\boxed{\text{f a}}$ will be $\boxed{(r \to a)}$

2. Check that the Applicative laws hold for the instance for $\boxed{\text{Maybe}}$ presented in the main body:

```
instance Applicative Maybe where
    pure                 = Just
    (Just f) <*> (Just x) = Just (f x)
    _        <*> _        = Nothing
```

3. Write $\boxed{\text{Applicative}}$ instances for

   - $\boxed{\text{Either e}}$, for a fixed $\boxed{\text{e}}$
   - $\boxed{((\to)\ r)}$, for a fixed $\boxed{\text{r}}$

K.2 Advanced *Monad* and *Applicative* exercises

1. What is the expected behavior of $\boxed{\text{sequence}}$ for the $\boxed{\text{Maybe}}$ monad?

2. Write a definition of $\boxed{(<*>)}$ using $\boxed{(>>=)}$ and $\boxed{\text{fmap}}$. Do not use do-notation.

3. Implement

```
liftA5 :: Applicative f => (a -> b -> c -> d -> e -> k)
-> f a -> f b -> f c -> f d -> f e -> f k
```

4. For the list functor, implement from scratch (that is, without using anything from $\boxed{\text{Applicative}}$ or $\boxed{\text{Monad}}$ directly) both $\boxed{(<*>)}$ and its version with the "wrong" sequencing of effects,

```
(<|*|>) :: Applicative f => f (a -> b) -> f a -> f b
```

5. Rewrite the definition of commutativity for a $\boxed{\text{Monad}}$;

```
liftA2 f u v = liftA2 (flip f) v u -- Commutativity
   -- Or, equivalently,
f <$> u <*> v = flip f <$> v <*> u
```

   using do-notation instead of $\boxed{\text{ap}}$ or $\boxed{\text{liftM2}}$.

6. Are the following applicative functors commutative?

   - $\boxed{\text{ZipList}}$
   - $\boxed{((\rightarrow)\ r)}$
   - $\boxed{\text{State s}}$ (Use the $\boxed{\text{newtype}}$ definition from the $\boxed{\text{State}}$ appendix).

   Hint: You may find the answer to exercise 5 (in this section) useful.

7. What is the result of `[2,7,8] *> [3,9]`? Try to guess without writing.)

8. Implement $\boxed{(<**>)}$ in terms of other $\boxed{\text{Applicative}}$ functions.

9. As we have just seen, some functors allow two legal implementations of $\boxed{(<*>)}$ which are only different in the sequencing of effects. Why there is not an analogous issue involving $\boxed{(>>=)}$?

The next few exercises concern the following tree data structure:[1]

```
data AT a = L a | B (AT a) (AT a)
```

10. Write $\boxed{\text{Functor}}$, $\boxed{\text{Applicative}}$ and $\boxed{\text{Monad}}$ instances for $\boxed{\text{AT}}$. Do not use shortcuts such as $\boxed{\text{pure = return}}$. The $\boxed{\text{Applicative}}$ and $\boxed{\text{Monad}}$ instances should match; in particular, $\boxed{(>>=)}$ should be equivalent to $\boxed{\text{ap}}$, which follows from the $\boxed{\text{Monad}}$ instance.

[1] In case you are wondering, "AT" stands for "apple tree".

11. Implement the following functions, using either the $\boxed{\text{Applicative}}$ instance, the $\boxed{\text{Monad}}$ one or neither of them, if neither is enough to provide a solution. Between Applicative and Monad, choose the *least* powerful one which is still good enough for the task. Justify your choice for each case in a few words.

   - `fructify :: AT a -> AT a`, which grows the tree by replacing each leaf $\boxed{\text{L}}$ with a branch $\boxed{\text{B}}$ containing two copies of the leaf.
   - `prune :: a -> (a -> Bool) -> AT a -> AT a`, with $\boxed{\text{prune z p t}}$ replacing a branch of $\boxed{\text{t}}$ with a leaf carrying the default value $\boxed{\text{z}}$ whenever any of the leaves directly on it satisfies the test $\boxed{\text{p}}$.
   - `reproduce :: (a -> b) -> (a -> b) -> AT a -> AT b`, with $\boxed{\text{reproduce f g t}}$ resulting in a new tree with two modified copies of $\boxed{\text{t}}$ on the root branch. The left copy is obtained by applying $\boxed{\text{f}}$ to the values in $\boxed{\text{t}}$, and the same goes for $\boxed{\text{g}}$ and the right copy.

12. There is another legal instance of $\boxed{\text{Applicative}}$ for $\boxed{\text{AT}}$ (the reversed sequencing version of the original one doesn't count). Write it.

    Hint: this other instance can be used to implement

    ```
    sagittalMap :: (a -> b) -> (a -> b) -> AT a -> AT b
    ```

    which, when given a branch, maps one function over the left child tree and the other over the right child tree.

13. Write implementations for $\boxed{\text{unit}}$ and $\boxed{(*\&*)}$ in terms of $\boxed{\text{pure}}$ and $\boxed{(<*>)}$, and vice-versa.

14. Formulate the law of commutative applicative functors,

    ```
    liftA2 f u v = liftA2 (flip f) v u -- Commutativity
       -- Or, equivalently,
    f <$> u <*> v = flip f <$> v <*> u
    ```

    in terms of the $\boxed{\text{Monoidal}}$ methods.

15. Write from scratch $\boxed{\text{Monoidal}}$ instances for:

    - $\boxed{\text{ZipList}}$
    - $\boxed{((\to)\ r)}$

K.3 *State* exercises

1. Implement a function `rollNDiceIO :: Int -> IO [Int]` that, given an integer (a number of die rolls), returns a list of that number of pseudo-random integers between 1 and 6.

2. Implement a function `rollDice :: StdGen -> ((Int, Int), StdGen)` that, given a generator, returns a tuple with our random numbers as first element and the last generator as the second.

3. Similarly to what was done for rollNDiceIO, implement a function

   ```
   rollNDice :: Int -> State StdGen [Int]
   ```

   that, given an integer, returns a list with that number of pseudo-random integers between 1 and 6.

4. Write an instance of Functor for State s. Your final answer should not use anything that mentions Monad in its type (that is, return, (>>=), etc.). Then, explain in a few words what the fmap you wrote does.

   (Hint: If you get stuck, have another look at the comments about liftM in the main body.)

5. Besides put and get, there are also

   ```
   modify :: (s -> s) -> State s ()
   ```

   which modifies the current state using a function, and

   ```
   gets :: (s -> a) -> State s a
   ```

   which produces a modified copy of the state while leaving the state itself unchanged. Write implementations for them.

6. If you are not convinced that State is worth using, try to implement a function equivalent to evalState allTypes without making use of monads, i.e. with an approach similar to clumsyRollDice above.

K.4 *MonadPlus* exercises

1. Prove the MonadPlus laws for Maybe and the list monad.

2. We could augment our above parser to involve a parser for any character:

```
-- | Consume a given character in the input, and return the character we
--   just consumed, paired with rest of the string. We use a do-block  so that
--   if the pattern match fails at any point, fail of the Maybe monad (i.e.
--   Nothing) is returned.
char :: Char -> String -> Maybe (Char, String)
char c s = do
  let (c':s') = s
  if c == c' then Just (c, s') else Nothing
```

   It would then be possible to write a $\boxed{\text{hexChar}}$ function which parses any valid hexadecimal character (0-9 or a-f). Try writing this function (hint: `map digit [0..9] :: [String -> Maybe Int]`).

K.5 Monad transformers exercises'

1. Why is it that the $\boxed{\text{lift}}$ function has to be defined separately for each monad, where as $\boxed{\text{liftM}}$ can be defined in a universal way?

2. $\boxed{\text{Identity}}$ is a trivial functor, defined in $\boxed{\text{Data.Functor.Identity}}$ as:

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

   It has the following Monad instance:

```
instance Monad Identity where
    return a = Identity a
    m >>= k  = k (runIdentity m)
```

   Implement a monad transformer $\boxed{\text{IdentityT}}$, analogous to $\boxed{\text{Identity}}$ but wrapping values of type $\boxed{\text{m a}}$ rather than $\boxed{\text{a}}$. Write at least its $\boxed{\text{Monad}}$ and $\boxed{\text{MonadTrans}}$ instances.

3. Implement `state :: MonadState s m => (s -> (a, s)) -> m a` in terms of $\boxed{\text{get}}$ and $\boxed{\text{put}}$.

4. Are $\boxed{\text{MaybeT (State s)}}$ and $\boxed{\text{StateT s Maybe}}$ equivalent? (Hint: one approach is comparing what the $\boxed{\text{run...T}}$ unwrappers produce in each case.)

K.6 Hask category exercises

1. As was mentioned, any partial order ($P, \leq$ is a category with objects the elements of P and a morphism between elements a and b iff $a \leq b$. Which of the above laws guarantees the transitivity of $\leq$?

2. Check the functor laws for the Maybe and list functors.

3. Verify that the list and $\boxed{\text{Maybe}}$ monads do in fact obey the first monad law,

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

   with some examples to see precisely how the layer flattening works.

4. Prove the second monad law, $\boxed{\text{join . fmap return} = \text{join . return} = \text{id}}$ for the $\boxed{\text{Maybe}}$ monad.

5. Convince yourself that the 3rd and 4th laws should hold true for any monad by exploring what they mean, in a similar style to how the first and second laws were explored.

6. In fact, the two versions of the laws we gave:

   ```
   -- Categorical:
   join . fmap join = join . join
   join . fmap return = join . return = id
   return . f = fmap f . return
   join . fmap (fmap f) = fmap f . join

   -- Functional:
   m >>= return = m
   return m >>= f = f m
   (m >>= f) >>= g = m >>= (\x -> f x >>= g)
   ```

   are entirely equivalent. We showed that we can recover the functional laws from the categorical ones. Go the other way; show that starting from the functional laws, the categorical laws hold. It may be useful to remember the following definitions:

   ```
   join m = m >>= id
   fmap f m = m >>= return . f
   ```

Appendix L

My solutions for the exercises

Solutions will be given in packs, one for each section

L.1 Basic *Functor* and *Applicative* solutions

1. Define instances of $\boxed{\text{Functor}}$ for the following types:

   - A rose tree, defined as: $\boxed{\text{data Tree a} = \text{Node a [Tree a]}}$
   - $\boxed{\text{Either e}}$ for a fixed $\boxed{\text{e}}$.
   - The function type $\boxed{((\rightarrow)\ r)}$. In this case, $\boxed{\text{f a}}$ will be $\boxed{(r \rightarrow a)}$

2. Check that the Applicative laws hold for the instance for $\boxed{\text{Maybe}}$ presented in the main body:

```
instance Applicative Maybe where
    pure                    = Just
    (Just f) <*> (Just x) = Just (f x)
    _        <*> _        = Nothing
```

3. Write $\boxed{\text{Applicative}}$ instances for

   - $\boxed{\text{Either e}}$, for a fixed $\boxed{\text{e}}$
   - $\boxed{((\rightarrow)\ r)}$, for a fixed $\boxed{\text{r}}$

```
{-# LANGUAGE TypeSynonymInstances #-}

import Control.Applicative



------------------------------------------------------------------------
--      FIRST EXERCISE OF BASIC FUNCTOR AND APPLICATIVE SECTION       --
------------------------------------------------------------------------

data Tree a = Node a [Tree a]
 deriving (Eq, Show)


instance Functor Tree where
  fmap f (Node n ts)  = Node (f n) ( map (fmap f) ts )





mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f (Node n [])     = Node (f n) []
mapTree f (Node n (t:ts)) = Node (f n) list
  where
    list  = (mapTree f t) : mapp (mapTree f) ts
    mapp _ [] = []
    mapp g (x:xs) = (g x):(mapp g xs)

-- NO HAY FORMA DE ESCRIBIR LOS DOS BUCLES RECURSIVOS EN UNA SOLA LINEA?

--  fmap f (Node n [])     = Node (f n) []
--  fmap f (Node n (t:ts)) = Node (f n) ????


n1 = Node 1 []
n2 = Node 2 []
n3 = Node 3 []
n4 = Node 4 []
n5 = Node 5 []
n6 = Node 6 []
n7 = Node 7 []

t1 = Node 10 [n1,n2,n3]
t2 = Node 11 [n4,n5]
t3 = Node 12 [n6]
t4 = Node 13 [n7]

t10 = Node 20 [t1,t2,t3,t4]


ej1 :: IO (Tree Int)
```

```
ej1 = do
  let t10' = fmap (100+) t10
  putStrLn (show t10)
  putStrLn (show t10')
  return t10'

-- Checking the correctness of the solution:
-- https://hackage.haskell.org/package/containers-0.5.7.1/docs/src/Data.Tree.html#line-74


------------------------------------------------------------------------


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

instance Functor (Either' a) where
  fmap _ (Left' x)  = Left' x
  fmap f (Right' y) = Right' (f y)


-- Checking the laws:

-- fmap id (Left x) = Left x == Left x                      OK
-- fmap id (Right y) = Right (id y) = Right y == Right y     OK

-- fmap (f.g) (Left x) = Left x  == Left x = fmap f (Left x) = fmap f (fmap g (Left x))  OK
-- fmap (f.g) (Right y) = Right ((f.g) y)
--   ==          --------------------------------> because   (f.g) y  ==  f (g y)        OK
--  Right (f (g y)) = fmap f (Right (g y)) = fmap f (fmap g (Right y))


------------------------------------------------------------------------

type FuncsWithFixedDomain r = (->) r

-- *Main> :k FuncsWithFixedDomain
-- FuncsWithFixedDomain :: * -> * -> *
-- *Main> :k FuncsWithFixedDomain Int
-- FuncsWithFixedDomain Int :: * -> *


-- instance Functor (FuncsWithFixedDomain r) where  ==>  ERROR

-- [1 of 1] Compiling Main             ( BasicFunctorAndApp.hs, interpreted )

-- BasicFunctorAndApp.hs:88:10:
--     Duplicate instance declarations:
--       instance Functor (FuncsWithFixedDomain r)
--         -- Defined at BasicFunctorAndApp.hs:88:10
--       instance Functor ((->) r) -- Defined in GHC.Base
-- Failed, modules loaded: none.
```

```
-- http://hackage.haskell.org/package/base-4.8.2.0/docs/src/GHC.Base.html#line-612


mapF :: (a -> b) -> (r -> a) -> (r -> b)
mapF alpha f =  alpha . f


-- *Main> (mapF (2+) length ) [1,2,3]
-- 5
-- (0.00 secs, 0 bytes)


-- Checking the laws:

-- mapF id f = id . f  ==  f                                                           OK

-- mapF (alpha.beta) f = (alpha.beta).f
--      ==          -------------------> because   (alpha.beta).f  == alpha.(beta.f)  OK
--  alpha.(beta.f) = mapF alpha (beta.f) = mapF alpha (mapF beta f)





------------------------------------------------------------------------
--     SECOND EXERCISE OF BASIC FUNCTOR AND APPLICATIVE SECTION       --
------------------------------------------------------------------------

data Maybe' a = Just' a | Nothing'

instance Functor Maybe' where
  fmap f Nothing' = Nothing'
  fmap f (Just' x) = Just' (f x)


instance Applicative Maybe' where
  pure = Just'
  (Just' f) <*> (Just' x) = Just' (f x)
  _        <*> _          = Nothing'



-- Checking the laws:

-- Identity:
--   (pure id) <*> Nothing   ==  Nothing
--   (pure id) <*> (Just x) = (Just id) <*> (Just x) = Just (id x)  ==  Just x
-- Homomorphism
--   (pure f) <*> (pure x) = (Just f) <*> (Just x) == Just (f x) == pure (f x)
-- Interchange
--   Nothing <*> (pure x) = Nothing <*> (Just x) == Nothing ==  (Just ($ y)) <*> Nothing
--   (Just f) <*> (pure x) = Just (f x)  ==  Just (f $ x) = (Just ($ x)) <*> (Just f)
```

```
-- Composition
--   pure (.) <*> Just u <*> Just v <*> Just a = Just (u.v) <*> Just a  ==  Just ((u.v) a)
--     ==
--    Just u <*> Just (v a) = Just u <*> (Just v <*> Just a)




------------------------------------------------------------------------
--      THIRD EXERCISE OF BASIC FUNCTOR AND APPLICATIVE SECTION       --
------------------------------------------------------------------------


instance Applicative (Either' e) where
  pure x = Right' x
  (Right' f) <*> (Right' x) = Right' (f x)
  _          <*> (Left' x)  = Left' x


-- Checking the laws:
--   analogous to the Maybe instance


------------------------------------------------------------------------



-- instance Applicative (FuncsWithFixedDomain r) where
--   pure x = const x
--   (alpha <*> f) rVal = alpha r (f rVal)


-- http://hackage.haskell.org/package/base-4.8.2.0/docs/src/GHC.Base.html#line-616
```

L.2 Advanced *Monad* and *Applicative* solutions

1. What is the expected behavior of `sequence` for the `Maybe` monad?

2. Write a definition of `(<*>)` using `(>>=)` and `fmap`. Do not use do-notation.

3. Implement

```
liftA5 :: Applicative f => (a -> b -> c -> d -> e -> k)
-> f a -> f b -> f c -> f d -> f e -> f k
```

4. For the list functor, implement from scratch (that is, without using anything from `Applicative` or `Monad` directly) both `(<*>)` and its version with the "wrong" sequencing of effects,

```
(<|*|>) :: Applicative f => f (a -> b) -> f a -> f b
```

5. Rewrite the definition of commutativity for a `Monad`;

```
liftA2 f u v = liftA2 (flip f) v u -- Commutativity
   -- Or, equivalently,
f <$> u <*> v = flip f <$> v <*> u
```

   using do-notation instead of `ap` or `liftM2`.

6. Are the following applicative functors commutative?

   - `ZipList`
   - `((->) r)`
   - `State s` (Use the `newtype` definition from the `State` appendix).

   Hint: You may find the answer to exercise 5 (in this section) useful.

7. What is the result of `[2,7,8] *> [3,9]`? Try to guess without writing.)

8. Implement `(<**>)` in terms of other `Applicative` functions.

9. As we have just seen, some functors allow two legal implementations of `(<*>)` which are only different in the sequencing of effects. Why there is not an analogous issue involving `(>>=)`?

The next few exercises concern the following tree data structure:[1]

```
data AT a = L a | B (AT a) (AT a)
```

10. Write `Functor`, `Applicative` and `Monad` instances for `AT`. Do not use shortcuts such as `pure = return`. The `Applicative` and `Monad` instances should match; in particular, `(>>=)` should be equivalent to `ap`, which follows from the `Monad` instance.

[1] In case you are wondering, "AT" stands for "apple tree".

11. Implement the following functions, using either the $\boxed{\text{Applicative}}$ instance, the $\boxed{\text{Monad}}$ one or neither of them, if neither is enough to provide a solution. Between Applicative and Monad, choose the *least* powerful one which is still good enough for the task. Justify your choice for each case in a few words.

   - `fructify :: AT a -> AT a`, which grows the tree by replacing each leaf $\boxed{\text{L}}$ with a branch $\boxed{\text{B}}$ containing two copies of the leaf.
   - `prune :: a -> (a -> Bool) -> AT a -> AT a`, with $\boxed{\text{prune z p t}}$ replacing a branch of $\boxed{\text{t}}$ with a leaf carrying the default value $\boxed{\text{z}}$ whenever any of the leaves directly on it satisfies the test $\boxed{\text{p}}$.
   - `reproduce :: (a -> b) -> (a -> b) -> AT a -> AT b`, with $\boxed{\text{reproduce f g t}}$ resulting in a new tree with two modified copies of $\boxed{\text{t}}$ on the root branch. The left copy is obtained by applying $\boxed{\text{f}}$ to the values in $\boxed{\text{t}}$, and the same goes for $\boxed{\text{g}}$ and the right copy.

12. There is another legal instance of $\boxed{\text{Applicative}}$ for $\boxed{\text{AT}}$ (the reversed sequencing version of the original one doesn't count). Write it.

    Hint: this other instance can be used to implement

    ```
    sagittalMap :: (a -> b) -> (a -> b) -> AT a -> AT b
    ```

    which, when given a branch, maps one function over the left child tree and the other over the right child tree.

13. Write implementations for $\boxed{\text{unit}}$ and $\boxed{(*\&*)}$ in terms of $\boxed{\text{pure}}$ and $\boxed{(<*>)}$, and vice-versa.

14. Formulate the law of commutative applicative functors,

    ```
    liftA2 f u v = liftA2 (flip f) v u -- Commutativity
       -- Or, equivalently,
    f <$> u <*> v = flip f <$> v <*> u
    ```

    in terms of the $\boxed{\text{Monoidal}}$ methods.

15. Write from scratch $\boxed{\text{Monoidal}}$ instances for:

    - $\boxed{\text{ZipList}}$
    - $\boxed{((\to)\ r)}$

```
import Control.Applicative
import Control.Monad


------------------------------------------------------------------------
--   FIRST EXERCISE OF THE ADVANCED MONAD AND APPLICATIVE SECTION     --
------------------------------------------------------------------------


-- sequence :: Monad m => [m a] -> m [a]

-- Looking at the type signature, the expected behavior could be:
-- sequence [Just 5, Just 6, Just 7, Nothing] == Just [5,6,7]
-- sequence [Nothing, Nothing, Nothing] == Nothing

-- However, taking a closer look,
-- sequence :: Monad m => [m a] -> m [a]
-- sequence = mapM id
--
-- mapM :: Monad m => (a -> m b) -> [a] -> m [b]
-- mapM f as = foldr k (return []) as
--   where
--     k a r = do { x <- f a; xs <- r; return (x:xs) }

-- The base case of the foldr call inside mapM is (return []), so it should be:
-- sequence [Nothing, Nothing, Nothing] == Just []


-- Finally, the real behavior is:
-- *Main> sequence [Nothing, Nothing, Nothing]
-- Nothing
-- (0.00 secs, 0 bytes)
-- *Main> sequence [Nothing, Nothing, Nothing, Just 5, Just 7]
-- Nothing
-- (0.00 secs, 0 bytes)
-- *Main> sequence [Just 5, Just 7]
-- Just [5,7]
-- (0.00 secs, 0 bytes)

-- To understand this:
--   in (MapM id) ,  we have  id :: m b -> m b
--   essentially, as soon as we get a Nothing in the do block, we have  Nothing >>= ...
--   which always ends up being Nothing




------------------------------------------------------------------------
--   SECOND EXERCISE OF THE ADVANCED MONAD AND APPLICATIVE SECTION    --
------------------------------------------------------------------------
```

```
-- (<*>) :: Applicative f => f (a -> b) -> f a -> f b
-- Recalling:
-- (>>=) :: Monad f => f a -> (a -> f b) -> f b
-- fmap  :: Functor f => (a -> b) -> f a -> f b


myApply :: (Functor m, Monad m) => m (a -> b) -> m a -> m b
myApply phi m =   phi >>= (\f -> fmap f m)


-- *Main> myApply (Just (2+)) (Just 3)
-- Just 5

-- *Main> myApply [(1+), (2*), id] [10,20,30]
-- [11,21,31,20,40,60,10,20,30]




------------------------------------------------------------------------
--   THIRD EXERCISE OF THE ADVANCED MONAD AND APPLICATIVE SECTION     --
------------------------------------------------------------------------


liftA5 :: Applicative f => (a -> b -> c -> d -> e -> k)
  -> f a -> f b -> f c -> f d -> f e -> f k

liftA5 func a b c d e = fmap func a <*> b <*> c <*> d <*> e
--                      (          )     )     )     )     )




------------------------------------------------------------------------
--   FOURTH EXERCISE OF THE ADVANCED MONAD AND APPLICATIVE SECTION    --
------------------------------------------------------------------------


myListApply :: [ a -> b ] -> [a] -> [b]
myListApply fs as  =  concatMap (\f -> map f as) fs


-- *Main> myListApply [(1+), (2*), id] [10,20,30]
-- [11,21,31,20,40,60,10,20,30]


fs <|*|> xs = concatMap (\x -> fmap ($ x) fs) xs
```

```
------------------------------------------------------------------------
--   FIFTH EXERCISE OF THE ADVANCED MONAD AND APPLICATIVE SECTION     --
------------------------------------------------------------------------


-- Commutativity:
--   liftA2 f u v  ==  liftA2 (flip f) v u
--  or equivalently
--   f <$> u <*> v  ==  flip f <$> v <*> u

-- Or equivalently:
--   do {x <- u; y <- v; return (f x y)}
--     ==
--   do {y <- v; x <- u; return ((flip f) y x)}
--     ==
--   do {y <- v; x <- u; return (f x y)}


-- *Main> let aux f u v = do {x <- u; y <- v; return (f x y)}
-- (0.02 secs, 0 bytes)
-- *Main> :t aux
-- aux :: Monad m => (t -> t1 -> b) -> m t -> m t1 -> m b
-- *Main> :t liftA2
-- liftA2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c
-- *Main> :t liftM2
-- liftM2 :: Monad m => (a1 -> a2 -> r) -> m a1 -> m a2 -> m r




------------------------------------------------------------------------
--   SIXTH EXERCISE OF THE ADVANCED MONAD AND APPLICATIVE SECTION     --
------------------------------------------------------------------------


-- Is  ZipList  (found in Control.Applicative) a commutative applicative functor?
--   newtype ZipList a = ZipList { getZipList :: [a] }
--   instance Applicative ZipList where
--     (ZipList fs) <*> (ZipList xs) = ZipList (zipWith ($) fs xs)
--     pure x                        = ZipList (repeat x)

-- (f <$> (ZipList l1)) <*> (ZipList l2)
--   ==  (ZipList (map f l1)) <*> ZipList l2
--   ==  ZipList (zipWith ($) (map f l1) l2)
--   ==  ZipList (zipWith ($) (map (flip f) l2) l1)
--   ==  (ZipList (map (flip f) l2)) <*> ZipList l1
--   == ( (flip f) <$> (ZipList l2) ) <*> (ZipList l1)


-- Is  ((->) r)  a commutative applicative functor?
-- instance Applicative ((->) a) where
--     pure = const
```

```
--     (<*>) f g x = f x (g x)

-- We have
--   f :: a -> b -> c
--   g :: r -> a
--   h :: r -> b
-- So
-- (f <$> g <*> h) x
--   ==  ((f.g) <*> h) x
--   ==  (f.g) x (h x)
--   ==  ((flip f).h) x (g x)
--   ==  ( ((flip f).h) <*>  g ) x
--   ==  ( ((flip f) <$> h) <*> g ) x


-- Is  State s  a commutative applicative functor?
--   No, because the order of computations affects the result

-- https://en.wikibooks.org/wiki/Haskell/Solutions/Applicative_functors



------------------------------------------------------------------------
--  SEVENTH EXERCISE OF THE ADVANCED MONAD AND APPLICATIVE SECTION    --
------------------------------------------------------------------------


-- [2,7,8] *> [3,9] ?

-- *Main> :t (*>)
-- (*>) :: Applicative f => f a -> f b -> f b
-- *Main> :t (<*>)
-- (<*>) :: Applicative f => f (a -> b) -> f a -> f b
-- *Main> :t ($>)

-- <interactive>:1:1:
--     Not in scope: $>
--     Perhaps you meant one of these:
--       >> (imported from Control.Monad), $! (imported from Prelude),
--       > (imported from Prelude)
-- *Main> :t (>>)
-- (>>) :: Monad m => m a -> m b -> m b

-- So it could be:
--   [2,7,8] *> [3,9]  ==  [3,9]
-- However:
--   *Main> [2,7,8] *> [3,9]
--   [3,9,3,9,3,9]
--   (0.02 secs, 0 bytes)
--   *Main>  [2,7,8] >> [3,9]
--   [3,9,3,9,3,9]
--   (0.00 secs, 0 bytes)
```

```
-- Because:
--   (*>) u v = pure (const id) <*> u <*> v
-- *Main> :t const
-- const :: a -> b -> a
-- *Main> :t id
-- id :: a -> a
-- *Main> :t (const id)
-- (const id) :: b -> a -> a
-- So:
--   [2,7,8] *> [3,9]
--   ==  [ (const id) ] <*> [2,7,8] <*> [3,9]
--   ==  [const id 2 , const id 7 , const id 8 ] <*> [3,9]
--   ==  [const id 2 3 , const id 2 9 , ... , const id 8 9]
--   ==  [3,9,3,9,3,9]

-- *Main> [2,7,8] <* [3,9]
-- [2,2,7,7,8,8]
-- (0.00 secs, 0 bytes)
-- *Main> [3,9] *> [2,7,8]
-- [2,7,8,2,7,8]
-- (0.00 secs, 0 bytes)

-- In conclusion, for lists:
--   l1 (*>) l2  repeats l2 as many times as length l1
--   l1 (<*) l2  repeats each element of l1 as many times as length l2




-------------------------------------------------------------------------
--   EIGHTH EXERCISE OF THE ADVANCED MONAD AND APPLICATIVE SECTION    --
-------------------------------------------------------------------------


-- (<**>) :: Applicative f => f a -> f (a -> b) -> f b
--   Recalling:  (<**>)  is NOT  flip (<*>)


myInvertedApply :: Applicative f => f a -> f (a -> b) -> f b
myInvertedApply = liftA2 (flip ($))


-- (searched hoogle because i was lazy)

-- Recalling:
-- Commutativity:
--   liftA2 f u v  ==  liftA2 (flip f) v u





-------------------------------------------------------------------------
```

```
--    NINTH EXERCISE OF THE ADVANCED MONAD AND APPLICATIVE SECTION    --
------------------------------------------------------------------------


-- from the Haskell documentation:
--   (=<<) :: Monad m => (a -> m b) -> m a -> m b
--   f =<< x         = x >>= f

-- *Main> :t (*>)
-- (*>) :: Applicative f => f a -> f b -> f b
-- *Main> :t (<*)
-- (<*) :: Applicative f => f a -> f b -> f a
-- *Main> :t (<*>)
-- (<*>) :: Applicative f => f (a -> b) -> f a -> f b
-- *Main> :t (<**>)
-- (<**>) :: Applicative f => f a -> f (a -> b) -> f b

-- It does not happen because the order of computations is fixed:
--   First, the monadic action x
--   Second, retrieve the hidden value in x and apply f
-- It doesn't make sense to think that we can evaluate any monadic action in f
-- first, because we need a value  a  from  m a  before




------------------------------------------------------------------------
--    TENTH EXERCISE OF THE ADVANCED MONAD AND APPLICATIVE SECTION    --
------------------------------------------------------------------------


data AT a = L a | B (AT a) (AT a)
  deriving (Show)

at1 :: AT Int
at1 = B (L 5) (B (L 2) (L 100))

at2 :: AT (Int -> Char)
at2 = let f=toEnum.(100+) in B (B (L f) (L f)) (B (L f) (B (L f) (L f)))

at3 :: AT (Int -> Int)
at3 = B (L (2*)) ( B (L (1+)) (L (3*)) )


-- Working with finite trees


mapAT :: (a -> b) -> AT a -> AT b
mapAT f (L a) = L (f a)
mapAT f (B l r) = B (mapAT f l) (mapAT f r)
```

```
instance Functor AT where
  fmap = mapAT

-- fmap id (L x) = L (id x)  ==  L x
-- fmap id (B l r) = B (fmap id l) (fmap id r)  ==  B l r
--   should be proved by induction (over the depth of the tree?)

-- fmap (f.g) (L x) = L ( f (g x) )
--   ==
-- fmap f (L (g x)) = (fmap f).(fmap g) (L x)



-- For each leaf in the tree of functions, substitute it with the tree
-- resulting from applying that function to the tree of values:
applyAT :: AT (a -> b) -> AT a -> AT b
applyAT at_fs at_xs = case at_fs of
  (L f)   -> mapAT f at_xs
  (B l r) -> B (applyAT l at_xs) (applyAT r at_xs)


instance Applicative AT where
  pure x = L x
  fs <*> xs = applyAT fs xs

-- pure id <*> at
--   = (L id) <*> at
--   = mapAT id at
--   = at           (checked in the Functor part)
--   == at          (as we wanted)

-- (pure f) <*> (pure x)
--   = (L f) <*> (L x)
--   = mapAT f (L x)
--   = L (f x)
--   == pure (f x)  (as we wanted)

-- pure (.) <*> gs <*> fs <*> xs
--   ==   ????
-- gs <*> (fs <*> xs)
--
-- pure (.) <*> gs <*> fs <*> xs
--   = (L (.)) <*> gs <*> fs <*> xs
--   = (mapAT (.) gs) <*> fs <*> xs
--   = case gs of
--     (L g) =>
--       = L (g.) <*> fs <*> xs
--         = case fs of
--           (L f) =>
--             = L (g.f) <*> xs
--             = mapAT (g.f) xs
--             == mapAT g (mapAT f xs)
```

```
--                 = gs <*> (fs <*> xs)
-- and the other cases are ‘trivial’:
--   applyAT (B l r) xs = B (applyAT l xs) (applyAT r xs)
-- so everything is decided in the leaves

-- fs <*> (pure y)
--   = fs <*> (L y)
--   == substitute each leaf (L f) in fs by [L (f y)]
--   = mapAT ($ y) fs
--   = L ($ y) <*> fs
--   = pure ($ y) <*> fs



bindAT :: AT a -> (a -> AT b) -> AT b
bindAT (L x) f   = f x
bindAT (B l r) f = B (bindAT l f) (bindAT r f)


instance Monad AT where
  return x = (L x)
  (>>=)    = bindAT


-- return a >>= f
--   = (L a) >>= f
--   = f a          (by the def of >>=)
--   == f a         (as we wanted)

-- at >>= return
--   = case at of
--     (L a) =>
--       = (L a) >>= return
--       = return a
--       == (L a)  (as we wanted)
--     (B l r) =>
--       = B (l>>=return) (r>>=return)
--         (reduced to the leaf case)

-- at >>= (\x -> f x >>= g)
--   ==   ????
-- (at >>= f) >>= g
--
-- case at of (L a):
--   = (L a) >>= (\x -> f x >>= g)
--   = (\x -> f x >>= g) a
--   = (f a) >>= g
--   == (f a) >>= g
--   = ((L a) >>= f) >>= g
--   = (at >>= f) >>= g
```

```
-- Is (<*>) equal to ap?
--   ap :: (Monad m) => m (a -> b) -> m a -> m b
--   ap m1 m2 = do {x1 <- m1; x2 <- m2; return (x1 x2)}
-- Translating the do block first
--   ap m1 m2  =  m1 >>= ( \x1 -> m2 >>= \x2 -> return (x1 x2) )
-- So:
--   ap at_fs at_xs
--     = at_fs >>= ( \x1 -> at_xs >>= \x2 -> return (x1 x2) )
--     case  at_fs of (L f)
--       = ( \x1 -> at_xs >>= \x2 -> return (x1 x2) ) f
--       = at_xs >>= (\x2 -> return (f x2))
--         case at_xs of (L x)
--           = (\x2 -> return (f x2)) x
--           = return (f x)
--           = L (f x)
--           == (L f) <*> (L x)




------------------------------------------------------------------------
--  ELEVENTH EXERCISE OF THE ADVANCED MONAD AND APPLICATIVE SECTION   --
------------------------------------------------------------------------


-- Grow the tree by replacing each leaf with a branch
-- containing two copies of the leaf.

fructify1 :: AT a -> AT a
fructify1 at =  at >>= (\a -> B (L a) (L a))

fructify1' :: AT a -> AT a
fructify1' at =  join ( fmap (\a -> B (L a) (L a)) at )

fructify2 :: AT a -> AT a
fructify2 at =  (fmap g at) <*> (B (L 0) (L 0))
  where g a = \_ -> a

fructify2' :: AT a -> AT a
fructify2' at =  (fmap const at) <*> (B (L "hi") (L "bye"))

fructify2'' :: AT a -> AT a
fructify2'' at = at <* (B (L True) (L False))

wrongFructify at = at *> (B (L 0) (L 0))



-- Replace a branch of a tree with a leaf carrying the default
-- value z whenever any of the leaves directly on it satisfies the test p

prune :: a -> (a -> Bool) -> AT a -> AT a
```

```
-- prune z p t = boolT ???
--   where boolT = fmap p t
-- None of the instances above allows to cut parts of the tree because
-- each of them only grows the tree(s).
prune z p (L x)             = if p x then (L z) else (L x)
prune z p t@(B (L x) (L y) ) = if (p x) || (p y) then (L z) else t
prune z p (B (L x) t)       = if p x then (L z) else B (L x) (prune z p t)
prune z p (B t (L y))       = if p y then (L z) else B (prune z p t) (L y)
prune z p (B l r)           = B (prune z p l) (prune z p r)


-- *Main> at1
-- B (L 5) (B (L 2) (L 100))
-- (0.00 secs, 12846520 bytes)
-- *Main> prune (-1) (<3) at1
-- B (L 5) (L (-1))
-- (0.00 secs, 0 bytes)
-- *Main> prune (-1) (<10) at1
-- L (-1)
-- (0.00 secs, 0 bytes)



-- Duplicate a tree applying two different functions

reproduce :: (a -> b) -> (a -> b) -> AT a -> AT b
reproduce f g t = B (fmap f t) (fmap g t)

reproduce2 :: (a -> b) -> (a -> b) -> AT a -> AT b
reproduce2 f g t = (B (L f) (L g)) <*> t

reproduce2' :: (a -> b) -> (a -> b) -> AT a -> AT b
reproduce2' f g t =  (B (L f) (L g)) >>= (\f -> fmap f t)

reproduce2'' :: (a -> b) -> (a -> b) -> AT a -> AT b
reproduce2'' f g t = (B (L f) (L g)) `ap` t




-------------------------------------------------------------------------
--   TWELFTH EXERCISE OF THE ADVANCED MONAD AND APPLICATIVE SECTION   --
-------------------------------------------------------------------------

-- First, the reversed sequencing version of (<*>)
--   (first, search for the leaves in the tree of values
--    second, pass each value to every function in the tree of funcs)
applyAT'' :: AT (a -> b) -> AT a -> AT b
applyAT'' at_fs (L x) = mapAT ($ x) at_fs
applyAT'' at_fs (B l r) = B (applyAT'' at_fs l) (applyAT'' at_fs r)

-- *Main> applyAT at3 at1
```

```
-- B (B (L 10) (B (L 4) (L 200))) (B (B (L 6) (B (L 3) (L 101))) (B (L 15) (B (L 6) (L 300))))
-- (0.00 secs, 0 bytes)
-- *Main> applyAT'' at3 at1
-- B (B (L 10) (B (L 6) (L 15))) (B (B (L 4) (B (L 3) (L 6))) (B (L 200) (B (L 101) (L 300))))
-- (0.00 secs, 0 bytes)


-- And the other version:

applyAT' :: AT (a -> b) -> AT a -> AT b
applyAT' (L f) at   = mapAT f at
applyAT' (B l r) at = B (applyAT' r at) (applyAT' l at)


-- *Main> applyAT at3 at1
-- B (B (L 10) (B (L 4) (L 200))) (B (B (L 6) (B (L 3) (L 101))) (B (L 15) (B (L 6) (L 300))))
-- (0.00 secs, 0 bytes)
-- *Main> applyAT' at3 at1
-- B (B (B (L 15) (B (L 6) (L 300))) (B (L 6) (B (L 3) (L 101)))) (B (L 10) (B (L 4) (L 200)))
-- (0.00 secs, 0 bytes)

-- pure id <*> t == t ??
-- pure id <*> t
--   = (L id) <*> t
--   = mapAT f t
--   == t

-- pure f <*> pure x == pure (f x) ??
-- pure f <*> pure x
--   = (L f) <*> (L x)
--   = mapAT f (L x)
--   = L (f x)
--   == pure (f x)

-- fs <*> pure x == pure ($ x) <*> fs ??
-- fs <*> pure x == pure ($ x) <*> fs
--   iff fs <*> (L x) == L ($ x) <*> fs
--   iff fs <*> (L x) == mapAT ($ x) fs
-- Now, fs <*> (L x) =
--   case fs of
--     (L f):
--       = (L f) <*> (L x)
--       = L (f x)
--       == L ($ x) <*> (L f)
--     (B l r):
--       = (B l r) <*> (L x)
--       = B (r <*> (L x)) (l <*> (L x))
--       =/= B (mapAT ($ x) l) (mapAT ($ x) r) ?!?!?!?!?
--       = mapAT ($ x) (B l r)
--       = L ($ x) <*> (B l r)

-- To solve that problem, i thought about:
```

```
-- -- First, we need an illegal version of fmap:

mapAT' :: (a -> b) -> AT a -> AT b
mapAT' f (L x)   = L (f x)
mapAT' f (B l r) = B (mapAT' f r) (mapAT' f l)

-- Occurs that
-- mapAT' id (B (L 1) (L 2))
--   = B (mapAT' id (L 2)) (mapAT' id (L 1))
--   = B (L 2) (L 1)
--   =/= B (L 1) (L 2)
-- However, the applicative instance seems ok   <-- NO:
--   pure id <*> t  =/=  t

-- So i went to
-- https://en.wikibooks.org/wiki/Haskell/Solutions/Applicative_functors#111
-- And found:

-- instance Applicative AT where
--     pure x                 = L x
--     L f       <*> tx        = fmap f tx
--     tf        <*> L x       = fmap ($ x) tf
--     B tfl tfr <*> B txl txr = B (tfl <*> txl) (tfr <*> txr)

-- " It only combines subtrees with matching positions in the tree structures.
-- The resulting behaviour is similar to that of ZipLists,
-- except that when the subtree shapes are different:
-- it inserts missing branches rather than removing extra ones
-- (and it couldn't be otherwise, since there are no empty ATs).
-- By the way, sagittalMap would have the exact same implementation of reproduce,
-- only using this instance. "

-- And seems ok:

-- pure id <*> t == t  OK
-- pure f <*> pure x == pure (f x)  OK
-- fs <*> pure x == pure ($ x) <*> fs  OK
-- pure (.) <*> gs <*> fs <*> as == gs <*> (fs <*> as)  OK


data AT' a = L' a | B' (AT' a) (AT' a)
  deriving (Show)

at1' :: AT' Int
at1' = B' (L' 5) (B' (L' 2) (L' 100))

at2' :: AT' (Int -> Char)
at2' = let f=toEnum.(100+) in B' (B' (L' f) (L' f)) (B' (L' f) (B' (L' f) (L' f)))

at3' :: AT' (Int -> Int)
at3' = B' (L' (2*)) ( B' (L' (1+)) (L' (3*)) )
```

```
instance Functor AT' where
  fmap f (L' x) = L' (f x)
  fmap f (B' l r) = B' (fmap f l) (fmap f r)

instance Applicative AT' where
  pure = L'
  L' f       <*> tx        = fmap f tx
  tf        <*> L' x       = fmap ($ x) tf
  B' tfl tfr <*> B' txl txr = B' (tfl <*> txl) (tfr <*> txr)



-- *Main> at3 <*> at1
-- B (B (L 10) (B (L 4) (L 200))) (B (B (L 6) (B (L 3) (L 101))) (B (L 15) (B (L 6) (L 300))))
-- (0.00 secs, 0 bytes)
-- *Main> at3' <*> at1'
-- B' (L' 10) (B' (L' 3) (L' 300))
-- (0.02 secs, 0 bytes)




------------------------------------------------------------------------
-- THIRTEENTH EXERCISE OF THE ADVANCED MONAD AND APPLICATIVE SECTION  --
------------------------------------------------------------------------

-- Recalling:
--   unit  :: Monoidal f => f ()
--   (*&*) :: Monoidal f => f a -> f b -> f (a,b)
--   pure  :: Applicative f => a -> f a
--   (<*>) :: Applicative f => f (a -> b) -> f a -> f b

-- IMPLEMENTING THE MONOIDAL CLASS:
-- (because its not defined in the Prelude)

class Functor f => Monoidal f where
  unit  :: f ()
  (*&*) :: f a -> f b -> f (a,b)


myUnit :: Applicative f => f ()
myUnit = pure ()

myOP :: Applicative f => f a -> f b -> f (a,b)
myOP a b = (fmap (,) a) <*> b


-- *Main> myOP [1,2,3] [4,5,6]
-- [(1,4),(1,5),(1,6),(2,4),(2,5),(2,6),(3,4),(3,5),(3,6)]
-- (0.02 secs, 0 bytes)
```

```
instance Monoidal [] where
  unit = [()]
  (*&*) l1 l2 = concatMap (((flip zip) l2) . repeat) l1

-- *Main> [1,2,3] *&* [4,5,6]
-- [(1,4),(1,5),(1,6),(2,4),(2,5),(2,6),(3,4),(3,5),(3,6)]



pureGivenMonoidal :: Monoidal f => a -> f a
pureGivenMonoidal x = fmap (\_ -> x) unit

applyGivenMonoidal :: Monoidal f => f (a -> b) -> f a -> f b
applyGivenMonoidal fs as = fmap (\(f,a) -> f a) (fs *&* as)




------------------------------------------------------------------------
-- FOURTEENTH EXERCISE OF THE ADVANCED MONAD AND APPLICATIVE SECTION  --
------------------------------------------------------------------------

-- Commutativity:
--   liftA2 f u v  ==  liftA2 (flip f) v u
--  or equivalently
--   f <$> u <*> v  ==  flip f <$> v <*> u
-- Or equivalently:
--   do {x <- u; y <- v; return (f x y)}
--     ==
--   do {y <- v; x <- u; return ((flip f) y x)}
--     ==
--   do {y <- v; x <- u; return (f x y)}

-- Or equivalently:
--   fmap (\(f,a) -> f a) ((f <$> u) *&* v)
--     ==
--   fmap (\(f,a) -> f a) ((flip f) <$> v *&* u)




------------------------------------------------------------------------
--  FIFTEENTH EXERCISE OF THE ADVANCED MONAD AND APPLICATIVE SECTION  --
------------------------------------------------------------------------

instance Monoidal ZipList where
  unit = ZipList [()]
  (*&*) (ZipList l1) (ZipList l2) = ZipList (zip l1 l2)


-- *Main> (,) <$> (ZipList [1,2,3,4]) <*> (ZipList [10,11,12])
```

```
-- ZipList {getZipList = [(1,10),(2,11),(3,12)]}
-- (0.00 secs, 0 bytes)
-- *Main> (ZipList [1,2,3,4]) *&* (ZipList [10,11,12])
-- ZipList {getZipList = [(1,10),(2,11),(3,12)]}
-- (0.00 secs, 0 bytes)



instance Monoidal ((->) r) where
  unit = const ()
  (*&*) f g = \r -> (f r, g r)


-- *Main> ((,) <$> (\n -> 2*n) <*> (\n -> 3*n)) 5
-- (10,15)
-- (0.00 secs, 0 bytes)
-- *Main> ((\n -> 2*n) *&* (\n -> 3*n)) 5
-- (10,15)
-- (0.00 secs, 0 bytes)
```

L.3 *State* exercises

1. Implement a function `rollNDiceIO :: Int -> IO [Int]` that, given an integer (a number of die rolls), returns a list of that number of pseudo-random integers between 1 and 6.

2. Implement a function `rollDice :: StdGen -> ((Int, Int), StdGen)` that, given a generator, returns a tuple with our random numbers as first element and the last generator as the second.

3. Similarly to what was done for $\boxed{\text{rollNDiceIO}}$, implement a function

   ```
   rollNDice :: Int -> State StdGen [Int]
   ```

   that, given an integer, returns a list with that number of pseudo-random integers between 1 and 6.

4. Write an instance of $\boxed{\text{Functor}}$ for $\boxed{\text{State s}}$. Your final answer should not use anything that mentions $\boxed{\text{Monad}}$ in its type (that is, $\boxed{\text{return}}$, $\boxed{\text{(>>=)}}$, etc.). Then, explain in a few words what the $\boxed{\text{fmap}}$ you wrote does.

   (Hint: If you get stuck, have another look at the comments about $\boxed{\text{liftM}}$ in the main body.)

5. Besides $\boxed{\text{put}}$ and $\boxed{\text{get}}$, there are also

   ```
   modify :: (s -> s) -> State s ()
   ```

   which modifies the current state using a function, and

   ```
   gets :: (s -> a) -> State s a
   ```

   which produces a modified copy of the state while leaving the state itself unchanged. Write implementations for them.

6. If you are not convinced that $\boxed{\text{State}}$ is worth using, try to implement a function equivalent to $\boxed{\text{evalState allTypes}}$ without making use of monads, i.e. with an approach similar to $\boxed{\text{clumsyRollDice}}$ above.

```
{-# LANGUAGE TypeSynonymInstances #-}

import Control.Monad
import Control.Monad.State
import Control.Applicative
import System.Random



------------------------------------------------------------------------
--               FIRST EXERCISE OF THE STATE SECTION                  --
------------------------------------------------------------------------

rollNDiceIO :: Int -> IO [Int]
rollNDiceIO n = (take n) <$> (getStdGen >>= f)
  where
    f = \ gen -> return (randomRs (1,6) gen)


{-

*Main> rollNDiceIO 10
Loading package array-0.5.0.0 ... linking ... done.
Loading package deepseq-1.3.0.2 ... linking ... done.
Loading package bytestring-0.10.4.0 ... linking ... done.
Loading package Win32-2.3.0.2 ... linking ... done.
Loading package old-locale-1.0.0.6 ... linking ... done.
Loading package time-1.4.2 ... linking ... done.
Loading package random-1.0.1.1 ... linking ... done.
[5,5,1,6,6,5,1,6,5,3]
(0.11 secs, 5620896 bytes)
*Main> rollNDiceIO 6
[5,5,1,6,6,5]
(0.00 secs, 0 bytes)
*Main> rollNDiceIO 10
[5,5,1,6,6,5,1,6,5,3]
(0.02 secs, 0 bytes)
*Main> rollNDiceIO 10
[5,5,1,6,6,5,1,6,5,3]
(0.00 secs, 0 bytes)
*Main> getStdGen
700125431 1
(0.00 secs, 0 bytes)
*Main> getStdGen
700125431 1
(0.00 secs, 0 bytes)
*Main> newStdGen
895916699 2147483398
(0.00 secs, 0 bytes)
*Main> getStdGen
700125432 40692
(0.00 secs, 0 bytes)
```

```
*Main> getStdGen
700125432 40692
(0.00 secs, 0 bytes)
*Main> newStdGen
895956713 40691
(0.00 secs, 0 bytes)
*Main> getStdGen
700125433 1655838864
(0.00 secs, 0 bytes)
*Main> rollNDiceIO 10
[4,5,4,3,5,6,6,5,2,5]
(0.00 secs, 0 bytes)

-}


rollNDiceIO' :: Int -> IO [Int]
rollNDiceIO' n = sequence (fmap randomRIO (replicate n (1,6)))

-- does exactly the same




-----------------------------------------------------------------------
--                SECOND EXERCISE OF THE STATE SECTION               --
-----------------------------------------------------------------------

{-
roll2Dice :: StdGen -> ((Int,Int) , StdGen)
roll2Dice gen = ((n1,n2) , b) where
  (gen1, gen2) = split gen
  n1 = fst $ randomR (1,6) gen1
  n2 = fst $ randomR (1,6) gen2
  b  = snd $ randomR (1,6) gen2

This version seems correct but gives an "ambiguous type" error
-}


roll2Dice' :: StdGen -> ((Int,Int) , StdGen)
roll2Dice' gen = ((n1,n2) , b) where
  (gen1, gen2) = split gen
  n1 = fst $ aux gen1
  n2 = fst $ aux gen2
  b  = snd $ aux gen2
  aux = (\g -> randomR (1,6) g)  :: StdGen -> (Int,StdGen)


{-

*Main> roll2Dice' (mkStdGen 0)
```

```
Loading package array-0.5.0.0 ... linking ... done.
Loading package deepseq-1.3.0.2 ... linking ... done.
Loading package bytestring-0.10.4.0 ... linking ... done.
Loading package Win32-2.3.0.2 ... linking ... done.
Loading package old-locale-1.0.0.6 ... linking ... done.
Loading package time-1.4.2 ... linking ... done.
Loading package random-1.0.1.1 ... linking ... done.
((6,1),1601120196 2147442707)
(0.08 secs, 6278672 bytes)
*Main> roll2Dice' (mkStdGen 0)
((6,1),1601120196 2147442707)
(0.02 secs, 0 bytes)
*Main> roll2Dice' (mkStdGen 38)
((6,2),166664317 2147442707)
(0.02 secs, 8181848 bytes)
*Main> roll2Dice' (mkStdGen 123)
((6,5),970416508 2147442707)
(0.00 secs, 0 bytes)
*Main> roll2Dice' (mkStdGen 3456789)
((2,1),1125608184 2147442707)
(0.00 secs, 0 bytes)
*Main> roll2Dice' (mkStdGen 4689273)
((3,4),1290960903 2147442707)
(0.00 secs, 0 bytes

-}


roll2Dice'' :: StdGen -> ((Int,Int) , StdGen)
roll2Dice'' gen = ((n1,n2) , res) where
  (n1,aux) = randomR (1,6) gen
  (n2,res) = randomR (1,6) aux

-- *Main> roll2Dice' (mkStdGen 0)
-- ((6,1),1601120196 2147442707)
-- (0.00 secs, 0 bytes)
-- *Main> roll2Dice'' (mkStdGen 0)
-- ((6,6),1601120196 1655838864)
-- (0.02 secs, 8290224 bytes)




-------------------------------------------------------------------------
--                THIRD EXERCISE OF THE STATE SECTION                  --
-------------------------------------------------------------------------

rollNDice :: Int -> State StdGen [Int]
rollNDice n = state ( \gen -> ((take n) $ (randomRs (1,6) gen) , (snd.next) gen) )


{-
```

```
*Main> rollNDice 10

<interactive>:21:1:
    No instance for (Show (State StdGen [Int]))
      arising from a use of print
    In a stmt of an interactive GHCi command: print it
(0.02 secs, 0 bytes)
*Main> runState (rollNDice 10) (mkStdGen 0)
Loading package transformers-0.4.3.0 ... linking ... done.
Loading package mtl-2.2.1 ... linking ... done.
([6,6,4,1,5,2,4,2,2,1],40014 40692)
(0.02 secs, 0 bytes)
*Main> runState (rollNDice 10) (mkStdGen 0)
([6,6,4,1,5,2,4,2,2,1],40014 40692)
(0.00 secs, 0 bytes)
*Main> runState (rollNDice 11) (mkStdGen 0)
([6,6,4,1,5,2,4,2,2,1,6],40014 40692)
(0.00 secs, 0 bytes)
*Main> runState (rollNDice 11) (mkStdGen 534621)
([3,1,5,6,2,4,3,3,6,1,2],2065012641 40692)
(0.00 secs, 0 bytes)

-}




------------------------------------------------------------------------
--                FOURTH EXERCISE OF THE STATE SECTION                --
------------------------------------------------------------------------


-- instance Functor (State s) where

{-
StateExercises.hs:188:10:
    Illegal instance declaration for Functor (State s)
      (All instance types must be of the form (T t1 ... tn)
       where T is not a synonym.
       Use TypeSynonymInstances if you want to disable this.)
    In the instance declaration for Functor (State s)
-}

-- After adding that at the top of this file:

{-
StateExercises.hs:190:10:
    Illegal instance declaration for Functor (State s)
      (All instance types must be of the form (T a1 ... an)
       where a1 ... an are *distinct type variables*,
       and each type variable appears at most once in the instance head.
```

```
      Use FlexibleInstances if you want to disable this.)
    In the instance declaration for Functor (State s)
-}

-- Redefining the State data (to avoid these problems):

data State' s a = St (s -> (a,s))

instance Functor (State' s) where
  fmap f (St g) = St ( \s -> ( f $ fst $ g s , snd $ g s) )


-- fmap id x == x ??
-- fmap id (St g)
--   = St ( \s -> ( fst $ g s , snd $ g s) )
--   = St ( \s -> g s)
--   == (St g)

-- fmap (g.f) x == (fmap g) . (fmap f) x ??
-- fmap (g.f) (St pr)
--   = ST ( \s -> ((f.g) $ fst $ pr s , snd $ pr s) )
--   == fmap g (St (\s -> (f $ fst $ pr s , snd $ pr s)))
--   = ((fmap g) . (fmap f)) x

-- This Functor instance works as follow:
--   fmap f (St pr)
--     with  f :: a -> b , pr :: s -> (a,s)
--   applies f to the first value of the pair returned by pr



------------------------------------------------------------------------
--               FIFTH EXERCISE OF THE STATE SECTION                  --
------------------------------------------------------------------------

myModify :: (s -> s) -> State s ()
myModify f = state ( \s -> (() , f s) )

-- *Main> runState (myModify (2*)) 5
-- ((),10)
-- (0.00 secs, 0 bytes)



myGets :: (s -> a) -> State s a
myGets f = state ( \s -> (f s , s) )

-- *Main> runState (myGets (toEnum :: Int -> Char)) 90
-- ('Z',90)
-- (0.00 secs, 0 bytes)
```

```
-----------------------------------------------------------------------
--               SIXTH EXERCISE OF THE STATE SECTION                --
-----------------------------------------------------------------------


-- evalState :: State s a -> s -> a

allTypes :: State StdGen (Int, Float, Char, Integer, Double, Bool, Int)
allTypes = liftM (,,,,,,) getRandom
                     `ap` getRandom
                     `ap` getRandom
                     `ap` getRandom
                     `ap` getRandom
                     `ap` getRandom
                     `ap` getRandom

getRandom :: Random a => State StdGen a
getRandom = state random



monsterRandom :: StdGen -> (Int, Float, Char, Integer, Double, Bool, Int)
monsterRandom gen = (n1,f1,c1,n2,d1,b1,n3)
  where
    (n1,g1) = random gen
    (f1,g2) = random g1
    (c1,g3) = random g2
    (n2,g4) = random g3
    (d1,g5) = random g4
    (b1,g6) = random g5
    (n3,_) = random g6



-- *Main> evalState allTypes (mkStdGen 0)
-- (-117157315039303149,0.4883204,'\260381',-2598893763451025729,0.30447780927171453,False,-525544148
-- (0.02 secs, 0 bytes)
-- *Main> monsterRandom (mkStdGen 0)
-- (-117157315039303149,0.4883204,'\260381',-2598893763451025729,0.30447780927171453,False,-525544148
-- (0.00 secs, 0 bytes)
```

L.4 *MonadPlus* exercises

1. Prove the MonadPlus laws for Maybe and the list monad.
2. We could augment our above parser to involve a parser for any character:

```
-- | Consume a given character in the input, and return the character we
--   just consumed, paired with rest of the string. We use a do-block  so that
--   if the pattern match fails at any point, fail of the Maybe monad (i.e.
--   Nothing) is returned.
char :: Char -> String -> Maybe (Char, String)
char c s = do
  let (c':s') = s
  if c == c' then Just (c, s') else Nothing
```

   It would then be possible to write a $\boxed{\text{hexChar}}$ function which parses any valid hexadecimal character (0-9 or a-f). Try writing this function

   (hint: `map digit [0..9] :: [String -> Maybe Int]`).

```
import Control.Monad

-----------------------------------------------------------------------
--              FIRST EXERCISE OF THE MONADPLUS SECTION              --
-----------------------------------------------------------------------


-- instance MonadPlus [] where
--   mzero = []
--   mplus = (++)

-- neutral element:
--  mzero `mplus` m = [] ++ m == m
--  m `mplus` mzero = m ++ [] == m

-- associativity
-- (l1 `mplus` l2) `mplus` l3
--   = (l1 ++ l2) ++ l3
--   == l1 ++ (l2 ++ l3)
--   = l1 `mplus` (l2 `mplus` l3)

-- interaction with the monad part

-- mzero >>= f  ==  mzero  ??
-- mzero >>= f
--   = [] >>= f
--   = concatMat f []
--   = []
--   == mzero

-- l >> mzero  ==  mzero  ??
-- l >> mzero
--   = l >> []
--   = l >>= (\_ -> [])
--   = concatMap (\_ -> []) l
--   = []
--   == mzero

-- (l1 `mplus` l2) >>= k
--   ==  ????
-- (l1 >>= k) `mplus` (l2 >>= k)

-- (l1 `mplus` l2) >>= k
--   = (l1 ++ l2) >>= k
--   = concatMap k (l1++l2)
--   == (concatMap k l1) ++ (concatMap k l2)
--   = (l1 >>= k) `mplus` (l2 >>= k)




-----------------------------------------------------------------------
```

```
--             SECOND EXERCISE OF THE MONADPLUS SECTION             --
----------------------------------------------------------------------


char :: Char -> String -> Maybe (Char, String)
char c s = do
  let (c':s') = s
  if c == c' then Just (c,s') else Nothing


digit :: Int -> String -> Maybe Int
digit i s | i > 9 || i < 0 = Nothing
          | otherwise      = do
  let (c:_) = s
  if [c] == show i then Just i else Nothing


hexChar :: String -> Maybe Char
hexChar s = (fmap (head . show) (isDigit s)) `mplus` isValidChar
  where
    funcList    = map digit [0..9]
    isDigit x   = msum (map ($ x) funcList)
    char' c s   = fmap fst (char c s)
    funcList'   = map char' ['a','b','c','d','e','f']
    isValidChar = msum (map ($ s) funcList')
```

L.5 Monad transformers exercises'

1. Why is it that the $\boxed{\text{lift}}$ function has to be defined separately for each monad, where as $\boxed{\text{liftM}}$ can be defined in a universal way?

2. $\boxed{\text{Identity}}$ is a trivial functor, defined in $\boxed{\text{Data.Functor.Identity}}$ as:

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

   It has the following Monad instance:

```
instance Monad Identity where
    return a = Identity a
    m >>= k  = k (runIdentity m)
```

   Implement a monad transformer $\boxed{\text{IdentityT}}$, analogous to $\boxed{\text{Identity}}$ but wrapping values of type $\boxed{\text{m a}}$ rather than $\boxed{\text{a}}$. Write at least its $\boxed{\text{Monad}}$ and $\boxed{\text{MonadTrans}}$ instances.

3. Implement `state :: MonadState s m => (s -> (a, s)) -> m a` in terms of $\boxed{\text{get}}$ and $\boxed{\text{put}}$.

4. Are $\boxed{\text{MaybeT (State s)}}$ and $\boxed{\text{StateT s Maybe}}$ equivalent? (Hint: one approach is comparing what the $\boxed{\text{run...T}}$ unwrappers produce in each case.)

```
------------------------------------------------------------------------
--         FIRST EXERCISE OF THE MONAD TRANSFORMERS SECTION           --
------------------------------------------------------------------------
```

L.6 Hask category exercises

1. As was mentioned, any partial order $(P, \leq$ is a category with objects the elements of P and a morphism between elements a and b iff $a \leq b$. Which of the above laws guarantees the transitivity of $\leq$?

2. Check the functor laws for the Maybe and list functors.

3. Verify that the list and $\boxed{\text{Maybe}}$ monads do in fact obey the first monad law,

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

   with some examples to see precisely how the layer flattening works.

4. Prove the second monad law, $\boxed{\text{join . fmap return} = \text{join . return} = \text{id}}$ for the $\boxed{\text{Maybe}}$ monad.

5. Convince yourself that the 3rd and 4th laws should hold true for any monad by exploring what they mean, in a similar style to how the first and second laws were explored.

6. In fact, the two versions of the laws we gave:

```
-- Categorical:
join . fmap join = join . join
join . fmap return = join . return = id
return . f = fmap f . return
join . fmap (fmap f) = fmap f . join

-- Functional:
m >>= return = m
return m >>= f = f m
(m >>= f) >>= g = m >>= (\x -> f x >>= g)
```

   are entirely equivalent. We showed that we can recover the functional laws from the categorical ones. Go the other way; show that starting from the functional laws, the categorical laws hold. It may be useful to remember the following definitions:

```
join m = m >>= id
fmap f m = m >>= return . f
```

```
-----------------------------------------------------------------------
--            FIRST EXERCISE OF THE HASK CATEGORY SECTION            --
-----------------------------------------------------------------------

-- Given a partially ordered set (P, <=), we can define a category whose
-- objects are the elements of P, and there is a morphism between elements
-- a and b  iff  a <= b

-- The transitivity of <= guarantees the existence of the composition law,
-- this is:
--   if f and g exist, with  f : a -> b  ,  g : b -> c
--   ==> a <= b  and  b <= c
--   ==> a <= c   (transitivity)
--   ==> exists h : a -> c
--         (so we can asign g.f = h)




-----------------------------------------------------------------------
--           SECOND EXERCISE OF THE HASK CATEGORY SECTION            --
-----------------------------------------------------------------------

-- Maybe functor:
{-
instance  Functor Maybe  where
    fmap _ Nothing       = Nothing
    fmap f (Just a)      = Just (f a)
-}

-- fmap id Nothing == Nothing
-- fmap id (Just x) = Just (id x) == (Just x)

-- fmap (g.f) Nothing
--   = Nothing
--   == fmap g Nothing
--   = fmap g (fmap f Nothing)

-- fmap (g.f) (Just x)
--   = Just ( g (f x) )
--   == fmap g (Just (f x))
--   = fmap g (fmap f (Just x))


-- List functor:
{-
instance Functor [] where
    {-# INLINE fmap #-}
    fmap = map
-}
```

```haskell
{-
map _ []     = []
map f (x:xs) = f x : map f xs
-}

-- map id [] == []
-- map id (x:xs)
--   = (id x) : map id xs
--   = x : map id xs
--   == (x:xs)

-- map (g.f) []
--   == []
--   = map g []
--   = map g (map f [])

-- map (g.f) (x:xs)
--   = (g (f x)) : map (g.f) xs
--   == map g ((f x):xs)
--   = map g (map f (x:xs))




-------------------------------------------------------------------------
--             THIRD EXERCISE OF THE HASK CATEGORY SECTION             --
-------------------------------------------------------------------------
{-
join x = x >>= id
-}

-- Maybe monad:
{-
instance  Monad Maybe  where
    (Just x) >>= k      = k x
    Nothing  >>= _      = Nothing
-}
{- occurs that
join :: Maybe (Maybe a) -> Maybe a
join Nothing         = Nothing
join (Just Nothing)  = Nothing
join (Just (Just x)) = Just x
-}

-- (join . fmap join) Nothing
--   = join Nothing
--   == Nothing
--   = join (join Nothing)

-- (join . fmap join) (Just Nothing)
--   == Nothing
--   = (join . join) (Just Nothing)
```

```
-- (join . fmap join) (Just (Just Nothing))
--   = join (Just (join (Just Nothing)))
--   = join (Just Nothing)
--   == Nothing
--   = join (Just Nothing)
--   = (join . join) (Just (Just Nothing))

-- (join . fmap join) (Just (Just (Just x)))
--   = join (Just (join (Just (Just x))))
--   = join (Just (Just x))
--   = Just x
--   == join (Just (Just x))
--   = (join . join) (Just (Just (Just x)))


-- List monad:
--   Some examples

{-
Prelude Control.Monad> (join . fmap join) [[]]
[]
(0.00 secs, 0 bytes)
Prelude Control.Monad> (join . fmap join) []
[]
(0.00 secs, 2959128 bytes)
Prelude Control.Monad> (join . fmap join) [[]]
[]
(0.00 secs, 0 bytes)
Prelude Control.Monad> (join . fmap join) [[[]]]
[]
(0.00 secs, 3707032 bytes)
Prelude Control.Monad> (join . fmap join) [[[[]]]]
[[]]
(0.02 secs, 0 bytes)
Prelude Control.Monad> (join . join) []
[]
(0.00 secs, 0 bytes)
Prelude Control.Monad> (join . join) [[]]
[]
(0.00 secs, 0 bytes)
Prelude Control.Monad> (join . join) [[[]]]
[]
(0.00 secs, 0 bytes)
Prelude Control.Monad> (join . join) [[[[]]]]
[[]]
(0.00 secs, 0 bytes)
-}
-- Prelude Control.Monad> fmap join [ [[1,2],[3]] , [[4,5],[6,7]] ]
-- [[1,2,3],[4,5,6,7]]     <-- CHECK THIS
-- (0.00 secs, 0 bytes)
-- Prelude Control.Monad> (join . fmap join) [ [[1,2],[3]] , [[4,5],[6,7]] ]
```

```
-- [1,2,3,4,5,6,7]
-- (0.00 secs, 0 bytes)
-- Prelude Control.Monad>  join [ [[1,2],[3]] , [[4,5],[6,7]] ]
-- [[1,2],[3],[4,5],[6,7]] <- AND THIS
-- (0.00 secs, 0 bytes)
-- Prelude Control.Monad> (join . join) [ [[1,2],[3]] , [[4,5],[6,7]] ]
-- [1,2,3,4,5,6,7]
-- (0.02 secs, 0 bytes)



-----------------------------------------------------------------------
--            FOURTH EXERCISE OF THE HASK CATEGORY SECTION            --
-----------------------------------------------------------------------

-- Second law:
--   join . fmap return = join . return = id

-- (join . fmap return) Nothing
--   = join Nothing
--   = Nothing
--   == join (Just Nothing)
--   = (join . return) Nothing

-- (join . fmap return) (Just x)
--   = join (Just (Just x))
--   = Just x
--   == join (Just (Just x))
--   = (join . return) (Just x)



-----------------------------------------------------------------------
--            FIFTH EXERCISE OF THE HASK CATEGORY SECTION             --
-----------------------------------------------------------------------

-- Third law:
--   return . f = fmap f . return

-- f :: a -> b        , (return . f)      = a -> m b
-- return :: a -> m a , (fmap f . return) = a -> m b

-- States that applying a function to a value and then
-- embedding the result into the monad is the same as
-- embedding the value into the monad and then mapping the function.



-- Fourth law:
--   join . fmap (fmap f) = fmap f . join
```

```
-- (fmap f) :: m a -> m b ,
-- fmap (fmap f) :: m (m a) -> m (m b),
-- join . fmap (fmap f) :: m (m a) -> m b
-- AND on the other side:
-- join :: m (m a) -> m a ,
-- fmap f . join :: m (m a) -> m b

-- States that, when having a 2 layer monadic value m (m a)
-- its the same joining and then mapping or
-- mapping to the inner layer and joining afterwards.




------------------------------------------------------------------------
--            SIXTH EXERCISE OF THE HASK CATEGORY SECTION             --
------------------------------------------------------------------------

-- Recalling:
--   join m   =  m >>= id
--   fmap f m =  m >>= return . f



-- First Law:
--   join . fmap join == join . join  ??

-- (join . fmap join) m
--   = join (fmap join m)
--   = join (m >>= return . join)
--   = (m >>= return . join) >>= id
--   = (m >>= id) >>= id     *****
--   = join (m >>= id)
--   = join (join m)
--   = (join . join) m

-- ***** OBS: we need to prove that return.join == id
--            (which is the second law)


-- Second Law:
--   join . fmap return == join . return == id ??

-- (join . fmap return) m
--   = join (fmap return m)
--   = join (m >>= return . return)
--   = join (return m)

-- (join . return) m
--   = join (return m)
--   = return m >>= id
```

```
--   = id m
--   == m


-- Third Law:
--   return . f = fmap f . return

-- (return  . f) x
--   = return (f x)

-- (fmap f . return) x
--   = fmap f (return x)
--   = (return x) >>= return . f
--   = (return . f) x


-- Fourth Law:
--   join . fmap (fmap f) = fmap f . join

-- (fmap f . join) m
--   = fmap f (join m)
--   = fmap f (m >>= id)
--   = (m >>= id) >>= return . f
--   = (m >>= id) >>= fmap f . return

--   = m >>= (\m' -> id m' >>= return . f)
--   = m >>= (\m' -> m' >>= return . f)

-- (join . fmap (fmap f)) m
--   = join (fmap (fmap f) m)
--   = join (m >>= return . (fmap f))
--   = join (m >>= \x -> return (fmap f x))
--   = (m >>= \x -> return (fmap f x)) >>= id
```

Appendix M

FAQS

Frequently Asked Questions, as found in high-quality webpages.

M.1 Where does the term “Monad” come from?

M.2 A monad is just a monoid in the category of endofunctors, what’s the problem?

M.3 How to extract value from monadic action?

M.4 How is $<*>$ pronounced?

M.5 Distinction between typeclasses MonadPlus, Alternative and Monoid?

M.6 Functions from ‘Alternative’ type class

M.7 Confused by the meaning of the ‘Alternative’ type class and its relationship with other type classes

M.8 What’s wrong with GHC Haskell’s current constraint system?

M.9 Lax monoidal functors with a different monoidal structure

StackExchange sign up log in tour help

English Language & Usage Stack Exchange is a question and answer site for linguists, etymologists, and serious English language enthusiasts. It's 100% free, no registration required.

Sign up

Here's how it works:

Anybody can ask a question

Anybody can answer

The best answers are voted up and rise to the top

Where does the term "Monad" come from?

I understand how monads work, and I use them on a routine basis. However, I've been wondering where the term actually comes from and what does it mean?

Edit: To clarify, I'm specifically referring to the origin of the term.

etymology terminology

edited Jun 21 '11 at 2:19
jozzas
295 1 4 11

asked Jun 20 '11 at 6:09
Kurios

migrated from programmers.stackexchange.com Jun 20 '11 at 20:05

This question came from our site for professional programmers interested in conceptual questions about software development.

This may help? en.wikipedia.org/wiki/Monad – Crazy Eddie Jun 20 '11 at 6:43

2 From your comments on the answers so far, it looks like the question could do with a bit more clarity. – Paul Butcher Jun 20 '11 at 8:44

3 For the benefit of anyone seeing this question after the migration, the question is specifically about this term, not other uses of the word. – camccann Jun 20 '11 at 20:20

Thanks to camccann for taking the time to give such en excellent answer. Sadly I wasn't logged in properly at the time I posted it, so now I can't upvote it or mark it as correct. – user10131 Jun 21 '11 at 0:47

If you want ownership of the question back, you might be able to get a moderator to help--I gather that merging unintentional duplicate accounts comes up fairly often. I'm not sure how to go about requesting it, though. But don't worry on my account, knowing I helped is all I need. :] – camccann Jun 21 '11 at 1:08

6 Answers

You seem to be asking about the origin of the term as used in category theory. The history of the term there is somewhat unclear, but it can at least be traced back a little ways:

- The term is sometimes attributed to Mac Lane, but this seems to be inaccurate; however, the widespread *use* of the term is probably due to his influential "Categories for the Working Mathematician", replacing the remarkably terrible term "triple".

> The frequent but unfortunate use of the word "triple" in this sense has achieved a maximum of needless confusion, what with the conflict with ordered triple, plus the use of associated terms such as "triple derived functors" for functors which are not three times derived from anything in the world. Hence the term monad.

- Mac Lane's use of the term was apparently prompted by J. P. May:

> The name "operad" is a word that I coined myself, spending a week thinking about nothing else. Besides having a nice ring to it, the name is meant to bring to mind both operations and monads. Incidentally, I persuaded MacLane to discard the term "triple" in favor of "monad" in his book "Categories for the working mathematician", which was being written about the same time. I was convinced that the notion of an operad was an important one, and I wanted the names to mesh.

- Elsewhere, Ross Street attributes the term to Jean Bénabou:

> Meanwhile Jean Bénabou had invented weak 2-categories, calling them bicategories. (...) He pointed out that a lax functor from the terminal category 1 to Cat was a category A equipped with a "standard construction" or "triple" (that is, a monoid in the

> monoidal category [A, A] of endofunctors of A where the tensor product is composition); he introduced the term monad for this concept.

The attribution to Bénabou is also mentioned here.

- The motivation for the term is to suggest a relationship with monoids, as can be deduced from the construction given in the quote above, and the Greek root "monos" comes second-hand. The connection to philosophy in general, or Leibniz in particular, is often asserted but never to my knowledge supported in any way. More likely if anything would be a connection to the term "monad" used in non-standard analysis, also related to Leibniz, but I'm not sure what the conceptual link there would be. An anecdote from Michael Barr relates the first use of the term:

> (...) The attendance consisted of practically everyone in the world who had any interest in categories, with the notable exception of Charles Ehresmann. (...) One day at lunch or dinner I happened to be sitting next to Jean Benabou and he turned to me and said something like "How about 'monad'?" I thought about and said it sounded pretty good to me. (Yes, I did.) So Jean proposed it to the general audience and there was general agreement.

The off-the-cuff nature of the suggestion, and immediate positive response from a large audience, suggests that there's probably no written record of the term being introduced formally. It's certainly possible that the word was borrowed from use in philosophy or elsewhere, but in any case there appears to be no connection more meaningful than the level of "cheap pun".

As far as I know, the only way you're going to get a better answer than that is by asking Bénabou himself.

edited Jun 21 '11 at 1:50

answered Jun 20 '11 at 20:15
camccann
339 2 7

2 excellent answer – FinnNk Jun 20 '11 at 20:20

1 I'm not sure I really care enough to follow up all those links, but I'm impressed. You must have taken some considerable trouble to chase all that down (please don't say *you* cared enough to have previously committed it to memory! :-) – FumbleFingers Jun 21 '11 at 2:16

1 @FumbleFingers: Haha, no! Just familiar enough with the subject matter to be *very efficient* at digging things up with Google. :] – camccann Jun 21 '11 at 2:29

Lawvere, I believe, suggests that it is a contraction of "monoidal triad" in particular. In fact, the Mac Lane citation, if one reads one sentence earlier, also makes this suggestion, since it mentions "triad" and "monoid" as well as "triple". – sclv Nov 23 '12 at 7:38

"The name is taken from the mathematical monad construct in category theory."

In math the name probably came from the greek word "monos" meaning "single", "unit"

http://en.wikipedia.org/wiki/Monad_(functional_programming)

answered Jun 20 '11 at 6:27
mko
211 1 1

4 But then the question is, where did category theory get the name? – bdonlan Jun 20 '11 at 6:28

Yeah, I found that wikipedia page before asking this, but I haven't been able to find the original source of the name. – Kurios Jun 20 '11 at 6:37

I believe that monads originated with Leibniz' metaphysical theory. Essentially, the monad acts as an interface between the worldly, corporeal and the spiritual, reflecting what happens on one side to the other and back.

Essentially an attempt to solve the mind-body problem.

As to why it was eventually snapped up in mathematical theor{y,ies} I do not know, but that is definitely what I think of when I hear "monad" (and monads in Haskell seem to share some of the qualities of Leibnizian monads).

answered Jun 20 '11 at 12:23
Vatine

1 I believe the connection is purely coincidental – FinnNk Jun 20 '11 at 20:20

1 No, Leibniz' monads are completely unrelated. – Marcin Jun 20 '11 at 20:24

I believe it is a backformation from dyad and triad.

A dyad is a couple, but not just any group of two. It is a group of two that forms a complete unit. A classical example is a group of friends with two people at the centre. They might be lovers, or roommates, classmates, or brothers. But everyone in the group is there because of one or the other of the dyad. Everyone has a tight connection to them. Often in a workplace there will be two people who form a dyad and the rest of the team forms around them. A triad is a group of three that rules something. Together the three of them form a ruling unit.

With those definitions in mind, what would a monad be? A single thing that is a thing all to itself. Sounds ok to me.

answered Jun 20 '11 at 15:35
Kate Gregory
7,988 1 22 36

1 Cf. *decade*, *Iliad*. The Greek suffix *-as* (stem *-ad-*) is used to—ehm, it is hard to pin down. I'd say it makes something into an abstract unit that normally isn't one, like Latin *-tas*, gen. *-tat-*, as in *trinitas* ("trinity"), *unitas* ("unity"), and *paucitas* ("paucity"; from *pauci*, "few"). – Cerberus Jun 20 '11 at 20:16

> "unity, arithmetical unit," 1610s, from L. monas (gen. monadis), from Gk. monas "unit," from monos "alone" (see mono-). In Leibnitz's philosophy, "an ultimate unit of being" (1748).

Reference

answered Jun 20 '11 at 15:57
Brad Christie

What the hell are monads? Your paragraph is "So, Monads"

answered Jun 20 '11 at 6:45
Sergey
172 2 8

1 Again, that says that it comes from the mathematical notion of a monad as well as defining the functions of the monad laws, but it doesn't give any insight to the origin of the name. – Kurios Jun 20 '11 at 6:52

StackExchange sign up log in tour help

Stack Overflow is a community of 4.7 million programmers, just like you, helping each other.

Join them; it only takes a minute:

Sign up

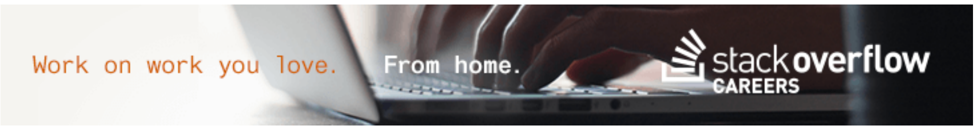
Join the Stack Overflow community to:

Ask programming questions

Answer and help your peers

Get recognized for your expertise

A monad is just a monoid in the category of endofunctors, what's the problem?



Who first said

> A monad is just a monoid in the category of endofunctors, what's the problem?

and on a less important note is this true and if so could you give an explanation (hopefully one that can be understood by someone who doesn't have much haskell experience).

haskell quotes monads category-theory monoids

edited Dec 24 '10 at 15:43

asked Oct 6 '10 at 6:55
Roman A. Taycher
3,122 ● 13 ● 42 ● 90

50 "hopefully one that can be understood by someone who doesn't have much haskell experience" Whether or not someone can understand this quote has little to do with his Haskell experience and a lot with his maths experience/knowledge about category theory. Also understanding this quote will tell you nothing about how monads in Haskell work and how to use them. So if that's your intention, you should not use this quote as a starting point (and probably forget that monads come from category theory altogether unless you want to understand why they are named monads). – sepp2k Oct 6 '10 at 8:03

38 IOW: it's a joke. – luqui Oct 6 '10 at 13:19

8 See "Categories for the Working Mathematician" – Don Stewart Oct 6 '10 at 15:27

10 You don't need to understand this to use monads in Haskell. From a practical perspective they are just a clever way to pass around "state" through some underground plumbing. – starblue Oct 7 '10 at 18:00

18 Actually, really understanding this quote *did* help me understand monads in a deeper sense, as well as monoids and functors. It only requires you to know other concepts which you *should* know anyway, to truly understand those concepts. And when you do, it nicely brings the concept to a single mental point. So ignore the stupid unconstructive comments above. All one needs, is a *proper* explanation of those concepts, before reading that quote. Then it't exactly the right thing to say. Which is the whole joke behind it. (That people don't know those concepts.) – Evi1M4chine Mar 3 '13 at 16:29

4 Answers

That particularly phrasing is by James Iry, from his highly entertaining *Brief, Incomplete and Mostly Wrong History of Programming Languages*, in which he fictionally attributes it to Philip Wadler.

The original quote is from Saunders Mac Lane in *Categories for the Working Mathematician*, one of the foundational texts of Category Theory. Here it is in context, which is probably the best place to learn exactly what it means.

But, I'll take a stab. The original sentence is this:

> All told, a monad in X is just a monoid in the category of endofunctors of X, with product × replaced by composition of endofunctors and unit set by the identity endofunctor.

X here is a category. Endofunctors are functors from a category to itself (which is usually *all* `Functor`s as far as functional programmers are concerned, since they're mostly dealing with just one category; the category of types--but I digress). But you could imagine another category which is the category of "endofunctors on *X*". This is a category in which the objects are endofunctors and the morphisms are natural transformations.

And of those endofunctors, some of them might be monads. Which ones are monads? Just exactly the ones which are *monoidal* in a particular sense. Instead of spelling out the exact mapping from monads to monoids (since Mac Lane does that far better than I could hope to), I'll just put their respective definitions side by side and let you compare:

A monoid is...

- A set, $\boldsymbol{S}$
- An operation, $\boldsymbol{\bullet : S \times S \to S}$
- An element of S, $\boldsymbol{e : 1 \to S}$

...satisfying these laws:

- $(a \bullet b) \bullet c = a \bullet (b \bullet c)$, for all a, b and c in S
- $e \bullet a = a = a \bullet e$, for all a in S

A monad is...

- An endofunctor, $\boldsymbol{T : X \to X}$ (in Haskell, a type constructor of kind `* -> *` with a `Functor` instance)
- A natural transformation, $\boldsymbol{\mu : T \times T \to T}$, where × means functor composition (also known as `join` in Haskell)
- A natural transformation, $\boldsymbol{\eta : I \to T}$, where I is the identity endofunctor on X (also known as `return` in Haskell)

...satisfying these laws:

- $\mu(\mu(T \times T) \times T)) = \mu(T \times \mu(T \times T))$
- $\mu(\eta(T)) = T = \mu(T(\eta))$

With a bit of squinting you can probably see that both of these definitions are instances of the same abstract concept (I think category theorists would say "monoid" is the abstract term, and my definition of "monoid" above is overly specific since it mentions sets and elements).

edited Aug 31 '15 at 0:48 answered Oct 6 '10 at 7:35

Tom Crockett
17.8k ●3 ●47 ●63

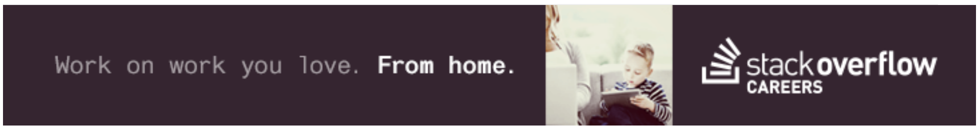
9 thanks for the explanation and thanks for the Brief, Incomplete and Mostly Wrong History of Programming Languages article. I thought it might be from there. Truly one of the greatest pieces of programming humor. – Roman A. Taycher Oct 6 '10 at 13:39

4 @Jonathan: In the classical formulation of a monoid, × means the cartesian product of sets. You can read more about that here: en.wikipedia.org/wiki/Cartesian_product, but the basic idea is that an element of $S \times T$ is a pair (s, t), where $s \in S$ and $t \in T$. So the signature of the monoidal product $\bullet : S \times S \to S$ in this context simply means a function that takes 2 elements of S as input and produces another element of S as an output. – Tom Crockett Oct 20 '10 at 8:19

8 I have to memorize this definition, to show off :p – Aivar Sep 14 '11 at 19:47

7 @TahirHassan - In the generality of category theory, we deal with opaque "objects" instead of sets, and so there is no a priori notion of "elements". But if you think about the category **Set** where the objects are sets and the arrows are functions, the elements of any set S are in one-to-one correspondence with the functions from any one-element set to S. That is, for any element **e** of **S**, there is exactly one function **f : 1 -> S**, where **1** is any one-element set... (cont'd) – Tom Crockett Nov 1 '12 at 23:22

8 @TahirHassan 1-element sets are themselves specializations of the more general category-theoretic notion of "terminal objects": a terminal object is any object of a category for which there is exactly one arrow from any other object to it (you can check that this is true of 1-element sets in **Set**). In category theory terminal objects are simply referred to as **1**; they are unique up to isomorphism so there is no point distinguishing them. So now we have a purely category-theoretical description of "elements of **S**" for any **S**: they are just the arrows from **1** to **S**! – Tom Crockett Nov 1 '12 at 23:26



Intuitively, I think that what the fancy math vocabulary is saying is that:

Monoid

A **monoid** is a set of objects, and a method of combining them. Well known monoids are:

- numbers you can add
- lists you can concatenate
- sets you can union

There are more complex examples also.

Further, *every* monoid has an **identity**, which is that "no-op" element that has no effect when you combine it with something else:

- 0 + 7 **==** 7 + 0 **==** 7
- [] ++ [1,2,3] **==** [1,2,3] ++ [] **==** [1,2,3]
- {} *union* {apple} **==** {apple} *union* {} **==** {apple}

Finally, a monoid must be **associative**. (you can reduce a long string of combinations anyway you want, as long as you don't change the left-to-right-order of objects) Addition is OK ((5+3)+1 == 5+(3+1)), but subtraction isn't ((5-3)-1 != 5-(3-1)).

Monad

Now, let's consider a special kind of set and a special way of combining objects.

Objects

Suppose your set contains objects of a special kind: **functions**. And these functions have an interesting signature: They don't carry numbers to numbers or strings to strings. Instead, each function carries a number to a list of numbers in a two-step process.

1. Compute 0 or more results
2. Combine those results unto a single answer somehow.

Examples:

- 1 -> [1] (just wrap the input)
- 1 -> [] (discard the input, wrap the nothingness in a list)
- 1 -> [2] (add 1 to the input, and wrap the result)
- 3 -> [4, 6] (add 1 to input, and multiply input by 2, and wrap the *multiple results*)

Combining Objects

Also, our way of combining functions is special. A simple way to combine function is *composition*: Let's take our examples above, and compose each function with itself:

- 1 -> [1] -> [[1]] (wrap the input, twice)
- 1 -> [] -> [] (discard the input, wrap the nothingness in a list, twice)
- 1 -> [2] -> [UH-OH!] (we can't "add 1" to a list!")
- 3 -> [4, 6] -> [UH-OH!] (we can't add 1 a list!)

Without getting too much into type theory, the point is that you can combine two integers to get an integer, but you can't always compose two functions and get a function of the same type. (Functions with type *a* -> *a* will compose, but *a*-> *[a]* won't.)

So, let's define a different way of combining functions. When we combine two of these functions, we don't want to "double-wrap" the results.

Here is what we do. When we want to combine two functions F and G, we follow this process (called *binding*):

1. Compute the "results" from F but don't combine them.
2. Compute the results from applying G to each of F's results separately, yielding a collection of collection of results.
3. Flatten the 2-level collection and combine all the results.

Back to our examples, let's combine (bind) a function with itself using this new way of "binding" functions:

- 1 -> [1] -> [1] (wrap the input, twice)
- 1 -> [] -> [] (discard the input, wrap the nothingness in a list, twice)
- 1 -> [2] -> [3] (add 1, then add 1 again, and wrap the result.)
- 3 -> [4,6] -> [5,8,7,12] (add 1 to input, and also multiply input by 2, keeping both results, then do it all again to both results, and then wrap the final results in a list.)

This more sophisticated way of combining functions *is* associative (following from how function composition is associative when you aren't doing the fancy wrapping stuff).

Tying it all together,

- a monad is a structure that defines a way to combine (the results of) functions,
- analogously to how a monoid is a structure that defines a way to combine objects,
- where the method of combination is associative,
- and where there is a special 'No-op' that can be combined with any *something* to result in *something* unchanged.

Notes

There are lots of ways to "wrap" results. You can make a list, or a set, or discard all but the first result while noting if there are no results, attach a sidecar of state, print a log message, etc, etc.

I've played a bit loose with the definitions in hopes of getting the essential idea across intuitively.

I've simplified things a bit by insisting that our monad operates on functions of type *a* -> *[a]*. In fact, monads work on functions of type *a* -> *m b*, but the generalization is kind of a technical detail that isn't the main insight.

edited Nov 30 '14 at 0:53 community wiki
9 revs, 3 users 91%

misterbee

19 Best explanation I've read. I finally think I'm starting to get this, after 3 years pottering with Haskell every few months. – chrisdew Oct 20 '11 at 8:46

9 This is a nice explanation of how every monad constitutes a *category* (the Kleisli category is what you're demonstrating--there is also the Eilenberg-Moore category). But due to the fact that you can't compose *any* two Kleisli arrows `a -> [b]` and `c -> [d]` (you can only do this if `b = c`), this doesn't quite describe a monoid. It's actually the flattening operation you described, rather than function composition, which is the "monoid operator". – Tom Crockett Dec 10 '11 at 19:35

4 I wish I could vote this up twice. – jwg Feb 6 '13 at 17:08

2 On the last note, it helps to remember, that a -> [a] is just a -> [] a. ([] is just type constructor too.) And so it can not only be seen as a -> m b, but [] is indeed an instance of the Monad class. – Evi1M4chine Mar 3 '13 at 17:34

2 This is the best and most grokkable explanation of monads and their mathematical background of monoids I have come across in literally weeks. This is what should be printed in every Haskell book when it comes to monads, hands down. UPVOTE! Maybe further get the piece of information, that monads are realized as parameterized typeclass instances wrapping whatever put in them in haskell, into the post. (At least that is how I understood them by now. Correct me if I am wrong. See haskell.org/haskellwiki/What_a_Monad_is_not) – sjas Dec 2 '13 at 19:20

This is an old question, but I feel there's a way to make the answer a bit more concrete with some code. At least, I'm better at Haskell than I am at category theory, so I find it easier to understand it this way :-P.

First, the extensions and libraries that we're going to use:

```
{-# LANGUAGE RankNTypes, TypeOperators #-}

import Control.Monad (join)
```

Of these, `RankNTypes` is the only one that's absolutely essential to the below. I once wrote an explanation of `RankNTypes` that some people seem to have found useful, so I'll refer to that.

Quoting Tom Crockett's excellent answer, we have:

> ## A monad is...
>
> - An endofunctor, ***T : X -> X***
> - A natural transformation, ***μ : T × T -> T***, where × means functor composition
> - A natural transformation, ***η : I -> T***, where *I* is the identity endofunctor on *X*
>
> ## ...satisfying these laws:
>
> - *μ(μ(T × T) × T)) = μ(T × μ(T × T))*
> - *μ(η(T)) = T = μ(T(η))*

How do we translate this to Haskell code? Well, let's start with the notion of a **natural transformation**:

```
-- | A natural transformations between two 'Functor' instances.  Law:
--
-- > fmap f . eta g == eta g . fmap f
--
-- Neat fact: the type system actually guarantees this law.
--
newtype f :-> g =
    Natural { eta :: forall x. f x -> g x }
```

A type of the form `f :-> g` is analogous to a function type, but instead of thinking of it as a *function* between two *types* (of kind `*`), think of it as a **morphism** between two **functors** (each of kind `* -> *`). Examples:

```
listToMaybe :: [] :-> Maybe
listToMaybe = Natural go
    where go [] = Nothing
          go (x:_) = Just x

maybeToList :: Maybe :-> []
maybeToList = Natural go
    where go Nothing = []
          go (Just x) = [x]

reverse' :: [] :-> []
reverse' = Natural reverse
```

Basically, in Haskell, natural transformations are functions from some type `f x` to another type `g x` such that the `x` type variable is "inaccessible" to the caller. So for example, `sort :: Ord a => [a] -> [a]` cannot be made into a natural transformation, because it's "picky" about which types we may instantiate for `a`. One intuitive way I often use to think of this is the following:

- A functor is a way of operating on the *content* of something without touching the *structure*.
- A natural transformation is a way of operating on the *structure* of something without touching or looking at the *content*.

Now, with that out of the way, let's tackle the clauses of the definition.

The first clause is "an endofunctor, ***T : X -> X***." Well, every `Functor` in Haskell is an endofunctor in what people call "the Hask category," whose objects are Haskell types (of kind `*`) and whose morphisms are Haskell functions. This sounds like a complicated statement, but it's actually a very trivial one. All it means is that that a `Functor f :: * -> *` gives you the means of constructing a type `f a :: *` for any `a :: *` and a function `fmap f :: f a -> f b` out of any `f :: a -> b`, and that these obey the functor laws.

Second clause: the `Identity` functor in Haskell (which comes with the Platform, so you can just import it) is defined this way:

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

instance Functor Identity where
    fmap f (Identity a) = Identity (f a)
```

So natural transformation ***η : I -> T*** from Tom Crockett's definition can be written this way for any `Monad` instance `t`:

```
return' :: Monad t => Identity :-> t
return' = Natural (return . runIdentity)
```

Third clause: the composition of two functors in Haskell can be defined this way (which also comes with the Platform):

```
newtype Compose f g a = Compose { getCompose :: f (g a) }

-- | The composition of two 'Functor's is also a 'Functor'.
instance (Functor f, Functor g) => Functor (Compose f g) where
    fmap f (Compose fga) = Compose (fmap (fmap f) fga)
```

So the natural transformation ***μ : T × T -> T*** from Tom Crockett's definition can be written like this:

```
join' :: Monad t => Compose t t :-> t
join' = Natural (join . getCompose)
```

The statement that this is a monoid in the category of endofunctors then means that `Compose` (partially applied to just its first two parameters) is associative, and that `Identity` is its identity element. I.e., that the following isomorphisms hold:

- `Compose f (Compose g h) ~= Compose (Compose f g) h`
- `Compose f Identity ~= f`
- `Compose Identity g ~= g`

These are very easy to prove because `Compose` and `Identity` are both defined as `newtype`, and the Haskell Reports define the semantics of `newtype` as an isomorphism between the type being defined and the type of the argument to the `newtype`'s data constructor. So for example, let's prove `Compose f Identity ~= f`:

```
Compose f Identity a
    ~= f (Identity a)                -- newtype Compose f g a = Compose (f (g a))
    ~= f a                           -- newtype Identity a = Identity a
Q.E.D.
```

edited Mar 21 '15 at 7:51
Lambda Fairy
4,444 ●2 ●20 ●40

answered May 2 '14 at 0:07
Luis Casillas
19.1k ●2 ●29 ●75

Your explanation is very clear and wonderful – Song Zhang Feb 1 '15 at 3:43

In the `Natural` newtype, I can't figure out what the `(Functor f, Functor g)` constraint is doing. Could you explain? – dfeuer Mar 20 '15 at 15:53

@dfeuer It's not really doing anything essential. – Luis Casillas Mar 20 '15 at 18:15

1 @LuisCasillas I've removed those `Functor` constraints since they don't seem necessary. If you disagree then feel free to add them back. – Lambda Fairy Mar 21 '15 at 8:03

Can you elaborate on what it means formally for the product of functors to be taken as composition? In particular, what are the projection morphisms for functor composition? My guess is that the product is only defined for a functor F against itself, F x F and only when `join` is defined. And that `join` is the projection morphism. But I'm not sure. – tksfz Apr 1 '15 at 21:54

It's quite possible that Iry had read From Monoids to Monads, a post in which Dan Piponi (sigfpe) derives monads from monoids in Haskell, with much discussion of category theory and explicit mention of "the category of endofunctors on Hask" . In any case, anyone who wonders what it means for a monad to be a monoid in the category of endofunctors might benefit from reading this derivation.

answered Sep 16 '15 at 6:58
hobbs
98.5k ●10 ●109 ●189

It's the other way round. I wrote that because I felt the need to explain Iry's comment. – sigfpe Nov 30 '15 at 22:15

1 @sigfpe darn. Well, thanks for dropping by to clear things up :) – hobbs Nov 30 '15 at 22:20

StackExchange sign up log in tour help

Stack Overflow is a community of 4.7 million programmers, just like you, helping each other.

Join them; it only takes a minute:

Sign up

Join the Stack Overflow community to:

Ask programming questions

Answer and help your peers

Get recognized for your expertise

How to extract value from monadic action

$ git push origin stackoverflowcareers Add repos

Is there a built-in function with signature `:: (Monad m) => m a -> a` ?

Hoogle tells that there is no such function.

Can you explain why?

haskell monads comonad

edited Oct 18 '14 at 14:01
Zoidberg
17.5k ●4 ●34 ●69

asked Dec 19 '11 at 21:22
ДМИТРИЙ МАЛИКОВ
10.8k ●5 ●35 ●89

2 Related: taking out a value out of a monad? haskell – Jan Dec 19 '11 at 21:26

13 There isn't, but there is a function that turns functions expecting an `a` into functions expecting an `m a` : `(=<<) :: Monad m => (a -> m b) -> (m a -> m b)` . Invert your expectations, and you will be fine. =) – Daniel Wagner Dec 19 '11 at 21:33

2 In the same vein as what Daniel Wagner said, `liftM :: Monad m => (a -> b) -> (m a -> m b)` allows a "regular" function to accept a monadic value as input, but in exchange it must output a monadic value rather than a "regular" value. – Dan Burton Dec 19 '11 at 22:39

7 Answers

A monad only supplies two functions:

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

Both of these return something of type `m a` , so there is no way to combine these in any way to get a function of type `Monad m => m a -> a` . To do that, you'll need more than these two functions, so you need to know more about `m` than that it's a monad.

For example, the `Identity` monad has `runIdentity :: Identity a -> a` , and several monads have similar functions, but there is no way to provide it generically. In fact, the inability to "escape" from the monad is essential for monads like `IO` .

answered Dec 19 '11 at 21:31
hammar
107k ●8 ●219 ●325

There is probably a better answer than this, but one way to see why you cannot have a type `(Monad m) => m a -> a` is to consider a null monad:

```
data Null a = Null

instance Monad Null where
    return a = Null
    ma >>= f = Null
```

Now `(Monad m) => m a -> a` means `Null a -> a` , ie getting something out of nothing. You can't do that.

answered Dec 19 '11 at 21:29
Owen
19.3k ●4 ●51 ●89

4 On the other hand this fact doesn't prevent `fromJust` from existence. It returns contents of a `Just` and raises an exception in case of `Nothing`. Your `Null` monad could simply always raise an exception on calling such imaginary monad unwrapping function. – Jan Dec 19 '11 at 21:32

11 @Jan: On the other other hand that doesn't make either function a good idea. `fromJust` is terrible and would be better not existing. – C. A. McCann Dec 19 '11 at 21:51

1 @C.A.McCann I +1 your comment partly because `fromJust` is bad... but mainly because you used "other other hand". – Adam Wagner Dec 19 '11 at 23:54

Agee with C.A. McCann; the fact that the proposed method for the `Monad` class would have to be implemented as raising an exception for many classes is evidence enough that it should not be part of the class. If we actually had some examples of "monad that we can extract from" that we wanted to cope, then we could create a subclass of Monad to tackle them and put the operation there. – Luis Casillas Dec 20 '11 at 0:26

`fromJust` has its uses, just not in robust code. I use it when there's no point in continuing on failure. For example, when the UI definition file is missing for an mplayer frontend. – György Andrasek Dec 20 '11 at 3:59

This doesn't exist because `Monad` is a pattern for composition, not a pattern for decomposition. You can always put more pieces together with the interface it defines. It doesn't say a thing about taking anything apart.

Asking why you can't take something out is like asking why Java's `Iterator` interface doesn't contain a method for adding elements to what it's iterating over. It's just not what the `Iterator` interface is for.

And your arguments about specific types having a kind of extract function follows in the exact same way. Some particular implementation of `Iterator` might have an `add` function. But since it's not what `Iterator` s are for, the presence that method on some particular instance is irrelevant.

And the presence of `fromJust` is just as irrelevant. It's not part of the behavior `Monad` is intended to describe. Others have given lots of examples of types where there is no value for `extract` to work on. But those types still support the intended semantics of `Monad`. This is important. It means that `Monad` is a more general interface than you are giving it credit for.

answered Dec 19 '11 at 21:56
Carl
15.2k ●2 ●38 ●57

> Is there a build-in function with signature `:: (Monad m) => m a -> a` ?

If Hoogle says there isn't...then there probably isn't, assuming your definition of "built in" is "in the base libraries".

> Hoogle tells that there is no such function. Can you explain why?

That's easy, because Hoogle didn't find any function in the base libraries that matches that type signature!

More seriously, I suppose you were asking for the monadic explanation. The issues are *safety* and *meaning*. (See also my previous thoughts on `magicMonadUnwrap :: Monad m  => m a -> a`)

Suppose I tell you I have a value which has the type `[Int]`. Since we know that `[]` is a monad, this is similar to telling you I have a value which has the type `Monad m => m Int`. So let's suppose you want to get the `Int` out of that `[Int]`. Well, which `Int` do you want? The first one? The last one? What if the value I told you about is actually an empty list? In that case, there isn't even an `Int` to give you! So for lists, it is *unsafe* to try and extract a single value willy-nilly like that. Even when it is safe (a non-empty list), you need a list-specific function (for example, `head`) to clarify what you *mean* by desiring `f :: [Int] -> Int`. Hopefully you can intuit from here that the *meaning* of `Monad m => m a -> a` is simply not well defined. It could hold multiple meanings for the same monad, or it could mean absolutely nothing at all for some monads, and sometimes, it's just simply not safe.

answered Dec 19 '11 at 23:02
Dan Burton
31.3k ●13 ●85 ●159

1 I don't see how `Monad m => m a -> a` lacks meaning in any way that wouldn't also apply to `>>=` or `return` or `fail`. You could always say that the "meaning" of the operation isn't well-defined in advance of knowing the full implementation that `m` provides for its inclusion in the Monad type class. For your list example, a function with type `Monad m => m a -> a` could very well mean any of the things you suggest - - and *any* of them *might* be valid. You could always use `newtype` to tweak the behavior for *your* application, but denying even the chance to do it seems too severe. – Mr. F Dec 21 '14 at 0:14

Suppose there was such a function:

```
extract :: Monad m => m a -> a
```

Now you could write a "function" like this:

```
appendLine :: String -> String
appendLine str = str ++ extract getLine
```

Unless the `extract` function was guaranteed never to terminate, this would violate referential transparency, because the result of `appendLine "foo"` would (a) depend on something other than `"foo"`, (b) evaluate to different values when evaluated in different contexts.

Or in simpler words, if there was an actually useful `extract` operation Haskell would not be purely functional.

answered Dec 20 '11 at 0:19
Luis Casillas
19.1k ●2 ●29 ●75

1 unsafePerformIO does just this. – Mr. F Dec 21 '14 at 0:28

1 Don't use that! Boooooooo!!!!!! – Luis Casillas Dec 22 '14 at 21:33

Because it may make no sense (actually, *does* make no sense in many instances).

For example, I might define a Parser Monad like this:

```
data Parser a = Parser (String ->[(a, String)])
```

Now there is absolutely no sensible default way to get a `String` out of a `Parser String`. Actually, there is no way at all to get a String out of this with just the Monad.

answered Dec 19 '11 at 21:27
Cubic
7,231 ●2 ●18 ●52

Well, technicaly there is unsafePerformIO for the IO monad.

But, as the name itself suggests, this function is evil and you should only use it if you *really* know what you are doing (and if you have to ask wether you know or not then you don't)

answered Dec 20 '11 at 20:53
hugomg
38.2k ●9 ●71 ●148

6 You shouldn't tell innocent people how to do evil stuff! – is7s Dec 20 '11 at 21:17

1 There is also `unsafeHead` for the List monad...(oh wait, it's just called `head`...but it is similarly, though not quite as drastically, unsafe.) – Dan Burton Dec 21 '11 at 5:23

-1 It doesn't answer the OP question. – mb14 Oct 18 '14 at 16:18

StackExchange sign up log in tour help

Stack Overflow is a community of 4.7 million programmers, just like you, helping each other.

Join them; it only takes a minute:

Sign up

Join the Stack Overflow community to:

Ask programming questions

Answer and help your peers

Get recognized for your expertise

Haskell: How is <*> pronounced?

Octopus Deploy
Build servers build. Octopus deploys.
What's the difference?
FREE FOR SMALL TEAMS

Sorry, I don't really know my math, so I'm curious how to pronounce the functions in the Applicative typeclass:

```
(<*>) :: f (a -> b) -> f a -> f b
(*>)  :: f a -> f b -> f b
(<*)  :: f a -> f b -> f a
```

(That is, if they weren't operators, what might they be called?)

As a side note, if you could rename `pure` to something more friendly to podunks like me, what would you call it?

haskell operators pronunciation

asked Jul 13 '10 at 23:33
J Cooper
8,641 ●4 ●46 ●81

5 @J Cooper... would you be able hear how we pronounce it? :) You might want to post a request on meta.stackoverflow.com for a voice recording and playback feature :). – Lirik Jul 13 '10 at 23:39

7 It's pronounced "Good grief, they were really running out of operators, weren't they?" Also, a good name for `pure` might be `makeApplicative`. – Chuck Jul 13 '10 at 23:44

2 @Lirik foh-net-iks – Tyler Jul 13 '10 at 23:45

4 (<*>) is the Control.Applicative version of Control.Monad's "ap", so "ap" is probably the most appropriate name. – Edward KMETT Jul 15 '10 at 5:20

8 i'd call it a cyclops, but that's just me. – RCIX Aug 6 '10 at 22:40

3 Answers

> Sorry, I don't really know my math, so I'm curious how to pronounce the functions in the Applicative typeclass

Knowing your math, or not, is largely irrelevant here, I think. As you're probably aware, Haskell borrows a few bits of terminology from various fields of abstract math, most notably Category Theory, from whence we get functors and monads. The use of these terms in Haskell diverges somewhat from the formal mathematical definitions, but they're usually close enough to be good descriptive terms anyway.

The `Applicative` type class sits somewhere between `Functor` and `Monad`, so one would expect it to have a similar mathematical basis. The documentation for the `Control.Applicative` module begins with:

> This module describes a structure intermediate between a functor and a monad: it provides pure expressions and sequencing, but no binding. (Technically, a strong lax monoidal functor.)

Hmm.

```
class (Functor f) => StrongLaxMonoidalFunctor f where
    . . .
```

Not quite as catchy as `Monad`, I think.

What all this basically boils down to is that `Applicative` doesn't correspond to any concept that's particularly *interesting* mathematically, so there's no ready-made terms lying around that capture the way it's used in Haskell. So, set the math aside for now.

If we want to know what to call `(<*>)` it might help to know what it basically means.

So what's up with `Applicative`, anyway, and why *do* we call it that?

What `Applicative` amounts to in practice is a way to lift *arbitrary* functions into a `Functor`. Consider the combination of `Maybe` (arguably the simplest non-trivial `Functor`) and `Bool` (likewise the simplest non-trivial data type).

```
maybeNot :: Maybe Bool -> Maybe Bool
maybeNot = fmap not
```

The function `fmap` lets us lift `not` from working on `Bool` to working on `Maybe Bool`. But what if we want to lift `(&&)`?

```
maybeAnd' :: Maybe Bool -> Maybe (Bool -> Bool)
maybeAnd' = fmap (&&)
```

Well, that's not what we want *at all*! In fact, it's pretty much useless. We can try to be clever and sneak another `Bool` into `Maybe` through the back...

```
maybeAnd'' :: Maybe Bool -> Bool -> Maybe Bool
maybeAnd'' x y = fmap ($ y) (fmap (&&) x)
```

...but that's no good. For one thing, it's wrong. For another thing, it's *ugly*. We could keep trying, but it turns out that there's *no way to lift a function of multiple arguments to work on an arbitrary `Functor`*. Annoying!

On the other hand, we could do it easily if we used `Maybe`'s `Monad` instance:

```
maybeAnd :: Maybe Bool -> Maybe Bool -> Maybe Bool
maybeAnd x y = do x' <- x
                  y' <- y
                  return (x' && y')
```

Now, that's a lot of hassle just to translate a simple function--which is why `Control.Monad` provides a function to do it automatically, `liftM2`. The 2 in its name refers to the fact that it works on functions of exactly two arguments; similar functions exist for 3, 4, and 5 argument functions. These functions are *better*, but not perfect, and specifying the number of arguments is ugly and clumsy.

Which brings us to the paper that introduced the Applicative type class. In it, the authors make essentially two observations:

- Lifting multi-argument functions into a `Functor` is a very natural thing to do
- Doing so doesn't require the full capabilities of a `Monad`

Normal function application is written by simple juxtaposition of terms, so to make "lifted application" as simple and natural as possible, the paper introduces *infix operators to stand in for application, lifted into the `Functor`*, and a type class to provide what's needed for that.

All of which brings us to the following point: `(<*>)` **simply represents function application--so why pronounce it any differently than you do the whitespace "juxtaposition operator"?**

But if that's not very satisfying, we can observe that the `Control.Monad` module also provides a function that does the same thing for monads:

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

Where `ap` is, of course, short for "apply". Since any `Monad` can be `Applicative`, and `ap` needs only the subset of features present in the latter, we can perhaps say that **if `(<*>)` weren't an operator, it should be called `ap`.**

We can also approach things from the other direction. The `Functor` lifting operation is called `fmap` because it's a generalization of the `map` operation on lists. What sort of function on lists would work like `(<*>)`? There's what `ap` does on lists, of course, but that's not particularly useful on its own.

In fact, there's a perhaps more natural interpretation for lists. What comes to mind when you look at the following type signature?

```
listApply :: [a -> b] -> [a] -> [b]
```

There's something just so tempting about the idea of lining the lists up in parallel, applying each function in the first to the corresponding element of the second. Unfortunately for our old friend `Monad`, this simple operation *violates the monad laws* if the lists are of different lengths. But it makes a fine `Applicative`, in which case `(<*>)` becomes a way of **stringing together a generalized version of `zipWith`, so perhaps we can imagine calling it `fzipWith`?**

This zipping idea actually brings us full circle. Recall that math stuff earlier, about monoidal functors? As the name suggests, these are a way of combining the structure of monoids and functors, both of which are familiar Haskell type classes:

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

class Monoid a where
    mempty :: a
    mappend :: a -> a -> a
```

What would these look like if you put them in a box together and shook it up a bit? From `Functor` we'll keep the idea of a *structure independent of its type parameter*, and from `Monoid` we'll keep the overall form of the functions:

```haskell
class (Functor f) => MonoidalFunctor f where
    mfEmpty :: f ?
    mfAppend :: f ? -> f ? -> f ?
```

We don't want to assume that there's a way to create an truly "empty" `Functor`, and we can't conjure up a value of an arbitrary type, so we'll fix the type of `mfEmpty` as `f ()`.

We also don't want to force `mfAppend` to need a consistent type parameter, so now we have this:

```haskell
class (Functor f) => MonoidalFunctor f where
    mfEmpty :: f ()
    mfAppend :: f a -> f b -> f ?
```

What's the result type for `mfAppend`? We have two arbitrary types we know nothing about, so we don't have many options. The most sensible thing is to just keep both:

```haskell
class (Functor f) => MonoidalFunctor f where
    mfEmpty :: f ()
    mfAppend :: f a -> f b -> f (a, b)
```

At which point `mfAppend` is now clearly a generalized version of `zip` on lists, and we can reconstruct `Applicative` easily:

```haskell
mfPure x = fmap (\() -> x) mfEmpty
mfApply f x = fmap (\(f, x) -> f x) (mfAppend f x)
```

This also shows us that `pure` is related to the identity element of a `Monoid`, so other good names for it might be anything suggesting a unit value, a null operation, or such.

That was lengthy, so to summarize:

- **`(<*>)` is just a modified function application, so you can either read it as "ap" or "apply", or elide it entirely the way you would normal function application.**
- **`(<*>)` also roughly generalizes `zipWith` on lists, so you can read it as "zip functors with", similarly to reading `fmap` as "map a functor with".**

The first is closer to the intent of the `Applicative` type class--as the name suggests--so that's what I recommend.

In fact, I encourage **liberal use, and non-pronunciation, of all lifted application operators**:

- `(<$>)`, which lifts a single-argument function into a `Functor`
- `(<*>)`, which chains a multi-argument function through an `Applicative`
- `(=<<)`, which binds a function that enters a `Monad` onto an existing computation

All three are, at heart, just regular function application, spiced up a little bit.

answered Jul 14 '10 at 1:55
C. A. McCann
63.7k ●15 ●163 ●277

14 This is a fantastic answer. Extremely informative and very well-written. – Colin Cochrane Jul 14 '10 at 2:09

5 @Colin Cochrane: Are you sure you didn't misspell "long-winded" there? :) But hey, I'll take it! I always feel that `Applicative` and the functional idiomatic style it promotes don't get enough love, so I couldn't resist the chance to extol its virtues a bit as a means to explain how I (don't) pronounce `(<*>)`. – C. A. McCann Jul 14 '10 at 2:16

4 Would that Haskell had syntax sugar for `Applicative`'s! Something like `[| f a b c d |]` (as suggested by the original paper). Then we wouldn't need the `<*>` combinator and you would refer to such an expression as an example of "function application in a functorial context" – Tom Crockett Jan 6 '11 at 0:09

1 @FredOverflow: No, I meant `Monad`. Or `Functor` or `Monoid` or anything else that has a well-established term involving fewer than three adjectives. "Applicative" is merely an uninspiring, albeit reasonably descriptive, name slapped onto something that rather needed one. – C. A. McCann Sep 23 '11 at 18:30

1 @pelotom: see [stackoverflow.com/questions/12014524/… where kind people showed me two ways to get almost that notation. – AndrewC Aug 22 '12 at 18:04

Since I have no ambitions of improving on C. A. McCann's technical answer, I'll tackle the more fluffy one:

> If you could rename `pure` to something more friendly to podunks like me, what would you call it?

As an alternative, especially since there is no end to the constant angst-and-betrayal-filled cried against the `Monad` version, called " `return` ", I propose another name, which suggests its function in a way that can satisfy the most imperative of imperative programmers, and the most functional of...well, hopefully, everyone can complain the same about: `inject` .

Take a value. "Inject" it into the `Functor` , `Applicative` , `Monad` , or what-have-you. I vote for " `inject` ", and I approved this message.

edited Sep 3 '14 at 13:20
TRiG
4,866 ●2 ●30 ●68

answered Jul 14 '10 at 22:15
BMeph
1,047 ●8 ●12

3 I usually lean toward something like "unit" or "lift", but those already have too many other meanings in Haskell. `inject` is an excellent name and probably better than mine, though as a minor side note, "inject" is used in--I think--Smalltalk and Ruby for a left-fold method of some sort. I never understood that choice of name, though... – C. A. McCann Jul 14 '10 at 23:45

3 This is a very old thread, but I think that `inject` in Ruby & Smalltalk is used because it is like you are "injecting" an operator between each element in the list. At least, that's how I always thought of it. – Jonathan Sterling Jun 7 '12 at 18:48

1 To *again* pick up that old side-thread: You're not injecting operators, you're replacing (eliminating) constructors that are already there. (Viewed the other way round, you're *injecting* old data into a new type.) For lists, elimination is just `foldr` . (You replace `(:)` and `[]` , where `(:)` takes 2 args and `[]` is a constant, hence `foldr (+) 0 (1:2:3:[])` ↝ `1+2+3+0` .) On `Bool` it's just `if` - `then` - `else` (two constants, pick one) and for `Maybe` it's called `maybe` … Haskell has no single name/function for this, as all have different types (in general elim is just recursion/induction) – nobody Mar 16 '13 at 3:56

I always liked `wrap` . Take a value and wrap it in a Functor, Applicative, Monad. It also works well when used in a sentence with concrete instances: `[]` , `Maybe` , etc. "It takes a value and wraps it in a `X` ".

answered Apr 18 '15 at 15:48
Peter Hall
4,162 ●1 ●17 ●50

StackExchange sign up log in tour help

Stack Overflow is a community of 4.7 million programmers, just like you, helping each other.

Join them; it only takes a minute:

Sign up

Join the Stack Overflow community to:

Ask programming questions

Answer and help your peers

Get recognized for your expertise

Distinction between typeclasses MonadPlus, Alternative, and Monoid?

The standard-library Haskell typeclasses `MonadPlus`, `Alternative`, and `Monoid` each provide two methods with essentially the same semantics:

- An empty value: `mzero`, `empty`, or `mempty`.
- An operator `a -> a -> a` that joins values in the typeclass together: `mplus`, `<|>`, or `mappend`.

All three specify these laws to which instances should adhere:

```
mempty `mappend` x = x
x `mappend` mempty = x
```

Thus, it seems the three typeclasses are all providing the *same* methods.

(`Alternative` also provides `some` and `many`, but their default definitions are usually sufficient, and so they're not too important in terms of this question.)

So, my query is: why have these three extremely similar classes? Is there any real difference between them, besides their differing superclass constraints?

haskell functional-programming typeclass applicative monoids

edited Apr 16 '12 at 2:06
Xenon
2,415 ●9 ●30

asked Apr 16 '12 at 1:53
00Davo
622 ●8 ●13

That's a good question. In particular, `Applicative` and `MonadPlus` seem to be *exactly* the same (modulo superclass constraints). – Peter Apr 16 '12 at 2:22

1 There's also `ArrowZero` and `ArrowPlus` for arrows. My bet: to make type signatures cleaner (which makes differing superclass constraints *the* real difference). – Cat Plus Plus Apr 16 '12 at 2:28

1 @CatPlusPlus: well, `ArrowZero` and `ArrowPlus` have kind `* -> * -> *`, which means you can pass them in for the arrow type once for a function that needs to use them for a multitude of types, to use a `Monoid` you'd have to require an instance of `Monoid` for each particular instantiation, and you'd have no guarantee they were handled in a similar way, the instances could be unrelated! – Edward KMETT Apr 16 '12 at 2:52

1 Answer

`MonadPlus` and `Monoid` serve different purposes.

A `Monoid` is parameterized over a type of kind `*`.

```
class Monoid m where
    mempty :: m
    mappend :: m -> m -> m
```

and so it can be instantiated for almost any type for which there is an obvious operator that is associative and which has a unit.

However, `MonadPlus` not only specifies that you have a monoidal structure, but also that that structure is related to how the `Monad` works, *and* that that structure doesn't care about the value contained in the monad, this is (in part) indicated by the fact that `MonadPlus` takes an argument of kind `* -> *`.

```
class Monad m => MonadPlus m where
    mzero :: m a
    mplus :: m a -> m a -> m a
```

In addition to the monoid laws, we have two potential sets of laws we can apply to `MonadPlus`.

Sadly, the community disagrees as to what they should be.

At the least we know

```
mzero >>= k = mzero
```

but there are two other competing extensions, the left (sic) distribution law

```
mplus a b >>= k = mplus (a >>= k) (b >>= k)
```

and the left catch law

```
mplus (return a) b = return a
```

So any instance of `MonadPlus` should satisfy one or both of these additional laws.

So what about `Alternative`?

`Applicative` was defined after `Monad`, and logically belongs as a superclass of `Monad`, but largely due to the different pressures on the designers back in Haskell 98, even `Functor` wasn't a superclass of `Monad` until 2015. Now we finally have `Applicative` as a superclass of `Monad` in GHC (if not yet in a language standard.)

Effectively, `Alternative` is to `Applicative` what `MonadPlus` is to `Monad`.

For these we'd get

```
empty <*> m = empty
```

analogously to what we have with `MonadPlus` and there exist similar distributive and catch properties, at least one of which you should satisfy.

Unfortunately, even `empty <*> m = empty` law is too strong a claim. It doesn't hold for Backwards, for instance!

When we look at MonadPlus, the empty >>= f = empty law is nearly forced on us. The empty construction can't have any 'a's in it to call the function `f` with anyways.

However, since `Applicative` is *not* a superclass of `Monad` and `Alternative` is *not* a superclass of `MonadPlus`, we wind up defining both instances separately.

Moreover, even if `Applicative` was a superclass of `Monad`, you'd wind up needing the `MonadPlus` class anyways, because even if we did obey

```
empty <*> m = empty
```

that isn't strictly enough to prove that

```
empty >>= f = empty
```

So claiming that something is a `MonadPlus` is stronger than claiming it is `Alternative`.

Now, by convention, the `MonadPlus` and `Alternative` for a given type should agree, but the `Monoid` may be *completely* different.

For instance the `MonadPlus` and `Alternative` for `Maybe` do the obvious thing:

```
instance MonadPlus Maybe where
    mzero = Nothing
    mplus (Just a) _  = Just a
    mplus _        mb = mb
```

but the `Monoid` instance lifts a semigroup into a `Monoid`. Sadly because there did not exist a `Semigroup` class at the time in Haskell 98, it does so by requring a `Monoid`, but not using its unit. ಠ_ಠ

```
instance Monoid a => Monoid (Maybe a) where
    mempty = Nothing
    mappend (Just a) (Just b) = Just (mappend a b)
    mappend Nothing x = x
    mappend x Nothing = x
    mappend Nothing Nothing = Nothing
```

TL;DR `MonadPlus` is a stronger claim than `Alternative`, which in turn is a stronger claim than `Monoid`, and while the `MonadPlus` and `Alternative` instances for a type should be related, the `Monoid` may be (and sometimes is) something completely different.

edited Jul 3 '15 at 7:07 answered Apr 16 '12 at 2:36
Edward KMETT
22.5k ●2 ●65 ●94

2 Excellent answer, however the last definition seems to be wrong, it doesn't satisfy `` mempty `mappend` x ≡ x ``. – Vitus Apr 16 '12 at 6:44

2 Great answer. Does anyone know of a (commonly used) type that has *different* `MonadPlus` and `Alternative` implementations? – Peter Apr 16 '12 at 11:05

6 @EdwardKmett: This answer seems to imply that there could be a `Monad` which is an `Alternative` but not a `MonadPlus`. I asked a question about finding a specific example of this; if you know of one, I'd love

to see it. – Antal Spector-Zabusky Oct 29 '12 at 13:36

2 Can you explain the left catch law for monadplus? It is apparently violated by []; should [] really ignore its second argument if its first is non-empty? – ben w Feb 13 '13 at 2:18

4 @benw left distribution is arguably the more sensible law, but it doesn't hold for some instances. left catch is an alternate law that those other instances tend to support, but which aren't supported by most of the others. Consequently, we really have 2 largely unrelated sets of laws being implemented by different instances, so `MonadPlus` is really two classes disguised as one because most people don't care. – Edward KMETT Feb 18 '13 at 22:01

176

StackExchange sign up log in tour help

Stack Overflow is a community of 4.7 million programmers, just like you, helping each other.

Join them; it only takes a minute:

Sign up

Join the Stack Overflow community to:

Ask programming questions

Answer and help your peers

Get recognized for your expertise

Functions from 'Alternative' type class [duplicate]

USE STACK OVERFLOW TO FIND THE BEST DEVELOPERS

stackoverflow CAREERS

Possible Duplicate:
Haskell: some and many
Haskell - What is Control.Applicative.Alternative good for?

What are the functions `some` and `many` in the `Alternative` type class useful for? Docs provide a recursive definition which I was unable to comprehend.

haskell functional-programming typeclass

asked Oct 6 '11 at 6:45
missingfaktor
53.2k 26 191 308

marked as duplicate by Landei, FUZxxl, hammar, Donal Fellows, YOU Oct 8 '11 at 3:25

This question has been asked before and already has an answer. If those answers do not fully address your question, please ask a new question.

@Landei: I read the answer in that thread, and I still don't get it. – missingfaktor Oct 6 '11 at 11:04

I just said this question is a duplicate, not that the original one had a good answer :-) Although it was good enough for me: I figured out that these functions are very likely not interesting for me... – Landei Oct 6 '11 at 11:32

@Landei: I am reaching about the same conclusion as you did. :-) – missingfaktor Oct 6 '11 at 11:43

If you are going to close this question, please merge it with @Landei's. Don't delete it. – missingfaktor Oct 7 '11 at 6:23

2 Answers

`some` and `many` can be defined as:

```
some f = (:) <$> f <*> many f
many f = some f <|> pure []
```

Perhaps it helps to see how `some` would be written with monadic `do` syntax:

```
some f = do
  x <- f
  xs <- many f
  return (x:xs)
```

So `some f` runs `f` once, then "many" times, and conses the results. `many f` runs `f` "some" times, or "alternatively" just returns the empty list. The idea is that they both run `f` as often as possible until it "fails", collecting the results in a list. The difference is that `some f` fails if `f` fails immediately, while `many f` will succeed and "return" the empty list. But what this all means exactly depends on how `<|>` is defined.

Is it only useful for parsing? Let's see what it does for the instances in base: `Maybe`, `[]` and `STM`.

First `Maybe`. `Nothing` means failure, so `some Nothing` fails as well and evaluates to `Nothing` while `many Nothing` succeeds and evaluates to `Just []`. Both `some (Just ())` and `many (Just ())` never return, because `Just ()` never fails! In a sense they evaluate to `Just (repeat ())`.

For lists, `[]` means failure, so `some []` evaluates to `[]` (no answers) while `many []` evaluates to `[[]]` (there's one answer and it is the empty list). Again `some [()]` and `many [()]` don't return. Expanding the instances, `some [()]` means `fmap (():) (many [()])` and `many [()]` means `some [()] ++ [[]]`, so you could say that `many [()]` is the same as `tails (repeat ())`.

For `STM`, failure means that the transaction has to be retried. So `some retry` will retry itself, while `many retry` will simply return the empty list. `some f` and `many f` will run `f` repeatedly until it retries. I'm not sure if this is useful thing, but I'm guessing it isn't.

So, for `Maybe`, `[]` and `STM` `many` and `some` don't seem to be that useful. It is only useful if the applicative has some kind of state that makes failure increasingly likely when running the same thing over and over. For parsers this is the input which is shrinking with every successful match.

edited Oct 6 '11 at 22:47

answered Oct 6 '11 at 22:39
Sjoerd Visscher
9,524 ●1 ●32 ●55

E.g. for parsing (see the "Applicative parsing by example" section).

answered Oct 6 '11 at 7:12
Alexey Romanov
57.5k ●13 ●136 ●268

1 I am not familiar with Parsec. I'd appreciate some explanation. – missingfaktor Oct 6 '11 at 7:41

2 As far as I understand, if you have a parser `p` for X, then `some p` is a parser for 0 or more X and `many p` is a parser for 1 or more X. – Ingo Oct 6 '11 at 9:08

2 @missingfaktor `some` and `many` are implementaed in terms of `<|>`. This combinator is useful also in other ways. Consider `Either`: `Just 0 <|> Just 1 = Just 0`, `Nothing <|> Just 2 = Just 2`, `Just 3 <|> Nothing = Just 3`, `Nothing <|> Nothing = Nothing` – FUZxxl Oct 6 '11 at 10:17

1 @missingfaktor: that is the usual application; I'm not sure if `Alternative` is used for anything else. You could say that "in general", `some` is used whenever you want something to run multiple times (but doesn't **have** to run), and `many` to run at least once. – ivanm Oct 6 '11 at 11:24

2 @Ingo @ivanm Note that you have `some` and `many` backwards. `some` is one or more (i.e. `+` in regexps) and `many` is zero or more (i.e. `*`). – Sjoerd Visscher Oct 6 '11 at 20:24

<div id="noscript-padding"></div>

StackExchange 3 • 3 help

Confused by the meaning of the 'Alternative' type class and its relationship to other type classes

I've been going through the Typeclassopedia to learn the type classes. I'm stuck understanding `Alternative` (and `MonadPlus`, for that matter).

The problems I'm having:

> the 'pedia says that "the Alternative type class is for Applicative functors which also have a monoid structure." I don't get this -- doesn't Alternative mean something totally different from Monoid? i.e. I understood the point of the Alternative type class as picking between two things, whereas I understood Monoids as being about combining things.
>
> why does Alternative need an `empty` method/member? I may be wrong, but it seems to not be used at all ... at least in the code I could find. And it seems not to fit with the theme of the class -- if I have two things, and need to pick one, what do I need an 'empty' for?
>
> why does the Alternative type class need an Applicative constraint, and why does it need a kind of `* -> *`? Why not just have `<|> :: a -> a -> a`? All of the instances could still be implemented in the same way ... I think (not sure). What value does it provide that Monoid doesn't?
>
> what's the point of the `MonadPlus` type class? Can't I unlock all of its goodness by just using something as both a `Monad` and `Alternative`? Why not just ditch it? (I'm sure I'm wrong, but I don't have any counterexamples)

Hopefully all those questions are coherent ... !

Bounty update: @Antal's answer is a great start, but Q3 is still open: what does Alternative provide that Monoid doesn't? I find this answer unsatisfactory since it lacks concrete examples, and a specific discussion of how the higher-kindedness of Alternative distinguishes it from Monoid.

If it's to combine applicative's effects with Monoid's behavior, why not just:

```
liftA2 mappend
```

This is even more confusing for me because many Monoid instances are exactly the same as the Alternative instances.

That's why I'm looking for **specific examples** that show why Alternative is necessary, and how it's different -- or means something different -- from Monoid.

haskell typeclass

edited Nov 3 '13 at 17:42
PeeHaa
38.5k • 28 • 123 • 203

asked Oct 26 '12 at 4:11
Matt Fenwick
22k • 7 • 72 • 133

2 Check out this question and the two questions linked within. – Rafael Caetano Oct 26 '12 at 4:44

Also see this answer. – Matt Fenwick Dec 3 '12 at 21:07

5 Answers

To begin with, let me offer short answers to each of these questions. I will then expand each into a longer detailed answer, but these short ones will hopefully help in navigating those.

1. No, `Alternative` and `Monoid` don't mean different things; `Alternative` is for types which have the structure both of `Applicative` and of `Monoid`. "Picking" and "combining" are two different intuitions for the same broader concept.
2. `Alternative` contains `empty` as well as `<|>` because the designers thought this would be useful, and because this gives rise to a monoid. In terms of picking, `empty` corresponds to making an impossible choice.
3. We need both `Alternative` and `Monoid` because the former obeys (or should) *more* laws than the latter; these laws relate the monoidal and applicative structure of the type constructor. Additionally, `Alternative` can't depend on the inner type, while `Monoid` can.
4. `MonadPlus` is slightly stronger than `Alternative`, as it must obey more laws; these laws relate the monoidal structure to the monadic structure in addition to the applicative structure. If you have instances of both, they should coincide.

Doesn't `Alternative` mean something totally different from `Monoid`?

Not really! Part of the reason for your confusion is that the Haskell `Monoid` class uses some pretty bad (well, insufficiently general) names. This is how a mathematician would define a monoid (being very explicit about it):

Definition. A *monoid* is a set M equipped with a distinguished element $\varepsilon \in M$ and a binary operator $\cdot : M \times M \to M$, denoted by juxtaposition, such that the following two conditions hold:

1. ε is the identity: for all $m \in M$, $m\varepsilon = \varepsilon m = m$.

2. $\cdot$ is associative: for all $m_1, m_2, m_3 \in M$, $(m_1 m_2) m_3 = m_1 (m_2 m_3)$.

That's it. In Haskell, ε is spelled `mempty` and · is spelled `mappend` (or, these days, `<>`), and the set *M* is the type `M` in `instance Monoid M where ...`.

Looking at this definition, we see that it says nothing about "combining" (or about "picking," for that matter). It says things about · and about *ε*, but that's it. Now, it's certainly true that combining things works well with this structure: *ε* corresponds to having no things, and $m_1 m_2$ says that if I glom m_1 and m_2's stuff together, I can get a new thing containing all their stuff. But here's an alternative intuition: *ε* corresponds to no choices at all, and $m_1 m_2$ corresponds to a choice between m_1 and m_2. This is the "picking" intuition. Note that both obey the monoid laws:

1. Having nothing at all and having no choice are both the identity.
   If I have no stuff and glom it together with some stuff, I end up with that same stuff again.

   If I have a choice between no choice at all (something impossible) and some other choice, I have to pick the other (possible) choice.
2. Glomming collections together and making a choice are both associative.
   If I have three collections of things, it doesn't matter if I glom the first two together and then the third, or the last two together and then the first; either way, I end up with the same total glommed collection.

   If I have a choice between three things, it doesn't matter if I (a) first choose between first-or-second and third and then, if I need to, between first and second, or (b) first choose between first and second-or-third and then, if I need to, between second and third. Either way, I can pick what I want.

(Note: I'm playing fast and loose here; that's why it's intuition. For instance, it's important to remember that · need not be commutative, which the above glosses over: it's perfectly possible that $m_1 m_2 \neq m_2 m_1$.)

Behold: both these sorts of things (and many others—is multiplying numbers really either "combining" *or* "picking"?) obey the same rules. Having an intuition is important to develop understanding, but it's the rules and definitions that determine what's *actually* going on.

And the best part is that these both of these intuitions can be interpreted by the same carrier! Let *M* be some set of sets (not a set of *all* sets!) containing the empty set, let *ε* be the empty set ∅, and let · be set union ∪. It is easy to see that ∅ is an identity for ∪, and that ∪ is associative, so we can conclude that $(M,\emptyset,\cup)$ is a monoid. Now:

1. If we think about sets as being collections of things, then ∪ corresponds to glomming them together to get more things—the "combining" intuition.
2. If we think about sets as representing possible actions, then ∪ corresponds to increasing your pool of possible actions to pick from—the "picking" intuition.

And this is exactly what's going on with `[]` in Haskell: `[a]` is a `Monoid` for all `a`, and `[]` as an applicative functor (and monad) is used to represent nondeterminism. Both the combining and the picking intuitions coincide at the same type: `mempty = empty = []` and `mappend = (<|>) = (++)`.

So the `Alternative` class is just there to represent objects which (a) are applicative functors, and (b) when instantiated at a type, have a value and a binary function on them which follow some rules. Which rules? The monoid rules. Why? Because it turns out to be useful :-)

> Why does `Alternative` need an empty method/member?

Well, the snarky answer is "because `Alternative` represents a monoid structure." But the real question is: *why* a monoid structure? Why not just a semigroup, a monoid without *ε*? One answer is to claim that monoids are just more useful. I think many people (but perhaps not Edward Kmett) would agree with this; almost all of the time, if you have a sensible `(<|>)` / `mappend` /·, you'll be able to define a sensible `empty` / `mempty` /*ε*. On the other hand, having the extra generality is nice, since it lets you place more things under the umbrella.

You also want to know how this meshes with the "picking" intuition. Keeping in mind that, in some sense, the right answer is "know when to abandon the 'picking' intuition," I think you *can* unify the two. Consider `[]`, the applicative functor for nondeterminism. If I combine two values of type `[a]` with `(<|>)`, that corresponds to nondeterministically picking either an action from the left or an action from the right. But sometimes, you're going to have no possible actions on one side—and that's fine. Similarly, if we consider parsers, `(<|>)` represents a parser which parses either what's on the left or what's on the right (it "picks"). And if you have a parser which always fails, that ends up being an identity: if you pick it, you immediately reject that pick and try the other one.

All this said, remember that it *would* be entirely possible to have a class almost like `Alternative`, but lacking `empty`. That would be perfectly valid—it could even be a superclass of `Alternative` —but happens not to be what Haskell did. Presumably this is out of a guess as to what's useful.

> Why does the `Alternative` type class need an `Applicative` constraint, and why does it need a kind of `* -> *`? … Why not just [use] `liftA2 mappend`?

Well, let's consider each of these three proposed changes: getting rid of the `Applicative` constraint for `Alternative`; changing the kind of `Alternative`'s argument; and using `liftA2 mappend` instead of `<|>` and `pure mempty` instead of `empty`. We'll look at this third change first, since it's the most different. Suppose we got rid of `Alternative` entirely, and replaced the class with two plain functions:

```
fempty :: (Applicative f, Monoid a) => f a
fempty = pure mempty

(>|<) :: (Applicative f, Monoid a) => f a -> f a -> f a
(>|<) = liftA2 mappend
```

We could even keep the definitions of `some` and `many`. And this *does* give us a monoid structure, it's true. But it seems like it gives us the wrong one . Should `Just fst >|< Just snd` fail, since `(a,a) -> a` isn't an instance of `Monoid`? No, but that's what the above code would result in. The monoid instance we *want* is one that's inner-type agnostic (to borrow terminology from Matthew Farkas-Dyck in a very related haskell-cafe discussion which asks some very similar questions); the `Alternative` structure is about a monoid determined by `f`'s structure, not the structure of `f`'s argument.

Now that we think we want to leave `Alternative` as some sort of type class, let's look at the two proposed ways to change it. If we change the kind, we *have* to get rid of the `Applicative` constraint; `Applicative` only talks about things of kind `* -> *`, and so there's no way to refer to it. That leaves two possible changes; the first, more minor, change is to get rid of the `Applicative` constraint but leave the kind alone:

```
class Alternative' f where
  empty' :: f a
  (<||>) :: f a -> f a -> f a
```

The other, larger, change is to get rid of the `Applicative` constraint and change the kind:

```
class Alternative'' a where
  empty'' :: a
  (<|||>) :: a -> a -> a
```

In both cases, we have to get rid of `some`/`many`, but that's OK; we can define them as standalone functions with the type `(Applicative f, Alternative' f) => f a -> f [a]` or `(Applicative f, Alternative'' (f [a])) => f a -> f [a]`.

Now, in the second case, where we change the kind of the type variable, we see that our class is exactly the same as `Monoid` (or, if you still want to remove `empty''`, `Semigroup`), so there's no advantage to having a separate class. And in fact, even if we leave the kind variable alone but remove the `Applicative` constraint, `Alternative` just becomes `forall a. Monoid (f a)`, although we can't write these quantified constraints in Haskell, not even with all the fancy GHC extensions. (Note that this expresses the inner-type–agnosticism mentioned above.) Thus, if we can make either of these changes, then we have no reason to keep `Alternative` (except for being able to express that quantified constraint, but that hardly seems compelling).

So the question boils down to "is there a relationship between the `Alternative` parts and the `Applicative` parts of an `f` which is an instance of both?" And while there's nothing in the documentation, I'm going to take a stand and say *yes*—or at the very least, there *ought* to be. I think that `Alternative` is supposed to obey some laws relating to `Applicative` (in addition to the monoid laws); in particular, I think those laws are something like

1. **Right distributivity (of `<*>`):** `(f <|> g) <*> a = (f <*> a) <|> (g <*> a)`
2. **Right absorption (for `<*>`):** `empty <*> a = empty`
3. **Left distributivity (of `fmap`):** `f <$> (a <|> b) = (f <$> a) <|> (f <$> b)`
4. **Left absorption (for `fmap`):** `f <$> empty = empty`

These laws appear to be true for `[]` and `Maybe`, and (pretending its `MonadPlus` instance is an `Alternative` instance) `IO`, but I haven't done any proofs or exhaustive testing. (For instance, I originally thought that *left* distributivity held for `<*>`, but this "performs the effects" in the wrong order for `[]`.) By way of analogy, though, it is true that `MonadPlus` is expected to obey similar laws (although there is apparently some ambiguity about which). I had originally wanted to claim a third law, which seems natural:

> **Left absorption (for `<*>`):** `a <*> empty = empty`

However, although I believe `[]` and `Maybe` obey this law, `IO` doesn't, and I think (for reasons that will become apparent in the next couple of paragraphs) it's best not to require it.

And indeed, it appears that Edward Kmett has some slides where he espouses a similar view; to get into that, we'll need to take brief digression involving some more mathematical jargon. The final slide, "I Want More Structure," says that "A Monoid is to an Applicative as a Right Seminearring is to an Alternative," and "If you throw away the argument of an Applicative, you get a Monoid, if you throw away the argument of an Alternative you get a RightSemiNearRing."

Right seminearrings? "How did right seminearrings get into it?" I hear you cry. Well,

Definition. A *right near-semiring* (also *right seminearring*, but the former seems to be used more on Google) is a quadruple $(R,+,\cdot,0)$ where $(R,+,0)$ is a monoid, $(R,\cdot)$ is a semigroup, and the following two conditions hold:

1. $\cdot$ is right-distributive over $+$: for all $r,s,t \in R$, $(s + t)r = sr + tr$.
2. 0 is right-absorbing for $\cdot$: for all $r \in R$, $0r = 0$.

A *left near-semiring* is defined analogously.

Now, this doesn't quite work, because `<*>` is not truly associative or a binary operator—the types don't match. I think this is what Edward Kmett is getting at when he talks about "throw[ing] away the argument." Another option might be to say (I'm unsure if this is right) that we actually want (`f a` , `<|>` , `<*>` , `empty`) to form a *right near-semiringoid*, where the "-oid" suffix indicates that the binary operators can only be applied to specific pairs of elements (à la *groupoids*). And we'd also want to say that (`f a` , `<|>` , `<$>` , `empty`) was a left near-semiringoid, although this could conceivably follow from the combination of the `Applicative` laws and the right near-semiringoid structure. But now I'm getting in over my head, and this isn't deeply relevant anyway.

At any rate, these laws, being *stronger* than the monoid laws, mean that perfectly valid `Monoid` instances would become invalid `Alternative` instances. There are (at least) two examples of this in the standard library: `Monoid a => (a,)` and `Maybe` . Let's look at each of them quickly.

Given any two monoids, their product is a monoid; consequently, tuples can be made an instance of `Monoid` in the obvious way (reformatting the base package's source):

```
instance (Monoid a, Monoid b) => Monoid (a,b) where
  mempty = (mempty, mempty)
  (a1,b1) `mappend` (a2,b2) = (a1 `mappend` a2, b1 `mappend` b2)
```

Similarly, we can make tuples whose first component is an element of a monoid into an instance of `Applicative` by accumulating the monoid elements (reformatting the base package's source):

```
instance Monoid a => Applicative ((,) a) where
  pure x = (mempty, x)
  (u, f) <*> (v, x) = (u `mappend` v, f x)
```

However, tuples aren't an instance of `Alternative` , because they can't be—the monoidal structure over `Monoid a => (a,b)` isn't present for all types `b` , and `Alternative` 's monoidal structure must be inner-type agnostic. Not only must `b` be a monad, to be able to express `(f <> g) <*> a` , we need to use the `Monoid` instance for functions, which is for functions of the form `Monoid b => a -> b` . And even in the case where we have all the necessary monoidal structure, it violates *all four* of the `Alternative` laws. To see this, let `ssf n = (Sum n, (<> Sum n))` and let `ssn = (Sum n, Sum n)` . Then, writing `(<>)` for `mappend` , we get the following results (which can be checked in GHCi, with the occasional type annotation):

1. Right distributivity:
   - `(ssf 1 <> ssf 1) <*> ssn 1 = (Sum 3, Sum 4)`
   - `(ssf 1 <*> ssn 1) <> (ssf 1 <*> ssn 1) = (Sum 4, Sum 4)`
2. Right absorption:
   - `mempty <*> ssn 1 = (Sum 1, Sum 0)`
   - `mempty = (Sum 0, Sum 0)`
3. Left distributivity:
   - `(<> Sum 1) <$> (ssn 1 <> ssn 1) = (Sum 2, Sum 3)`
   - `((<> Sum 1) <$> ssn 1) <> ((<> Sum 1) <$> ssn 1) = (Sum 2, Sum 4)`
4. Left absorption:
   - `(<> Sum 1) <$> mempty = (Sum 0, Sum 1)`
   - `mempty = (Sum 1, Sum 1)`

Next, consider `Maybe` . As it stands, `Maybe` 's `Monoid` and `Alternative` instances *disagree*. (Although the haskell-cafe discussion I mention at the beginning of this section proposes changing this, there's an `Option` newtype from the semigroups package which would produce the same effect.) As a `Monoid` , `Maybe` lifts semigroups into monoids by using `Nothing` as the identity; since the base package doesn't have a semigroup class, it just lifts monoids, and so we get (reformatting the base package's source):

```
instance Monoid a => Monoid (Maybe a) where
  mempty = Nothing
  Nothing `mappend` m       = m
  m       `mappend` Nothing = m
  Just m1 `mappend` Just m2 = Just (m1 `mappend` m2)
```

On the other hand, as an `Alternative` , `Maybe` represents prioritized choice with failure, and so we get (again reformatting the base package's source):

```
instance Alternative Maybe where
  empty = Nothing
  Nothing <|> r = r
  l       <|> _ = l
```

And it turns out that only the latter satisfies the `Alternative` laws. The `Monoid` instance fails less badly than `(,)` 's; it *does* obey the laws with respect to `<*>` , although almost by accident—it comes form the behavior of the only instance of `Monoid` for functions, which (as mentioned above), lifts functions that return monoids into the reader applicative functor. If you work it out (it's all very mechanical), you'll find that right distributivity and right absorption for `<*>` all hold for both the `Alternative` and `Monoid` instances, as does left absorption for `fmap` . And left distributivity for `fmap` does hold for the `Alternative` instance, as follows:

```
f <$> (Nothing <|> b)
  = f <$> b                        by the definition of (<|>)
  = Nothing <|> (f <$> b)          by the definition of (<|>)
  = (f <$> Nothing) <|> (f <$> b)  by the definition of (<$>)

f <$> (Just a <|> b)
  = f <$> Just a                   by the definition of (<|>)
  = Just (f a)                     by the definition of (<$>)
```

```
  = Just (f a) <|> (f <$> b)       by the definition of (<|>)
  = (f <$> Just a) <|> (f <$> b)   by the definition of (<$>)
```

However, it fails for the `Monoid` instance; writing `(<>)` for `mappend`, we have:

> `(<> Sum 1) <$> (Just (Sum 0) <> Just (Sum 0)) = Just (Sum 1)`
>
> `((<> Sum 1) <$> Just (Sum 0)) <> ((<> Sum 1) <$> Just (Sum 0)) = Just (Sum 2)`

Now, there is one caveat to this example. If you only require that `Alternative`s be compatibility with `<*>`, and not with `<$>`, then `Maybe` is fine. Edward Kmett's slides, mentioned above, don't make reference to `<$>`, but I think it seems reasonable to require laws with respect to it as well; nevertheless, I can't find anything to back me up on this.

Thus, we can conclude that being an `Alternative` is a *stronger* requirement than being a `Monoid`, and so it requires a different class. The purest example of this would be a type with an inner-type agnostic `Monoid` instance and an `Applicative` instance which were incompatible with each other; however, there aren't any such types in the base package, and I can't think of any. (It's possible none exist, although I'd be surprised.) Nevertheless, these inner-type gnostic examples demonstrate why the two type classes must be different.

> What's the point of the `MonadPlus` type class?

`MonadPlus`, like `Alternative`, is a strengthening of `Monoid`, but with respect to `Monad` instead of `Applicative`. According to Edward Kmett in his answer to the question "Distinction between typeclasses `MonadPlus`, `Alternative`, and `Monoid`?", `MonadPlus` is *also* stronger than `Alternative`: the law `empty <*> a`, for instance, doesn't imply that `empty >>= f`. AndrewC provides two examples of this: `Maybe` and its dual. The issue is complicated by the fact that there are two potential sets of laws for `MonadPlus`. It is universally agreed that `MonadPlus` is supposed to form a monoid with `mplus` and `mempty`, and it's supposed to satisfy the *left zero* law, `mempty >>= f = mempty`. However, some `MonadPlus`ses satisfy *left distribution*, `mplus a b >>= f = mplus (a >>= f) (b >>= f)`; and others satisfy *left catch*, `mplus (return a) b = return a`. (Note that left zero/distribution for `MonadPlus` are analogous to right distributivity/absorption for `Alternative`; `(<*>)` is more analogous to `(=<<)` than `(>>=)`.) Left distribution is probably "better," so any `MonadPlus` instance which satisfies left catch, such as `Maybe`, is an `Alternative` but not the first kind of `MonadPlus`. And since left catch relies on ordering, you can imagine a newtype wrapper for `Maybe` whose `Alternative` instance is *right*-biased instead of *left*-biased: `a <|> Just b = Just b`. This will satisfy neither left distribution nor left catch, but will be a perfectly valid `Alternative`.

However, since any type which *is* a `MonadPlus` ought to have its instance coincide with its `Alternative` instance (I believe this is required in the same way that it is required that `ap` and `(<*>)` are equal for `Monad`s that are `Applicative`s), you could imagine defining the `MonadPlus` class instead as

```
class (Monad m, Alternative m) => MonadPlus' m
```

The class doesn't need to declare new functions; it's just a promise about the laws obeyed by `empty` and `(<|>)` for the given type. This design technique isn't used in the Haskell standard libraries, but is used in some more mathematically-minded packages for similar purposes; for instance, the lattices package uses it to express the idea that a lattice is just a join semilattice and a meet semilattice over the same type which are linked by absorption laws.

The reason you can't do the same for `Alternative`, even if you wanted to guarantee that `Alternative` and `Monoid` always coincided, is because of the kind mismatch. The desired class declaration would have the form

```
class (Applicative f, forall a. Monoid (f a)) => Alternative''' f
```

but (as mentioned far above) not even GHC Haskell supports quantified constraints.

Also, note that having `Alternative` as be a superclass of `MonadPlus` would require `Applicative` being a superclass of `Monad`, so good luck getting that to happen. If you run into that problem, there's always the `WrappedMonad` newtype, which turns any `Monad` into an `Applicative` in the obvious way; there's an `instance MonadPlus m => Alternative (WrappedMonad m) where ...` which does exactly what you'd expect.

edited Oct 29 '12 at 23:26 answered Oct 26 '12 at 6:03
Antal Spector-Zabusky
22.9k ●4 ●45 ●103

Thank you, this helps a ton. I'm still stuck on point 3 though -- what value Alternative has over Monoid ... will have to think more about that one. – Matt Fenwick Oct 26 '12 at 12:42

1 @MattFenwick I'm in the same boat. `Monoid` I understand, but I'm not sure why we need a seperate and basically identical `Alternative` / `MonadPlus`. My big guess is hitory, but sometimes they have different semantics (which can be useful, but isn't a great argument for having both, IMHO). – singpolyma Oct 26 '12 at 16:16

I love hysterical raisins. – AndrewC Oct 29 '12 at 3:12

1 For the record - I was the one who fought the windmill about leaving `<>` only in `Data.Semigroup`. Edward folded early. – Yitz Nov 3 '13 at 19:10

2 from 2015: Alternative is a superclass of MonadPlus hackage.haskell.org/package/base-4.8.0.0/docs/... – sam boosalis May 14 '15 at 21:37

```
import Data.Monoid
import Control.Applicative
```

Let's trace through an example of how Monoid and Alternative interact with the `Maybe` functor and the `ZipList` functor, but let's start from scratch, partly to get all the definitions fresh in our minds, partly to stop from switching tabs to bits of hackage all the time, but mainly so I can run this past ghci to correct my typos!

```
(<>) :: Monoid a => a -> a -> a
(<>) = mappend -- I'll be using <> freely instead of `mappend`.
```

Here's the Maybe clone:

```
data Perhaps a = Yes a | No  deriving (Eq, Show)

instance Functor Perhaps where
   fmap f (Yes a) = Yes (f a)
   fmap f No      = No

instance Applicative Perhaps where
   pure a = Yes a
   No    <*> _     = No
   _     <*> No    = No
   Yes f <*> Yes x = Yes (f x)
```

and now ZipList:

```
data Zip a = Zip [a]  deriving (Eq,Show)

instance Functor Zip where
   fmap f (Zip xs) = Zip (map f xs)

instance Applicative Zip where
   Zip fs <*> Zip xs = Zip (zipWith id fs xs)   -- zip them up, applying the fs to the xs
   pure a = Zip (repeat a)   -- infinite so that when you zip with something, lengths
don't change
```

Structure 1: combining elements: Monoid

Maybe clone

First let's look at `Perhaps String`. There are two ways of combining them. Firstly concatenation

```
(<++>) :: Perhaps String -> Perhaps String -> Perhaps String
Yes xs <++> Yes ys = Yes (xs ++ ys)
Yes xs <++> No     = Yes xs
No     <++> Yes ys = Yes ys
No     <++> No     = No
```

Concatenation works inherently at the String level, not really the Perhaps level, by treating `No` as if it were `Yes []`. It's equal to `liftA2 (++)`. It's sensible and useful, but maybe we could generalise from just using `++` to using any way of combining - any Monoid then!

```
(<++>) :: Monoid a => Perhaps a -> Perhaps a -> Perhaps a
Yes xs <++> Yes ys = Yes (xs `mappend` ys)
Yes xs <++> No     = Yes xs
No     <++> Yes ys = Yes ys
No     <++> No     = No
```

This monoid structure for `Perhaps` tries to work as much as possible at the `a` level. Notice the `Monoid a` constraint, telling us we're using structure from the `a` level. This isn't an Alternative structure, it's a derived (lifted) Monoid structure.

```
instance Monoid a => Monoid (Perhaps a) where
   mappend = (<++>)
   mempty = No
```

Here I used the structure of the data a to add structure to the whole thing. If I were combining `Set`s, I'd be able to add an `Ord a` context instead.

ZipList clone

So how should we combine *elements* with a zipList? What should these zip to if we're combining them?

```
   Zip ["HELLO","MUM","HOW","ARE","YOU?"]
<> Zip ["this", "is", "fun"]
=  Zip ["HELLO" ? "this",   "MUM" ? "is",   "HOW" ? "fun"]

mempty = ["","","","",..]   -- sensible zero element for zipping with ?
```

But what should we use for `?`. I say the only sensible choice here is `++`. Actually, for lists, `(<>) = (++)`

```
   Zip [Just 1,  Nothing, Just 3, Just 4]
<> Zip [Just 40, Just 70, Nothing]
 =  Zip [Just 1 ? Just 40,    Nothing ? Just 70,    Just 3 ? Nothing]

mempty = [Nothing, Nothing, Nothing, .....]  -- sensible zero element
```

But what can we use for `?` I say that we're meant to be combining elements, so we should use the element-combining operator from Monoid again: `<>`.

```
instance Monoid a => Monoid (Zip a) where
   Zip as `mappend` Zip bs = Zip (zipWith (<>) as bs) -- zipWith the internal mappend
   mempty = Zip (repeat mempty)  -- repeat the internal mempty
```

This is the only sensible way of combining the elements using a zip - so it's the only sensible monoid instance.

Interestingly, that doesn't work for the Maybe example above, because Haskell doesn't know how to combine `Int`s - should it use `+` or `*`? To get a Monoid instance on numerical data, you wrap them in `Sum` or `Product` to tell it which monoid to use.

```
Zip [Just (Sum 1),   Nothing,       Just (Sum 3), Just (Sum 4)] <>
Zip [Just (Sum 40),  Just (Sum 70), Nothing]
= Zip [Just (Sum 41),Just (Sum 70), Just (Sum 3)]

   Zip [Product 5,Product 10,Product 15]
<> Zip [Product 3, Product 4]
 =  Zip [Product 15,Product 40]
```

Key point

Notice the fact that the type in a Monoid has kind `*` is exactly what allows us to put the `Monoid a` context here - we could also add `Eq a` or `Ord a`. In a Monoid, the raw elements matter. A Monoid instance is *designed* to let you manipulate and combine the data inside the structure.

Structure 2: higher-level choice: Alternative

A choice operator is similar, but also different.

Maybe clone

```
(<||>) :: Perhaps String -> Perhaps String -> Perhaps String
Yes xs <||> Yes ys = Yes xs   -- if we can have both, choose the left one
Yes xs <||> No     = Yes xs
No     <||> Yes ys = Yes ys
No     <||> No     = No
```

Here there's *no concatenation* - we didn't use `++` at all - this combination works purely at the `Perhaps` level, so let's change the type signature to

```
(<||>) :: Perhaps a -> Perhaps a -> Perhaps a
Yes xs <||> Yes ys = Yes xs   -- if we can have both, choose the left one
Yes xs <||> No     = Yes xs
No     <||> Yes ys = Yes ys
No     <||> No     = No
```

Notice there's no constraint - we're not using the structure from the `a` level, just structure at the `Perhaps` level. This is an Alternative structure.

```
instance Alternative Perhaps where
   (<|>) = (<||>)
   empty = No
```

ZipList clone

How should we choose between two ziplists?

```
Zip [1,3,4] <|> Zip [10,20,30,40] = ????
```

It would be very tempting to use `<|>` on the elements, but we can't because the type of the elements isn't available to us. Let's start with the `empty`. It can't use an element because we don't know the type of the elements when defining an Alternative, so it has to be `Zip []`. We need it to be a left (and preferably right) identity for `<|>`, so

```
Zip [] <|> Zip ys = Zip ys
Zip xs <|> Zip [] = Zip xs
```

There are two sensible choices for `Zip [1,3,4] <|> Zip [10,20,30,40]`:

1. `Zip [1,3,4]` because it's first - consistent with Maybe
2. `Zip [10,20,30,40]` because it's longest - consistent with `Zip []` being discarded

Well that's easy to decide: since `pure x = Zip (repeat x)`, both lists might be infinite, so comparing them for length might never terminate, so it has to be pick the first one. Thus the only sensible Alternative instance is:

```
instance Alternative Zip where
   empty = Zip []
   Zip [] <|> x = x
   Zip xs <|> _ = Zip xs
```

This is the only sensible Alternative we could have defined. Notice how different it is from the Monoid instance, because we couldn't mess with the elements, we couldn't even look at them.

Key Point

Notice that because `Alternative` takes a constructor of kind `* -> *` there is *no possible way* to add an `Ord a` or `Eq a` or `Monoid a` context. An Alternative is **not allowed** to use any information about the data inside the structure. You cannot, no matter how much you would like to, *do* anything to the data, except possibly throw it away.

Key point: What's the difference between Alternative and Monoid?

Not a lot - they're both monoids, but to summarise the last two sections:

`Monoid *` instances make it possible to combine internal data. `Alternative (* -> *)` instances make it impossible. Monoid provides flexibility, Alternative provides guarantees. The kinds `*` and `(* -> *)` are the main drivers of this difference. Having them both allows you to use both sorts of operations.

This is the right thing, and our two flavours are both appropriate. The Monoid instance for `Perhaps String` represents putting together all characters, the Alternative instance represents a choice between Strings.

There is nothing wrong with the Monoid instance for Maybe - it's doing its job, *combining* data. There's nothing wrong with the Alternative instance for Maybe - it's doing its job, *choosing* between things.

The Monoid instance for Zip combines its elements. The Alternative instance for Zip is forced to choose one of the lists - the first non-empty one.

It's good to be able to do both.

What's the Applicative context any use for?

There's some interaction between choosing and applying. See Antal S-Z's laws stated in his question or in the middle of his answer here.

From a practical point of view, it's useful because Alternative is something that is used for some Applicative Functors to choose. The functionality was being used for Applicatives, and so a general interface class was invented. Applicative Functors are good for representing computations that produce values (IO, Parser, Input UI element,...) and some of them have to handle failure - Alternative is needed.

Why does Alternative have `empty`?

> why does Alternative need an empty method/member? I may be wrong, but it seems to not be used at all ... at least in the code I could find. And it seems not to fit with the theme of the class -- if I have two things, and need to pick one, what do I need an 'empty' for?

That's like asking why addition needs a 0 - if you want to add stuff, what's the point in having something that doesn't add anything? The answer is that 0 is the crucual pivotal number around which everything revolves in addition, just like 1 is crucial for multiplication, `[]` is crucial for lists (and `y=e^x` is crucial for calculus). In practical terms, you use these do-nothing elements to start your building:

```
sum = foldr (+) 0
concat = foldr (++) []
msum = foldr (`mappend`) mempty          -- any Monoid
whichEverWorksFirst = foldr (<|>) empty  -- any Alternative
```

Can't we replace MonadPlus with Monad+Alternative?

> what's the point of the MonadPlus type class? Can't I unlock all of its goodness by just using something as both a Monad and Alternative? Why not just ditch it? (I'm sure I'm wrong, but I don't have any counterexamples)

You're not wrong, there aren't any counterexamples!

Your interesting question has got Antal S-Z, Petr Pudlák and I delved into what the relationship between MonadPlus and Applicative really is. The answer, here and here is that anything that's a `MonadPlus` (in the left distribution sense - follow links for details) is also an `Alternative`, but not the other way around.

This means that if you make an instance of Monad and MonadPlus, it satisfies the conditions for Applicative and Alternative anyway. This means if you follow the rules for MonadPlus (with left dist), you may as well have made your Monad an Applicative and used Alternative.

If we remove the MonadPlus class, though, we remove a sensible place for the rules to be documented, and you lose the ability to specify that something's Alternative without being MonadPlus (which technically we ought to have done for Maybe). These are theoretical reasons. The practical reason is that it would break existing code. (Which is also why neither Applicative nor Functor are superclasses of Monad.)

Aren't Alternative and Monoid the same? Aren't Alternative and Monoid completely different?

> the 'pedia says that "the Alternative type class is for Applicative functors which also have a monoid structure." I don't get this -- doesn't Alternative mean something totally different from Monoid? i.e. I understood the point of the Alternative type class as picking between two

> things, whereas I understood Monoids as being about combining things.

Monoid and Alternative are two ways of getting one object from two in a sensible way. Maths doesn't care whether you're choosing, combining, mixing or blowing up your data, which is why Alternative was referred to as a Monoid for Applicative. You seem to be at home with that concept now, but you now say

> for types that have both an Alternative and a Monoid instance, the instances are intended to be the same

I disagree with this, and I think my Maybe and ZipList examples are carefully explained as to why they're different. If anything, I think it should be rare that they're the same. I can only think of one example, plain lists, where this is appropriate. That's because lists are a fundamental example of a monoid with `++`, but also lists are used in some contexts as an indeterminate choice of elements, so `<|>` should also be `++`.

answered Nov 1 '12 at 10:03
AndrewC
24.8k ●4 ●54 ●95

Summary

> We need to define (instances that provide the same operations as) Monoid instances for some applicative functors, that genuinely combine at the applicative functor level, and not just lifting lower level monoids. The example error below from `litvar = liftA2 mappend literal variable` shows that `<|>` cannot in general be defined as `liftA2 mappend`; `<|>` works in this case by combining parsers, not their data.
>
> If we used Monoid directly, we'd need language extensions to define the instances. `Alternative` is higher kinded so you can make these instances without requiring language extensions.

Example: Parsers

Let's imagine we're parsing some declarations, so we import everything we're going to need

```
import Text.Parsec
import Text.Parsec.String
import Control.Applicative ((<$>),(<*>),liftA2,empty)
import Data.Monoid
import Data.Char
```

and think about how we'll parse a type. We choose simplistic:

```
data Type = Literal String | Variable String  deriving Show
examples = [Literal "Int",Variable "a"]
```

Now let's write a parser for literal types:

```
literal :: Parser Type
literal = fmap Literal $ (:) <$> upper <*> many alphaNum
```

Meaning: parse an `upper`case character, then `many alphaNum`eric characters, combine the results into a single String with the pure function `(:)`. Afterwards, apply the pure function `Literal` to turn those `String`s into `Type`s. We'll parse variable types exactly the same way, except for starting with a `lower`case letter:

```
variable :: Parser Type
variable = fmap Variable $ (:) <$> lower <*> many alphaNum
```

That's great, and `parseTest literal "Bool" == Literal "Bool"` exactly as we'd hoped.

Question 3a: If it's to combine applicative's effects with Monoid's behavior, why not just `liftA2 mappend`

Edit:Oops - forgot to actually use `<|>`!
Now let's combine these two parsers using Alternative:

```
types :: Parser Type
types = literal <|> variable
```

This can parse any Type: `parseTest types "Int" == Literal "Bool"` and `parseTest types "a" == Variable "a"`. This combines the two *parsers*, not the two *values*. That's the sense in which it works at the Applicative Functor level rather than the data level.

However, if we try:

```
litvar = liftA2 mappend literal variable
```

that would be asking the compiler to combine the two *values* that they generate, at the data level. We get

```
No instance for (Monoid Type)
  arising from a use of `mappend'
Possible fix: add an instance declaration for (Monoid Type)
```

```
In the first argument of `liftA2', namely `mappend'
In the expression: liftA2 mappend literal variable
In an equation for `litvar':
    litvar = liftA2 mappend literal variable
```

So we found out the first thing; the Alternative class does something genuinely different to `liftA2 mappend`, becuase it combines objects at a different level - it combines the parsers, not the parsed data. If you like to think of it this way, it's combination at the genuinely higher-kind level, not merely a lift. I don't like saying it that way, because `Parser Type` has kind `*`, but it is true to say we're combining the `Parser` s, not the `Type` s.

(Even for types with a Monoid instance, `liftA2 mappend` won't give you the same parser as `<|>`. If you try it on `Parser String` you'll get `liftA2 mappend` which parses one after the other then concatenates, versus `<|>` which will try the first parser and default to the second if it failed.)

Question 3b: In what way does Alternative's `<|> :: f a -> f a -> f a` differ from Monoid's `mappend :: b -> b -> b`?

Firstly, you're right to note that it doesn't provide new functionality over a Monoid instance.

Secondly, however, there's an issue with using Monoid directly: Let's try to use `mappend` on parsers, at the same time as showing it's the same structure as `Alternative`:

```
instance Monoid (Parser a) where
    mempty = empty
    mappend = (<|>)
```

Oops! We get

```
Illegal instance declaration for `Monoid (Parser a)'
  (All instance types must be of the form (T t1 ... tn)
   where T is not a synonym.
   Use -XTypeSynonymInstances if you want to disable this.)
In the instance declaration for `Monoid (Parser a)'
```

So if you have an applicative functor `f`, the `Alternative` instance shows that `f a` is a monoid, but you could only declare that as a `Monoid` with a language extension.

Once we add `{-# LANGUAGE TypeSynonymInstances #-}` at the top of the file, we're fine and can define

```
typeParser = literal `mappend` variable
```

and to our delight, it works: `parseTest typeParser "Yes" == Literal "Yes"` and `parseTest typeParser "a" == Literal "a"`.

Even if you don't have any synonyms (`Parser` and `String` are synonyms, so they're out), you'll still need `{-# LANGUAGE FlexibleInstances #-}` to define an instance like this one:

```
data MyMaybe a = MyJust a | MyNothing deriving Show
instance Monoid (MyMaybe Int) where
   mempty = MyNothing
   mappend MyNothing x = x
   mappend x MyNothing = x
   mappend (MyJust a) (MyJust b) = MyJust (a + b)
```

(The monoid instance for Maybe gets around this by lifting the underlying monoid.)

Making a standard library unnecessarily dependent on language extensions is clearly undesirable.

So there you have it. Alternative is just Monoid for Applicative Functors (and isn't just a lift of a Monoid). It needs the higher-kinded type `f a -> f a -> f a` so you can define one without language extensions.

Your other Questions, for completeness:

1. Why does Alternative need an empty method/member?
   Because having an identity for an operation is sometimes useful. For example, you can define `anyA = foldr (<|>) empty` without using tedious edge cases.

2. what's the point of the MonadPlus type class? Can't I unlock all of its goodness by just using something as both a Monad and Alternative? No. I refer you back to the question you linked to:

> Moreover, even if Applicative was a superclass of Monad, you'd wind up needing the MonadPlus class anyways, because obeying `empty <*> m = empty` isn't strictly enough to prove that `empty >>= f = empty`.

....and I've come up with an example: Maybe. I explain in detail, with proof in this answer to Antal's question. For the purposes of this answer, it's worth noting that I was able to use >>= to make the MonadPlus instance that broke the Alternative laws.

Monoid structure is useful. Alternative is the best way of providing it for Applicative Functors.

edited Oct 29 '12 at 22:48 answered Oct 29 '12 at 2:30

AndrewC
24.8k ● 4 ● 54 ● 95

1 @MattFenwick These aren't silly questions. Alternative *is* the same as a monoid instance for Parser, yes. I show that (a) `(<|>)` is not equal to `liftA2 mappend`, addressing your question why we don't just do that, and (b) that you'd need a language extension to define that monoid instance, which is why there's a separate class, addressing your main question. – AndrewC Oct 29 '12 at 14:27

@MattFenwick So sorry - I realise now I never actually used <|> so there really weren't any <|> examples to contrast with how using Monoid didn't work! I've changed mainly the start of section 3a but also a bit of 3b. – AndrewC Oct 29 '12 at 20:47

@MattFenwick hopefully now I actually *included* the examples, it should make more sense! – AndrewC Oct 29 '12 at 20:51

I won't cover MonadPlus because there is disagreement about its laws.

After trying and failing to find any meaningful examples in which the structure of an Applicative leads naturally to an Alternative instance that disagrees with its Monoid instance*, I finally came up with this:

Alternative's laws are more strict than Monoid's, because the result *cannot* depend on the inner type. This excludes a large number of Monoid instances from being Alternatives. These datatypes allow partial (meaning that they only work for some inner types) Monoid instances which are forbidden by the extra 'structure' of the `* -> *` kind. Examples:

> the standard Maybe instance for Monoid assumes that the inner type is Monoid => not an Alternative
>
> ZipLists, tuples, and functions can all be made Monoids, *if* their inner types are Monoids => not Alternatives
>
> sequences that have at least one element -- cannot be Alternatives because there's no `empty`:
>
> ```
> data Seq a
>     = End a
>     | Cons a (Seq a)
>   deriving (Show, Eq, Ord)
> ```

On the other hand, some data types cannot be made Alternatives because they're `*`-kinded:

> unit -- `()`
>
> `Ordering`
>
> numbers, booleans

My inferred conclusion: **for types that have both an Alternative and a Monoid instance, the instances are intended to be the same.** See also this answer.

---

excluding Maybe, which I argue doesn't count because its standard instance should not require Monoid for the inner type, in which case it would be identical to Alternative

edited Dec 3 '12 at 21:08

community wiki
4 revs
Matt Fenwick

> I understood the point of the Alternative type class as picking between two things, whereas I understood Monoids as being about combining things.

If you think about this for a moment, they are the same.

The `+` combines things (usually numbers), and it's type signature is `Int -> Int -> Int` (or whatever).

The `<|>` operator selects between alternatives, and it's type signature is also the same: take two matching things and return a combined thing.

answered Oct 26 '12 at 12:00
MathematicalOrchid
28.5k ● 9 ● 64 ● 132

<div><img src="/posts/13080606/ivc/82c9" class="dno" alt="" width="0" height="0"></div>

<div id="noscript-warning">Stack Overflow works best with JavaScript enabled<img src="http://pixel.quantserve.com/pixel/p-c1rF4kxgLUzNc.gif" alt="" class="dno"></div>

StackExchange sign up log in tour help

Stack Overflow is a community of 4.7 million programmers, just like you, helping each other.

Join them; it only takes a minute:

Sign up

Join the Stack Overflow community to:

Ask programming questions

Answer and help your peers

Get recognized for your expertise

What's wrong with GHC Haskell's current constraint system?

I've heard that there are some problems with Haskell's "broken" constraint system, as of GHC 7.6 and below. What's "wrong" with it? Is there a comparable existing system that overcomes those flaws?

For example, both edwardk and tekmo have run into trouble (e.g. this comment from tekmo).

haskell typeclass

edited Jan 15 '13 at 5:07
tobyodavies 10.3k ●2 ●26 ●48

asked Oct 9 '12 at 17:34
Dan Burton 31.3k ●13 ●85 ●159

4 While I'm sure there's an interesting question in here, in its current form it's essentially "What problems have edwardk and tekmo run into?" , which can only really be answered by those people. As such, I don't think this question is a good fit for SO in its current form. – hammar Oct 9 '12 at 17:47

5 I seems like "what problems exist that anyone has run into?" is more the intent here. Anyone who's run into similar problems could, I expect, recognize that the question just as well as the specific people whose complaints are mentioned. – C. A. McCann Oct 9 '12 at 18:22

3 Yes, @C.A.McCann captured my intent fairly well, though I'm not particularly looking for "what problems have you run into?" so much as "what is the underlying problem?" I expect a good answer will elaborate on what the current constraint system *is*, what its weaknesses are, and whether there are existing plans to improve on it. – Dan Burton Oct 9 '12 at 18:56

6 I started a discussion at /r/haskell. I was under the impression that there was an obvious, well-understood flaw, but apparently this is not the case. – Dan Burton Oct 9 '12 at 21:07

2 @C.A.McCann what is LtU? – Cetin Sert Oct 10 '12 at 13:40

2 Answers

Ok, I had several discussions with other people before posting here because I wanted to get this right. They all showed me that all the problems I described boil down to the lack of polymorphic constraints.

The simplest example of this problem is the `MonadPlus` class, defined as:

```
class MonadPlus m where
    mzero :: m a
    mplus :: m a -> m a -> m a
```

... with the following laws:

```
mzero `mplus` m = m

m `mplus` mzero = m

(m1 `mplus` m2) `mplus` m3 = m1 `mplus` (m2 `mplus` m3)
```

Notice that these are the `Monoid` laws, where the `Monoid` class is given by:

```
class Monoid a where
    mempty :: a
    mappend :: a -> a -> a

mempty `mplus` a = a

a `mplus` mempty = a

(a1 `mplus` a2) `mplus` a3 = a1 `mplus` (a2 `mplus` a3)
```

So why do we even have the `MonadPlus` class? The reason is because Haskell forbids us from writing constraints of the form:

```
(forall a . Monoid (m a)) => ...
```

So Haskell programmers must work around this flaw of the type system by defining a separate class to handle this particular polymorphic case.

However, this isn't always a viable solution. For example, in my own work on the `pipes` library, I frequently encountered the need to pose constraints of the form:

```
(forall a' a b' b . Monad (p a a' b' b m)) => ...
```

Unlike the `MonadPlus` solution, I cannot afford to switch the `Monad` type class to a different type class to get around the polymorphic constraint problem because then users of my library would lose `do` notation, which is a high price to pay.

This also comes up when composing transformers, both monad transformers and the proxy transformers I include in my library. We'd like to write something like:

```
data Compose t1 t2 m r = C (t1 (t2 m) r)

instance (MonadTrans t1, MonadTrans t2) => MonadTrans (Compose t1 t2) where
    lift = C . lift . lift
```

This first attempt doesn't work because `lift` does not constrain its result to be a `Monad`. We'd actually need:

```
class (forall m . Monad m => Monad (t m)) => MonadTrans t where
    lift :: (Monad m) => m r -> t m r
```

... but Haskell's constraint system does not permit that.

This problem will grow more and more pronounced as Haskell users move on to type constructors of higher kinds. You will typically have a type class of the form:

```
class SomeClass someHigherKindedTypeConstructor where
    ...
```

... but you will want to constrain some lower-kinded derived type constructor:

```
class (SomeConstraint (someHigherKindedTypeConstructor a b c))
    => SomeClass someHigherKindedTypeConstructor where
    ...
```

However, without polymorphic constraints, that constraint is not legal. I've been the one complaining about this problem the most recently because my `pipes` library uses types of very high kinds, so I run into this problem constantly.

There are workarounds using data types that several people have proposed to me, but I haven't (yet) had the time to evaluate them to understand which extensions they require or which one solves my problem correctly. Somebody more familiar with this issue could perhaps provide a separate answer detailing the solution to this and why it works.

edited Oct 11 '12 at 20:16

answered Oct 11 '12 at 15:29

Gabriel Gonzalez
27.1k ●3 ●48 ●99

[a follow-up to Gabriel Gonzalez answer]

The right notation for constraints and quantifications in Haskell is the following:

```
<functions-definition> ::= <functions> :: <quantified-type-expression>

<quantified-type-expression> ::= forall <type-variables-with-kinds> . (<constraints>) =>
<type-expression>

<type-expression> ::= <type-expression> -> <quantified-type-expression>
                    | ...

...
```

Kinds can be omitted, as well as `forall`s for rank-1 types:

```
<simply-quantified-type-expression> ::= (<constraints-that-uses-rank-1-type-variables>) =>
<type-expression>
```

For example:

```
{-# LANGUAGE Rank2Types #-}

msum :: forall m a. Monoid (m a) => [m a] -> m a
msum = mconcat

mfilter :: forall m a. (Monad m, Monoid (m a)) => (a -> Bool) -> m a -> m a
```

```
mfilter p ma = do { a <- ma; if p a then return a else mempty }

guard :: forall m. (Monad m, Monoid (m ())) => Bool -> m ()
guard True = return ()
guard False = mempty
```

or without `Rank2Types` (since we only have rank-1 types here), and using `CPP` (j4f):

```
{-# LANGUAGE CPP #-}

#define MonadPlus(m, a) (Monad m, Monoid (m a))

msum :: MonadPlus(m, a) => [m a] -> m a
msum = mconcat

mfilter :: MonadPlus(m, a) => (a -> Bool) -> m a -> m a
mfilter p ma = do { a <- ma; if p a then return a else mempty }

guard :: MonadPlus(m, ()) => Bool -> m ()
guard True = return ()
guard False = mempty
```

The "problem" is that we can't write

```
class (Monad m, Monoid (m a)) => MonadPlus m where
  ...
```

or

```
class forall m a. (Monad m, Monoid (m a)) => MonadPlus m where
  ...
```

That is, `forall m a. (Monad m, Monoid (m a))` can be used as a standalone constraint, but can't be aliased with a new one-parametric typeclass for `*->*` types.

This is because the typeclass defintion mechanism works like this:

```
class (constraints[a, b, c, d, e, ...]) => ClassName (a b c) (d e) ...
```

i.e. the **rhs** side introduce type variables, not the lhs or `forall` at the lhs.

Instead, we need to write 2-parametric typeclass:

```
{-# LANGUAGE MultiParamTypeClasses, FlexibleContexts, FlexibleInstances #-}

class (Monad m, Monoid (m a)) => MonadPlus m a where
  mzero :: m a
  mzero = mempty
  mplus :: m a -> m a -> m a
  mplus = mappend

instance MonadPlus [] a
instance Monoid a => MonadPlus Maybe a

msum :: MonadPlus m a => [m a] -> m a
msum = mconcat

mfilter :: MonadPlus m a => (a -> Bool) -> m a -> m a
mfilter p ma = do { a <- ma; if p a then return a else mzero }

guard :: MonadPlus m () => Bool -> m ()
guard True = return ()
guard False = mzero
```

Cons: we need to specify second parameter every time we use `MonadPlus`.

Question: how

```
instance Monoid a => MonadPlus Maybe a
```

can be written if `MonadPlus` is one-parametric typeclass? `MonadPlus Maybe` from `base`:

```
instance MonadPlus Maybe where
   mzero = Nothing
   Nothing `mplus` ys  = ys
   xs      `mplus` _ys = xs
```

works not like `Monoid Maybe`:

```
instance Monoid a => Monoid (Maybe a) where
  mempty = Nothing
  Nothing `mappend` m = m
  m `mappend` Nothing = m
  Just m1 `mappend` Just m2 = Just (m1 `mappend` m2) -- < here
```

:

```
(Just [1,2] `mplus` Just [3,4]) `mplus` Just [5,6] => Just [1,2]
(Just [1,2] `mappend` Just [3,4]) `mappend` Just [5,6] => Just [1,2,3,4,5,6]
```

Analogically, `forall m a b n c d e. (Foo (m a b), Bar (n c d) e)` gives rise for (7 - 2 * 2)-parametric typeclass if we want `*` types, (7 - 2 * 1)-parametric typeclass for `* -> *` types, and (7 - 2 * 0) for `* -> * -> *` types.

answered Oct 11 '12 at 19:43

JJJ
1,756 ●4 ●18

StackExchange sign up log in tour help

Stack Overflow is a community of 4.7 million programmers, just like you, helping each other.

Join them; it only takes a minute:

Sign up

Join the Stack Overflow community to:

Ask programming questions

Answer and help your peers

Get recognized for your expertise

Lax monoidal functors with a different monoidal structure

Add projects to your stackoverflow CAREERS profile.

Applicative functors are well-known and well-loved among Haskellers, for their ability to apply functions in an effectful context.

In category-theoretic terms, it can be shown that the methods of `Applicative`:

```
pure :: a -> f a
(<*>) :: f (a -> b) -> f a -> f b
```

are equivalent to having a `Functor f` with the operations:

```
unit :: f ()
(**) :: (f a, f b) -> f (a,b)
```

the idea being that to write `pure` you just replace the `()` in `unit` with the given value, and to write `(<*>)` you squish the function and argument into a tuple and then map a suitable application function over it.

Moreover, this correspondence turns the `Applicative` laws into natural monoidal-ish laws about `unit` and `(**)`, so in fact an applicative functor is precisely what a category theorist would call a lax monoidal functor (lax because `(**)` is merely a natural transformation and not an isomorphism).

Okay, fine, great. This much is well-known. But that's only one family of lax monoidal functors – those respecting the monoidal structure of the *product*. A lax monoidal functor involves two choices of monoidal structure, in the source and destination: here's what you get if you turn product into sum:

```
class PtS f where
  unit :: f Void
  (**) :: f a -> f b -> f (Either a b)

-- some example instances
instance PtS Maybe where
  unit = Nothing
  Nothing ** Nothing = Nothing
  Just a ** Nothing = Just (Left a)
  Nothing ** Just b = Just (Right b)
  Just a ** Just b = Just (Left a) -- ick, but it does satisfy the laws

instance PtS [] where
  unit = []
  xs ** ys = map Left xs ++ map Right ys
```

It seems like turning sum into other monoidal structures is made less interesting by `unit :: Void -> f Void` being uniquely determined, so you really have more of a semigroup going on. But still:

- Are other lax monoidal functors like the above studied or useful?
- Is there a neat alternative presentation for them like the `Applicative` one?

haskell functor applicative category-theory

edited May 26 '14 at 22:02

asked Apr 26 '14 at 20:38
Ben Millwood
4,723 • 11 • 37

When you say `Void` in the type of `PtS.unit`, don't you mean Empty, since it should be a unit for `Either`? – Dominique Devriese Apr 27 '14 at 7:04

Never mind, you probably intended `Void` to represent the empty type. I was confused because the name void in C-like languages corresponds to the unit type, which you write as `()`. – Dominique Devriese Apr 27 '14 at 7:20

Yep, sorry for the confusion, but there's reasonable precedent for it: hackage.haskell.org/package/void – Ben Millwood Apr 27 '14 at 11:10

2 Ack, it's arguably C's usage of void that is wrong, not yours ;) – Dominique Devriese Apr 27 '14 at 17:25

3 Answers

The "neat alternative presentation" for `Applicative` is based on the following two equivalencies

```
pure a = fmap (const a) unit
unit = pure ()

ff <*> fa = fmap (\(f,a) -> f a) $ ff ** fa
fa ** fb = pure (,) <*> fa <*> fb
```

The trick to get this "neat alternative presentation" for `Applicative` is the same as the trick for `zipWith` - replace explicit types and constructors in the interface with things that the type or constructor can be passed into to recover what the original interface was.

```
unit :: f ()
```

Is replaced with `pure` which we can substitute the type `()` and the constructor `() :: ()` into to recover `unit`.

```
pure :: a  -> f a
pure    () :: f ()
```

And similarly (though not as straightforward) for substituting the type `(a,b)` and the constructor `(,) :: a -> b -> (a,b)` into `liftA2` to recover `**`.

```
liftA2 :: (a -> b -> c) -> f a -> f b -> f c
liftA2    (,)           :: f a -> f b -> f (a,b)
```

`Applicative` then gets the nice `<*>` operator by lifting function application `($) :: (a -> b) -> a -> b` into the functor.

```
(<*>) :: f (a -> b) -> f a -> f b
(<*>) = liftA2 ($)
```

To find a "neat alternative presentation" for `PtS` we need to find

- something we can substitute the type `Void` into to recover `unit`
- something we can substitute the type `Either a b` and the constructors `Left :: a -> Either a b` and `Right :: b -> Either a b` into to recover `**`

(If you notice that we already have something the constructors `Left` and `Right` can be passed to you can probably figure out what we can replace `**` with without following the steps I used; I didn't notice this until after I solved it)

unit

This immediately gets us an alternative to `unit` for sums:

```
empty :: f a
empty = fmap absurd unit

unit :: f Void
unit = empty
```

operator

We'd like to find an alternative to `(**)`. There is an alternative to sums like `Either` that allows them to be written as functions of products. It shows up as the visitor pattern in object oriented programming languages where sums don't exist.

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

{-# LANGUAGE RankNTypes #-}
type Sum a b = forall c. (a -> c) -> (b -> c) -> c
```

It's what you would get if you changed the order of `either`'s arguments and partially applied them.

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

toSum :: Either a b -> Sum a b
toSum e = \forA forB -> either forA forB e

toEither :: Sum a b -> Either a b
toEither s = s Left Right
```

We can see that `Either a b ≅ Sum a b`. This allows us to rewrite the type for `(**)`

```
(**) :: f a -> f b -> f (Either a b)
(**) :: f a -> f b -> f (Sum a b)
(**) :: f a -> f b -> f ((a -> c) -> (b -> c) -> c)
```

Now it's clear what `**` does. It delays `fmap`ing something onto both of its arguments, and combines the results of those two mappings. If we introduce a new operator, `<||> :: f c -> f c -> f c` which simply assumes that the `fmap`ing was done already, then we can see that

```
fmap (\f -> f forA forB) (fa ** fb) = fmap forA fa <||> fmap forB fb
```

Or back in terms of `Either`:

```
fa ** fb = fmap Left fa <||> fmap Right fb
fa1 <||> fa2 = fmap (either id id) $ fa1 ** fa2
```

So we can express everything `PtS` can express with the following class, and everything that could implement `PtS` can implement the following class:

```
class Functor f => AlmostAlternative f where
    empty  :: f a
    (<||>) :: f a -> f a -> f a
```

This is almost certainly the same as the `Alternative` class, except we didn't require that the `Functor` be `Applicative`.

Conclusion

It's just a `Functor` that is a `Monoid` for all types. It'd be equivalent to the following:

```
class (Functor f, forall a. Monoid (f a)) => MonoidalFunctor f
```

The `forall a. Monoid (f a)` constraint is pseudo-code; I don't know a way to express constraints like this in Haskell.

edited Apr 29 '14 at 3:58 answered Apr 27 '14 at 7:00
Cirdec
16.1k ●1 ●24 ●62

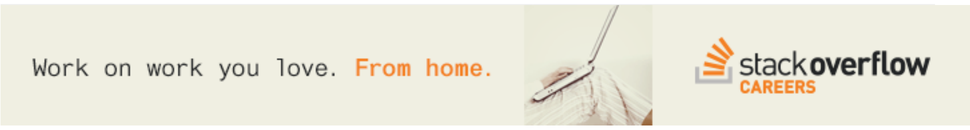
+1 for actually analysing and argumenting your answer ;). – Dominique Devriese Apr 27 '14 at 7:17

Perfect! Now annoyed I didn't spot this myself :P (and also, curse you, you've given me a nontrivial choice of which answer to accept) – Ben Millwood Apr 27 '14 at 11:08

(Small point: you describe sums as analogous to "functions of products", do you mean "products of functions"?) – Ben Millwood Apr 27 '14 at 11:18

No, I mean't functions of products. A sum is a function that accepts the product of two functions. You get data out of the sum by passing in two functions (the product of two functions) - one for what to do for the first option, one for what to do with the second. If the sum is the first option, it takes the first function from the product, applies it to what the sum contains, and returns the result. If the sum is the second option, it takes the second function from the product, applies it to what the sum contains, and returns the result. A sum is a function that takes a product of functions. – Cirdec Apr 27 '14 at 16:12

Oh, I see what you mean, yes. – Ben Millwood Apr 27 '14 at 17:32



Before you can even talk about monoidal functors, you need to make sure you're in a monoidal category. It so happens that **Hask** is a monoidal category in the following way:

- `()` as identity
- `(,)` as bifunctor
- Identify isomorphic types, i.e. `(a,()) ≅ ((),a) ≅ a`, and `(a,(b,c)) ≅ ((a,b),c)`.

Like you observed, it's also a monoidal category when you exchange `()` for `Void` and `(,)` for `Either`.
However, monoidal doesn't get you very far – what makes **Hask** so powerful is that it's cartesian closed. That gives us currying and related techniques, without which applicative would be pretty much useless.

A monoidal category can be cartesian closed iff its identity is a terminal object, i.e. a type *onto* which there exists precisely one (of course, we disregard □ here) arrow. There is one function `A -> ()` for any type `A`, namely `const ()`. There is no function `A -> Void` however. Instead, `Void` is the *initial object*: there exists precisely one arrow *from* it, namely the `absurd :: Void -> a` method. Such a monoidal category can't be cartesian closed then.
Now, of course, you can switch between initial and terminal easily by turning around the arrow direction. That always places you in the dual structure, so we get a cocartesian closed category. But that means you also need to flip the arrows in your monoidal functors. Those are called decisive functors then (and generalise comonads). With Conor's ever-so-amazing naming scheme,

```
class (Functor f) => Decisive f where
  nogood :: f Void -> Void
  orwell :: f (Either s t) -> Either (f s) (f t)
```

edited Apr 26 '14 at 22:16 answered Apr 26 '14 at 21:44
leftaroundabout
42.6k ●3 ●68 ●142

2 I'm aware this doesn't really answer your question. A monoidal functor WRT coproducts might still be interesting in some way, but I suppose troubles like `unit` being trivial as you say largely hampers this. – leftaroundabout Apr 26 '14 at 21:53

I knew you must be the author of this post was by the time I finished reading the first sentence. – Cirdec

Apr 27 '14 at 4:51

My background in category theory is very limited, but FWIW, your PtS class reminds me of the `Alternative` class, which looks essentially like this:

```
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a
```

The only problem of course is that `Alternative` is an extension of `Applicative`. However, perhaps one can imagine it being presented separately, and the combination with `Applicative` is then quite reminiscent of a functor with a non-commutative ring-like structure, with the two monoid structures as the operations of the ring? There are also distributivity laws between `Applicative` and `Alternative` IIRC.

edited Apr 27 '14 at 7:17

answered Apr 27 '14 at 6:02

Dominique Devriese
2,044 ●1 ●5 ●15

+1 for seeing straight through the problem. After working through the problem by hand, I arrived at the same conclusion for my answer. – Cirdec Apr 27 '14 at 7:04