A free foo happens to be the simplest thing that satisfies all of the 'foo' laws. That is to say it satisfies exactly the laws necessary to be a foo and nothing extra.

A forgetful functor is one that "forgets" part of the structure as it goes from one category to another.

Given functors F : D -> C, and G : C -> D, we say F -| G, F is left adjoint to G, or G is right adjoint to F whenever forall a, b: F a -> b is isomorphic to a -> G b, where the arrows come from the appropriate categories.

Formally, a free functor is left adjoint to a forgetful functor.

***The Free Monoid***

Let us start with a simpler example, the free monoid.

Take a monoid, which is defined by some carrier set T, a binary function to mash a pair of elements together f :: T → T → T, and a unit :: T, such that you have an associative law, and an identity law: f(unit,x) = x = f(x,unit).

You can make a functor U from the category of monoids (where arrows are monoid homomorphisms, that is, they ensure they map unit to unit on the other monoid, and that you can compose before or after mapping to the other monoid without changing meaning) to the category of sets (where arrows are just function arrows) that 'forgets' about the operation and unit, and just gives you the carrier set.

Then, you can define a functor F from the category of sets back to the category of monoids that is left adjoint to this functor. That functor is the functor that maps a set a to the monoid [a], where unit = [], and mappend = (++).

So to review our example so far, in pseudo-Haskell:

U : Mon → Set -- is our forgetful functor

U (a,mappend,mempty) = a

F : Set → Mon -- is our free functor

F a = ([a],(++),[])

Then to show F is free, need to demonstrate that it is left adjoint to U, a forgetful functor, that is, as we mentioned above, we need to show that

F a → b is isomorphic to a → U b

now, remember the target of F is in the category Mon of monoids, where arrows are monoid homomorphisms, so we need a to show that a monoid homomorphism from [a] → b can be described precisely by a function from a → b.

In Haskell, we call the side of this that lives in Set (er, Hask, the category of Haskell types that we pretend is Set), just foldMap, which when specialized from Data.Foldable to Lists has type Monoid m => (a → m) → [a] → m.

There are consequences that follow from this being an adjunction. Notably that if you forget then build up with free, then forget again, its just like you forgot once, and we can use this to build up the monadic join. since UFUF ~ U(FUF) ~ UF, and we can pass in the identity monoid homomorphism from [a] to [a] through the isomorphism that defines our adjunction,get that a list isomorphism from [a] → [a] is a function of type a -> [a], and this is just return for lists.

You can compose all of this more directly by describing a list in these terms with:

newtype List a = List (forall b. Monoid b => (a -> b) -> b)

***The Free Monad***

So what is a *Free Monad*?

Well, we do the same thing we did before, we start with a forgetful functor U from the category of monads where arrows are monad homomorphisms to a category of endofunctors where the arrows are natural transformations, and we look for a functor that is left adjoint to that.

So, how does this relate to the notion of a free monad as it is usually used?

Knowing that something is a free monad, Free f, tells you that giving a monad homomorphism from Free f -> m, is the same thing (isomorphic to) as giving a natural transformation (a functor homomorphism) from f -> m. Remember F a -> b must be isomorphic to a -> U b for F to be left adjoint to U. U here mapped monads to functors.

F is at least isomorphic to the Free type I use in my free package on hackage.

We could also construct it in tighter analogy to the code above for the free list, by defining

class Algebra f x where

phi :: f x -> x

newtype Free f a = Free (forall x. Algebra f x => (a -> x) -> x)

***Cofree Comonads***

We can construct something similar, by looking at the right adjoint to a forgetful functor assuming it exists. A cofree functor is simply /right adjoint/ to a forgetful functor, and by symmetry, knowing something is a cofree comonad is the same as knowing that giving a comonad homomorphism from w -> Cofree f is the same thing as giving a natural transformation from w -> f.