Perhaps Not The Answer You Were Expecting But You Asked For It

(An Accidental Blook)

Conor McBride

July 17, 2018

Contents

Chapter 1

Haskell Curiosities

1.1 What is () in Haskell, exactly?

tl;dr `()` does not add a "null" value to every type, hell no; `()` is a "dull" value in a type of its own: `()`.

Let me step back from the question a moment and address a common source of confusion. A key thing to absorb when learning Haskell is the distinction between its *expression* language and its *type* language. You're probably aware that the two are kept separate. But that allows the same symbol to be used in both, and that is what is going on here. There are simple textual cues to tell you which language you're looking at. You don't need to parse the whole language to detect these cues.

The top level of a Haskell module lives, by default, in the expression language. You define functions by writing equations between expressions. But when you see *foo* :: *bar* in the expression language, it means that *foo* is an expression and *bar* is its type. So when you read `() :: ()`, you're seeing a statement which relates the `()` in the expression language with the `()` in the type language. The two `()` symbols mean different things, because they are not in the same language. This repetition often causes confusion for beginners, until the expression/type language separation installs itself in their subconscious, at which point it becomes helpfully mnemonic.

The keyword `data` introduces a new datatype declaration, involving a careful mixture of the expression and type languages, as it says first what the new type is, and secondly what its values are.

In such a declaration, type constructor *TyCon* is being added to the type language and the *ValCon* value constructors are being added to the expression language (and its pattern sublanguage). In a `data` declaration, the things which stand in argument places for the *ValCon* s tell you the types given to the arguments when that *ValCon* is used in expressions. For example,

```
data Tree a = Leaf | Node (Tree a) a (Tree a)
```

declares a type constructor `Tree` for binary tree types storing elements at nodes, whose values are given by value constructors `Leaf` and `Node`. I like to colour type constructors (Tree) blue and value constructors (Leaf, Node) red. There should be no blue in expressions and (unless you're using advanced features) no red in types. The built-in type `Bool` could be declared,

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

adding blue `Bool` to the type language, and red `True` and `False` to the expression language. Sadly, my markdown-fu is inadequate to the task of adding the colours to this post, so you'll just have to learn to add the colours in your head.

The "unit" type uses `()` as a special symbol, but it works as if declared

```
data () = ()  -- the left () is blue; the right () is red
```

meaning that a notionally blue `()` is a type constructor in the type language, but that a notionally red `()` is a value constructor in the expression language, and indeed `() :: ()`. [It is not the only example of such a pun. The types of larger tuples follow the same pattern: pair syntax is as if given by

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

adding (,) to both type and expression languages. But I digress.

So the type `()`, often pronounced "Unit", is a type containing one value worth speaking of: that value is written `()` but in the expression language, and is sometimes pronounced "void". A type with only one value is not very interesting. A value of type `()` contributes zero bits of information: you already know what it must be. So, while there is nothing special about type `()` to indicate side effects, it often shows up as the value component in a monadic type. Monadic operations tend to have types which look like

where the return type is a type application: the function tells you which effects are possible and the argument tells you what sort of value is produced by the operation. For example

```
put :: s -> State s ()
```

which is read (because application associates to the left ["as we all did in the sixties", Roger Hindley]) as

```
put :: s -> (State s) ()
```

has one value input type `s`, the effect-monad `State s`, and the value output type `()`. When you see `()` as a value output type, that just means "this operation is used only for its *effect*; the value delivered is uninteresting". Similarly

```
putStr :: String -> IO ()
```

delivers a string to `stdout` but does not return anything exciting.

The `()` type is also useful as an element type for container-like structures, where it indicates that the data consists just of a *shape*, with no interesting payload. For example, if `Tree` is declared as above, then `Tree ()` is the type of binary tree shapes, storing nothing of interest at nodes. Similarly `[()]` is the type of lists of dull elements, and if there is nothing of interest in a list's elements, then the only information it contributes is its length.

To sum up, `()` is a type. Its one value, `()`, happens to have the same name, but that's ok because the type and expression languages are separate. It's useful to have a type representing "no information" because, in context (e.g., of a monad or a container), it tells you that only the context is interesting.

1.2 What does () mean in Haskell?

`()` means "Boring". It means the boring type which contains one thing, also boring. There is nothing interesting to be gained by comparing one element of the boring type with another, because there is nothing to learn about an element of the boring type by giving it any of your attention.

It is very different from the empty type, called (by people I wish had chosen a better name like the one I suggested) in Haskell `Void`. The empty type is very exciting, because if somebody ever gives you a value belonging to it, you know that you are already dead and in Heaven and that anything you want is yours.

But if somebody gives you a value in `()`, don't get excited. Just throw it away.

Sometimes it's fun to take type constructors parametrised by an "element type" and fill the parameter in with `()`. You can see just what information is inherent in the type constructor, rather than coming from the elements. E.g, `Maybe ()` is a version of `Bool`, with `Just ()` or

`Nothing`. Also, `[()]` amounts to the (possibly infinite) natural numbers: the only information you have is a length.

So, `()` means "Boring", but it's often a clue that something interesting is happening somewhere else.

1.3 Implement the function `lines` in Haskell

Suppose you had already knew the result of `lines` for all but the first character of your input. How would you add the first character onto that result?

```
charon :: Char -> [[Char]] -> [[Char]]
charon '\n'     css  = [] : css    -- to begin with newline, insert blank line
charon c         []  = [[c]]       -- very last char is in line of its own
charon c (cs : css)  = (c : cs) : css  -- else put char in first line
```

And with that mystery solved,

```
lines = foldr charon []
```

For years, I have made students bang their fists on the furniture and chant "*what* do you *do* with the *emp*ty *list*? *what* do you *do* with *x* cons *xs*?". Sometimes it helps.

1.4 Boolean Expression evaluation (subtle type error)

Your suspicion, "I think there is a problem in the data BExp declaration itself", is correct. What you've written doesn't mean what I suspect you hope it does. The error is at the far right end (as errors often are) so I had to scroll to find it. Using more vertical space, let's see.

```
data BExp
  = Eq AExp AExp
  | Lt AExp AExp
  | Gt AExp AExp
  | ELt AExp AExp
  | EGt AExp AExp
  | And BExp BExp
  | Or BExp BExp
  | Bool
```

And it's the last that is the big problem. It's harder to spot because although you tell *us*, "I want the type to be :: BExp -> Bool", you do not tell the compiler. If you had done the decent thing and communicated your intention by writing an explicit type signature, the error report might have been more helpful. Your program begins

```
evalBExp True = True
```

and that is enough to convince the typechecker that the intended type is

```
evalBExp :: Bool -> Bool
```

because `True :: Bool`. When line 3 shows up with

```
evalBExp (Eq a1 a2) = evalAExp (a1) == evalAExp (a2)
```

suddenly it wonders why you're giving `evalBExp` a `BExp` instead of a `Bool`.

Now, I suspect that you have the impression that your final clause in `BExp`

```
| Bool
```

makes `True :: BExp` and `False :: BExp`, but that's not what it does at all. Instead, you will discover that you have a nullary data constructor `Bool :: BExp` whose name is coincident with the datatype `Bool` but lives in an entirely separate namespace. I believe your intention is to embed the values of `Bool` into `BExp` silently, but Haskell does not allow such subtle subtyping. To achieve the intended effect, you will need a constructor which explicitly packs up a `Bool`, so try

```
data BExp
  = ...
  | BVal Bool
```

and

```
evalBExp :: BExp -> Bool
evalBExp (BVal b) = b
...
```

to get closer to your plan.

You are, of course, free to use the name `Bool` for the `BVal` constructor, and so write

```
data BExp
  = ...
  | Bool Bool
```

where the first `Bool` is a data constructor and the second is a type constructor, but I would find that choice confusing.

1.5 Purely functional data structures for text editors

I don't know whether this suggestion is "good" for sophisticated definitions of "good", but it's easy and fun. I often set an exercise to write the core of a text editor in Haskell, linking with rendering code that I provide. The data model is as follows.

First, I define what it is to be a cursor inside a list of `x`-elements, where the information available at the cursor has some type `m`. (The `x` will turn out to be `Char` or `String`.)

```
type Cursor x m = (Bwd x, m, [x])
```

This `Bwd` thing is just the backward "snoc-lists". I want to keep strong spatial intuitions, so I turn things around in my code, not in my head. The idea is that the stuff nearest the cursor is the most easily accessible. That's the spirit of The Zipper.

```
data Bwd x = B0 | Bwd x :< x deriving (Show, Eq)
```

I provide a gratuitous singleton type to act as a readable marker for the cursor...

```
data Here = Here deriving Show
```

...and I can thus say what it is to be somewhere in a `String`

```
type StringCursor = Cursor Char Here
```

Now, to represent a buffer of multiple lines, we need `String`s above and below the line with the cursor, and a `StringCursor` in the middle, for the line we're currently editing.

```
type TextCursor = Cursor String StringCursor
```

This `TextCursor` type is all I use to represent the state of the edit buffer. It's a two layer zipper. I provide the students with code to render a viewport on the text in an ANSI-escape-enabled shell window, ensuring that the viewport contains the cursor. All they have to do is implement the code that updates the `TextCursor` in response to keystrokes.

```
handleKey :: Key -> TextCursor -> Maybe (Damage, TextCursor)
```

where `handleKey` should return `Nothing` if the keystroke is meaningless, but otherwise deliver `Just` an updated `TextCursor` and a "damage report", the latter being one of

```
data Damage
  = NoChange       -- use this if nothing at all happened
  | PointChanged   -- use this if you moved the cursor but kept the text
  | LineChanged    -- use this if you changed text only on the current line
  | LotsChanged    -- use this if you changed text off the current line
  deriving (Show, Eq, Ord)
```

(If you're wondering what the difference is between returning `Nothing` and returning `Just (NoChange, ...)`, consider whether you also want the editor to go beep.) The damage report tells the renderer how much work it needs to do to bring the displayed image up to date.

The `Key` type just gives a readable datatype representation to the possible keystrokes, abstracting away from the raw ANSI escape sequences. It's unremarkable.

I provide the students with a big clue about to go up and down with this data model by offering these pieces of kit:

```
deactivate :: Cursor x Here -> (Int, [x])
deactivate c = outward 0 c where
  outward i (B0, Here, xs)       = (i, xs)
  outward i (xz :< x, Here, xs)  = outward (i + 1) (xz, Here, x : xs)
```

The `deactivate` function is used to shift focus out of a `Cursor`, giving you an ordinary list, but telling you where the cursor *was*. The corresponding `activate` function attempts to place the cursor at a given position in a list:

```
activate :: (Int, [x]) -> Cursor x Here
activate (i, xs) = inward i (B0, Here, xs) where
  inward _ c@(_, Here, [])     = c  -- we can go no further
  inward 0 c                   = c  -- we should go no further
  inward i (xz, Here, x : xs)  = inward (i - 1) (xz :< x, Here, xs)  -- and on!
```

I offer the students a deliberately incorrect and incomplete definition of `handleKey`

```
handleKey :: Key -> TextCursor -> Maybe (Damage, TextCursor)
handleKey (CharKey c)  (sz,
                        (cz, Here, cs),
                        ss)
  = Just (LineChanged, (sz,
                        (cz, Here, c : cs),
                        ss))
handleKey _ _ = Nothing
```

which just handles ordinary character keystrokes but makes the text come out backwards. It's easy to see that the character `c` appears *right* of `Here`. I invite them to fix the bug and add functionality for the arrow keys, backspace, delete, return, and so on.

It may not be the most efficient representation ever, but it's purely functional and enables the code to conform concretely to our spatial intuitions about the text that's being edited.

1.6 What are some motivating examples for Cofree CoMonad in Haskell?

Let's just recap the definition of the `Cofree` datatype.

```
data Cofree f a = a :< f (Cofree f a)
```

That's at least enough to diagnose the problem with the example. When you wrote

```
1 :< [2, 3]
```

you made a small error that's reported rather more subtly than is helpful. Here, `f = []` and `a` is something numeric, because `1 :: a`. Correspondingly you need

```
[2, 3] :: [Cofree [] a]
```

and hence

```
2 :: Cofree [] a
```

which *could* be ok if `Cofree [] a` were also and instance of `Num`. Your definition thus acquires a constraint which is unlikely to be satisfied, and indeed, when you *use* your value, the attempt to satisfy the constraint fails.

Try again with

```
1 :< [2 :< [], 3 :< []]
```

and you should have better luck.

Now, let's see what we've got. Start by keeping it simple. What's `Cofree f ()`? What, in particular, is `Cofree [] ()`? The latter is isomorphic to the fixpoint of `[]`: the tree structures where each node is a list of subtrees, also known as "unlabelled rose trees". E.g.,

```
() :< [  () :< [  () :< []
                ,  () :< []
                ]
      ,  () :< []
      ]
```

Similarly, `Cofree Maybe ()` is more or less the fixpoint of `Maybe`: a copy of the natural numbers, because `Maybe` gives us either zero or one position into which to plug a subtree.

```
zero :: Cofree Maybe ()
zero = () :< Nothing
succ :: Cofree Maybe () -> Cofree Maybe ()
succ n = () :< Just n
```

An important trivial case is `Cofree (Const y) ()`, which is a copy of `y`. The `Const y` functor gives *no* positions for subtrees.

```
pack :: y -> Cofree (Const y) ()
pack y = () :< Const y
```

Next, let's get busy with the other parameter. It tells you what sort of label you attach to each node. Renaming the parameters more suggestively

```
data Cofree nodeOf label = label :< nodeOf (Cofree nodeOf label)
```

When we label up the `(Const y)` example, we get *pairs*

```
pair :: x -> y -> Cofree (Const y) x
pair x y = x :< Const y
```

When we attach labels to the nodes of our numbers, we get *nonempty* lists

```
one :: x -> Cofree Maybe x
one = x :< Nothing
cons :: x -> Cofree Maybe x -> Cofree Maybe x
cons x xs = x :< Just xs
```

And for lists, we get *labelled* rose trees.

```
0 :< [  1 :< [  3 :< []
             ,  4 :< []
             ]
     ,  2 :< []
     ]
```

These structures are always "nonempty", because there is at least a top node, even if it has no children, and that node will always have a label. The `extract` operation gives you the label of the top node.

```
extract :: Cofree f a -> a
extract (a :< _) = a
```

That is, `extract` throws away the *context* of the top label.

Now, the `duplicate` operation *decorates* every label with *its own* context.

```
duplicate :: Cofree f a -> Cofree f (Cofree f a)
duplicate a :< fca = (a :< fca) :< fmap duplicate fca  -- f's fmap
```

We can get a `Functor` instance for `Cofree f` by visiting the whole tree

```
fmap :: (a -> b) -> Cofree f a -> Cofree f b
fmap g (a :< fca) = g a :< fmap (fmap g) fca
    --                      ^^^^  ^^^^
    --                f's fmap  ||||
    --                          (Cofree f)'s fmap, used recursively
```

It's not hard to see that

```
fmap extract . duplicate = id
```

because `duplicate` decorates every node with its context, then `fmap extract` throws away the decoration.

Note that `fmap` gets to look only at the labels of the input to compute the labels of the output. Suppose we wanted to compute output labels depending on each input label *in its context*? E.g., given an unlabelled tree, we might want to label each node with the size of its entire subtree. Thanks to the `Foldable` instance for `Cofree f`, we should be able to count nodes with.

```
length :: Foldable f => Cofree f a -> Int
```

So that means

```
fmap length . duplicate :: Cofree f a -> Cofree f Int
```

The key idea of comonads is that they capture "things with some context", and they let you apply context-dependent maps everywhere.

```
extend :: Comonad c => (c a -> b) -> c a -> c b
extend f = fmap f       -- context-dependent map everywhere
           .            -- after
           duplicate    -- decorating everything with its context
```

Defining `extend` more directly saves you the trouble of duplication (although that amounts only to sharing).

```
extend :: (Cofree f a -> b) -> Cofree f a -> Cofree f b
extend g ca@(_ :< fca) = g ca :< fmap (extend g) fca
```

And you can get `duplicate` back by taking

```
duplicate = extend id -- the output label is the input label in its context
```

Moreover, if you pick `extract` as the thing to do to each label-in-context, you just put each label back where it came from:

```
extend extract = id
```

These "operations on labels-in-context" are called "co-Kleisli arrows",

```
g :: c a -> b
```

and the job of `extend` is to interpret a co-Kleisli arrow as a function on whole structures. The `extract` operation is the identity co-Kleisli arrow, and it's interpreted by `extend` as the identity function. Of course, there is a co-Kleisli composition

```
(=<=) :: Comonad c => (c s -> t) -> (c r -> s) -> (c r -> t)
(g =<= h) = g . extend h
```

and the comonad laws ensure that `=<=` is associative and absorbs `extract`, giving us the co-Kleisli category. Moreover we have

```
extend (g =<= h)  =  extend g . extend h
```

so that `extend` is a *functor* (in the categorical sense) from the co-Kleisli category to sets-and-functions. These laws are not hard to check for `Cofree`, as they follow from the `Functor` laws for the node shape.

Now, one useful way to see a structure in a cofree comonad is as a kind of "game server". A structure

```
a :< fca
```

represents the state of the game. A move in the game consists either of "stopping", in which case you get the `a`, or of "continuing", by choosing a subtree of the `fca`. For example, consider

```
Cofree ((->) move) prize
```

A client for this server must either stop, or continue by giving a `move`: it's a *list* of `move`s. The game is played out as follows:

```
play :: [move] -> Cofree ((->) move) prize -> prize
play []       (prize :< _) = prize
play (m : ms) (_     :< f) = play ms (f m)
```

Perhaps a `move` is a `Char` and the `prize` is the result of parsing the character sequence.

If you stare hard enough, you'll see that `[move]` is a version of `Free ((,) move) ()`. Free monads represent client strategies. The functor `((,) move)` amounts to a command interface with only the command "send a `move`". The functor `((->) move)` is the corresponding structure "respond to the sending of a `move`".

Some functors can be seen as capturing a command interface; the free monad for such a functor represents programs that make commands; the functor will have a "dual" which represents how to respond to commands; the cofree comonad of the dual is the general notion of environment in which programs that make commands can be run, with the label saying what to do if the program stops and returns a value, and the substructures saying how to carry on running the program if it issues a command.

For example,

```
data Comms x = Send Char x | Receive (Char -> x)
```

describes being allowed to send or receive characters. Its dual is

```
data Responder x = Resp {ifSend :: Char -> x, ifReceive :: (Char, x)}
```

As an exercise, see if you can implement the interaction

```
chatter :: Free Comms x -> Cofree Responder y -> (x, y)
```

1.7 Find indices of things in lists

I'd use `zip` and a list comprehension.

```
indicesOf :: Eq a => a -> [a] -> [Int]
indicesOf a as = [i | (b, i) <- zip as [0..], b == a]
```

Zipping with `[0..]` is a standard way to label every element with an index, and then it's a simple query.

1.8 How do I extend this mergeWords function to any number of strings?

The puzzle is effectively to merge a *list* of words, a character at a time, into lines with trailing newline characters.

```
mergeWords :: [String] -> String
```

We need to take a list like

```
[ "hello"
, "jim"
, "nice"
, "day"
]
```

and rearrange it into the lists of things at a given position

```
[ "hjnd"
, "eiia"
, "lmcy"
, "le"
, "o"
]
```

That's what the library function `transpose` does.

And then we need to make a single string which treats those as lines, separated by newlines. Which is what `unlines` does.

So

```
mergeWords = unlines . transpose
```

and we're done.

1.9 Example of UndecidableInstances yielding nonterminating typecheck

There's a classic, somewhat alarming example (involving interaction with functional dependencies) in this paper from HQ:

```
class Mul a b c | a b -> c where
  mul :: a -> b -> c
instance Mul a b c => Mul a [b] [c] where
  mul a = map (mul a)

f b x y = if b then mul x [y] else y
```

We need `mul x [y]` to have the same type as `y`, so, taking `x :: x` and `y :: y`, we need

```
instance Mul x [y] y
```

which, according to the given instance, we can have. Of course, we must take `y ~ [z]` for some `z` such that

```
instance Mul x y z
```

i.e.

```
instance Mul x [z] z
```

and we're in a loop.

It's disturbing, because that `Mul` instance looks like its recursion is structurally *reducing*, unlike the clearly pathological instance in Petr's answer. Yet it makes GHC loop (with the boredom threshold kicking in to avoid hanging).

The trouble, as I'm sure I've mentioned somewhere somewhen, is that the identification `y ~ [z]` is made in spite of the fact that `z` depends functionally on `y`. If we used a functional notation for the functional dependency, we might see that the constraint says `y ~ Mul x [y]` and reject the substitution as in violation of the occurrence check.

Intrigued, I thought I'd give this a whirl,

```
class Mul' a b where
  type MulT a b
  mul' :: a -> b -> MulT a b

instance Mul' a b => Mul' a [b] where
  type MulT a [b] = [MulT a b]
  mul' a = map (mul' a)

g b x y = if b then mul' x [y] else y
```

With `UndecidableInstances` enabled, it takes quite a while for the loop to time out. With `UndecidableInstances` disabled, the instance is still accepted and the typechecker still loops, but the cutoff happens much more quickly.

So… funny old world.

1.10 Why do 3 and x (which was assigned 3) have different inferred types in Haskell?

There's another factor here, mentioned in some of the links which acfoltzer includes, but it might be worth making explicit here. You're encountering the effect of the monomorphism restriction. When you say

```
let x = 5
```

you make a *top-level* definition of a *variable*. The MR insists that such definitions, when otherwise unaccompanied by a type signature, should be specialized to a monomorphic value by choosing (hopefully) suitable default instances for the unresolved type variables. By contrast, when you use `:t` to ask for an inferred type, no such restriction or defaulting is imposed. So

```
> :t 3
3 :: (Num t) => t
```

because `3` is indeed overloaded: it is admitted by any numeric type. The defaulting rules choose `Integer` as the default numeric type, so

```
> let x = 3
> :t x
x :: Integer
```

But now let's switch off the MR.

```
> :set -XNoMonomorphismRestriction
> let y = 3
> :t y
y :: (Num t) => t
```

Without the MR, the definition is just as polymorphic as it can be, just as overloaded as `3`. Just checking…

```
> :t y * (2.5 :: Float)
y * (2.5 :: Float) :: Float
> :t y * (3 :: Int)
y * (3 :: Int) :: Int
```

Note that the polymorphic `y = 3` is being differently specialized in these uses, according to the `fromInteger` method supplied with the relevant `Num` instance. That is, `y` is not associated with a particular representation of `3`, but rather a scheme for constructing representations of `3`. Naïvely compiled, that's a recipe for slow, which some people cite as a motivation for the MR.

I'm (locally pretending to be) neutral on the debate about whether the monomorphism restriction is a lesser or greater evil. I always write type signatures for top-level definitions, so there is no ambiguity about what I'm trying to achieve and the MR is beside the point.

When trying to learn how the type system works, it's really useful to separate the aspects of type inference which

1. 'follow the plan', specializing polymorphic definitions to particular use cases: a fairly robust matter of constraint-solving, requiring basic unification and instance resolution by backchaining; and
2. 'guess the plan', generalizing types to assign a polymorphic type scheme to a definition with no type signature: that's quite fragile, and the more you move past the basic Hindley-Milner discipline, with type classes, with higher-rank polymorphism, with GADTs, the stranger things become.

It's good to learn how the first works, and to understand why the second is difficult. Much of the weirdness in type inference is associated with the second, and with heuristics like the monomorphism restriction trying to deliver useful default behaviour in the face of ambiguity.

1.11 Use case for rank-3 (or higher) polymorphism?

I may be able to help, although such beast are inevitably a bit involved. Here's a pattern I sometimes use in developing well-scoped syntax with binding and de Bruijn indexing, bottled.

```
mkRenSub ::
  forall v t x y.                          -- variables represented by v, terms by t
    (forall x. v x -> t x) ->                -- how to embed variables into terms
    (forall x. v x -> v (Maybe x)) ->        -- how to shift variables
    (forall i x y.                           -- for thingies, i, how to traverse terms...
      (forall z. v z -> i z) ->                 -- how to make a thingy from a variable
      (forall z. i z -> t z) ->                 -- how to make a term from a thingy
      (forall z. i z -> i (Maybe z)) ->         -- how to weaken a thingy
      (v x -> i y) ->                        -- ...turning variables into thingies
      t x -> t y) ->                         -- wherever they appear
    ((v x -> v y) -> t x -> t y, (v x -> t y) -> t x -> t y)
                                                -- acquire renaming and substitution
mkRenSub var weak mangle = (ren, sub) where
  ren = mangle id var weak         -- take thingies to be vars to get renaming
  sub = mangle var id (ren weak)   -- take thingies to be terms to get substitution
```

Normally, I'd use type classes to hide the worst of the gore, but if you unpack the dictionaries, this is what you'll find.

The point is that `mangle` is a rank-2 operation which takes a notion of thingy equipped with suitable operations polymorphic in the variable sets over which they work: operations which map variables to thingies get turned into term-transformers. The whole thing shows how to use `mangle` to generate both renaming and substitution.

Here's a concrete instance of that pattern:

```
data Id x = Id x

data Tm x
  = Var (Id x)
  | App (Tm x) (Tm x)
  | Lam (Tm (Maybe x))

tmMangle :: forall i x y.
             (forall z. Id z -> i z) ->
             (forall z. i z -> Tm z) ->
             (forall z. i z -> i (Maybe z)) ->
             (Id x -> i y) -> Tm x -> Tm y
tmMangle v t w f (Var i) = t (f i)
tmMangle v t w f (App m n) = App (tmMangle v t w f m) (tmMangle v t w f n)
tmMangle v t w f (Lam m) = Lam (tmMangle v t w g m) where
  g (Id Nothing) = v (Id Nothing)
  g (Id (Just x)) = w (f (Id x))

subst :: (Id x -> Tm y) -> Tm x -> Tm y
subst = snd (mkRenSub Var (\ (Id x) -> Id (Just x)) tmMangle)
```

We implement the term traversal just once, but in a very general way, then we get substitution by deploying the mkRenSub pattern (which uses the general traversal in two different ways).

For another example, consider polymorphic operations between type operators

```
type (f :-> g) = forall x. f x -> g x
```

An `IMonad` (indexed monad) is some `m :: (* -> *) -> * -> *` equipped with polymorphic operators

```
ireturn :: forall p. p :-> m p
iextend :: forall p q. (p :-> m q) -> m p :-> m q
```

so those operations are rank 2.

Now any operation which is parametrized by an arbitrary indexed monad is rank 3. So, for example, constructing the usual monadic composition,

```
compose :: forall m p q r. IMonad m => (q :-> m r) -> (p :-> m q) -> p :-> m r
compose qr pq = iextend qr . pq
```

relies on rank 3 quantification, once you unpack the definition of `IMonad`.

Moral of story: once you're doing higher order programming over polymorphic/indexed notions, your dictionaries of useful kit become rank 2, and your generic programs become rank 3. This is, of course, an escalation that can happen again.

1.12 Why don't Haskell compilers facilitate deterministic memory management?

It is sensible to ask whether functional programming languages can do less GC by tracking usage. Although the general problem of whether some data can safely be discarded is undecidable (because of conditional branching), it's surely plausible to work harder statically and find more opportunities for direct deallocation.

It's worth paying attention to the work of Martin Hofmann and the team on the Mobile Resource Guarantees project, who made type-directed memory (de/re)allocation a major theme. The thing that makes their stuff work, though, is something Haskell doesn't have in its type system — linearity. If you know that a function's input data are *secret* from the rest of the computation, you can reallocate the memory they occupy. The MRG stuff is particularly nice because it manages a realistic exchange rate between deallocation for one type and allocation for another which turns into good old-fashioned pointer-mangling underneath a purely functional exterior. In fact, lots of lovely parsimonious mutation algorithms (e.g. pointer-reversing traversal, overwrite-the-tail-pointer construction, etc) can be made to look purely functional (and checked for nasty bugs) using these techniques.

In effect, the linear typing of resources gives a conservative but mechanically checkable approximation to the kind of usage analysis that might well help to reduce GC. Interesting questions then include how to mix this treatment cleanly (deliberate adverb choice) with the usual persistent deal. It seems to me that quite a lot of intermediate data structures has an initial single-threaded phase in recursive computation, before being either shared or dropped when the computation finishes. It may be possible to reduce the garbage generated by such processes.

TL;DR There are good typed approaches to usage analysis which cut GC, but Haskell has the wrong sort of type information just now to be particularly useful for this purpose.

1.13 How does ArrowLoop work? Also, mfix?

In this code, they key piece is the `delay 0` arrow in the `rec` block. To see how it works, it helps to think of values as varying over time and time as chopped into slices. I think of the slices as 'days'. The `rec` block explains how each day's computation works. It's organised by *value*, rather than by *causal order*, but we can still track causality if we're careful. Crucially, we must make sure (without any help from the *types*) that each day's work relies on the past but not the future. The one-day `delay 0` buys us time in that respect: it shifts its input signal one day later, taking care of the first day by giving the value 0. The delay's input signal is 'tomorrow's `next`'.

```
rec     output <- returnA -< if reset then 0 else next
        next <- delay 0 -< output+1
```

So, looking at the arrows and their outputs, we're delivering *today's* `output` but *tomorrow's* `next`. Looking at the inputs, we're relying on *today's* `reset` and `next` values. It's clear that we can deliver those outputs from those inputs without time travel. The `output` is today's `next` number unless we `reset` to 0; tomorrow, the `next` number is the successor of today's `output`. Today's `next` value thus comes from yesterday, unless there was no yesterday, in which case it's 0.

At a lower level, this whole setup works because of Haskell's laziness. Haskell computes by a demand-driven strategy, so if there is a sequential order of tasks which respects causality, Haskell will find it. Here, the `delay` establishes such an order.

Be aware, though, that Haskell's type system gives you very little help in ensuring that such an order exists. You're free to use loops for utter nonsense! So your question is far from trivial. Each time you read or write such a program, you do need to think 'how can this possibly work?'. You need to check that `delay` (or similar) is used appropriately to ensure that information is demanded only when it can be computed. Note that *constructors*, especially `(:)` can act like delays, too: it's not unusual to compute the tail of a list, apparently given the whole list (but being careful only to inspect the head). Unlike imperative programming, the lazy functional style allows you to organize your code around concepts other than the sequence of events, but it's a freedom that demands a more subtle awareness of time.

1.14 What does ⇒ mean in a type signature?

Here's another way of looking at it. Some of a function's arguments are invisible, others are visible. A type `input -> output` tells us that a visible `input` is expected as an argument. A type `(Constraint) => output` tells us that some invisible information is expected. They're not interchangeable, because the visible arguments must be written, and the invisible arguments must not be written. The invisible arguments are for the compiler to figure out for himself (well, he sounds like a himself to me), and he insists on puzzling them out: he refuses just to be told what they are!

Secretly, the full type of this `tell` example is

```
tell :: forall (a :: *). (Show a) => [a] -> String
```

What I've done is to make clear where this `a` variable comes in and what kind of a thing it is. You can also read this as a "deal": `tell` offers to work for all types `a` which satisfy the demand `(Show a)`.

In order for a usage of `tell` to make sense, it needs three things. Two of them are invisible and one is visible. That is, when you use `tell`, you make the visibile argument explicit, and the compiler tries to fill in the invisible parts. Let's work through that type more slowly.

```
tell :: forall (a :: *).   -- the type of elements to tell          (invisible)
        (Show a) =>        -- how to make a String from one element (invisible)
        [a] ->             -- the list of elements to be told       (visible)
        String             -- the String made by showing all the elements
```

So, when you use `tell`, e.g.,

```
tell [True, False]
```

you give only the visible argument: the list `[True, False]` of things to tell, and the compiler figures out the invisible arguments. He knows that `True` and `False` are both values of type `Bool`, so that means

```
[True, False] :: [Bool]
```

which is how the compiler figures out that the `a` in the type of `tell` must be `Bool`, making `[a] = [Bool]`

(By the way, about `[True, False] :: [Bool]`. Left of ::, the square brackets, [..], make list values. Right of ::, the square brackets, [..], make a type of lists. They may just look black on a grey background to you, but my brain colours the value-making brackets red and the type-making brackets blue. They are entirely different. I wish I could colour code on this site. I digress.)

So, now, the other invisible argument must satisfy this `(Show a)` thing, which we now know is specifically `(Show Bool)` because we figured out that `a` is `Bool`. We call this part of the type a "constraint", but really it's a demand not just that a fact is true, but that some useful stuff exists. The stuff that is being demanded here is that there is a function

```
show :: Bool -> String
```

That's the function used to turn the individual elements `True` and `False` into `String`s in the process of evaluating `tell [True, False]`.

The identifier `Show` is the name of a *type class* and *show* is the method of that type class. A type `class` specifies an interface of operations which must be implemented for each `instance`. The invisible argument to the function is a record (or "dictionary") packaging the implementations of those operations for the type in question (here, the implementation of `show`). Without that information, the compiled code would have no way to do its job, but we don't have to write that information, because the compiler can (at least in this case) search through the instances it knows about and fill in the right one for the job.

So, we don't just have invisible type arguments (which are inferred at compile time, then erased before run time), signalled by lowercase type variables, or more explicitly by `forall` *blah* `..` We also have invisible implementations of type class operations (which are looked up at compile time, to provide vital run time information). So, something very important happens *between inferring and erasing types*: while the compiler still knows the types, it uses them to figure out which invisible implementations will be needed at run time, so that we can get away with not writing them ourselves.

Zooming out, => in a type documents our expectation that the compiler will make use of type information to guide the generation of run time code that we don't have to bother writing. That's a nice little bit of win, right there.

Ulterior motive for type system hackers. The message that the invisible-visible distinction is in a different place from the erasable-useful distinction is one that some people have not yet received. That is the classic Hindley-Milner position, but the fact is that these distinctions are orthogonal, and the sooner people learn to enjoy that, the better.

1.15 Meaning of `Double` and `Floating` point?

A useful thing to try interactively in `ghci` is the `:info <something>` command, which can sometimes tell you helpful things.

```
> :info Floating
class Fractional a => Floating a where
  pi :: a
  exp :: a -> a
  log :: a -> a
  sqrt :: a -> a
  (**) :: a -> a -> a
  ---------------------------------------- loads more stuff
        -- Defined in 'GHC.Float'
```

```
instance Floating Float -- Defined in 'GHC.Float'
instance Floating Double -- Defined in 'GHC.Float'
```

What does this mean? `Floating` is a *type class*. There is more than one type of floating point numbers. Indeed, two come as standard: `Float` and `Double`, where `Double` gives you twice the precision of `Float`. The `a` in `Floating a` stands for any type, and the big list of operations (including `sqrt`) is an interface which any instance of the class must implement. The fact that `sqrt` is in the interface for `Floating` means that it can always and only be used for instances of `Floating`. That is, to you its type is given as you say

```
sqrt :: Floating a => a -> a
```

The => syntax signals a *constraint*, here `Floating a` to its left. The type says

> for any type `a` which is an instance of `Floating`, given an input of type `a`, the output will have type `a`

You can specialize this type by filling in `a` with any type for which the constraint `Floating a` can be satisfied, so the following are both true

```
sqrt :: Float -> Float
sqrt :: Double -> Double
```

Now, `Float` and `Double` are represented by different bit-patterns, so the computational mechanisms for taking square roots is different in each case. It's handy not to have to remember different names for the different versions used for different types. The "constraint" `Floating a` really stands for the record (or *dictionary*) of the implementations for type `a` of all the operations in the interface. What the type of `sqrt` is really saying is

> given a type `a` and a dictionary of implementations for all the `Floating` operations, I'll take an `a` and give you an `a`

and it works by extracting the relevant `sqrt` implementation from the dictionary and using it on the given input.

So the => signals a function type with an invisible dictionary input just as -> signals a function type with a visible value input. You don't (indeed, you can't) write the dictionary when you use the function: the compiler figures it out from the type. When we write

```
sqrt :: Double -> Double
```

we mean the general `sqrt` function invisibly applied to the `Floating Double` dictionary.

1.16 Haskell terminology: meaning of type vs. data type, are they synonyms?

A **type** (in Haskell) is a piece of syntax which can meaningfully be put right of `::` to classify an expression left of `::`. Each syntactic component of a type is itself classified by a **kind**, where the kind of types (which classify expressions) is `*`. *Some people are happy to use the word "type" to refer to any component of the type syntax, whether or not its kind allows it to classify expressions.*

The syntax of types can be extended by various declaration forms.

1. A **type synonym**, e.g., `type Foo x y z = [x] -> IO (y, z)`, adds type components of fully applied form `Foo` *x y* `z`, which expand macro-fashion in accordance with their defining equation.

2. A **data** declaration, e.g., `data Goo x y z = ThisGoo x | ThatGoo (Goo y z x)` introduces a fresh **type constructor** symbol `Goo` to the syntax of types, which is used to build the types which classify values generated by the **data constructors**, here `ThisGoo` and `ThatGoo`.
3. A **newtype** declaration, e.g., `newtype Noo x y z = MkNoo (x, [y], z)` makes a copy of an existing type which is distinguished from the original in the syntax of types.

A type is **polymorphic** if it contains **type variables** which can be substituted with other type components: the values classified by polymorphic types can be **specialized** to any substitution instance of the type variables. E.g. append `(++) :: [a] -> [a] -> [a]` works on lists whose elements have the same type, but any type will do. Values with polymorphic types are often referred to as "polymorphic values".

Sometimes, "data type" is used to mean, quite simply, a type introduced by a `data` declaration. In this sense, all data types are types, but not all types are data types. Examples of types which are not data types include `IO ()` and `Int -> Int`. Also, `Int` is not a data type in *this* sense: it is a hardwired **primitive** type. For the avoidance of doubt, some people call these types **algebraic** data types, because the constructors give an algebra, meaning "a bunch of operations for building values by combining other values". A "polymorphic data type" is a data type with type variables in it, such as `[(a, Bool)]`, by contrast with `[Int]`. Sometimes people talk about "declaring a polymorphic data type" or say things like "`Maybe` is a polymorphic data type" when they really just mean that the type constructor has parameters (and can thus be used to form polymorphic types): pedantically, one does declare a polymorphic data type, but not any old polymorphic datatype, rather a type constructor applied to formal parameters).

Of course, all first-class values classified by types are in some sense "data", and in Haskell, types are not used to classify anything which is not a first-class value, so in that looser sense, every "type" is a "data type". The distinction becomes more meaningful in languages where there are things other than data which have types (e.g., methods in Java).

Informal usage is often somewhere in the middle and not very well defined. People are often driving at some sort of distinction between functions or processes and the sort of stuff (the "data") on which they operate. Or they might think of data as being "understood in terms of the way they're made" (and exposing their representation, e.g. by pattern matching) rather than "understood in terms of the way they're used". This last usage of "data" sits a little uncomfortably with the notion of an **abstract** data type, being a type which hides the representation of the underlying stuff. Representation-hiding abstract data types thus contrast rather strongly with representation-exposing algebraic data types, which is why it is rather unfortunate that "ADT" is casually used as an abbreviation for both.

The upshot, I'm afraid, is vague.

1.17 Can you formulate the Bubble sort as a monoid or semigroup?

I'm using my phone with a poor network connection, but here goes.

tl;dr bubblesort is insertion sort is the monoidal "crush" for the monoid of ordered lists with merging.

Ordered lists form a monoid.

```
newtype OL x = OL [x]
instance Ord x => Monoid (OL x) where
  mempty = OL []
  mappend (OL xs) (OL ys) = OL (merge xs ys) where
    merge [] ys = ys
    merge xs [] = xs
    merge xs@(x : xs') ys@(y : ys')
```

```
        | x <= y = x : merge xs' ys
        | otherwise = y : merge xs ys'
```

Insertion sort is given by

```
isort :: Ord x => [x] -> OL x
isort = foldMap (OL . pure)
```

because insertion is exactly merging a singleton list with another list. (Mergesort is given by building a balanced tree, then doing the same foldMap.)

What has this to do with bubblesort? Insertion sort and bubblesort have exactly the same comparison strategy. You can see this if you draw it as a sorting network made from compare-and-swap boxes. Here, data flows downward and lower inputs to boxes [n] go left:

```
| | | |
[1] | |
| [2] |
[3] [4]
| [5] |
[6] | |
| | | |
```

If you perform the comparisons in the sequence given by the above numbering, cutting the diagram in / slices, you get insertion sort: the first insertion needs no comparison; the second needs comparison 1; the third 2,3; the last 4,5,6.

But if, instead, you cut in slices…

```
| | | |
[1] | |
| [2] |
[4] [3]
| [5] |
[6] | |
| | | |
```

…you are doing bubblesort: first pass 1,2,3; second pass 4,5; last pass 6.

1.18 Is this a correctly implemented mergesort in Haskell?

No, that's not **mergeSort**. That's **insertionSort**, which is essentially the same algorithm as **bubbleSort**, depending on how you stare at it. At each step, a singleton list is `merge`d with the accumulated ordered-list-so-far, so, effectively, the element of that singleton is inserted.

As other commenters have already observed, to get **mergeSort** (and in particular, its efficiency), it's necessary to divide the problem repeatedly into roughly equal parts (rather than "one element" and "the rest"). The "official" solution gives a rather clunky way to do that. I quite like

```
foldr (\ x (ys, zs) -> (x : zs, ys)) ([], [])
```

as a way to split a list in two, not in the middle, but into elements in even and odd positions.

If, like me, you like to have structure up front where you can see it, you can make ordered lists a `Monoid`.

```
import Data.Monoid
import Data.Foldable
import Control.Newtype

newtype Merge x = Merge {merged :: [x]}
instance Newtype (Merge x) [x] where
  pack = Merge
  unpack = merged

instance Ord x => Monoid (Merge x) where
  mempty = Merge []
  mappend (Merge xs) (Merge ys) = Merge (merge xs ys) where
    -- merge is as you defined it
```

And now you have insertion sort just by

```
ala' Merge foldMap (:[]) :: [x] -> [x]
```

One way to get the divide-and-conquer structure of mergeSort is to make it a data structure: binary trees.

```
data Tree x = None | One x | Node (Tree x) (Tree x) deriving Foldable
```

I haven't enforced a balancing invariant here, but I could. The point is that the same operation as before has another type

```
ala' Merge foldMap (:[]) :: Tree x -> [x]
```

which merges lists collected from a treelike arrangement of elements. To obtain said arrangements, think "what's cons for `Tree`?" and make sure you keep your balance, by the same kind of twistiness I used in the above "dividing" operation.

```
twistin :: x -> Tree x -> Tree x   -- a very cons-like type
twistin x None        = One x
twistin x (One y)     = Node (One x) (One y)
twistin x (Node l r)  = Node (twistin x r) l
```

Now you have mergeSort by building a binary tree, then merging it.

```
mergeSort :: Ord x => [x] -> [x]
mergeSort = ala' Merge foldMap (:[]) . foldr twistin None
```

Of course, introducing the intermediate data structure has curiosity value, but you can easily cut it out and get something like

```
mergeSort :: Ord x => [x] -> [x]
mergeSort []   = []
mergeSort [x]  = [x]
mergeSort xs   = merge (mergeSort ys) (mergeSort zs) where
  (ys, zs) = foldr (\ x (ys, zs) -> (x : zs, ys)) ([], []) xs
```

where the tree has become the recursion structure of the program.

1.19 Haskell type system nuances (ambiguity)

This is a variant of the notorious `show . read` problem. The classic version uses

```
read :: Read a => String -> a
show :: Show a => a -> String
```

so the composition might seem to be a plain old String transducer

```
moo :: String -> String
moo = show . read
```

except that there is no information in the program to determine the type in the middle, hence what to `read` and then `show`.

```
Ambiguous type variable 'b' in the constraints:
  'Read b' arising from a use of 'read' at ...
  'Show b' arising from a use of 'show' at ...
Probable fix: add a type signature that fixes these type variable(s)
```

Please not that ghci does a bunch of crazy extra defaulting, resolving ambiguity arbitrarily.

```
> (show . read) "()"
"()"
```

Your `C` class is a variant of `Read`, except that it decodes an `Int` instead of reading a `String`, but the problem is essentially the same.

Type system enthusiasts will note that underconstrained type variables are not *per se* a big deal. It's ambiguous *instance inference* that's the issue here. Consider

```
poo :: String -> a -> a
poo _ = id

qoo :: (a -> a) -> String
qoo _ = ""

roo :: String -> String
roo = qoo . poo
```

In the construction of `roo`, it is never determined what the type in the middle must be, nor is `roo` polymorphic in that type. Type inference neither solves nor generalizes the variable! Even so,

```
> roo "magoo"
""
```

it's not a problem, because the construction is *parametric* in the unknown type. The fact that the type cannot be determined has the consequence that the type cannot *matter*.

But unknown *instances* clearly do matter. The completeness result for Hindley-Milner type inference relies on parametricity and is thus lost when we add overloading. Let us not weep for it.

1.20 Understanding a case of Haskell Type Ambiguity

This is a classic problem. The "ad hoc" polymorphism of type classes makes type inference incomplete, and you've just been bitten. Let's look at the pieces.

```
read    :: Read x => String -> x
flatten :: NestedList a -> [a]
print   :: Show y => y -> IO ()
```

and we'll also have machine-generated instances for

```
Read a => Read (NestedList a)
Show a => Show (NestedList a)
Read a => Read [a]
Show a => Show [a]
```

Now let's solve the equations we get when we try to build the composition.

```
print   .     flatten              .   read

      y = [a]         NestedList a = x
```

That means we need

```
Show [a]                    Read (NestedList a)
```

and thus

```
Show a                      Read a
```

and we've used all our information without determining `a`, and hence the relevant `Read` and `Show` instances.

As J. Abrahamson has already suggested, you need to do something which determines the `a`. There are lots of ways to do it. I tend to prefer type annotations to writing strange terms whose only purpose is to make a type more obvious. I second the proposal to give a type to one of the components in the composition, but I'd probably pick `(read :: String -> NestedList Int)`, as that's the operation which introduces the ambiguously typed thing.

1.21 Why does Haskell use $\rightarrow$ instead of $=$?

It would be unfortunate to write

```
(0, _) = []
```

because that is not true.

In the tradition of Robert Recorde, we try to write equations only when we intend the left-hand side to equal the right-hand side. So we write

```
dup x = (x, x)
```

to make `dup x` equal to `(x, x)`, or

```
dup = \ x -> (x, x)
```

to make `dup` equal to the function which maps `x` to `(x, x)`, but not

```
\ x = (x, x)
```

because there is no way to make `x` equal `(x, x)`.

We depart from the tradition only slightly when we allow "falling through", e.g.,

```
f 0 = 1
f n = 2 * f (n - 1)
```

but only in the sense that the second line has a silent "otherwise".

1.22 Is it possible to make a type an instance of a class if its type parameters are in the wrong order?

Biased am I, but I think this is a great opportunity to make use of Control.Newtype, a little piece of kit that's a mere "cabal install newtype" away.

Here's the deal. You want to flip around type constructors to get your hands on functoriality (for example) in a different parameter. Define a newtype

```
newtype Flip f x y = Flip (f y x)
```

and add it to the `Newtype` class thus

```
instance Newtype (Flip f x y) (f y x) where
  pack = Flip
  unpack (Flip z) = z
```

The `Newtype` class is just a directory mapping newtypes to their unvarnished equivalents, providing handy kit, e.g. `op Flip` is the inverse of `Flip`: you don't need to remember what you called it.

For the problem in question, we can now do stuff like this:

```
data Bif x y = BNil | BCons x y (Bif x y) deriving Show
```

That's a two parameter datatype which happens to be functorial in both parameters. (Probably, we should make it an instance of a Bifunctor class, but anyway...) We can make it a `Functor` twice over: once for the last parameter...

```
instance Functor (Bif x) where
  fmap f BNil = BNil
  fmap f (BCons x y b) = BCons x (f y) (fmap f b)
```

...and once for the first:

```
instance Functor (Flip Bif y) where
  fmap f (Flip BNil) = Flip BNil
  fmap f (Flip (BCons x y b)) = Flip (BCons (f x) y (under Flip (fmap f) b))
```

where `under p f` is a neat way to say `op p . f . p`.

I tell you no lies: let us try.

```
someBif :: Bif Int Char
someBif = BCons 1 'a' (BCons 2 'b' (BCons 3 'c' BNil))
```

and then we get

```
*Flip> fmap succ someBif
BCons 1 'b' (BCons 2 'c' (BCons 3 'd' BNil))
*Flip> under Flip (fmap succ) someBif
BCons 2 'a' (BCons 3 'b' (BCons 4 'c' BNil))
```

In these circumstances, there really are many ways the same thing can be seen as a `Functor`, so it's right that we have to make some noise to say which way we mean. But the noise isn't all that much if you're systematic about it.

1.23 Functor type variables for Flip data type

The future is now, when you (use ghc 8 and) switch on a flag or two

```
Prelude> :set -XPolyKinds -XFlexibleInstances
```

Let us declare

```
Prelude> newtype Flip f a b = MkFlip (f b a)
```

and then enquire

```
Prelude> :kind Flip
Flip :: (k1 -> k -> *) -> k -> k1 -> *

Prelude> :type MkFlip
MkFlip
  :: forall k k1 (b :: k) (f :: k -> k1 -> *) (a :: k1).
     f b a -> Flip f a b
```

The *type* constructor `Flip` takes two implicit arguments, being `k` and `k1`, and three explicit arguments, being a binary function producing a type, then its two arguments in reverse order. The arguments to this function are of unconstrained type (old people can say "kind" if they like), but it certainly returns a type (in the strict sense of "thing in `*`", rather than the uselessly vague sense of "any old rubbish right of `::`") because it is certainly used as a type in the declaration of `MkFlip`.

The *data* constructor, `MkFlip`, takes *five* implicit arguments (exactly the arguments of `Flip`) and one explicit argument, being some data in `f b a`.

What's going on is Hindley-Milner type inference one level up. Constraints are collected (e.g., `f b a` must inhabit `*` because a constructor argument must inhabit `f b a`) but otherwise a most general type is delivered: `a` and `b` could be anything, so their types are generalised as `k1` and `k`.

Let's play the same game with the constant type constructor:

```
Prelude> newtype K a b = MkK a

Prelude> :kind K
K :: * -> k -> *

Prelude> :type MkK
MkK :: forall k (b :: k) a. a -> K a b
```

We see that `a :: *` but `b` can be any old rubbish (and for that matter, `k :: *`, as these days, `* :: *`). Clearly, `a` is actually used as the type of a thing, but `b` is not used at all, hence unconstrained.

We may then declare

```
Prelude> instance Functor (Flip K b) where fmap f (MkFlip (MkK a)) = MkFlip (MkK (f a))
```

and ask

```
Prelude> :info Flip
...
instance [safe] forall k (b :: k). Functor (Flip K b)
```

which tells us that the unused `b` can still be any old rubbish. Because we had

```
K    ::   * -> k -> *
Flip :: (k1 -> k -> *) -> k -> k1 -> *
```

we can unify `k1 = *` and get

```
Flip K :: k -> * -> *
```

and so

```
Flip K b :: * -> *
```

for any old `b`. A `Functor` instance is thus plausible, and indeed deliverable, with the function acting on the packed up `a` element, corresponding to the argument of `Flip K b` which becomes the *first* argument of `K`, hence the type of the stored element.

Unification-based type inference is alive and (fairly) well, right of `::`.

1.24 Why does `product []` return 1?

Lists form a *monoid* structure, with associative binary operation `++` and neutral element `[]`. That is, we have

```
[] ++ xs = xs = xs ++ []     (xs ++ ys) ++ zs = xs ++ (ys ++ zs)
```

Meanwhile, numbers have lots of monoid structure, but the relevant one here is that where the operation is `*` and the neutral element is `1`.

```
1 * x = x = x * 1            (x * y) * z = x * (y * z)
```

The `product` function is not only a map from lists of numbers to numbers: it's a *monoid homomorphism*, reflecting the list monoid structure in the numerical monoid. Crucially,

```
product (xs ++ ys) = product xs * product ys
```

and

```
product [] = 1
```

In fact, to get the former, we pretty much have the latter forced upon us.

1.25 Minimum of Two Maybes

It is possible to satisfy the specification using operators from `Control.Applicative`.

```
myMin :: Ord x => Maybe x -> Maybe x -> Maybe x
myMin a b = min <$> a <*> b <|> a <|> b
```

where the <|> for `Maybe` implements "preference"

```
Nothing <|> b  = b
a       <|> _  = a
```

The thing is

```
min <$> Just a <*> Just b = Just (min a b)
```

but

```
min <$> Just a <*> Nothing = Nothing
```

which has resulted in some incorrect answers to this question. Using <|> allows you to prefer the computed `min` value when it's available, but recover with either individual when only one is `Just`.

But you should ask if it is appropriate to use `Maybe` in this way. With the inglorious exception of its `Monoid` instance, `Maybe` is set up to model failure-prone computations. What you have here is the extension of an existing `Ord` with a "top" element.

```
data Topped x = Val x | Top deriving (Show, Eq, Ord)
```

and you'll find that `min` for `Topped x` is just what you need. It's good to think of types as not just the representation of data but the equipment of data with structure. `Nothing` usually represents some kind of failure, so it might be better to use a different type for your purpose.

1.26 Is the equivalent of Haskell's Foldable and Traversable simply a sequence in Clojure?

No. Whilst any kind of `Functor` representing finite sequences of elements will be `Traversable` (hence `Foldable`), there are plenty of other structures which are `Traversable`, but which aren't sequence-like, in that they don't have an obvious notion of concatenation. There will be a way to obtain the sequence of contained elements, but the structure may consist of more than just that sequence.

What `Traversable f` means, in effect, is that structures with type `f x` contain finitely many elements of type `x`, and that there is some way to `traverse` the structure visiting each element of `x` exactly once. So things like "terms in a syntax, seen as containing variables" can be `Traversable`.

```
data Term x
  = Var x
  | Val Integer
  | Add (Term x) (Term x)

instance Traversable Term where
  traverse f (Var x)    = pure Var <*> f x
  traverse f (Val i)    = pure (Val i)
  traverse f (Add s t)  = pure Add <*> traverse f s <*> traverse f t
```

You can always use `traverse` to do operations on all elements. We get `fmap` by taking `pure = id` and <*> to be ordinary application.

```
instance Functor Term where
  fmap = fmapDefault
```

where

```
fmap :: (x -> y) -> Term x -> Term y
```

implements *simultaneous renaming*.

Meanwhile, the `Foldable` instance

```
instance Foldable Term where
  foldMap = foldMapDefault
```

takes `pure` to give the neutral element of some monoid and <*> to the combining operation, so we get reduce-like operations. E.g.,

```
foldMap (:[]) :: Term x -> [x]
```

gives the list of variables occurring in a term. That is we can always obtain the sequence of elements from `Traversable` data, but the data might have structure other than those elements. `Term`s have structure other than variables (their `Val`s and `Add`s), and it's not so clear what "cons" means for syntax trees.

So, while more structures than sequences are `Traversable`, the `Traversable` interface offers you fewer sequence-like operations. The point of `Traversable` is to generalize *map*-like and *reduce*-like operations, not to capture *list*-ness.

1.27 How do you keep track of multiple properties of a string without traversing it multiple times?

I'd start by defining the traditional "indicator function"

```
indicate :: Num a => Bool -> a
indicate b = if b then 1 else 0
```

so that

```
indicate . isVowel :: Char -> Integer
```

Next, I'd get hold of two key pieces of kit from `Control.Arrow`

```
(&&&) :: (x -> y) -> (x -> z) -> x -> (y, z)
(***) :: (a -> b) -> (c -> d) -> (a, c) -> (b, d)
```

so (remembering that some characters are neither vowels nor consonants)

```
(indicate . isVowel) &&& (indicate . isConsonant)
  :: Char -> (Integer, Integer)
```

And then I'd grab hold of `Sum` from `Data.Monoid`.

```
(Sum . indicate . isVowel) &&& (Sum . indicate . isConsonant)
  :: Char -> (Sum Integer, Sum Integer)
getSum *** getSum :: (Sum Integer, Sum Integer) -> (Integer, Integer)
```

Now I deploy `foldMap`, because we're doing some sort of monoidal "crush".

```
(getSum *** getSum) .
foldMap ((Sum . indicate . isVowel) &&& (Sum . indicate . isConsonant))
  :: String -> (Integer, Integer)
```

Then I remember that I wrote some code which got turned into `Control.Newtype` and I discover the following is missing but should be there.

```
instance (Newtype n o, Newtype n' o') => Newtype (n, n') (o, o') where
  pack = pack *** pack
  unpack = unpack *** unpack
```

And now I need only write

```
ala' (Sum *** Sum) foldMap ((indicate . isVowel) &&& (indicate . isConsonant))
  :: String -> (Integer, Integer)
```

The key gadget is

```
ala' :: (Newtype n o, Newtype n' o') =>
 (o -> n) -> ((a -> n) -> b -> n') -> (a -> o) -> b -> o'
-- ^-packer    ^-higher-order operator  ^-action-on-elements
```

where the packer's job is to select the newtype with the correct behavioral instance and also determine the unpacker. It's exactly designed to support working locally at a more specific type that signals the intended structure.

1.28 Finding a leaf with value x in a binary tree

Your language, about what you thought the program should *do*, suggests to me that you need help to escape from the trap of imperative thinking. Let me try to offer some help, based on thinking about what things *are*, not what things *do*.

For `findpath (Leaf y) x`, you're heading in the right direction. You just need to give `if` a lowercase `i`, and think about what the correct `Path` to a `Leaf` must be.

Now, let's think about the other possibility. You know more than that it's some `t`. You know that you're really trying to figure out what

```
findpath (Node l r) x
```

is (what it =, indeed), because that's the other possibility for a `BTree`. Think of splitting the problem by asking "Is this `BTree` a `(Leaf y)` or a `(Node l r)`?" as one conceptual step of program design. Now, in order to figure out what the above left-hand side equals, you're entitled to some recursively computed information, namely what

```
findpath l x
```

and

```
findpath r x
```

are. If you know `Path` information for both `l` and `r`, can you say what the `Path` for the whole `Node l r` is? Let me rephrase that question by writing it in Haskell:

```
findpath :: Eq a => BTree a -> a -> Path
findpath (Leaf y)   x = if y==x then ??? else Nothing
findpath (Node l r) x = nodepath (findpath l x) (findpath r x) where
  nodepath :: Path -> Path -> Path
  nodepath ???
```

I have expressed my question by introducing a *helper function* `nodepath` which takes as arguments the recursively computed information. Now you can try to implement `nodepath` by pattern matching on those two paths for the left and right subtrees, respectively. If you know whether they are `(Just p)` or `Nothing`, then you should be able to say what the path for the whole node must be.

Lesson one, the useful thoughts are of the form: "If this is like such-and-such, then that must be so-and-so.". Being, not doing.

Lesson two, the basic method of programming over a datatype is: split into constructor cases (`Leaf` versus `Node`, `Just` versus `Nothing`); collect useful information from any substructures by recursive calls; say what the value for the whole structure must be.

If you follow my advice and figure out what `nodepath` should be, you may find that it's simple enough not to merit being a separate named definition. In that case, just replace the `nodepath` call with its meaning and cut out the `where`-clause. But it's good to start by introducing `nodepath`, as it expresses a useful conceptual step towards solving the problem.

1.29 Taking from a list until encountering a duplicate

Your point that `takeWhile` doesn't work because you have no contextual information for the individual elements suggests the following strategy: get it.

This answer of mine makes reference to the decorate-with-context operation, which I called `picks` (because it shows you all the way to pick one element on which to focus). It's the general decorate-with-its-context operation that we just ought to have for free for every containery thing. For lists, it is

```
picks :: [x] -> [(x, ([x], [x]))] -- [(x-here, ([x-before], [x-after]))]
picks [] = []
picks (x : xs) = (x, ([], xs)) : [(y, (x : ys, zs)) | (y, (ys, zs)) <- picks xs]
```

and it works perfectly well for infinite lists, while we're about it.

Now have a go with

```
takeUntilDuplicate :: Eq x => [x] -> [x]
takeUntilDuplicate = map fst . takeWhile (\ (x, (ys, _)) -> not (elem x ys)) . picks
```

(Curiously, I'm disturbed that the above one-liner is rejected for ambiguity of `Eq` if not given the above type signature. I'm tempted to ask a question about it, here. *Oh, it's the monomorphism restriction. How annoying.*)

Remark. It makes a lot of sense to (and I normally would) represent the "elements before" component that `picks` delivers using snoc-lists (lists which grow on the right), the better to preserve sharing and visual left-to-right-ness.

1.30 RankNTypes and PolyKinds (quantifier alternation issues)

Let's be bloody. We must quantify everything and give the domain of quantification. Values have types; type-level things have kinds; kinds live in `BOX`.

```
f1 :: forall (k :: BOX).
      (forall (a :: k) (m :: k -> *). m a -> Int)
      -> Int

f2 :: (forall (k :: BOX) (a :: k) (m :: k -> *). m a -> Int)
      -> Int
```

Now, in neither example type is `k` quantified explicitly, so ghc is deciding where to put that `forall (k :: BOX)`, based on whether and where `k` is mentioned. I am not totally sure I understand or am willing to defend the policy as stated.

Ørjan gives a good example of the difference in practice. Let's be bloody about that, too. I'll write `/\ (a :: k). t` to make explicit the abstraction that corresponds to `forall`, and `f @ type` for the corresponding application. The game is that we get to pick the `@`-ed arguments, but we have to be ready to put up with whatever `/\`-ed arguments the devil may choose.

We have

```
x :: forall (a :: *) (m :: * -> *). m a -> Int
```

and may accordingly discover that `f1 x` is really

```
f1 @ * (/\ (a :: *) (m :: * -> *). x @ a @ m)
```

However, if we try to give `f2 x` the same treatment, we see

```
f2 (/\ (k :: BOX) (a :: k) (m :: k -> *). x @ ?m0 @ ?a0)
?m0 :: *
?a0 :: * -> *
where  m a = m0 a0
```

The Haskell type system treats type application as purely syntactic, so the only way that equation can be solved is by identifying the functions and identifying the arguments

```
(?m0 :: * -> *) = (m :: k -> *)
(?a0 :: *)      = (a :: k)
```

but those equations are not even well kinded, because `k` is not free to be chosen: it's being `/\`-ed not `@`-ed.

Generally, to get to grips with these uber-polymorphic types, it's good to write out all the quantifiers and then figure out how that turns into your game against the devil. Who chooses what, and in what order. Moving a `forall` inside an argument type changes its chooser, and can often make the difference between victory and defeat.

1.31 How to write this case expression with the view pattern syntax?

They're not equivalent. The `case` version has one `readMaybe`, the view pattern version has two. For every `readMaybe`, the compiler has to infer which type is the target of the attempt to read. When the code says

```
parse xs x = case readMaybe x of
  Just x  -> Right (x : xs)
  Nothing -> Left "Syntax error
```

the GHC detective notices that in your `Just x` case, `x` ends up consed to `xs`, and so must take whatever type the elements of `xs` have. And that's good work.

But when you write

```
parse xs (readMaybe -> Just x ) = Right (x : xs)
parse xs (readMaybe -> Nothing) = Left "Syntax error"
```

you create two separate find-the-target-type problems, one for each use of `readMaybe`. The first of these is solved in just the same way as in the `case` case, but for the second, read individually,

```
parse xs (readMaybe -> Nothing) = Left "Syntax error"
```

there is just no clue *what* it is that you are failing to read, and no reason to believe it is the same thing as in the line above.

Generally, it is inappropriate to use view patterns unless there is only one outcome of interest. They are the wrong syntax if you want to do an intermediate computation *once*, but analyse the result into more than one case. I am happy to remain on the record that I consider them a misfeature for this reason.

1.32 Recursive Type Families

Type inference is, by default, a guessing game. Haskell's surface syntax makes it rather awkward to be explicit about which types should instantiate a `forall`, even if you know what you want.

This is a legacy from the good old days of Damas-Milner completeness, when ideas interesting enough to require explicit typing were simply forbidden.

Let's imagine we're allowed to make type application explicit in patterns and expressions using an Agda-style `f {a = x}` notation, selectively accessing the type parameter corresponding to `a` in the type signature of `f`. Your

```
idT = StateT $ \ s -> idT
```

is supposed to mean

```
idT {a = a}{m = m} = StateT $ \ s -> idT {a = a}{m = m}
```

so that the left has type `C a a (StateT s m) r` and the right has type `StateT s (C a a m) r`, which are equal by definition of the type family and joy radiates over the world. But that's not the meaning of what you wrote. The "variable-rule" for invoking polymorphic things requires that each `forall` is instantiated with a fresh existential type variable which is then solved by unification. So what your code means is

```
idT {a = a}{m = m} = StateT $ \ s -> idT {a = a'}{m = m'}
  -- for a suitably chosen a', m'
```

The available constraint, after computing the type family, is

```
C a a m ~ C a' a' m'
```

but that doesn't simplify, nor should it, because there is no reason to assume `C` is injective. And the maddening thing is that the machine cares more than you about the possibility of finding a most general solution. You have a suitable solution in mind already, but the problem is to achieve *communication* when the default assumption is *guesswork*.

There are a number of strategies which might help you out of this jam. One is to use data families instead. Pro: injectivity no problem. Con: structor. (Warning, untested speculation below.)

```
class MonadPipe m where
  data C a b (m :: * -> *) r
  idT :: C a a m r
  (<-<) :: C b c m r -> C a b m r -> C a c m r

instance (MonadPipe m) => MonadPipe (StateT s m) where
  data C a b (StateT s m) r = StateTPipe (StateT s (C a b m) r)
  idT = StateTPipe . StateT $ \ s -> idT
  StateTPipe (StateT f) <-< StateTPipe (StateT g) =
    StateTPipe . StateT $ \ s - f s <-< g s
```

Another con is that the resulting data family is not automatically monadic, nor is it so very easy to unwrap it or make it monadic in a uniform way.

I'm thinking of trying out a pattern for these things where you keep your type family, but define a newtype wrapper for it

```
newtype WrapC a b m r = WrapC {unwrapC :: C a b m r}
```

then use `WrapC` in the types of the operations to keep the typechecker on the right track. I don't know if that's a good strategy, but I plan to find out, one of these days.

A more direct strategy is to use proxies, phantom types, and scoped type variables (although this example shouldn't need them). (Again, speculation warning.)

```
data Proxy (a :: *) = Poxy
data ProxyF (a :: * -> *) = PoxyF

class MonadPipe m where
  data C a b (m :: * -> *) r
  idT :: (Proxy a, ProxyF m) -> C a a m r
  ...

instance (MonadPipe m) => MonadPipe (StateT s m) where
  data C a b (StateT s m) r = StateTPipe (StateT s (C a b m) r)
  idT pp = StateTPipe . StateT $ \ s -> idT pp
```

That's just a crummy way of making the type applications explicit. Note that some people use `a` itself instead of `Proxy a` and pass `undefined` as the argument, thus failing to mark the proxy as such in the type and relying on not accidentally evaluating it. Recent progress with `PolyKinds` may at least mean we can have just one kind-polymorphic phantom proxy type. Crucially, the `Proxy` type constructors are injective, so my code really is saying "same parameters here as there".

But there are times when I wish that I could drop to the System FC level in source code, or otherwise express a clear manual override for type inference. Such things are quite standard in the dependently typed community, where nobody imagines that the machine can figure everything out without a nudge here and there, or that hidden arguments carry no information worth inspecting. It's quite common that hidden arguments to a function can be suppressed at usage sites but need to be made explicit within the definition. The present situation in Haskell is based on a cultural assumption that "type inference is enough" which has been off the rails for a generation but still somehow persists.

1.33 Determine whether a value is a function in Haskell

Parametricity says no. The only functions of type

```
a -> Bool
```

are constant functions.

However, with a bit of ad hoc polymorphism and a bit more chutzpah, you can do this:

```
{-# LANGUAGE OverlappingInstances, FlexibleInstances #-}

class Sick x where
  isFunc :: x -> Bool

instance Sick (a -> b) where
  isFunc _ = True

instance Sick x where
  isFunc _ = False
```

and then it looks like you have

```
*Sick> isFunc 3
False
*Sick> isFunc id
True
```

But it does seem like a peculiar thing to do. What use is the resulting `Bool` to you?

1.34 Automatic Functor Instance (not)

Working in ghci, if you make the incantation

```
Prelude> :set -XDeriveFunctor
```

then the compiler will become as clever as you are hoping for, if not quite as enthusiastic. You will need to invoke the functionality, thus,

```
Prelude> data Foo a = Foo a deriving (Show, Functor)
```

(the `Show` is just for printing the output, below) and then you will be able to do stuff like

```
Prelude> fmap (++"bar") (Foo "foo")
Foo "foobar"
```

In a module, you achieve the same by adding the pragma

```
{-# LANGUAGE DeriveFunctor #-}
```

before the `module` declaration. It's good for at least the more straightforward `Functor` instances, but you can fool it into a false negative.

```
Prelude> data Boo a = Boo (Either a Bool) deriving Functor

<interactive>:9:43:
    Can't make a derived instance of 'Functor Boo':
      Constructor 'Boo' must use the type variable only as the
        last argument of a data type
      In the data declaration for 'Boo'
```

Meanwhile

```
data Goo a = Goo (Either Bool a) deriving Functor
```

is ok, and the machinery has clearly been hacked to work with pairing, as

```
data Woo a = Woo (a, Bool) deriving Functor
```

is permitted.
So it's not as clever as it could be, but it's better than a poke in the eye.

1.35 How do I apply the first partial function that works in Haskell?

In `Data.Monoid`, a `newtype` copy of `Maybe`, called `First`, has the "take the first `Just`" behaviour.

If you were looking for a function of type

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

with the behaviour you describe, it would simply be

```
fold
```

from `Data.Foldable`, because the monoid behaviour for `a ->` does the pointwise lifting needed: the `Monoid` for `a -> First b` is exactly picking the first application outcome whic works. Sadly (for my tears over this have been many), to get `Maybe` instead of `First` takes a little more work.

Note that the pointwise lifting, yanking `a ->` out through `[]`, is just the sort of job for `sequenceA`, so

```
(asum .) . sequenceA
```

will do the job.

It's good to get the monoid structure you need cued from the type: in this case, accessing the `Alternative` behaviour with `asum` will have to do.

1.36 'Zipping' a plain list with a nested list

Will this do?

```
flip . (evalState .) . traverse . traverse . const . state $ head &&& tail
```

EDIT: let me expand on the construction...

The essential centre of it is `traverse . traverse`. If you stare at the problem with sufficiently poor spectacles, you can see that it's "do something with the elements of a container of containers". For that sort of thing, `traverse` (from `Data.Traversable`) is a very useful gadget (ok, I'm biased).

```
traverse :: (Traversable f, Applicative a) => (s -> a t) -> f s -> a (f t)
```

or, if I change to longer but more suggestive type variables

```
traverse :: (Traversable containerOf, Applicative doingSomethingToGet) =>
              (s -> doingSomethingToGet t) ->
              containerOf s -> doingSomethingToGet (containerOf t)
```

Crucially, `traverse` preserves the structure of the container it operates on, whatever that might be. If you view `traverse` as a higher-order function, you can see that it gives back an operator on containers whose type fits with the type of operators on elements it demands. That's to say `(traverse . traverse)` makes sense, and gives you structure-preserving operations on *two* layers of container.

```
traverse . traverse ::
  (Traversable g, Traversable f, Applicative a) => (s -> a t) -> g (f s) -> a (g (f t))
```

So we've got the key gadget for structure-preserving "do something" operations on lists of lists. The `length` and `splitAt` approach works fine for lists (the structure of a list is given by its length), but the essential characteristic of lists which enables that approach is already pretty much bottled by the `Traversable` class.

Now we need to figure out how to "do something". We want to replace the old elements with new things drawn successively from a supply stream. If we were allowed the side-effect of updating the supply, we could say what to do at each element: "return `head` of supply, updating supply with its `tail`". The `State s` monad (in `Control.Monad.State` which is an instance of `Applicative`, from `Control.Applicative`) lets us capture that idea. The type `State s a` represents computations which deliver a value of type `a` whilst mutating a state of type `s`. Typical such computations are made by this gadget.

```
state     :: (s -> (a, s)) -> State s a
```

That's to say, given an initial state, just compute the value and the new state. In our case, `s` is a stream, `head` gets the value, `tail` gets the new state. The `&&&` operator (from `Control.Arrow`) is a nice way to glue two functions on the same data to get a function making a pair. So

```
head &&& tail :: [x] -> (x, [x])
```

which makes

```
state $ head &&& tail :: State [x] x
```

and thus

```
const . state $ head &&& tail :: u -> State [x] x
```

explains what to "do" with each element of the old container, namely ignore it and take a new element from the head of the supply stream.

Feeding that into `(traverse . traverse)` gives us a big mutatey traversal of type

```
f (g u) -> State [x] (f (g x))
```

where `f` and `g` are any `Traversable` structures (e.g. lists).

Now, to extract the function we want, taking the initial supply stream, we need to unpack the state-mutating computation as a function from initial state to final value. That's what this does:

```
evalState :: State s a -> s -> a
```

So we end up with something in

```
f (g u) -> [x] -> f (g x)
```

which had better get flipped if it's to match the original spec.

tl;dr The `State [x]` monad is a readymade tool for describing computations which read and update an input stream. The `Traversable` class captures a readymade notion of structure-preserving operation on containers. The rest is plumbing (and/or golf).

1.37 Bunched accumulations

I see we're accumulating over some data structure. I think `foldMap`. I ask "Which `Monoid`"? It's some kind of lists of accumulations. Like this

```
newtype Bunch x = Bunch {bunch :: [x]}
instance Semigroup x => Monoid (Bunch x) where
  mempty = Bunch []
  mappend (Bunch xss) (Bunch yss) = Bunch (glom xss yss) where
    glom [] yss = yss
    glom xss [] = xss
    glom (xs : xss) (ys : yss) = (xs <> ys) : glom xss yss
```

Our underlying elements have some associative operator <>, and we can thus apply that operator pointwise to a pair of lists, just like `zipWith` does, except that when we run out of one of the lists, *we don't truncate*, rather we just take the other. Note that `Bunch` is a name I'm introducing for purposes of this answer, but it's not that unusual a thing to want. I'm sure I've used it before and will again.

If we can translate

```
0 -> Bunch [[0]]           -- single 0 in place 0
1 -> Bunch [[],[1]]        -- single 1 in place 1
2 -> Bunch [[],[],[2]]     -- single 2 in place 2
3 -> Bunch [[],[],[],[3]]  -- single 3 in place 3
...
```

and `foldMap` across the input, then we'll get the right number of each in each place. There should be no need for an upper bound on the numbers in the input to get a sensible output, as long as you are willing to interpret `[]` as "the rest is silence". Otherwise, like Procrustes, you can pad or chop to the length you need.

Note, by the way, that when `mappend`'s first argument comes from our translation, we do a bunch of `([]++)` operations, a.k.a. `id`s, then a single `([i]++)`, a.k.a. `(i:)`, so if `foldMap` is right-nested (which it is for lists), then we will always be doing cheap operations at the left end of our lists.

Now, as the question works with lists, we might want to introduce the `Bunch` structure only when it's useful. That's what `Control.Newtype` is for. We just need to tell it about `Bunch`.

```
instance Newtype (Bunch x) [x] where
  pack = Bunch
  unpack = bunch
```

And then it's

```
groupInts :: [Int] -> [[Int]]
groupInts = ala' Bunch foldMap (basis !!) where
  basis = ala' Bunch foldMap id [iterate ([]:) [], [[[i]] | i <- [0..]]]
```

What? Well, without going to town on what `ala'` is in general, its impact here is as follows:

```
ala' Bunch foldMap f = bunch . foldMap (Bunch . f)
```

meaning that, although `f` is a function to lists, we accumulate as if `f` were a function to `Bunch`es: the role of `ala'` is to insert the correct `pack` and `unpack` operations to make that just happen.

We need `(basis !!) :: Int -> [[Int]]` to be our translation. Hence `basis :: [[[Int]]]` is the list of images of our translation, computed on demand at most once each (i.e., the translation, *memoized*).

For this `basis`, observe that we need these two infinite lists

```
[ []                    [ [[0]]
, [[]]                  , [[1]]
, [[],[]]               , [[2]]
, [[],[],[]]            , [[3]]
...                     ...
```

combined `Bunch`wise. As both lists have the same length (infinity), I could also have written

```
basis = zipWith (++) (iterate ([]:) []) [[[i]] | i <- [0..]]
```

but I thought it was worth observing that this also is an example of `Bunch` structure.

Of course, it's very nice when something like `accumArray` hands you exactly the sort of accumulation you need, neatly packaging a bunch of grungy behind-the-scenes mutation. But the general recipe for an accumulation is to think "What's the `Monoid`?" and "What do I do with each element?". That's what `foldMap` asks you.

1.38 Functor on Phantom Type

A crucial skill in functional programming is grasping, in a given situation, the right way to do nothing. Classic errors arise, e.g., from giving `[]` meaning "no solutions" as the base case of a recursive search, when `[[]]` meaning "one trivial solution" is needed. Thus also `Functor` instances for phantom types, i.e. constant functors, are useful as the apparently trivial base case of a larger pattern.

I can present the general toolkit for building container-like structures as follows:

```
newtype K a x = K a deriving Functor             -- K for konstant
{- fmap _ (K a) = K a -}

newtype I x = I x deriving Functor               -- I for identity
{- fmap k (I x) = I (k x) -}

newtype P f g x = P (f x, g x) deriving Functor  -- P for product
{- will give (Functor f, Functor g) => Functor (P f g), such that
   fmap k (P (fx, gx)) = P (fmap k fx, fmap k gx) -}

newtype S f g x = S (Either (f x) (g x))         -- S for sum
instance (Functor f, Functor g) => Functor (S f g) where
  fmap k (S (Left  fx))  = S (Left  (fmap k fx))
  fmap k (S (Right gx))  = S (Right (fmap k gx))
```

Now, any recursive data structure can be presented as a top node which acts as a container for substructures.

```
data Data f = Node (f (Data f))
```

For example, if I want to make binary trees with numbers at the leaves, I can write

```
type Tree = S (K Int) (P I I)
```

to indicate that the node structure for a tree is either a leaf with an `Int` and *no subtrees* or a fork with a pair of subtrees. I need `K` to point out the *absence* of recursive substructures. The type of trees is then `Data Tree`.

The usual recursion scheme for these things is

```
fold :: Functor f => (f t -> t) -> Data f -> t
fold k (Node fd) = k (fmap (fold k) fd)
```

We don't need to do any work to instantiate that for trees, because `Tree` is already a `Functor`, as it was built from functorial components. The trivial `fmap` for `K Int` amounts to saying that the recursion stops when you reach a leaf.

Of course, these "encoded" data types make it harder to see what you're doing when you write ordinary programs by pattern matching. That's where the `PatternSynonyms` extension comes to your rescue. You can say

```
pattern Leaf i    = Node (S (Left (K i)))
pattern Fork l r  = Node (S (Right (P (I l, I r))))
```

to recover the usual interface. I recommend leaving off the outer `Node`, to fit with the way `fold` strips `Node` for you.

```
pattern Leaf i = S (Left (K i))
pattern Fork l r = S (Right (P (I l, I r)))

add :: Data Tree -> Int
add = fold $ \ t -> case t of
  Leaf i   -> i
  Fork x y -> x + y
```

I've barely scratched the surface of the kinds of generic functionality you can roll out to lots of data types whenever you can develop them just for `K`, `I`, `P` and `S`. The `K` cases are always trivial, but they have to be there.

Similar considerations apply to the `Void` data type (in `Data.Void`). Why on earth would we bother to introduce a data type with no elements worth speaking of? To model the impossible cases of a larger scheme.

1.39 Creating an Interpreter (with store) in Haskell

It's a little difficult to answer your question, because you didn't actually ask one. Let me just pick out a few of the things that you've said, in order to give you a few clues.

> I am not sure if I need to use `evalE` in this problem or not. I have written it in a previous problem. The signature for `evalE` is `evalE :: Expression -> Store -> (Value, Store)`
>
> ```
> evalS (Expr e) s         = ... Not sure what to do, here.
> ```

What does it mean to execute a statement which consists of an expression? If it has something to do with evaluating the expression, then the fact that you have an expression evaluator available might help with "what to do, here".

Next, compare the code you've been given for "while" (which contains a fine example of a sensible thing to do with an expression, by the way)...

```
evalS w@(While e s1) s = case (evalE e s) of`
  (BoolVal True,s')  -> let s'' = evalS s1 s' in evalS w s''
  (BoolVal False,s') -> s'
  _                  -> error "Condition must be a BoolVal"
```

...and compare it with your code for "if"

```
evalS (If e s1 s2) s     = do
   x <- evalE e
   case x of
      BoolVal True -> evalS s1
      BoolVal False -> evalS s2
```

Your code is in a rather different style — the "monadic" style. Where are you getting that from? It would make sense if the types of the evaluators were something like

```
evalE :: Expression -> State Store Value
evalS :: Statement  -> State Store ()
```

The monadic style is a very nice way to thread the mutating store through the evaluation process without talking about it too much. E.g., your `x <- evalE e` means "let `x` be the result of evaluating `e` (quietly receiving the initial store and passing along the resulting store)". It's a good way to work which I expect you'll explore in due course.

But those aren't the types you've been given, and the monadic style is not appropriate. You have

```
evalE :: Expression -> Store -> (Value, Store)
evalS :: Statement  -> Store ->         Store
```

and the example code threads the store explicitly. Look again

```
evalS w@(While e s1) s = case (evalE e s) of`
  (BoolVal True,s')  -> let s'' = evalS s1 s' in evalS w s''
  (BoolVal False,s') -> s'
  _                  -> error "Condition must be a BoolVal"
```

See? `evalS` receives its initial store, `s`, explicitly, and uses it explicitly in `evalE e s`. The resulting new store is called `s'` in both `case` branches. If the loop is over, then `s'` is given back as the final store. Otherwise, `s'` is used as the store for one pass through the loop body, `s1`, and the store `s''` resulting from that is used for the next time around the loop, `w`.

Your code will need to be similarly explicit in the way it names and uses the store at each stage of evaluation. Let's walk through.

```
evalS Skip s             = show s -- I am assuming that since Skip returns an empty Stri
```

You assume incorrectly. The `evalS` function does not return a `String`, empty or otherwise: it returns a `Store`. Now, which `Store`? Your initial store is `s`: how will the store after "skip" relate to `s`?

```
evalS (Sequence s1 s2) s = evalS s1 >> evalS s2 -- sequence1 then sequence2. I am not qu
```

Again, that's a monadic approach which does not fit with this question. You need to thread the store, intially `s`, through the process of evaluating statements `s1` and `s2` in sequence. The "while" case has a good example of how to do something like that.

```
evalS (Expr e) s         = ... Not sure what to do, here.
```

Again, the "while" example shows you one way to extract a value and an updated store by evaluating an expression. Food for thought, isn't it?

```
evalS (If e s1 s2) s     = ...
```

Now "if" starts out by evaluating a condition, rather a lot like "while", no?

So, my advice amounts to this:

- drop the monadic style code for now, but come back to it later when it's appropriate;
- read the example implementation of "while" and understand how it treats expressions and sequences of statements, passing the store explicitly;
- deploy the similar techniques to implement the other constructs.

The person who set the question has been kind enough to give you code which gives an example of everything you will need. Please reciprocate that kindness by comprehending and then taking the hint!

1.40 Existential type wrappers necessity

Your first attempt is *not* using existential types. Rather your

```
lists :: [(Int, forall a. Show a => Int -> a)]
```

demands that the second components can deliver an element of *any* showable type that I choose, not just *some* showable type that you choose. You're looking for

```
lists :: [(Int, exists a. Show a * (Int -> a))]  -- not real Haskell
```

but that's not what you've said. The datatype packaging method allows you to recover `exists` from `forall` by currying. You have

```
HRF :: forall a. Show a => (Int -> a) -> HRF
```

which means that to build an `HRF` value, you must supply a triple containing a type `a`, a `Show` dictionary for `a` and a function in `Int -> a`. That is, the `HRF` constructor's type effectively curries this non-type

```
HRF :: (exists a. Show a * (Int -> a)) -> HRF   -- not real Haskell
```

You might be able to avoid the datatype method by using rank-n types to Church-encode the existential

```
type HRF = forall x. (forall a. Show a => (Int -> a) -> x) -> x
```

but that's probably overkill.

1.41 Non-linear Patterns in Type-Level Functions

Haskell's type-level language is a purely first-order language, in which "application" is just another constructor, rather than a thing which computes. There are binding constructs, like `forall`, but the notion of equality for type-level stuff is fundamentally mere alpha-equivalence: structural up to renaming of bound variables. Indeed, the whole of our constructor-class machinery, monads, etc relies on being able to take an application `m v` apart unambiguously.

Type-level functions don't really live in the type-level language as first-class citizens: only their full applications do. We end up with an equational (for the ~ notion of equality) theory of type-level expressions in which constraints are expressed and solved, but the underlying notion of *value* that these expressions denote is always first-order, and thus always equippable with equality.

Hence it always makes sense to interpret repeated pattern variables by a structural equality test, which is exactly how pattern matching was designed in its original 1969 incarnation, as an extension to another language rooted in a fundamentally first-order notion of value, LISP.

1.42 Initial algebra for rose trees

I would discourage talk of "the Hask Category" because it subconsciously conditions you against looking for other categorical structure in Haskell programming.

Indeed, rose trees can be seen as the fixpoint of an endofunctor on types-and-functions, a category which we might be better to call `Type`, now that `Type` is the type of types. If we give ourselves some of the usual functor kit…

```
newtype K a   x = K a deriving Functor           -- constant functor
newtype P f g x = P (f x, g x) deriving Functor  -- products
```

…and fixpoints…

```
newtype FixF f = InF (f (FixF f))
```

…then we may take

```
type Rose a = FixF (P (K a) [])
pattern Node :: a -> [Rose a] -> Rose a
pattern Node a ars = InF (P (K a, ars))
```

The fact that `[]` is itself recursive does not prevent its use in the formation of recursive datatypes via `Fix`. To spell out the recursion explicitly, we have nested fixpoints, here with bound variable names chosen suggestively:

```
Rose a = μrose. a * (μlist. 1 + (rose * list))
```

Now, by the time we've arrived in the second fixpoint, we have a type formula

```
1 + (rose * list)
```

which is functorial (indeed, strictly positive) in both `rose` and `list`. One might say it is a `Bifunctor`, but that's unnecessary terminology: it's a functor from `(Type, Type)` to `Type`. You can make a `Type -> Type` functor by taking a fixpoint in the second component of the pair, and that's just what happened above.

The above definition of `Rose` loses an important property. It is not true that

```
Rose :: Type -> Type   -- GHC might say this, but it's lying
```

merely that `Rose x :: Type` if `x :: Type`. In particular,

```
Functor Rose
```

is not a well typed constraint, which is a pity, as intuitively, rose trees ought to be functorial in the elements they store.

You can fix this by building `Rose` as itself being the fixpoint of a `Bifunctor`. So, in effect, by the time we get to lists, we have *three* type variables in scope, `a`, `rose` and `list`, and we have functoriality in all of them. You need a *different* fixpoint type constructor, and a *different* kit for building `Bifunctor` instances: for `Rose`, life gets easier because the `a` parameter is not used in the inner fixpoint, but in general, to define bifunctors as fixpoints requires trifunctors, and off we go!

This answer of mine shows how to fight the proliferation by showing how *indexed* types are *closed* under a fixpoint-of-functor construction. That's to say, work not in `Type` but in `i -> Type` (for the full variety of index types `i`) and you're ready for mutual recursion, GADTs, and so on.

So, zooming out, rose trees are given by mutual fixpoints, which have a perfectly sensible categorical account, provided you see which categories are actually at work.

Chapter 2

Pattern Matching

2.1 Algorithm for typechecking ML-like pattern matching?

Here's a sketch of an algorithm. It's also the basis of Lennart Augustsson's celebrated technique for compiling pattern matching efficiently. (The paper is in that incredible FPCA proceedings (LNCS 201) with oh so many hits.) The idea is to reconstruct an exhaustive, non-redundant analysis by repeatedly splitting the most general pattern into constructor cases.

In general, the problem is that your program has a possibly empty bunch of 'actual' patterns {p1, .., pn}, and you want to know if they cover a given 'ideal' pattern q. To kick off, take q to be a variable x. The invariant, initially satisfied and subsequently maintained, is that each pi is σiq for some substitution σi mapping variables to patterns.

How to proceed. If n=0, the bunch is empty, so you have a possible case q that isn't covered by a pattern. Complain that the ps are not exhaustive. If σ1 is an injective renaming of variables, then p1 catches every case that matches q, so we're warm: if n=1, we win; if n>1 then oops, there's no way p2 can ever be needed. Otherwise, we have that for some variable x, σ1x is a constructor pattern. In that case split the problem into multiple subproblems, one for each constructor cj of x's type. That is, split the original q into multiple ideal patterns qj = [x:=cj y1 .. yarity(cj)]q, and refine the patterns accordingly for each qj to maintain the invariant, dropping those that don't match.

Let's take the example with {`[], x :: y :: zs`} (using `::` for `cons`). We start with

```
xs covering  {[], x :: y :: zs}
```

and we have [xs := []] making the first pattern an instance of the ideal. So we split xs, getting

```
[] covering {[]}
x :: ys covering {x :: y :: zs}
```

The first of these is justified by the empty injective renaming, so is ok. The second takes [x := x, ys := y :: zs], so we're away again, splitting ys, getting.

```
x :: [] covering {}
x :: y :: zs covering {x :: y :: zs}
```

and we can see from the first subproblem that we're banjaxed.

The overlap case is more subtle and allows for variations, depending on whether you want to flag up any overlap, or just patterns which are completely redundant in a top-to-bottom priority order. Your basic rock'n'roll is the same. E.g., start with

```
xs covering {[], ys}
```

with [xs := []] justifying the first of those, so split. Note that we have to refine ys with constructor cases to maintain the invariant.

```
[] covering {[], []}
x :: xs covering {y :: ys}
```

Clearly, the first case is strictly an overlap. On the other hand, when we notice that refining an actual program pattern is needed to maintain the invariant, we can filter out those strict refinements that become redundant and check that at least one survives (as happens in the `::` case here).

So, the algorithm builds a set of ideal exhaustive overlapping patterns q in a way that's motivated by the actual program patterns p. You split the ideal patterns into constructor cases whenever the actual patterns demand more detail of a particular variable. If you're lucky, each actual pattern is covered by disjoint nonempty sets of ideal patterns and each ideal pattern is covered by just one actual pattern. The tree of case splits which yield the ideal patterns gives you the efficient jump-table driven compilation of the actual patterns.

The algorithm I've presented is clearly terminating, but if there are datatypes with no constructors, it can fail to accept that the empty set of patterns is exhaustive. This is a serious issue in dependently typed languages, where exhaustiveness of conventional patterns is undecidable: the sensible approach is to allow "refutations" as well as equations. In Agda, you can write (), pronounced "my Aunt Fanny", in any place where no constructor refinement is possible, and that absolves you from the requirement to complete the equation with a return value. Every exhaustive set of patterns can be made *recognizably* exhaustive by adding in enough refutations.

Anyhow, that's the basic picture.

2.2 Haskell Pattern Matching

The datatype declaration for `Expr` gives rise, systematically, to a set of patterns which cover all possible things that a value of type `Expr` can *be*. Let's do the translation

```
data Expr           -- any e :: Expr must be one of
  = T               -- T
  | Var Variable    -- (Var x)          -- where x :: Variable
  | And Expr Expr   -- (And e1 e2)      -- where e1 :: Expr, e2 :: Expr
  | Not Expr        -- (Not e1)         -- where e1 :: Expr
```

You can see that the `T`, `Var`, `And` and `Not` that head up each `data` clause are **constructors**, and live in the value language; the rest of the things in each clause live in the type language, saying what type each component of an `Expr` must have. Each of the corresponding **patterns** consists of the constructor applied to **pattern variables** standing for components which have the given types. Basically, the patterns that show up on the left-hand side of a function are made by repeatedly refining pattern variables to the patterns that their values can possibly take, *as indicated by their type*.

Writing a function by pattern matching does not consist of saying what to *do*: it consists of saying what the output *is* for the possible cases of what the input *is*. You need to analyse the input into cases where you can easily say what the output must be. So, start with one general case...

```
v :: Expr -> [Variable]
v e = undefined
```

...and refine it. Ask "Can you tell what it is yet?". We can't tell what `v e` is without knowing more about `e`. So we'd better *split* e. We know that `e :: Expr`, so we know what patterns its value can match. Make four copies of your program line, and in each, replace `e` by one of the four possible pattern listed above.

```
v :: Expr -> [Variable]
v T           = undefined
v (Var x)     = undefined
v (And e1 e2) = undefined
v (Not e1)    = undefined
```

Now, in each case, can you tell what the output is? The handy thing is that you can make use of recursive calls on components. Assume you already know what `vars e1` and `vars e2` are when you're trying to say what `v (And e1 e2)` must be. If you get the steps right, the program will be correct.

I find it's often helpful to think in terms of concrete examples. Take your test example.

```
v (Not (And (Var "y") T))
```

That's supposed to be `["y"]`, right? Which pattern does it match?

```
Not e1   -- with e1 = And (Var "y") T
```

What's

```
v e1
```

? Looking at it, it had better be

```
["y"]
```

In this example, what's `v (Not e1)` in terms of `v e1`? The very same. That might suggest a suitable expression to replace `undefined` in

```
v (Not e1) = undefined  -- can you tell what this is now?
```

(Of course, a suggestive example is just a good start, not a guarantee of correctness.)

The takeaway messages: (1) build patterns by splitting pattern variables, figuring out the possible patterns by looking at the declaration of the type; (2) assume that recursive calls on components give you the right answer, then try to construct the right answer for the whole problem.

Shameless plug: shplit is a simple tool I built for my students, capturing message (1) mechanically.

2.3 Complex pattern matching

When I was a wee boy back in the 1980s, I implemented a functional language with complex patterns in that style. It amounted to allowing ++ in patterns. The resulting patterns were ambiguous, so matching involved a backtracking search process: the programmer could effectively specify whether to minimize or maximize the length of the prefix matching the pattern left of ++. The language had a form of "pattern guards", so that a candidate match could be tested to see if a subsequent computation succeeded and rejected in the case of failure. The resulting programs were often in-your-face obvious as to their meaning. It was a lot of fun.

These days, when faced with such problems, I reach for `span`, and if that won't cut it, I use parser combinators.

```
span :: (a -> Bool) -> [a] -> ([a], [a])
```

> span, applied to a predicate p and a list xs, returns a tuple where first element is longest prefix (possibly empty) of xs of elements that satisfy p and second element is the remainder of the list

So, in particular `span (/= ',')` will split a `String` into whatever is before the first comma (or the whole thing if there is no comma), and the rest (starting with the comma if there is one).

And if that won't cut it, I use parser combinators.

But I always remember how it used to be easy.

2.4 How to return an element before I entered?

It is well observed in the comments that your `otherwise` (which is just a synonym for `True`) needs an = sign like any other guard, but I'd make a few other adjustments.

Partial functions `head` and `tail` are probably better avoided, especially as there is a good way to solve this problem with pattern matching.

```
elementBefore :: Eq a => a -> [a] -> Maybe a
elementBefore elt (x : xs@(y : _)) | y == elt = Just x
                                   | otherwise = elementBefore elt xs
elementBefore _ _ = Nothing
```

The key is the use of `@` to make an "as-pattern", simultaneously naming the tail of the list `xs` (for use if we're unlucky) and matching it as `(y : _)` (so we can see if we've won).

When I was a child, my father and I would have written something like this

```
elementBefore elt (_ ++ x : elt : _) = Just x
elementBefore _ _ = Nothing
```

but that has always been too simple to be valid Haskell.

2.5 Buzzard Bazooka Zoom

The specification is not entirely clear, but it sounds like you want to collect all the characters which occur three places after a `'Z'` in the input, so that from

```
"BUZZARD BAZOOKA ZOOM"
```

we get

```
"RDKM"
```

Without a clearer presentation of the problem, it is difficult to give precise advice. But I hope I can help you get past some of the small irritations, so that you can engage with the actual logic of the problem.

Let's start with the type. You have

```
someFun :: String => String -> String -> String
```

but left of => is the place for *properties* of type expressions, usually involving variables that could stand for lots of types, such as `Eq a` (meaning that whatever type `a` is, we can test equality). `String` is a type, not a property, so it cannot stand left of =>. Drop it. That gives

```
someFun  :: String -- input
         -> String -- accumulating the output (?)
         -> String -- output
```

It is not clear whether you really need an accumulator. Suppose you know the output for

```
"ZARD BAZOOKA BOOM"  -- "DKM", right?
```

Can you compute the output for

```
"ZZARD BAZOOKA BOOM"  -- "RDKM"
```

? Just an extra `'R'` on the front, right? You're using tail recursion to *do* the next thing, when it is usually simpler to think about what things should *be*. If you know what the output *is* for the tail of the list, then say what the output *is* for the whole of the list. Why not just map input to output directly, so

```
someFun :: String -> String
```

Now, pattern matching, start with the simplest possible pattern

```
someFun s = undefined
```

Can you see enough about the input to determine the output? Clearly not. It matters whether the input is empty or has a first character. Split into two cases.

```
someFun ""       = undefined
someFun (c : s) = undefined   -- c is the first Char, s is the rest of the String
```

It also matters whether the first character is `'Z'` or not. Be careful to use *single* quotes for `Char` and *double* quotes for `String`: they are different types.

```
someFun ""          = undefined
someFun ('Z' : s)   = undefined   -- the first Char is Z
someFun (c : s)     = undefined
```

In the case wit `'Z'`, you also want to make sure that `s` has at least three characters, and we care about the third, so

```
someFun ""                           = undefined   -- input empty
someFun ('Z' : s@(_ : _ : d : _))    = undefined   -- first is 'Z' and d is 3 later
someFun (c : s)                      = undefined   -- input nonempty
```

The `@` is an "as pattern", allowing me to name the whole tail `s` and also check that it matches `(_ : _ : d : _)`, grabbing the third character after the `'Z'`.

So far, I've given no thought to the output, just what I need to see about the input. Let's figure out what the output must be. In the first case, empty input gives empty output

```
someFun ""                           = ""
someFun ('Z' : s@(_ : _ : d : _))    = undefined   -- first is 'Z' and d is 3 later
someFun (c : s)                      = undefined   -- input nonempty
```

and in the other two cases, we can assume that `someFun s` already tells us the output for the tail of the list, so we just need to figure out how to finish the output for the whole list. In the last line, the output for the tail is just what we want.

```
someFun ""                           = ""
someFun ('Z' : s@(_ : _ : d : _))    = undefined   -- first is 'Z' and d is 3 later
someFun (c : s)                      = someFun s
```

But in the case where we've found that `d` is three places after the initial `'Z'`, we need to make sure `d` is at the start of the output.

```
someFun ""                           = ""
someFun ('Z' : s@(_ : _ : d : _))    = d : someFun s
someFun (c : s)                      = someFun s
```

Just checking:

```
*Main> someFun "BUZZARD BAZOOKA ZOOM"
"RDKM"
```

The key idea is to figure out how to express the output for the whole input in terms of the output for its pieces: what *it is*, not what *to do*. Here, you can assume that the output for the tail, `s` is correctly computed, so you just need to figure out whether you have anything extra to return.

2.6 Why ++ is not allowed in pattern matching?

This is a deserving question, and it has so far received sensible answers (mutter only constructors allowed, mutter injectivity, mutter ambiguity), but there's still time to change all that.

We can say what the rules are, but most of the explanations for why the rules are what they are start by over-generalising the question, addressing why we can't pattern match against any old function (mutter Prolog). This is to ignore the fact that ++ isn't any old function: it's a (spatially) *linear* plugging-stuff-together function, induced by the zipper-structure of lists. Pattern matching is about taking stuff apart, and indeed, notating the process in terms of the plugger-togetherers and pattern variables standing for the components. Its motivation is clarity. So I'd like

```
lookup :: Eq k => k -> [(k, v)] -> Maybe v
lookup k (_ ++ [(k, v)] ++ _) = Just v
lookup _ _                    = Nothing
```

and not only because it would remind me of the fun I had thirty years ago when I implemented a functional language whose pattern matching offered exactly that.

The objection that it's ambiguous is a legitimate one, but not a dealbreaker. Plugger-togetherers like ++ offer only finitely many decompositions of finite input (and if you're working on infinite data, that's your own lookout), so what's involved is at worst *search*, rather than *magic* (inventing arbitrary inputs that arbitrary functions might have thrown away). Search calls for some means of prioritisation, but so do our ordered matching rules. Search can also result in failure, but so, again, can matching.

We have a sensible way to manage computations offering alternatives (failure and choice) via the `Alternative` abstraction, but we are not used to thinking of pattern matching as a form of such computation, which is why we exploit `Alternative` structure only in the *expression* language. The noble, if quixotic, exception is match-failure in `do`-notation, which calls the relevant `fail` rather than necessarily crashing out. Pattern matching is an attempt to compute an environment suitable for the evaluation of a 'right-hand side' expression; failure to compute such an environment is already handled, so why not choice?

(**Edit:** I should, of course, add that you only really need search if you have more than one stretchy thing in a pattern, so the proposed `xs++[x]` pattern shouldn't trigger any choices. Of course, it takes time to find the end of a list.)

Imagine there was some sort of funny bracket for writing `Alternative` computations, e.g., with `(|)` meaning `empty`, `(|a1|a2|)` meaning `(|a1|) <|> (|a2|)`, and a regular old `(|f s1 .. sn|)` meaning `pure f <*> s1 .. <*> sn`. One might very well also imagine `(|case a of {p1 -> a1; .. pn->an}|)` performing a sensible translation of search-patterns (e.g. involving ++) in terms of `Alternative` combinators. We could write

```
lookup :: (Eq k, Alternative a) => k -> [(k, v)] -> a k
lookup k xs = (|case xs of _ ++ [(k, v)] ++ _ -> pure v|)
```

We may obtain a reasonable language of search-patterns for any datatype generated by fixpoints of differentiable functors: symbolic differentiation is exactly what turns tuples of structures into choices of possible substructures. Good old ++ is just the sublists-of-lists example (which is confusing, because a list-with-a-hole-for-a-sublist looks a lot like a list, but the same is not true for other datatypes).

Hilariously, with a spot of `LinearTypes`, we might even keep hold of holey data by their holes as well as their root, then plug away destructively in constant time. It's scandalous behaviour only if you don't notice you're doing it.

Chapter 3

Recursion

3.1 What are paramorphisms?

Yes, that's `para`. Compare with catamorphism, or `foldr`:

```
para  :: (a -> [a] -> b -> b) -> b -> [a] -> b
foldr :: (a ->        b -> b) -> b -> [a] -> b

para  c n (x : xs) = c x xs (para c n xs)
foldr c n (x : xs) = c x    (foldr c n xs)
para  c n []       = n
foldr c n []       = n
```

Some people call paramorphisms "primitive recursion" by contrast with catamorphisms (`foldr`) being "iteration".

Where `foldr`'s two parameters are given a recursively computed value for each recursive subobject of the input data (here, that's the tail of the list), `para`'s parameters get both the original subobject and the value computed recursively from it.

An example function that's nicely expressed with `para` is the collection of the proper suffices of a list.

```
suff :: [x] -> [[x]]
suff = para (\ x xs suffxs -> xs : suffxs) []
```

so that

```
suff "suffix" = ["uffix", "ffix", "fix", "ix", "x", ""]
```

Possibly simpler still is

```
safeTail :: [x] -> Maybe [x]
safeTail = para (\ _ xs _ -> Just xs) Nothing
```

in which the "cons" branch ignores its recursively computed argument and just gives back the tail. Evaluated lazily, the recursive computation never happens and the tail is extracted in constant time.

You can define `foldr` using `para` quite easily; it's a little trickier to define `para` from `foldr`, but it's certainly possible, and everyone should know how it's done!

```
foldr c n =       para  (\ x  xs  t ->           c x    t)       n
para  c n = snd . foldr (\ x (xs, t) -> (x : xs, c x xs t)) ([], n)
```

The trick to defining `para` with `foldr` is to reconstruct a *copy* of the original data, so that we gain access to a copy of the tail at each step, even though we had no access to the original. At the end, `snd` discards the copy of the input and gives just the output value. It's not very efficient, but if you're interested in sheer expressivity, `para` gives you no more than `foldr`. If you use this `foldr`-encoded version of `para`, then `safeTail` will take linear time after all, copying the tail element by element.

So, that's it: `para` is a more convenient version of `foldr` which gives you immediate access to the tail of the list as well as the value computed from it.

In the general case, working with a datatype generated as the recursive fixpoint of a functor

```
data Fix f = In (f (Fix f))
```

you have

```
cata :: Functor f => (f         t  -> t) -> Fix f -> t
para :: Functor f => (f (Fix f, t) -> t) -> Fix f -> t

cata phi (In ff) = phi (fmap (cata phi) ff)
para psi (In ff) = psi (fmap keepCopy   ff) where
  keepCopy x = (x, para psi x)
```

and again, the two are mutually definable, with `para` defined from `cata` by the same "make a copy" trick

```
para psi = snd . cata (\ fxt -> (In (fmap fst fxt), psi fxt))
```

Again, `para` is no more expressive than `cata`, but more convenient if you need easy access to substructures of the input.

Edit: I remembered another nice example.

Consider binary search trees given by `Fix TreeF` where

```
data TreeF sub = Leaf | Node sub Integer sub
```

and try defining insertion for binary search trees, first as a `cata`, then as a `para`. You'll find the `para` version much easier, as at each node you will need to insert in one subtree but preserve the other as it was.

3.2 Why can you reverse list with foldl, but not with foldr in Haskell

Every `foldl` is a `foldr`.

Let's remember the definitions.

```
foldr :: (a -> s -> s) -> s -> [a] -> s
foldr f s []       = s
foldr f s (a : as) = f a (foldr f s as)
```

That's the standard issue one-step iterator for lists. I used to get my students to bang on the tables and chant "What do you do with the empty list? What do you do with `a : as`"? And that's how you figure out what `s` and `f` are, respectively.

If you think about what's happening, you see that `foldr` effectively computes a big composition of `f a` functions, then applies that composition to `s`.

```
foldr f s [1, 2, 3]
= f 1 . f 2 . f 3 . id $ s
```

Now, let's check out `foldl`

```
foldl :: (t -> a -> t) -> t -> [a] -> t
foldl g t []       = t
foldl g t (a : as) = foldl g (g t a) as
```

That's also a one-step iteration over a list, but with an accumulator which changes as we go. Let's move it last, so that everything to the left of the list argument stays the same.

```
flip . foldl :: (t -> a -> t) -> [a] -> t -> t
flip (foldl g) []       t = t
flip (foldl g) (a : as) t = flip (foldl g) as (g t a)
```

Now we can see the one-step iteration if we move the = one place leftward.

```
flip . foldl :: (t -> a -> t) -> [a] -> t -> t
flip (foldl g) []       = \ t -> t
flip (foldl g) (a : as) = \ t -> flip (foldl g) as (g t a)
```

In each case, we compute *what we would do if we knew the accumulator*, abstracted with `\ t ->`. For `[]`, we would return `t`. For `a : as`, we would process the tail with `g t a` as the accumulator.

But now we can transform `flip (foldl g)` into a `foldr`. Abstract out the recursive call.

```
flip . foldl :: (t -> a -> t) -> [a] -> t -> t
flip (foldl g) []       = \ t -> t
flip (foldl g) (a : as) = \ t -> s (g t a)
  where s = flip (foldl g) as
```

And now we're good to turn it into a `foldr` where type `s` is instantiated with `t -> t`.

```
flip . foldl :: (t -> a -> t) -> [a] -> t -> t
flip (foldl g) = foldr (\ a s -> \ t -> s (g t a)) (\ t -> t)
```

So `s` says "what `as` would do with the accumulator" and we give back `\ t -> s (g t a)` which is "what `a : as` does with the accumulator". Flip back.

```
foldl :: (t -> a -> t) -> t -> [a] -> t
foldl g = flip (foldr (\ a s -> \ t -> s (g t a)) (\ t -> t))
```

Eta-expand.

```
foldl :: (t -> a -> t) -> t -> [a] -> t
foldl g t as = flip (foldr (\ a s -> \ t -> s (g t a)) (\ t -> t)) t as
```

Reduce the `flip`.

```
foldl :: (t -> a -> t) -> t -> [a] -> t
foldl g t as = foldr (\ a s -> \ t -> s (g t a)) (\ t -> t) as t
```

So we compute "what we'd do if we knew the accumulator", and then we feed it the initial accumulator.

It's moderately instructive to golf that down a little. We can get rid of `\ t ->`.

```
foldl :: (t -> a -> t) -> t -> [a] -> t
foldl g t as = foldr (\ a s -> s . (`g` a)) id as t
```

Now let me reverse that composition using >>> from `Control.Arrow`.

```
foldl :: (t -> a -> t) -> t -> [a] -> t
foldl g t as = foldr (\ a s -> (`g` a) >>> s) id as t
```

That is, `foldl` computes a big *reverse* composition. So, for example, given `[1,2,3]`, we get

```
foldr (\ a s -> (`g` a) >>> s) id [1,2,3] t
= ((`g` 1) >>> (`g` 2) >>> (`g` 3) >>> id) t
```

where the "pipeline" feeds its argument in from the left, so we get

```
((`g` 1) >>> (`g` 2) >>> (`g` 3) >>> id) t
= ((`g` 2) >>> (`g` 3) >>> id) (g t 1)
= ((`g` 3) >>> id) (g (g t 1) 2)
= id (g (g (g t 1) 2) 3)
= g (g (g t 1) 2) 3
```

and if you take `g = flip (:)` and `t = []` you get

```
flip (:) (flip (:) (flip (:) [] 1) 2) 3
= flip (:) (flip (:) (1 : []) 2) 3
= flip (:) (2 : 1 : []) 3
= 3 : 2 : 1 : []
= [3, 2, 1]
```

That is,

```
reverse as = foldr (\ a s -> (a :) >>> s) id as []
```

by instantiating the *general* transformation of `foldl` to `foldr`.

For mathochists only. Do `cabal install newtype` and import `Data.Monoid`, `Data.Foldable` and `Control.Newtype`. Add the tragically missing instance:

```
instance Newtype (Dual o) o where
  pack = Dual
  unpack = getDual
```

Observe that, on the one hand, we can implement `foldMap` by `foldr`

```
foldMap :: Monoid x => (a -> x) -> [a] -> x
foldMap f = foldr (mappend . f) mempty
```

but also vice versa

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f = flip (ala' Endo foldMap f)
```

so that `foldr` accumulates in the monoid of composing endofunctions, but now to get `foldl`, we tell `foldMap` to work in the `Dual` monoid.

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl g = flip (ala' Endo (ala' Dual foldMap) (flip g))
```

What is `mappend` for `Dual (Endo b)`? Modulo wrapping, it's exactly the reverse composition, `>>>`.

3.3 Can fold be used to create infinite lists?

The `foldl` and `foldr` functions are list-*consumers*. As svenningsson's answer rightly points out, `unfoldr` is a list-*producer* which is suitable for capturing the *co*-recursive structure of `fibs`.

However, given that `foldl` and `foldr` are polymorphic in their return types, i.e. what they're producing by consuming a list, it is reasonable to ask whether they might be used to consume one list and produce another. Might any of these produced lists be infinite?

Looking at the definition of `foldl`

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f a []        = a
foldl f a (b : bs)  = foldl f (f a b) bs
```

we see that for `foldl` to produce anything at all, the list it consumes must be finite. Thus if `foldl f a` produces infinite output, it is because `a` is infinite or because `f` sometimes performs infinite list generation.

It's a different story with `foldr`

```
foldr :: (b -> a -> a) -> a -> [b] -> a
foldr f a []        = a
foldr f a (b : bs)  = f b (foldr f a bs)
```

which admits the lazy possibility that `f` might generate some output for each `b` consumed from the input. Operations like

```
map g = foldr (\ b gbs -> g b : gbs) []   -- golfers prefer ((:) . g)
stutter = foldr (\ x xxs -> x : x : xxs) []
```

producing a little bit of output for each input, deliver infinite output from infinite input.

A cheeky person can thus express any infinitary recursion as a non-recursive `foldr` on an infinite list. E.g.,

```
foldr (\ _ fibs -> 1 : 1 : zipWith (+) fibs (tail fibs)) undefined [1..]
```

(Edit: or, for that matter

```
foldr (\_ fib a b -> a : fib b (a + b)) undefined [1..] 1 1
```

which is closer to the definition in the question.)

although this observation, whilst true, is hardly indicative of a healthy programming style.

3.4 How do I give a Functor instance to a datatype built for general recursion schemes?

This is an old sore for me. The crucial point is that your `ExprF` is functorial in *both* its parameters. So if we had

```
class Bifunctor b where
  bimap :: (x1 -> y1) -> (x2 -> y2) -> b x1 x2 -> b y1 y2
```

then you could define (or imagine a machine defining for you)

```
instance Bifunctor ExprF where
  bimap k1 k2 (Val a)    = Val (k1 a)
  bimap k1 k2 (Add x y)  = Add (k2 x) (k2 y)
```

and now you can have

```
newtype Fix2 b a = MkFix2 (b a (Fix2 b a))
```

accompanied by

```
map1cata2 :: Bifunctor b => (a -> a') -> (b a' t -> t) -> Fix2 b a -> t
map1cata2 e f (MkFix2 bar) = f (bimap e (map1cata2 e f) bar)
```

which in turn gives you that when you take a fixpoint in one of the parameters, what's left is still functorial in the other

```
instance Bifunctor b => Functor (Fix2 b) where
  fmap k = map1cata2 k MkFix2
```

and you sort of get what you wanted. But your `Bifunctor` instance isn't going to be built by magic. And it's a bit annoying that you need a different fixpoint operator and a whole new kind of functor. The trouble is that you now have *two* sorts of substructure: "values" and "subexpressions".

And here's the turn. There is a notion of functor which is *closed* under fixpoints. Turn on the kitchen sink (especially `DataKinds`) and

```
type s :-> t = forall x. s x -> t x

class FunctorIx (f :: (i -> *) -> (o -> *)) where
  mapIx :: (s :-> t) -> f s :-> f t
```

Note that "elements" come in a kind indexed over `i` and "structures" in a kind indexed over some other `o`. We take `i`-preserving functions on elements to `o` preserving functions on structures. Crucially, `i` and `o` can be different.

The magic words are "1, 2, 4, 8, time to exponentiate!". A type of kind `*` can easily be turned into a trivially indexed GADT of kind `() -> *`. And two types can be rolled together to make a GADT of kind `Either () () -> *`. That means we can roll both sorts of substructure together. In general, we have a kind of type level `either`.

```
data Case :: (a -> *) -> (b -> *) -> Either a b -> * where
  CL :: f a -> Case f g (Left a)
  CR :: g b -> Case f g (Right b)
```

equipped with its notion of "map"

```
mapCase :: (f :-> f') -> (g :-> g') -> Case f g :-> Case f' g'
mapCase ff gg (CL fx) = CL (ff fx)
mapCase ff gg (CR gx) = CR (gg gx)
```

So we can refunctor our bifactors as `Either`-indexed `FunctorIx` instances.

And now we can take the fixpoint of any node structure `f` which has places for either elements `p` or subnodes. It's just the same deal we had above.

```
newtype FixIx (f :: (Either i o -> *) -> (o -> *))
              (p :: i -> *)
              (b :: o)
  = MkFixIx (f (Case p (FixIx f p)) b)

mapCata :: forall f p q t. FunctorIx f =>
  (p :-> q) -> (f (Case q t) :-> t) -> FixIx f p :-> t
mapCata e f (MkFixIx node) = f (mapIx (mapCase e (mapCata e f)) node)
```

But now, we get the fact that `FunctorIx` is closed under `FixIx`.

```
instance FunctorIx f => FunctorIx (FixIx f) where
  mapIx f = mapCata f MkFixIx
```

Functors on indexed sets (with the extra freedom to vary the index) can be very precise and very powerful. They enjoy many more convenient closure properties than `Functor`s do. I don't suppose they'll catch on.

3.5 Are there (term-transforming) morphisms in Haskell?

For reference, here are the terms...

```
data Term a =
      Var a
    | Lambda a (Term a)
    | Apply (Term a) (Term a)
```

(I note that the representation of variables is abstracted, which is often a good plan.)

...and here is the proposed function.

```
fmap' :: (Term a → Term a) → Term a → Term a
fmap' f (Var v) = f (Var v)
fmap' f (Lambda v t) = Lambda v (fmap' f t)
fmap' f (Apply t1 t2) = Apply (fmap' f t1) (fmap' f t2)
```

What bothers me about this definition is that `f` is only ever applied to terms of form `(Var v)`, so you might as well implement *substitution*.

```
subst :: (a → Term a) → Term a → Term a
subst f (Var v) = f v
subst f (Lambda v t) = Lambda v (subst f t)
subst f (Apply t1 t2) = Apply (subst f t1) (subst f t2)
```

If you took slightly more care to distinguish bound from free variables, you'd be able to make `Term` a `Monad` with substitution implementing `(>>=)`. In general, terms can have a `Functor` structure for renaming and a `Monad` structure for substitution. There's a lovely paper by Bird and Paterson about that, but I digress.

Meanwhile, if you *do* want to act other than at variables, one general approach is to use general purpose traversal toolkits like uniplate, as augustss suggests. Another possibility, perhaps slightly more disciplined, is to work with the 'fold' for your type.

```
tmFold :: (x -> y) -> (x -> y -> y) -> (y -> y -> y) -> Term x -> y
tmFold v l a (Var x)       = v x
tmFold v l a (Lambda x t)  = l x (tmFold v l a t)
tmFold v l a (Apply t t')  = a (tmFold v l a t) (tmFold v l a t')
```

Here, `v`, `l` and `a` define an alternative *algebra* for your `Term`-forming operations, only acting on `y`, rather than `Term x`, explaining how to handle variables, lambdas and applications. You might choose `y` to be `m (Term x)` for some suitable monad `m` (e.g., threading an environment for the variables), rather than just `Term x` itself. Each subterm is processed to give a `y`, then the appropriate function is chosen to produce the `y` for the whole term. The fold captures the standard recursion pattern.

Ordinary first-order datatypes (and some well-behaved higher-order datatypes) can all be equipped with fold-operators. At a cost to readability, you can even write the fold operator once and for all.

```
data Fix f = In (f (Fix f))

fixFold :: Functor f => (f y -> y) -> Fix f -> y
fixFold g (In xf) = g (fmap (fixFold g) xf)

data TermF a t
  = VarF a
  | LambdaF a t
  | ApplyF t t

type Term a = Fix (TermF a)
```

Unlike your recursive `Term a`, this `TermF a t` explains how to make *one* layer of a term, with `t` elements in the subterm places. We get back the recursive `Term` structure by using the recursive `Fix` type. We lose a little cosmetically, in that each layer has an extra `In` wrapping it. We can define

```
var x      = In (VarF x)
lambda x t = In (LambdaF x t)
apply t t' = In (Apply x t t')
```

but we can't use those definitions in pattern matching. The payoff, though, is that we can use the generic `fixFold` at no extra cost. To compute a `y` from a term, we need only give a function of type

```
TermF a y -> y
```

which (just like `v`, `l`, and `a` above) explains how to handle any term whose subterms have already been processed to values of type `y`. By being explicit in types about what one layer consists of, we can tap into the general pattern of working layer by layer.

3.6 Is this Fibonacci sequence function recursive?

My, what a rat's nest of subtle terminological distinctions. What is "this"?

```
fib=0:1:zipWith (+) fib (tail fib)
```

It is not a recursive function. It is not recursive data. It is a recursive definition.

What is being defined?

```
fib
```

What type of thing is `fib`, according to this definition?

```
[Integer]
```

A list of integers (or perhaps a list of any old numeric stuff).

Is `fib` a function? No, it is a list. Is `fib` recursively defined? Yes. Would `fib` be recursively defined if we replaced `zipWith` by a nonrecursive function of the same type (e.g. `\ f xs ys -> xs`)? Yes, although it would be a different recursively defined list.

Is `fib` a cyclic list? No. Does "recursive data structure" mean "cyclic data structure"? Not according to Hoare's paper, "Recursive Data Structures": http://portal.acm.org/book_gateway.cfm?id=63445&type=pdf&p217-hoare.pdf&coll=&dl=&CFID=15151515&CFTOKEN=6184618

In a typed setting, "recursive data structure" means no more or less than "inhabitant of a recursively defined type". Correspondingly `"fred"` is a recursive data structure, even though it is not recursively defined, and indeed it can be acted upon by recursive functions such as `++`.

The phrase "recursive function" means "recursively defined function". The phrase "recursive value" means "recursively defined value", such as exist in nonstrict languages: strict languages have the "value recursion" problem.

And if you think that's pedantic, try defining `fib` that way in a *total* programming language, and you'll discover that the notion of "recursive definition" splits into "definition by structural recursion" (consuming data in a way which stops) and "definition by guarded corecursion" (producing data in a way which goes), and that `fib` is of the latter variety. In that setting, the productivity of `fib` depends crucially on the laziness of `zipWith`. In the Haskell setting, of course, you don't need to worry about any of that stuff to figure out what sort of definition something is, just to figure out whether it has half a chance of actually working.

3.7 Can someone explain this lazy Fibonacci solution?

Your "Because" is not telling the whole story. You're truncating the lists at "the story so far" and evaluating eagerly, then wondering where the rest comes from. That's not quite to grasp what's really going on, so good question.

What gets computed when you make the definition

```
fibs = 0 : 1 : zipWith (+) fibs (drop 1 fibs)
```

? Very little. Computation starts once you begin to *use* the list. Lazy computation happens only on demand.

What's a demand? You get to ask "are you `[]` or `x : xs`?" and if it's the latter, you get a handle on the pieces.

When we ask that question of `fibs`, we get that

```
fibs = x0 : xs0
x0  = 0
xs0 = 1 : zipWith (+) fibs (drop 1 fibs)
```

but that means (substituting for `fibs` and then `x0`)

```
xs0 = 1 : zipWith (+) (0 : xs0) (drop 1 (0 : xs0))
```

and when we ask again, we get that

```
xs0 = x1 : xs1
x1  = 1
xs1 = zipWith (+) (0 : xs0) (drop 1 (0 : xs0))
```

so

```
xs1 = zipWith (+) (0 : 1 : xs1) (drop 1 (0 : 1 : xs1))
```

but now it gets interesting, because we have to do some work. Just enough work to answer the question, mind? When we look at `xs1`, we force `zipWith` which forces `drop`.

```
xs1 = zipWith (+) (0 : 1 : xs1) (drop 1 (0 : 1 : xs1))
    = zipWith (+) (0 : 1 : xs1) (1 : xs1)
    = (0 + 1) : zipWith (+) (1 : xs1) xs1
```

so

```
xs1 = x2 : xs2
x2  = 0 + 1 = 1
xs2 = zipWith (+) (1 : xs1) xs1
    = zipWith (+) (1 : 1 : xs2) (1 : xs2)
```

See? We've maintained that we still know the first two elements of one zipped list, and the first element of the other. That means we'll be able to deliver the next output *and* refresh our "buffer". When we look at `xs2`, we get

```
xs2 = zipWith (+) (1 : 1 : xs2) (1 : xs2)
    = (1 + 1) : zipWith (1 : xs2) xs2
xs2 = x3 : xs3
x3  = 1 + 1 = 2
xs3 = zipWith (1 : xs2) xs2
    = zipWith (1 : 2 : xs3) (2 : xs3)
```

and we're good to go again!

Each time we demand the next element, we also move one step further away from `zipWith` running out of elements, which is just as well, just in the nick of time.

None of the discipline that makes values show up in the nick of time is expressed in the types. At the moment, it's for programmers to make sure that well typed programs don't go wrong by running out of data when a demand is made. (I have plans to do something about that, but I'll try not to digress here.)

The key is that lazy, "on demand" computation means that we *don't* have to truncate lists to just the elements we can see when the process starts. We just need to know that we can always take the next step.

3.8 Monoidal folds on fixed points

Here's the essence of a solution. I've switched on

```
{-# LANGUAGE DeriveFunctor, DeriveFoldable, DeriveTraversable, PatternSynonyms #-}
```

Let's just recap fixpoints and catamorphisms.

```
newtype Fix f = In {out :: f (Fix f)}

cata :: Functor f => (f t -> t) -> Fix f -> t
cata alg = alg . fmap (cata alg) . out
```

The algebra, `alg :: f t -> t`, takes a node where the children have already been replaced by a `t` value, then returns the `t` for the parent. The `cata` operator works by unpacking the parent node, processing all its children recursively, then applying `alg` to finish the job.

So, if we want to count leaves in such a structure, we can start like this:

```
leaves :: (Foldable f, Functor f) => Fix f -> Integer
leaves = cata sumOrOne where
  -- sumOrOne :: f Integer -> Integer
```

The algebra, `sumOrOne` can see the number of leaves in each child of the parent node. We can use `cata` because `f` is a `Functor`. And because `f` is `Foldable`, we can compute the total number of leaves in the children.

```
  sumOrOne fl = case sum fl of
    ...
```

There are then two possibilities: if the parent has no children, its leaf sum will be `0`, which we can detect, but that means the parent is itself a leaf, so `1` should be returned. Otherwise, the leaf sum will be nonzero, in which case the parent is not a leaf, so its leaf sum is indeed the total leaf sum of its children. That gives us

```
leaves :: (Foldable f, Functor f) => Fix f -> Integer
leaves = cata sumOrOne where
  sumOrOne fl{- number of leaves in each child-} = case sum fl of
    0 -> 1  -- no leaves in my children means I am a leaf
    l -> l  -- otherwise, pass on the total
```

A quick example, based on Hutton's Razor (the expression language with integers and addition, which is often the simplest thing that illustrates the point). The expressions are generated from Hutton's functor.

```
data HF h = Val Int | h :+: h deriving (Functor, Foldable, Traversable)
```

I introduce some pattern synonyms to recover the look and feel of a bespoke type.

```
pattern V x    = In (Val x)
pattern s :+ t = In (s :+: t)
```

I cook up a quick example expression, with some leaves that are three levels deep.

```
example :: Fix HF
example = (V 1 :+ V 2) :+ ((V 3 :+ V 4) :+ V 5)
```

Sure enough

```
Ok, modules loaded: Leaves.
*Leaves> leaves example
5
```

An alternative approach is to be functorial and foldable in substructures of interest, in this case, stuff at leaves. (We get exactly the free monads.)

```
data Tree f x = Leaf x | Node (f (Tree f x)) deriving (Functor, Foldable)
```

Once you've made the leaf/node separation part of your basic construction, you can visit the leaves directly with `foldMap`. Throwing in a bit of `Control.Newtype`, we get

```
ala' Sum foldMap (const 1)  :: Foldable f => f x -> Integer
```

which is below the Fairbairn Threshold (i.e., short enough not to need a name and all the clearer for not having one).

The trouble, of course, is that data structures are often functorial in "substructures of interest" in multiple interesting but conflicting ways. Haskell isn't always the best at letting us access "found functoriality": we somehow have to predict the functoriality we need when we parametrise a data type at declaration time. But there's still time to change all that...

3.9 List Created Evaluating List Elements

Here's my thought process for this problem... We want to chop a list into 'chains' (so a list of lists), given a test to see if two elements link up.

```
chains :: (x -> x -> Bool) -> [x] -> [[x]]
```

I don't remember any such thing in the library, so I decide to roll my own. I want to identify a suitable recursion strategy for processing the list.

Can I just think about elements? No: I quickly rule out `map` and `foldMap`, as elements don't seem to be treated independently of each other in this problem.

Next, I ask 'Does the output type have a *list algebra*?'. That may not sound like an obvious thing to think, phrased that way, but it unpacks to the following sensible question. Are there 'nil' and 'cons' operations that build up outputs (lists of chains), instead of inputs (lists)? If so, I can use `foldr` to transform input nil-and-cons into output nil-and-cons, like this.

```
chains :: (x -> x -> Bool) -> [x] -> [[x]]
chains link = foldr chCons chNil where
  -- chNil :: [[x]]
  -- chCons :: x -> [[x]] -> [[x]]
```

It's clear what `chNil` has to be, as I'm grouping the original elements. Empty in? Empty out!

```
chains :: (x -> x -> Bool) -> [x] -> [[x]]
chains link = foldr chCons [] where
  -- chCons :: x -> [[x]] -> [[x]]
```

Can I write `chCons`? Suppose I get a list of chains: how do I add a new element? Well, if there's a front chain I can link to then I should grow that chain, otherwise I should start a new chain. So I have a special case for a nonempty chain at the start of a nonempty list of chains, and a default to cons a singleton.

```
chains :: (x -> x -> Bool) -> [x] -> [[x]]
chains link = foldr chCons [] where
  chCons y (xs@(x : _) : xss) | link y x  = (y : xs) : xss
  chCons y xss                            = [y] : xss
```

And we're home!

```
> chains (\ x y -> x + 1 == y) [1,2,3,4,5,6,8,9,10]
[[1,2,3,4,5,6],[8,9,10]]
```

A bunch of operators has an *algebra* for a given type if you can implement those operators for values of that type. The constructors of a datatype are just one algebra, one implementation of a bunch of operators, building values in that very datatype. A good way to compute with inputs from a datatype is to implement its algebra for your desired type of outputs. The point of `foldr` is to capture this 'find the algebra' pattern, and it's right on the money for this problem.

3.10 Functions of GADTs

The error you report is not the only error.

Let's put on the special spectacles which show the things usually kept invisible by "type inference".

Firstly, the data constructor:

```
Simple :: forall a. (Typeable a, Show a) =>
          Message -> (String -> a) -> Question
```

Effectively, a value of type `Question` looks like

```
Simple {a}{typeableDict4a}{showDict4a} message parser
```

where I've written the invisible things in braces. The constructor packs up a type and the two typeclass dictionaries that give the implementations for the members of `Typeable` and `Show`.

Now let's have the main program. I've renamed the type variable to make a point.

```
runQuestion :: forall b. (Typeable b, Show b) => Question -> IO b
```

The type to be given back is chosen by the caller of `runQuestion`, separately from whatever type is packed inside the argument of type `Question`. Now let's fill in the invisible components in the program itself.

```
runQuestion {b}{typeableDict4b}{showDict4b}
  (Simple {a}{typeableDict4a}{showDict4a} message parser) = do
                        -- so parser :: String -> a
      putStrLn message  -- ok, as message :: String
      ans <- getLine    -- ensures ans :: String
  return $ parser ans   -- has type IO a, not IO b
```

The `parser` computes a value of the type `a` packed up in the `Question`, which is totally separate from the type `b` passed directly to `runQuestion`. The program does not typecheck because there's a conflict between two types which can be made different by the program's caller.

Meanwhile, let's look at `print`

```
print :: forall c. Show c => c -> IO ()
```

When you write

```
main = getLine >>= (runQuestion . getQuestion) >>= print
```

you get

```
main = getLine >>=
  (runQuestion {b}{typeableDict4b}{showDict4b} . getQuestion) >>=
  print {b}{showDict4b}
```

and as the return type of `runQuestion {b}` is `IO b`, it must be the case that `print`'s `c` type is the same as `runQuestion`'s `b` type, but other than that, there is *nothing* to determine which type `b` is, or why it is an instance either of `Typeable` or `Show`. With the type annotation, the need for `Typeable` shows up first (in the `runQuestion` call); without, the need for `Show` in `print` causes the complaint.

The real problem, is that somehow, you seem to want `runQuestion` to deliver a thing in whatever type is hidden inside the question, as if you could somehow write a (dependently typed) program like

```
typeFrom :: Question -> *
typeFrom (Simple {a}{typeableDict4a}{showDict4a} message parser) = a

runQuestion :: (q :: Question) -> IO (typeFrom q)
```

That's a perfectly sensible thing to want, but it isn't Haskell: there's no way to name "the type packed up inside that argument". Everything which involves that type has to live in the scope of the case analysis or pattern match which exposes it. It's your attempt to do the `print` outside that scope that won't be allowed.

Chapter 4

Applicative Functors

4.1 Where to find programming exercises for applicative functors?

It seems amusing to post some questions as an answer. This is a fun one, on the interplay between `Applicative` and `Traversable`, based on sudoku.

(1) Consider

data Triple a = Tr a a a

Construct

```
instance Applicative Triple
instance Traversable Triple
```

so that the `Applicative` instance does "vectorization" and the `Traversable` instance works left-to-right. Don't forget to construct a suitable `Functor` instance: check that you can extract this from either of the `Applicative` or the `Traversable` instance. You may find

```
newtype I x = I {unI :: x}
```

useful for the latter.

(2) Consider

newtype (:.) f g x = Comp {comp :: f (g x)}

Show that

```
instance (Applicative f, Applicative g) => Applicative (f :. g)
instance (Traversable f, Traversable g) => Traversable (f :. g)
```

Now define

```
type Zone = Triple :. Triple
```

Suppose we represent a `Board` as a vertical zone of horizontal zones

```
type Board = Zone :. Zone
```

Show how to rearrange it as a horizontal zone of vertical zones, and as a square of squares, using the functionality of `traverse`.

(3) Consider

newtype Parse x = Parser {parse :: String -> [(x, String)]} deriving Monoid

or some other suitable construction (noting that the library `Monoid` behaviour for |Maybe| is inappropriate). Construct

```
instance Applicative Parse
instance Alternative Parse  -- just follow the `Monoid`
```

and implement

```
ch :: (Char -> Bool) -> Parse Char
```

which consumes and delivers a character if it is accepted by a given predicate.

(4) Implement a parser which consumes any amount of whitespace, followed by a single digit (0 represents blanks)

square :: Parse Int

Use `pure` and `traverse` to construct

```
board :: Parse (Board Int)
```

(5) Consider the constant functors

newtype K a x = K {unK :: a}

and construct

```
instance Monoid a => Applicative (K a)
```

then use `traverse` to implement

```
crush :: (Traversable f, Monoid b) => (a -> b) -> f a -> b
```

Construct `newtype` wrappers for `Bool` expressing its conjunctive and disjunctive monoid structures. Use `crush` to implement versions of `any` and `all` which work for any `Traversable` functor.

(6) Implement

duplicates :: (Traversable f, Eq a) => f a -> [a]

computing the list of values which occur more than once. (Not completely trivial.) (There's a lovely way to do this using differential calculus, but that's another story.)

(7) Implement

complete :: Board Int -> Bool ok :: Board Int -> Bool

which check if a board is (1) full only of digits in [1..9] and (2) devoid of duplicates in any row, column or box.

4.2 N-ary tree traversal

4.2.1 First Attempt: Hard Work

For the case of n-ary trees, there are *three* things going on: numbering elements, numbering trees, and numbering *lists* of trees. It would help to treat them separately. Types first:

```
aNumber   :: a                -- thing to number
          -> Int              -- number to start from
          -> ( (a, Int)       -- numbered thing
             , Int            -- next available number afterwards
             )

ntNumber  :: NT a             -- thing to number
          -> Int              -- number to start from
          -> ( NT (a, Int)    -- numbered thing
             , Int            -- next available number afterwards
             )

ntsNumber :: [NT a]           -- thing to number
          -> Int              -- number to start from
          -> ( [NT (a, Int)]  -- numbered thing
             , Int            -- next available number afterwards
             )
```

Notice that all three types share the same pattern. When you see that there is a pattern that you are following, apparently by coincidence, you know you have an opportunity to learn something. But let's press on for now and learn later.

Numbering an element is easy: copy the starting number into the output and return its successor as the next available.

```
aNumber a i = ((a, i), i + 1)
```

For the other two, the pattern (there's that word again) is

1. split the input into its top-level components
2. number each component in turn, threading the numbers through

It's easy to do the first with pattern matching (inspecting the data visually) and the second with `where` clauses (grabbing the two parts of the output).

For trees, a top level split gives us two components: an element and a list. In the where clause, we call the appropriate numbering functions as directed by those types. In each case, the "thing" output tells us what to put in place of the "thing" input. Meanwhile, we thread the numbers through, so the starting number for the whole is the starting number for the first component, the "next" number for the first component starts the second, and the "next" number from the second is the "next" number for the whole.

```
ntNumber (N a ants) i0  = (N ai aints, i2) where
  (ai,    i1) = aNumber   a    i0
  (aints, i2) = ntsNumber ants i1
```

For lists, we have two possibilities. An empty list has no components, so we return it directly without using any more numbers. A "cons" has two components, we do exactly as we did before, using the appropriate numbering functions as directed by the type.

```
ntsNumber []           i  = ([], i)
ntsNumber (ant : ants) i0 = (aint : aints, i2) where
  (aint,  i1) = ntNumber  ant  i0
  (aints, i2) = ntsNumber ants i1
```

Let's give it a go.

```
> let ntree = N "eric" [N "lea" [N "kristy" [],N "pedro" [] ,N "rafael" []],N "anna" [],
> ntNumber ntree 0
(N ("eric",0) [N ("lea",1) [N ("kristy",2) [],N ("pedro",3) [],N ("rafael",4) []],N ("a
```

So we're there. But are we happy? Well, I'm not. I have the annoying sensation that I wrote pretty much the same type three times and pretty much the same program twice. And if I wanted to do more element-numbering for differently organised data (e.g., your binary trees), I'd have to write the same thing again again. Repetitive patterns in Haskell code are *always* missed opportunities: it's important to develop a sense of self-criticism and ask whether there's a neater way.

4.2.2 Second Attempt: Numbering and Threading

Two of the repetitive patterns we saw, above, are 1. the similarity of the types, 2. the similarity of the way the numbers get threaded.

If you match up the types to see what's in common, you'll notice they're all

```
input -> Int -> (output, Int)
```

for different inputs and outputs. Let's give the largest common component a name.

```
type Numbering output = Int -> (output, Int)
```

Now our three types are

```
aNumber   :: a      -> Numbering (a, Int)
ntNumber  :: NT a   -> Numbering (NT (a, Int))
ntsNumber :: [NT a] -> Numbering [NT (a, Int)]
```

You often see such types in Haskell:

```
            input  -> DoingStuffToGet output
```

Now, to deal with the threading, we can build some helpful tools to work with and combine `Numbering` operations. To see which tools we need, look at how we combine the outputs after we've numbered the components. The "thing" parts of the outputs are always built by applying some functions which don't get numbered (data constructors, usually) to some "thing" outputs from numberings.

To deal with the functions, we can build a gadget that looks a lot like our `[]` case, where no actual numbering was needed.

```
steady :: thing -> Numbering thing
steady x i = (x, i)
```

Don't be put off by the way the type makes it look as if `steady` has only one argument: remember that `Numbering thing` abbreviates a function type, so there really is another `->` in there. We get

```
steady [] :: Numbering [a]
steady [] i = ([], i)
```

just like in the first line of `ntsNumber`.

But what about the other constructors, `N` and `(:)`? Ask `ghci`.

```
> :t steady N
steady N :: Numbering (a -> [NT a] -> NT a)
> :t steady (:)
steady (:) :: Numbering (a -> [a] -> [a])
```

We get numbering operations with *functions* as outputs, and we want to generate the arguments to those function by more numbering operations, producing one big overall numbering operation with the numbers threaded through. One step of that process is to feed a numbering-generated function one numbering-generated input. I'll define that as an infix operator.

```
($$) :: Numbering (a -> b) -> Numbering a -> Numbering b
infixl 2 $$
```

Compare with the type of the explicit application operator, `$`

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

This `$$` operator is "application for numberings". If we can get it right, our code becomes

```
ntNumber  :: NT a -> Numbering (NT (a, Int))
ntNumber  (N a ants)   i = (steady N $$ aNumber a $$ ntsNumber ants) i

ntsNumber :: [NT a] -> Numbering [NT (a, Int)]
ntsNumber []           i = steady [] i
ntsNumber (ant : ants) i = (steady (:) $$ ntNumber ant $$ ntsNumber ants) i
```

with `aNumber` as it was (for the moment). This code just does the data reconstruction, plugging together the constructors and the numbering processes for the components. We had better give the definition of `$$` and make sure it gets the threading right.

```
($$) :: Numbering (a -> b) -> Numbering a -> Numbering b
(fn $$ an) i0 = (f a, i2) where
  (f, i1) = fn i0
  (a, i2) = an i1
```

Here, our old threading *pattern* gets done *once*. Each of `fn` and `an` is a function, expecting a starting number, and the whole of `fn $$ sn` is a function, which gets the starting number `i0`. We thread the numbers through, collecting first the function, then the argument. We then do the actual application and hand back the final "next" number.

Now, notice that in every line of code, the `i` input is fed in as the argument to a numbering process. We can simplify this code by just talking about the *processes*, not the *numbers*.

```
ntNumber  :: NT a -> Numbering (NT (a, Int))
ntNumber  (N a ants)   = steady N $$ aNumber a $$ ntsNumber ants

ntsNumber :: [NT a] -> Numbering [NT (a, Int)]
ntsNumber []           = steady []
ntsNumber (ant : ants) = steady (:) $$ ntNumber ant $$ ntsNumber ants
```

One way to read this code is to filter out all the `Numbering`, `steady` and `$$` uses.

```
ntNumber  :: NT a -> ......... (NT (a, Int))
ntNumber  (N a ants)   = ...... N .. (aNumber a) .. (ntsNumber ants)

ntsNumber :: [NT a] -> ......... [NT (a, Int)]
ntsNumber []           = ...... []
ntsNumber (ant : ants) = ...... (:) .. (ntNumber ant) .. (ntsNumber ants)
```

and you see it just looks like a preorder traversal, reconstructing the original data structure after processing the elements. We're doing the right thing with the *values*, provided `steady` and `$$` are correctly combining the *processes*.

We could try to do the same for `aNumber`

```
aNumber  :: a -> Numbering a
aNumber a = steady (,) $$ steady a $$ ????
```

but the `????` is where we actually need the number. We could build a numbering process that fits in that hole: a numbering process that *issues the next number*.

```
next :: Numbering Int
next i = (i, i + 1)
```

That's the essence of numbering, the "thing" output is the number to be used now (which is the starting number), and the "next" number output is the one after. We may write

```
aNumber a = steady (,) $$ steady a $$ next
```

which simplifies to

```
aNumber a = steady ((,) a) $$ next
```

In our filtered view, that's

```
aNumber a = ...... ((,) a) .. next
```

What we've done is to bottle the idea of a "numbering process" and we've built the right tools to do *ordinary functional programming* with those processes. The threading pattern turns into the definitions of `steady` and `$$`.

Numbering is not the only thing that works this way. Try this...

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

...and you also get some more stuff. I just want to draw attention to the types of `pure` and `<*>`. They're a lot like `steady` and `$$`, but they are not just for `Numbering`. `Applicative` is the type class for *every* kind of process which works that way. I'm not saying "learn `Applicative` now!", just suggesting a direction of travel.

4.2.3 Third Attempt: Type-Directed Numbering

So far, our solution is directed towards one particular data structure, `NT a`, with `[NT a]` showing up as an auxiliary notion because it's used in `NT a`. We can make the whole thing a bit more plug-and-play if we focus on one layer of the type at a time. We defined numbering a list of trees in terms of numbering trees. In general, we know how to number a list of *stuff* if we know how to number each item of *stuff*.

If we know how to number an `a` to get `b`, we should be able to number a *list* of `a` to get a *list* of `b`. We can abstract over "how to process each item".

```
listNumber :: (a -> Numbering b) -> [a] -> Numbering [b]
listNumber na []       = steady []
listNumber na (a : as) = steady (:) $$ na a $$ listNumber na as
```

and now our old list-of-trees-numbering function becomes

```
ntsNumber :: [NT a] -> Numbering [NT (a, Int)]
ntsNumber = listNumber ntNumber
```

which is hardly worth naming. We can just write

```
ntNumber :: NT a -> Numbering (NT (a, Int))
ntNumber (N a ants) = steady N $$ aNumber a $$ listNumber ntNumber ants
```

We can play the same game for the trees themselves. If you know how to number stuff, you know how to number a tree of stuff.

```
ntNumber' :: (a -> Numbering b) -> NT a -> Numbering (NT b)
ntNumber' na (N a ants) = steady N $$ na a $$ listNumber (ntNumber' na) ants
```

Now we can do things like this

```
myTree :: NT [String]
myTree = N ["a", "b", "c"] [N ["d", "e"] [], N ["f"] []]

> ntNumber' (listNumber aNumber) myTree 0
(N [("a",0),("b",1),("c",2)] [N [("d",3),("e",4)] [],N [("f",5)] []],6)
```

Here, the node data is now itself a list of things, but we've been able to number those things individually. Our equipment is more adaptable because each component aligns with one layer of the type.

Now, try this:

```
> :t traverse
traverse :: (Applicative f, Traversable t) => (a -> f b) -> t a -> f (t b)
```

It's an awful lot like the thing we just did, where `f` is `Numbering` and `t` is sometimes lists and sometimes trees.

The `Traversable` class captures what it means to be a type-former that lets you thread some sort of process through the stored elements. Again, the pattern you're using is very common and has been anticipated. Learning to use `traverse` saves a lot of work.

4.2.4 Eventually...

...you'll learn that a thing to do the job of `Numbering` already exists in the library: it's called `State Int` and it belongs to the `Monad` class, which means it must also be in the `Applicative` class. To get hold of it,

```
import Control.Monad.State
```

and the operation which kicks off a stateful process with its initial state, like our feeding-in of `0`, is this thing:

```
> :t evalState
evalState :: State s a -> s -> a
```

Our `next` operation becomes

```
next' :: State Int Int
next' = get <* modify (1+)
```

where `get` is the process that accesses the state, `modify` makes a process that changes the state, and `<*` means "but also do".

If you start you file with the language extension pragma

```
{-# LANGUAGE DeriveFunctor, DeriveFoldable, DeriveTraversable #-}
```

you can declare your datatype like this

```
data NT a = N a [NT a] deriving (Show, Functor, Foldable, Traversable)
```

and Haskell will write `traverse` for you.

Your program then becomes one line...

```
evalState (traverse (\ a -> pure ((,) a) <*> get <* modify (1+)) ntree) 0
--                  ^ how to process one element ^^^^^^^^^^^^^^^
--          ^ how to process an entire tree of elements ^^^^^^^^^
--        ^ processing your particular tree ^^^^^^^^^^^^^^^^^^^^^^^^^
-- ^ kicking off the process with a starting number of 0 ^^^^^^^^^^^^^^^^
```

...but the journey to that one line involves a lot of "bottling the pattern" steps, which takes some (hopefully rewarding) learning.

4.3 Partial application of functions and currying, how to make a better code instead of a lot of maps?

All the suggestions so far are good. Here's another, which might seem a bit weird at first, but turns out to be quite handy in lots of other situations.

Some type-forming operators, like `[]`, which is the operator which maps a type of elements, e.g. `Int` to the type of lists of those elements, `[Int]`, have the property of being `Applicative`. For lists, that means there is some way, denoted by the operator, `<*>`, pronounced "apply", to turn *lists* of functions and *lists* of arguments into *lists* of results.

```
(<*>) :: [s -> t] -> [s] -> [t]    -- one instance of the general type of <*>
```

rather than your ordinary application, given by a blank space, or a `$`

```
($)   :: (s -> t) ->  s  ->  t
```

The upshot is that we can do ordinary functional programming with lists of things instead of things: we sometimes call it "programming in the list *idiom*". The only other ingredient is that, to cope with the situation when some of our components are individual things, we need an extra gadget

```
pure :: x -> [x]   -- again, one instance of the general scheme
```

which wraps a thing up as a list, to be compatible with `<*>`. That is `pure` moves an ordinary value into the applicative idiom.

For lists, `pure` just makes a singleton list and `<*>` produces the result of every pairwise application of one of the functions to one of the arguments. In particular

```
pure f <*> [1..10] :: [Int -> Int -> Int -> Int -> Int]
```

is a list of functions (just like `map f [1..10]`) which can be used with `<*>` again. The rest of your arguments for `f` are not listy, so you need to `pure` them.

```
pure f <*> [1..10] <*> pure 1 <*> pure 2 <*> pure 3 <*> pure 4
```

For lists, this gives

```
[f] <*> [1..10] <*> [1] <*> [2] <*> [3] <*> [4]
```

i.e. the list of ways to make an application from the f, one of the [1..10], the 1, the 2, the 3 and the 4.

The opening `pure f <*> s` is so common, it's abbreviated `f <$> s`, so

```
f <$> [1..10] <*> [1] <*> [2] <*> [3] <*> [4]
```

is what would typically be written. If you can filter out the `<$>`, `pure` and `<*>` noise, it kind of looks like the application you had in mind. The extra punctuation is only necessary because Haskell can't tell the difference between a listy computation of a bunch of functions or arguments and a non-listy computation of what's intended as a single value but happens to be a list. At least, however, the components are in the order you started with, so you see more easily what's going on.

Esoterica. (1) in my (not very) private dialect of Haskell, the above would be

```
(|f [1..10] (|1|) (|2|) (|3|) (|4|)|)
```

where each *idiom bracket*, `(|f a1 a2 ... an|)` represents the application of a pure function to zero or more arguments which live in the idiom. It's just a way to write

```
pure f <*> a1 <*> a2 ... <*> an
```

Idris has idiom brackets, but Haskell hasn't added them. Yet.

(2) In languages with algebraic effects, the idiom of nondeterministic computation is not the same thing (to the typechecker) as the data type of lists, although you can easily convert between the two. The program becomes

f (range 1 10) 2 3 4

where range nondeterministically chooses a value between the given lower and upper bounds. So, nondetermism is treated as a *local* side-effect, not a data structure, enabling operations for failure and choice. You can wrap nondeterministic computations in a *handler* which give meanings to those operations, and one such handler might generate the list of all solutions. That's to say, the extra notation to explain what's going on is pushed to the boundary, rather than peppered through the entire interior, like those `<*>` and `pure`.

Managing the boundaries of things rather than their interiors is one of the few good ideas our species has managed to have. But at least we can have it over and over again. It's why we farm instead of hunting. It's why we prefer static type checking to dynamic tag checking. And so on...

4.4 Translating monad to applicative

I'd write

```
integer :: Parser Integer
integer = read <$ many1 space <*> many1 digit
```

There's a bunch of left associative (like application) parser-building operators `<$>`, `<*>`, `<$`, `<*`. The thing in the far left should be the pure function which assembles the result value from the component values. The thing on the right of each operator should be a parser, collectively giving the components of the grammar left-to-right. Which operator to use depends on two choices, as follows.

```
  the thing to the right is    signal  / noise
  _________________________
  the thing to the left is \
                            +--------------------
                   pure /   |   <$>       <$
                 a parser   |   <*>       <*
```

So, having chosen `read :: String -> Integer` as the pure function which is going to deliver the semantics of the parser, we can classify the leading space as "noise" and the bunch of digits as "signal", hence

```
 read <$ many1 space <*> many1 digit
 (..)    (.........)     (.........)
 pure    noise parser     |
 (.................)      |
     parser              signal parser
 (..............................)
                     parser
```

You can combine multiple possibilities with

```
p1 <|> ... <|> pn
```

and express impossibility with

```
empty
```

It's seldom necessary to name components in parsers, and the resulting code looks more like a grammar with added semantics.

4.5 Applicatives compose, monads don't

If we compare the types

```
(<*>) :: Applicative a => a (s -> t) -> a s -> a t
(>>=) :: Monad m =>       m s -> (s -> m t) -> m t
```

we get a clue to what separates the two concepts. That `(s -> m t)` in the type of `(>>=)` shows that a value in `s` can determine the behaviour of a computation in `m t`. Monads allow interference between the value and computation layers. The `(<*>)` operator allows no such interference: the function and argument computations don't depend on values. This really bites. Compare

```
miffy :: Monad m => m Bool -> m x -> m x -> m x
miffy mb mt mf = do
  b <- mb
  if b then mt else mf
```

which uses the result of some effect to decide between two *computations* (e.g. launching missiles and signing an armistice), whereas

```
iffy :: Applicative a => a Bool -> a x -> a x -> a x
iffy ab at af = pure cond <*> ab <*> at <*> af where
  cond b t f = if b then t else f
```

which uses the value of `ab` to choose between *the values of* two computations `at` and `af`, having carried out both, perhaps to tragic effect.

The monadic version relies essentially on the extra power of `(>>=)` to choose a computation from a value, and that can be important. However, supporting that power makes monads hard to compose. If we try to build 'double-bind'

```
(>>>>==) :: (Monad m, Monad n) => m (n s) -> (s -> m (n t)) -> m (n t)
mns >>>>== f = mns >>-{-m-} \ ns -> let nmnt = ns >>= (return . f) in ???
```

we get this far, but now our layers are all jumbled up. We have an `n (m (n t))`, so we need to get rid of the outer `n`. As Alexandre C says, we can do that if we have a suitable

```
swap :: n (m t) -> m (n t)
```

to permute the `n` inwards and `join` it to the other `n`.

The weaker 'double-apply' is much easier to define

```
(<<**>>) :: (Applicative a, Applicative b) => a (b (s -> t)) -> a (b s) -> a (b t)
abf <<**>> abs = pure (<*>) <*> abf <*> abs
```

because there is no interference between the layers.

Correspondingly, it's good to recognize when you really need the extra power of `Monad`s, and when you can get away with the rigid computation structure that `Applicative` supports.

Note, by the way, that although composing monads is difficult, it might be more than you need. The type `m (n v)` indicates computing with `m`-effects, then computing with `n`-effects to a `v`-value, where the `m`-effects finish before the `n`-effects start (hence the need for `swap`). If you just want to interleave `m`-effects with `n`-effects, then composition is perhaps too much to ask!

4.6 Examples Separating Functor, Applicative and Monad

My style may be cramped by my phone, but here goes.

```
newtype Not x = Kill {kill :: x -> Void}
```

cannot be a Functor. If it were, we'd have

```
kill (fmap (const ()) (Kill id)) () :: Void
```

and the Moon would be made of green cheese.

Meanwhile

```
newtype Dead x = Oops {oops :: Void}
```

is a functor

```
instance Functor Dead where
  fmap f (Oops corpse) = Oops corpse
```

but cannot be applicative, or we'd have

```
oops (pure ()) :: Void
```

and Green would be made of Moon cheese (which can actually happen, but only later in the evening).

(Extra note: `Void`, as in `Data.Void` is an empty datatype. If you try to use `undefined` to prove it's a Monoid, I'll use `unsafeCoerce` to prove that it isn't.)

Joyously,

```
newtype Boo x = Boo {boo :: Bool}
```

is applicative in many ways, e.g., as Dijkstra would have it,

```
instance Applicative Boo where
  pure _ = Boo True
  Boo b1 <*> Boo b2 = Boo (b1 == b2)
```

but it cannot be a Monad. To see why not, observe that return must be constantly `Boo True` or `Boo False`, and hence that

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

cannot possibly hold.

Oh yeah, I nearly forgot

```
newtype Thud x = The {only :: ()}
```

is a Monad. Roll your own.

Plane to catch...

4.7 Parsec: Applicatives vs Monads

It might be worth paying attention to the key semantic difference between `Applicative` and `Monad`, in order to determine when each is appropriate. Compare types:

```
(<*>) :: m (s -> t) -> m s -> m t
(>>=) :: m s -> (s -> m t) -> m t
```

To deploy $<*>$, you choose two computations, one of a function, the other of an argument, then their values are combined by application. To deploy $>>=$, you choose one computation, and you explain how you will make use of its resulting values to choose the next computation. It is the difference between "batch mode" and "interactive" operation.

When it comes to parsing, `Applicative` (extended with failure and choice to give `Alternative`) captures the *context-free* aspects of your grammar. You will need the extra power that `Monad` gives you only if you need to inspect the parse tree from part of your input in order to decide what grammar you should use for another part of your input. E.g., you might read a format descriptor, then an input in that format. Minimizing your usage of the extra power of monads tells you which value-dependencies are essential.

Shifting from parsing to parallelism, this idea of using $>>=$ only for essential value-dependency buys you clarity about opportunities to spread load. When two computations are combined with $<*>$, neither need wait for the other. Applicative-when-you-can-but-monadic-when-you-must is the formula for speed. The point of `ApplicativeDo` is to automate the dependency analysis of code which has been written in monadic style and thus accidentally oversequentialised.

Your question also relates to coding style, about which opinions are free to differ. But let me tell you a story. I came to Haskell from Standard ML, where I was used to writing programs in direct style even if they did naughty things like throw exceptions or mutate references. What was I doing in ML? Working on an implementation of an ultra-pure type theory (which may not be named, for legal reasons). When working *in* that type theory, I couldn't write direct-style

programs which used exceptions, but I cooked up the applicative combinators as a way of getting as close to direct style as possible.

When I moved to Haskell, I was horrified to discover the extent to which people seemed to think that programming in pseudo-imperative do-notation was just punishment for the slightest semantic impurity (apart, of course, from non-termination). I adopted the applicative combinators as a style choice (and went even closer to direct style with "idiom brackets") long before I had a grasp of the semantic distinction, i.e., that they represented a useful weakening of the monad interface. I just didn't (and still don't) like the way do-notation requires fragmentation of expression structure and the gratuitous naming of things.

That's to say, the same things that make functional code more compact and readable than imperative code also make applicative style more compact and readable than do-notation. I appreciate that `ApplicativeDo` is a great way to make more applicative (and in some cases that means *faster*) programs that were written in monadic style that you haven't the time to refactor. But otherwise, I'd argue applicative-when-you-can-but-monadic-when-you-must is also the better way to see what's going on.

4.8 Refactoring do notation into applicative style

All these operators are left associative; the $<$ and/or $>$ points to things which contribute values; it's `$` for thing-to-left-is-pure-value and $\star$ for thing-to-left-is-applicative-computation.

My rule of thumb for using these operators goes as follows. First, list the components of your grammatical production and classify them as "signal" or "noise" depending on whether they contribute semantically important information. Here, we have

```
char '('      -- noise
buildExpr     -- signal
char ')'      -- noise
```

Next, figure out what the "semantic function" is, which takes the values of the signal components and gives the value for the whole production. Here, we have

```
id     -- pure semantic function, then a bunch of component parsers
       char '('      -- noise
       buildExpr     -- signal
       char ')'      -- noise
```

Now, each component parser will need to be attached to what comes before it with an operator, but which?

- always start with $<$
- next `$` for the first component (as the pure function's just before), or $\star$ for every other component
- then comes $>$ if the component is *signal* or ' ' if it's *noise*

So that gives us

```
id     -- pure semantic function, then a bunch of parsers
   <$  char '('      -- first, noise
   <*> buildExpr     -- later, signal
   <*  char ')'      -- later, noise
```

If the semantic function is `id`, as here, you can get rid of it and use $\star>$ to glue noise to the front of the signal which is `id`'s argument. I usually choose not to do that, just so that I can see the semantic function sitting clearly at the beginning of the production. Also, you can build a choice between such productions by interspersing $<|>$ and you don't need to wrap any of them in parentheses.

4.9 Zip with default values instead of dropping values?

There is some structure to this problem, and here it comes. I'll be using this stuff:

```
import Control.Applicative
import Data.Traversable
import Data.List
```

First up, lists-with-padding are a useful concept, so let's have a type for them.

```
data Padme m = (:-) {padded :: [m], padder :: m} deriving (Show, Eq)
```

Next, I remember that the truncating-`zip` operation gives rise to an `Applicative` instance, in the library as `newtype ZipList` (a popular example of a non-`Monad`). The `Applicative ZipList` amounts to a decoration of the monoid given by infinity and minimum. `Padme` has a similar structure, except that its underlying monoid is positive numbers (with infinity), using one and maximum.

```
instance Applicative Padme where
  pure = ([] :-)
  (fs :- f) <*> (ss :- s) = zapp fs ss :- f s where
    zapp  []        ss        = map f ss
    zapp  fs        []        = map ($ s) fs
    zapp  (f : fs)  (s : ss)  = f s : zapp fs ss
```

I am obliged to utter the usual incantation to generate a default `Functor` instance.

```
instance Functor Padme where fmap = (<*>) . pure
```

Thus equipped, we can pad away! For example, the function which takes a ragged list of strings and pads them with spaces becomes a one liner.

```
deggar :: [String] -> [String]
deggar = transpose . padded . traverse (:- ' ')
```

See?

```
*Padme> deggar ["om", "mane", "padme", "hum"]
["om   ","mane ","padme","hum  "]
```

4.10 `sum3` with `zipWith3` in Haskell

I've seen this sort of question before, here: https://stackoverflow.com/q/21349408/828361 My answer to that question also pertains here.

The `ZipList` applicative Lists with a designated padding element are applicative (the applicative grown from the 1 and max monoid structure on positive numbers).

```
data Padme m = (:-) {padded :: [m], padder :: m} deriving (Show, Eq)

instance Applicative Padme where
  pure = ([] :-)
  (fs :- f) <*> (ss :- s) = zapp fs ss :- f s where
    zapp  []        ss        = map f ss
    zapp  fs        []        = map ($ s) fs
    zapp  (f : fs)  (s : ss)  = f s : zapp fs ss

-- and for those of you who don't have DefaultSuperclassInstances
instance Functor Padme where fmap = (<*>) . pure
```

Now we can pack up lists of numbers with their appropriate padding

```
pad0 :: [Int] -> Padme Int
pad0 = (:- 0)
```

And that gives

```
padded ((\x y z -> x+y+z) <$> pad0 [1,2,3] <*> pad0 [4,5] <*> pad0 [6])
= [11,7,3]
```

Or, with the Idiom Brackets that aren't available, you vould write

```
padded (|pad0 [1,2,3] + (|pad0 [4,5] + pad0 6|)|)
```

meaning the same.

`Applicative` gives you a good way to bottle the essential idea of "padding" that this problem demands.

4.11 What is the 'Const' applicative functor useful for?

It's rather useful when combined with `Traversable`.

```
getConst . traverse Const :: (Monoid a, Traversable f) => f a -> a
```

That's the general recipe for glomming a bunch of stuff together. It was one of the use cases which convinced me that it was worth separating `Applicative` from `Monad`. I needed stuff like generalized `elem`

```
elem :: Eq x => x -> Term x -> Bool
```

to do occur-checking for a `Traversable Term` parametrized by the representation of free variables. I kept changing the representation of `Term` and I was fed up modifying a zillion traversal functions, some of which were doing accumulations, rather than effectful mapping. I was glad to find an abstraction which covered both.

4.12 Applicative instance for free monad

Will this do?

```
instance (Functor f) => Applicative (Free f) where
  pure = Return
  Return f  <*> as  = fmap f as
  Roll faf  <*> as  = Roll (fmap (<*> as) faf)
```

The plan is to act only at the leaves of the tree which produces the function, so for `Return`, we act by applying the function to all the argument values produced by the argument action. For `Roll`, we just do to all the sub-actions what we intend to do to the overall action.

Crucially, what we do when we reach `Return` is already set before we start. We don't change our plans depending on where we are in the tree. That's the hallmark of being `Applicative`: the structure of the computation is fixed, so that values depend on values but actions don't.

4.13 Examples of a monad whose Applicative part can be better optimized than the Monad part

Perhaps the canonical example is given by the vectors.

```
data Nat = Z | S Nat deriving (Show, Eq, Ord)

data Vec :: Nat -> * -> * where
  V0    ::                  Vec Z x
  (:>)  :: x -> Vec n x ->  Vec (S n) x
```

We can make them applicative with a little effort, first defining singletons, then wrapping them in a class.

```
data Natty :: Nat -> * where
  Zy  :: Natty Z
  Sy  :: Natty n -> Natty (S n)

class NATTY (n :: Nat) where
  natty :: Natty n

instance NATTY Z where
  natty = Zy

instance NATTY n => NATTY (S n) where
  natty = Sy natty
```

Now we may develop the `Applicative` structure

```
instance NATTY n => Applicative (Vec n) where
  pure   = vcopies natty
  (<*>)  = vapp

vcopies :: forall n x. Natty n -> x -> Vec n x
vcopies  Zy      x  =  V0
vcopies  (Sy n)  x  =  x :> vcopies n x

vapp :: forall n s t. Vec n (s -> t) -> Vec n s -> Vec n t
vapp  V0         V0         = V0
vapp  (f :> fs)  (s :> ss)  = f s :> vapp fs ss
```

I omit the `Functor` instance (which should be extracted via `fmapDefault` from the `Traversable` instance).

Now, there is a `Monad` instance corresponding to this `Applicative`, but what is it? *Diagonal thinking! That's what's required!* A vector can be seen as the tabulation of a function from a finite domain, hence the `Applicative` is just a tabulation of the K- and S-combinators, and the `Monad` has a `Reader`-like behaviour.

```
vtail :: forall n x. Vec (S n) x -> Vec n x
vtail (x :> xs) = xs

vjoin :: forall n x. Natty n -> Vec n (Vec n x) -> Vec n x
vjoin Zy     _                  = V0
vjoin (Sy n) ((x :> _) :> xxss) = x :> vjoin n (fmap vtail xxss)
```

```
instance NATTY n => Monad (Vec n) where
  return    = vcopies natty
  xs >>= f  = vjoin natty (fmap f xs)
```

You might save a bit by defining >>= more directly, but any way you cut it, the monadic behaviour creates useless thunks for off-diagonal computations. Laziness might save us from slowing down by an armageddon factor, but the zipping behaviour of the <*> is bound to be at least a little cheaper than taking the diagonal of a matrix.

4.14 How arbitrary is the "ap" implementation for monads?

There are at least three relevant aspects to this question.

1. Given a `Monad m` instance, what is the specification of its necessary `Applicative m` superclass instance? *Answer*: `pure` is `return`, <*> is `ap`, so

   ```
   mf <*> ms == do f <- mf; s <- ms; return (f s)
   ```

   Note that this specification is not a law of the `Applicative` class. It's a requirement on `Monad`s, to ensure consistent usage patterns.

2. Given that specification (by candidate implementation), is `ap` the only acceptable implementation. *Answer*: resoundingly, **no**. The value dependency permitted by the type of >>= can sometimes lead to inefficient execution: there are situations where <*> can be made more efficient than `ap` because you don't need to wait for the first computation to finish before you can tell what the second computation is. The "applicative do" notation exists exactly to exploit this possibility.

3. Do any other candidate instances for `Applicative` satisfy the `Applicative` laws, even though they disagree with the required `ap` instances? *Answer*: yes. The "backwards" instance proposed by the question is just such a thing. Indeed, as another answer observes, any applicative can be turned backwards, and the result is often a different beast.

For a further example and exercise for the reader, note that nonempty lists are monadic in the way familiar from ordinary lists.

```
data Nellist x = x :& Maybe (Nellist x)

necat :: Nellist x -> Nellist x -> Nellist x
necat (x :& Nothing) ys = x :& Just ys
necat (x :& Just xs) ys = x :& Just (necat xs ys)

instance Monad Nellist where
  return x = x :& Nothing
  (x :& Nothing) >>= k = k x
  (x :& Just xs) >>= k = necat (k x) (xs >>= k)
```

Find at least *four* behaviourally distinct instances of `Applicative Nellist` which obey the applicative laws.

4.15 Applicative without a functor (for arrays)

Reading the comments, I'm a little worried that size is under the carpet here. Is there a sensible behaviour when sizes mismatch?

Meanwhile, there may be something you can sensibly do along the following lines. Even if your arrays aren't easy to make polymorphic, you can make an `Applicative` instance like this.

```
data ArrayLike x = MkAL {sizeOf :: Int, eltOf :: Int -> x}

instance Applicative ArrayLike where
  pure x                  = MkAL maxBound  (pure x)
  MkAL i f <*> MkAL j g   = MkAL (min i j) (f <*> g)
```

(Enthusiasts will note that I've taken the product of the `(Int ->)` applicative with that induced by the (`maxBound`, `min`) monoid.)

Could you make a clean correspondence

```
imAL :: Image -> ArrayLike Float
alIm :: ArrayLike Float -> Image
```

by projection and tabulation? If so, you can write code like this.

```
alIm $ (f <$> imAL a1 <*> ... <*> imAL an)
```

Moreover, if you then want to wrap that pattern up as an overloaded operator,

```
imapp :: (Float -> ... -> Float) -> (Image -> ... -> Image)
```

it's a standard exercise in typeclass programming! (Ask if you need more of a hint.)

The crucial point, though, is that the wrapping strategy means you don't need to monkey with your array structures in order to put functional superstructure on top.

4.16 Does this simple Haskell function already have a well-known name? (strength)

If the `Traversable` and `Foldable` instances for `(,) x)` were in the library (and I suppose I must take some blame for their absence)...

```
instance Traversable ((,) x) where
  traverse f (x, y) = (,) x <$> f y

instance Foldable ((,) x) where
  foldMap = foldMapDefault
```

...then this (sometimes called 'strength') would be a specialisation of `Data.Traversable.sequence`.

```
sequence :: (Traversable t, Monad m) => t (m a) -> m (t a)
```

so

```
sequence :: (Monad m) => ((,) x) (m a) -> m (((,) x) a)
```

i.e.

```
sequence :: (Monad m) => (x, m a) -> m (x, a)
```

In fact, sequence doesn't really use the full power of `Monad`: `Applicative` will do. Moreover, in this case, pairing-with-x is linear, so the `traverse` does only `<$>` rather than other random combinations of `pure` and `<*>`, and (as has been pointed out elsewhere) you only need `m` to have functorial structure.

4.17 Why is `((,) r)` a Functor that is NOT an Applicative?

Suppose we have

```
pure :: forall r a. a -> (r, a)
```

then, in particular, we have

```
magic :: forall r. r
magic = fst (pure ())
```

Now, we can specialise the type variable `r` to get

```
magic :: Void
```

where `Void` is the datatype with no constructors, which means

```
magic = undefined
```

but as type variables (and the types which specialise them) play no run time role, that means `magic` is *always* undefined.

We've discovered that `((,) r)` can be `Applicative` only for *inhabited* `r`. And there's more. With any such instance, we can write

```
munge :: r -> r -> r
munge r0 r1 = fst ( pure (\ _ _ -> ()) <*> (r0, ()) <*> (r1, ()) )
```

to define a binary operator on `r`. The `Applicative` laws tell us effectively that `munge` must be an associative operator that absorbs `magic` on either side.

That's to say there *is* a sensible instance

```
instance Monoid r => Applicative ((,) r) where
  pure a              = (mempty, a)
  (r0, f) <*> (r1, s) = (mappend r0 r1, f s)
```

(exactly what you get when you take `pure=return; (<*>)=ap` from the `Monad (Writer r)`).

Of course, some pedants would argue that it is legal (if unhelpful) to define

```
instance Monoid r where
  mempty = undefined
  mappend _ _ = undefined
  -- Monoid laws clearly hold
```

but I would argue that any sensible type class instance should contribute nontrivially to the defined fragment of the language.

4.18 Applicative Rewriting (for reader)

Being an idle goodfornothing, I thought I would make a computer do the expansion for me. So into GHCi, I typed

```
let pu x = "(\\_ -> " ++ x ++ ")"
let f >*< a = "(\\g -> " ++ f ++ " g (" ++ a ++ " g))"
```

So now I have funny versions of `pure` and `<*>` which map strings which look like expressions to string which look like more complicated expressions. I then defined, similarly, the analogue of `sequenceA`, replacing functions by strings.

```
let sqa [] = pu "[]" ; sqa (f : fs) = (pu "(:)" >*< f) >*< sqa fs
```

I was then able to generate the expanded form of the example as follows

```
putStrLn $ sqa ["(+3)","(+2)"] ++ " 3"
```

which duly printed

```
(\g -> (\g -> (\_ -> (:)) g ((+3) g)) g ((\g -> (\g -> (\_ -> (:)) g ((+2) g)) g  ((\_ -
```

This last, copied to the prompt, yielded

```
[6,5]
```

Comparing the output from my "metaprogram" with the attempt in the question shows a shorter initial prefix of lambdas, arising from a shallower nesting of `<*>` operations. Remember, it's

```
(pure (:) <*> (+3)) <*> ((pure (:) <*> (+2)) <*> pure [])
```

so the outer `(:)` should be only three lambdas deep. I suspect the proposed expansion may correspond to a differently bracketed version of the above, perhaps

```
pure (:) <*> (+3) <*> pure (:) <*> (+2) <*> pure []
```

Indeed, when I evaluate

```
putStrLn $ pu "(:)" >*< "(+3)" >*< pu "(:)" >*< "(+2)" >*< pu "[]" ++ " 3 "
```

I get

```
(\g -> (\g -> (\g -> (\g -> (\_ -> (:)) g ((+3) g)) g ((\_ -> (:)) g)) g ((+2) g)) g ((\
```

which looks like it matches the (updated)

```
(\d -> (\c -> (\b -> (\a -> (\_ -> (:)) a ((+3) a)) b ((\_ -> (:)) b)) c ((+2) c)) d ((\
```

I hope this machine-assisted investigation helps to clarify what's going on.

4.19 Serialised Diagonalisation

Not every type is serializable. How can you establish an isomorphism between `String -> String` and `String`? If you give me `Read` and `Show` instances for `String -> String`, I can find a function which isn't serialisable like this:

```
evil :: String -> String
evil s = map succ (read s s ++ " evil")
```

Suppose

```
read (show evil) = evil
```

We get

```
evil (show evil)
  = map succ (read (show evil) (show evil) ++ " evil")
  = map succ (evil (show evil) ++ " evil")
  = map succ (evil (show evil)) ++ "!fwjm"
```

so if `evil (show evil)` is defined, then it has a first character `c` satisfying `c = succ c`, which is impossible.

In general, functions can't be serialized. Sometimes, we write datatypes which pack up functions, so not every datatype is serializable either. E.g.,

```
data Psychiatrist
  = Listen (String -> Psychiatrist)
  | Charge Int
```

Sometimes, even for these types, you might choose to provide partial implementations of `Read` (with some cases missing) and `Show` (e.g., with placeholders for or tabulations of functions), but there is no canonical way to choose them or reason why you would expect both.

As others have mentioned, serious serialization is the preserve of `Serialize`. I tend to use `Show` and `Read` for diagnostic purposes, especially trying things out in ghci. For that purpose, `Show` is by far more useful, because ghci has a *Haskell* parser to do the reading.

4.20 Applicative style for infix operators?

SHE lets you write

```
(|a ++ (|b ++ c|)|)
```

if that's any use. Of course, there's some overhead to introducing a preprocessing layer.

4.21 Where is the Monoid in Applicative?

Perhaps the monoid you're looking for is this one.

```
newtype AppM f m = AppM (f m) deriving Show

instance (Applicative f, Monoid m) => Monoid (AppM f m) where
  mempty                      = AppM (pure mempty)
  mappend (AppM fx) (AppM fy) = AppM (pure mappend <*> fx <*> fy)
```

As a comment, below, observes, it can be found in the reducers library under the name `Ap`. It's fundamental to `Applicative`, so let's unpack it.

Note, in particular, that because `()` is trivially a `Monoid`, `AppM f ()` is a `Monoid`, too. And that's the monoid lurking behind `Applicative f`.

We could have insisted on `Monoid (f ())` as a superclass of `Applicative`, but that would have fouled things up royally.

```
> mappend (AppM [(),()]) (AppM [(),(),()])
AppM [(),(),(),(),(),()]
```

The monoid underlying `Applicative []` is *multiplication* of natural numbers, whereas the 'obvious' monoidal structure for lists is concatenation, which specialises to *addition* of natural numbers.

Mathematics warning. Dependent types warning. Fake Haskell warning.

One way to see what's going on is to consider those Applicatives which happen to be *containers* in the dependently typed sense of Abbott, Altenkirch and Ghani. We'll have these in Haskell sometime soon. I'll just pretend the future has arrived.

```
data (<|) (s :: *)(p :: s -> *) (x :: *) where
  (:<|:) :: pi (a :: s) -> (p a -> x) -> (s <| p) x
```

The data structure `(s <| p)` is characterised by

- **Shapes** `s` which tell you what the container looks like.
- **Positions** `p` which tell you *for a given shape* where you can put data.

The above type says that to give data for such a structure is to pick a shape, then fill all the positions with data.

The container presentation of `[]` is `Nat <| Fin` where

```
data Nat = Z | S Nat
data Fin (n :: Nat) where
  FZ :: Fin (S n)
  FS :: Fin n -> Fin (S n)
```

so that `Fin n` has exactly `n` values. That is, the shape of a list is its *length*, and that tells you how many elements you need to fill up the list.

You can find the shapes for a Haskell `Functor f` by taking `f ()`. By making the data trivial, the positions don't matter. Constructing the GADT of positions generically in Haskell is rather more difficult.

Parametricity tells us that a polymorphic function between containers in

```
forall x. (s <| p) x -> (s' <| p') x
```

must be given by

- a function `f :: s -> s'` mapping input shapes to output shapes
- a function `g :: pi (a :: s) -> p' (f a) -> p a` mapping (for a given input shape) output positions back to the input positions where the output element will come from.

<!- ->

```
morph f g (a :<|: d) = f a :<|: (d . g a)
```

(Secretly, those of us who have had our basic Hancock training also think of "shapes" as "commands" and "positions" as "valid responses". A morphism between containers is then exactly a "device driver". But I digress.)

Thinking along similar lines, what does it take to make a container `Applicative`? For starters,

```
pure :: x -> (s <| p) x
```

which is equivalently

```
pure :: (() <| Const ()) x -> (s <| p) x
```

That has to be given by

```
f :: () -> s   -- a constant in s
g :: pi (a :: ()) -> p (f ()) -> Const () a  -- trivial
```

where `f = const neutral` for some

```
neutral :: s
```

Now, what about

```
(<*>) :: (s <| p) (x -> y) -> (s <| p) x -> (s <| p) y
```

? Again, parametricity tells us two things. Firstly, the only useful data for calculating the output shapes are the two input shapes. We must have a function

```
outShape :: s -> s -> s
```

Secondly, the only way we can fill an output position with a `y` is to pick a position from the first input to find a function in 'x -> y' and then a position in the second input to obtain its argument.

```
inPos :: pi (a :: s)(b :: s) -> p (outShape a b) -> (p a, p b)
```

That is, we can always identify the pair of input positions which determine the output in an output position.

The applicative laws tell us that `neutral` and `outShape` must obey the monoid laws, and that, moreover, we can lift monoids as follows

```
mappend (a :<|: f) (b :<|: g) = outShape a b :<|: \ z ->
  let (x, y) = inPos a b z
  in  mappend (f x) (g y)
```

There's something more to say here, but for that, I need to contrast two operations on containers.

Composition

```
(s <| p) . (s' <| p')  =  ((s <| p) s') <| \ (a :<|: f) -> Sigma (p a) (p' . f)
```

where `Sigma` is the type of dependent pairs

```
data Sigma (p :: *)(q :: p -> *) where
  Pair :: pi (a :: p) -> q a -> Sigma p q
```

What on earth does that mean?

- you choose an outer shape
- you choose an inner shape for each outer position
- a composite position is then the pair of an outer position and an inner position appropriate to the inner shape that sits there

Or, in Hancock

- you choose an outer command
- you can wait to see the outer response before choosing the inner command
- a composite response is then a response to the outer command, followed by a response to the inner command chosen by your strategy

Or, more blatantly

- when you make a list of lists, the inner lists can have different lengths

The `join` of a `Monad` flattens a composition. Lurking behind it is not just a monoid on shapes, but an *integration* operator. That is,

```
join :: ((s <| p) . (s <| p)) x -> (s <| p) x
```

requires

```
integrate :: (s <| p) s -> s
```

Your free monad gives you strategy trees, where you can use the result of one command to choose the rest of your strategy. As if you're interacting at a 1970s teletype.

Meanwhile...

Tensor

The tensor (also due to Hancock) of two containers is given by

```
(s <| p) >< (s' <| p')  =  (s, s') <| \ (a, b) -> (p a, p' b)
```

That is

- you choose two shapes
- a position is then a pair of positions, one for each shape

or

- you choose two commands, without seeing any responses
- a response is then the pair of responses

or

- `[] >< []` is the type of *rectangular matrices*: the 'inner' lists must all have the same length

The latter is a clue to why $><$ is very hard to get your hands on in Haskell, but easy in the dependently typed setting.

Like composition, tensor is a monoid with the identity functor as its neutral element. If we replace the composition underlying `Monad` by tensor, what do we get?

```
pure :: Id x -> (s <| p) x
mystery :: ((s <| p) >< (s <| p)) x -> (s <| p) x
```

But whatever can `mystery` be? It's not a mystery, because we know there's a rather rigid way to make polymorphic functions between containers. There must be

```
f :: (s, s) -> s
g :: pi ((a, b) :: (s, s)) -> p (f (a, b)) -> (p a, p b)
```

and those are exactly what we said determined $<\star>$ earlier.

`Applicative` is the notion of effectful programming generated by tensor, where `Monad` is generated by composition. The fact that you don't get to/need to wait for the outer response to choose the inner command is why `Applicative` programs are more readily parallelizable.

Seeing `[] >< []` as rectangular matrices tells us why $<\star>$ for lists is built on top of multiplication.

The free applicative functor is the free monoid with knobs on. For containers,

```
Free (s <| p) = [s] <| All p
```

where

```
All p [] = ()
All p (x : xs) = (p x, All p xs)
```

So a "command" is a big list of commands, like a deck of punch cards. You don't get to see any output before you choose your card deck. The "response" is your lineprinter output. It's the 1960s.

So there you go. The very nature of `Applicative`, tensor not composition, demands an underlying monoid, and a recombination of elements compatible with monoids.

4.22 Applicatives from Monoids including min and max

The basic reason why `Applicative []` has the generate-all-possible-combinations behaviour, rather than any kind of zippy behaviour, is that `Applicative` is a superclass of `Monad` and is intended to behave in accordance with the `Monad` instance when one exists. `Monad []` treats lists as failure-and-prioritized-choice, so the `Applicative []` instance does, too. People often refactor monadic code using the applicative interface to reduce the number of intermediate names needed for values, and to increase opportunities for parallelism. It would be pretty scary if that caused a significant shift in the functional semantics.

That aside, the truth is, you're spoilt for choice for `Applicative []` instances, and even more so if you consider empty/nonempty and finite/coinductive/infinite variations. Why is that?

Well, as I mentioned in this answer, every `Applicative f` begins its life as a `Monoid (f ())`, combining the *shapes* of the data, before we start to worry about the *values*. Lists are a case in point.

`[()]` is basically the type of numbers. Numbers are monoids in lots of ways.

Taking `Applicative []` from `Monad []` amounts to choosing the monoid generated by `1` and $*$.

Meanwhile, `Applicative ZipList` exploits Haskell's coinductive conflation and amounts to choosing the monoid generated by infinity and minimum.

The question proposes an instance which is not lawful, but is close to one that is. You'll notice $<*>$ isn't defined for an empty list of functions, but for nonempty lists of functions, it pads out to match the list of arguments. Asymmetrically, it truncates when the arguments run out. Something's not quite right.

Two candidate fixes follow.

One is to truncate on empty on both sides, and then you must take `pure = repeat` and you have `ZipList`.

The other is to rule out empty lists and pad on both sides. Then you get the `Applicative` generated from the `Monoid` on *positive* numbers generated by 1 and *maximum*. So it's not `ZipList` at all. That's the thing I called `PadMe` in this answer. The reason you need to rule out 0 is that for every position in the output of $<*>$, you need to point to the position in both inputs where the function and its arguments (respectively) come from. You can't pad if you have nothing to pad with.

It's a fun game. Pick a `Monoid` on numbers and see if you can grow it into an `Applicative` for lists!

Chapter 5

Monads

5.1 Why we use monadic functions $a \to m\ b$

In a sense, you're right. As every monad `m` is a functor, we can use `fmap f` with a function `f :: a -> b` to turn an `m a` into an `m b`, but there's a catch. What's `b`?

I like to think of such an `m` as meaning "plan-to-get", where "plans" involve some sort of additional interaction beyond pure computation. If you have a "plan-to-get `Int`" and you want a "plan-to-get `String`", you can use `fmap` with a function in `Int -> String`, but the type of that function tells you that getting the `String` from the `Int` involves no further interaction.

That isn't always so: perhaps the `Int` is a student registration number and the `String` is their name, so the plan to convert from one to the other needs an external lookup in some table. Then I don't have a pure function from `Int` to `String`, but rather a pure function from `Int` to "plan-to-get `String`". If I `fmap` that across my "plan-to-get `Int`", that's fine, but I end up with "plan-to-get (plan-to-get `String`)" and I need to `join` the outer and inner plans.

The general situation is that we have enough information to compute the plan to get more. That's what `a -> m b` models. In particular, we have `return :: a -> m a`, which turns the information we have into the plan that gives us exactly that information by taking no further action, and we have `(>=>) :: (a -> m b) -> (b -> m c) -> (a -> m c)` which composes two such things. We also have that `(>=>)` is associative and absorbs `return` on left and right, much the way `;` is associative and absorbs `skip` in classic imperative programming.

It's more convenient to build larger plans from smaller ones using this compositional approach, keeping the number of "plan-to-get" layers a consistent *one*. Otherwise, you need to build up an *n*-layer plan with `fmap`, then do the right number of `join`s on the outside (which will be a brittle property of the plan).

Now, as Haskell is a language with a concept of "free variable" and "scope", the `a` in

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

representing the "overall input information" can just be taken to come from the scope of things we already have, leaving

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

and we get back "bind", which is the tool that presents the compositional structure in the most programmer-friendly form, resembling a local definition.

To sum up, you can work with `a -> b`, but often you need `b` to be "plan-to-get something", and that's the helpful thing to choose if you want to build plans compositionally.

FIXME: `http://i.stack.imgur.com/7ts7a.jpg`

Figure 5.1: tree of trees

FIXME: `http://i.stack.imgur.com/Dw5vO.jpg`

Figure 5.2: enter image description here

5.2 Monads with Join instead of Bind

Without plumbing the depths of metaphor, might I suggest to read a typical monad `m` as "strategy to produce a", so the type `m value` is a first class "strategy to produce a value". Different notions of computation or external interaction require different types of strategy, but the general notion requires some regular structure to make sense:

- if you already have a value, then you have a strategy to produce a value (`return :: v -> m v`) consisting of nothing other than producing the value that you have;
- if you have a function which transforms one sort of value into another, you can lift it to strategies (`fmap :: (v -> u) -> m v -> m u`) just by waiting for the strategy to deliver its value, then transforming it;
- if you have a strategy to produce a strategy to produce a value, then you can construct a strategy to produce a value (`join :: m (m v) -> m v`) which follows the outer strategy until it produces the inner strategy, then follows that inner strategy all the way to a value.

Let's have an example: leaf-labelled binary trees...

```
data Tree v = Leaf v | Node (Tree v) (Tree v)
```

...represent strategies to produce stuff by tossing a coin. If the strategy is `Leaf v`, there's your `v`; if the strategy is `Node h t`, you toss a coin and continue by strategy `h` if the coin shows "heads", `t` if it's "tails".

```
instance Monad Tree where
  return = Leaf
```

A strategy-producing strategy is a tree with tree-labelled leaves: in place of each such leaf, we can just graft in the tree which labels it...

```
  join (Leaf tree) = tree
  join (Node h t)  = Node (join h) (join t)
```

...and of course we have `fmap` which just relabels leaves.

```
instance Functor Tree where
  fmap f (Leaf x)    = Leaf (f x)
  fmap f (Node h t)  = Node (fmap f h) (fmap f t)
```

Here's an strategy to produce a strategy to produce an `Int`.

Toss a coin: if it's "heads", toss another coin to decide between two strategies (producing, respectively, "toss a coin for producing 0 or producing 1" or "produce 2"); if it's "tails" produce a third ("toss a coin for producing 3 or tossing a coin for 4 or 5").

That clearly `join`s up to make a strategy producing an `Int`.

What we're making use of is the fact that a "strategy to produce a value" can itself be seen as a value. In Haskell, the embedding of strategies as values is silent, but in English, I use quotation marks to distinguish using a strategy from just talking about it. The `join` operator expresses the

FIXME: `http://i.stack.imgur.com/BFJSD.jpg`

Figure 5.3: jigsaw pieces for computations

strategy "somehow produce then follow a strategy", or "if you are *told* a strategy, you may then *use* it".

(Meta. I'm not sure whether this "strategy" approach is a suitably generic way to think about monads and the value/computation distinction, or whether it's just another crummy metaphor. I do find leaf-labelled tree-like types a useful source of intuition, which is perhaps not a surprise as they're the *free* monads, with just enough structure to be monads at all, but no more.)

PS The type of "bind"

```
(>>=) :: m v -> (v -> m w) -> m w
```

says "if you have a strategy to produce a `v`, and for each v a follow-on strategy to produce a `w`, then you have a strategy to produce a `w`". How can we capture that in terms of `join`?

```
mv >>= v2mw = join (fmap v2mw mv)
```

We can relabel our `v`-producing strategy by `v2mw`, producing instead of each `v` value the `w`-producing strategy which follows on from it — ready to `join`!

5.3 Using return versus not using return in the list monad

To see why you get the particular answers that arise, the desugaring explanations are very helpful. Let me supplement them with a little general advice about developing perceptions of Haskell code.

Haskell's type system makes no distinction between two separable "moral" purposes:

- `[x]` the type of *values* which are lists with elements drawn from `x`
- `[x]` the type of *computations* of elements of `x` which allow prioritized choice

The fact that these two notions have the same representation does not mean that they play the same roles. In `f1`, the `[x, x+1]` is playing the role of computation, so the possibilities it generates are merged into the choice generated by the whole computation: that's what the `>>=` of the list monad does. In `f2`, however, the `[x, x+1]` is playing the role of value, so that the whole computation generates a prioritized choice between two values (which happen to be list values).

Haskell does not use types to make this distinction [and you may have guessed by now that I think it should, but that's another story]. Instead, it uses syntax. So you need to train your head to perceive the value and computation roles when you read code. The `do` notation is a special syntax for constructing *computations*. What goes inside the `do` is built from the following template kit:

The three blue pieces make `do`-computations. I've marked the computation holes in blue and the value holes in red. This is not meant to be a complete syntax, just a guide to how to perceive pieces of code in your mind.

Indeed, you may write any old expression in the blue places provided it has a suitably monadic type, and the computation so generated will be merged into the overall computation using `>>=` as needed. In your `f1` example, your list is in a blue place and treated as prioritized choice.

Similarly, you may write expressions in red places which may very well have monadic types (like lists in this case), but they will be treated as values all the same. That's what happens in `f2`: as it were, the result's outer brackets are blue, but the inner brackets are red.

Train your brain to make the value/computation separation when you read code, so that you know instinctively which parts of the text are doing which job. Once you've reprogrammed your head, the distinction between `f1` and `f2` will seem completely normal!

5.4 Example showing monads don't compose

For a small concrete counterexample, consider the terminal monad.

```
data Thud x = Thud
```

The `return` and >>= just go `Thud`, and the laws hold trivially.

Now let's also have the writer monad for Bool (with, let's say, the xor-monoid structure).

```
data Flip x = Flip Bool x

instance Monad Flip where
   return x = Flip False x
   Flip False x  >>= f = f x
   Flip True x   >>= f = Flip (not b) y where Flip b y = f x
```

Er, um, we'll need composition

```
newtype (:.:) f g x = C (f (g x))
```

Now try to define...

```
instance Monad (Flip :.: Thud) where  -- that's effectively the constant `Bool` functor
  return x = C (Flip ??? Thud)
  ...
```

Parametricity tells us that `???` can't depend in any useful way on `x`, so it must be a constant. As a result, `join . return` is necessarily a constant function also, hence the law

```
join . return = id
```

must fail for whatever definitions of `join` and `return` we choose.

5.5 The Pause monad

Here's how I'd go about it, using *free* monads. Er, um, what are they? They're trees with actions at the nodes and values at the leaves, with >>= acting like tree grafting.

```
data f :^* x
  = Ret x
  | Do (f (f :^* x))
```

It's not unusual to write F*X for such a thing in the mathematics, hence my cranky infix type name. To make an instance, you just need `f` to be something you can map over: any `Functor` will do.

```
instance Functor f => Monad ((:^*) f) where
  return = Ret
  Ret x  >>= k  = k x
  Do ffx >>= k  = Do (fmap (>>= k) ffx)
```

That's just "apply `k` at all the leaves and graft in the resulting trees". These can trees represent *strategies* for interactive computation: the whole tree covers every possible interaction with the environment, and the environment chooses which path in the tree to follow. Why are they *free*? They're just trees, with no interesting equational theory on them, saying which strategies are equivalent to which other strategies.

Now let's have a kit for making Functors which correspond to a bunch of commands we might want to be able to do. This thing

```
data (:>>:) s t x = s :? (t -> x)

instance Functor (s :>>: t) where
  fmap k (s :? f) = s :? (k . f)
```

captures the idea of getting a value in `x` after *one* command with input type `s` and output type `t`. To do that, you need to choose an input in `s` and explain how to continue to the value in `x` given the command's output in `t`. To map a function across such a thing, you tack it onto the continuation. So far, standard equipment. For our problem, we may now define two functors:

```
type Modify s  = (s -> s) :>>: ()
type Yield     = () :>>: ()
```

It's like I've just written down the value types for the commands we want to be able to do!

Now let's make sure we can offer a *choice* between those commands. We can show that a choice between functors yields a functor. More standard equipment.

```
data (:+:) f g x = L (f x) | R (g x)

instance (Functor f, Functor g) => Functor (f :+: g) where
  fmap k (L fx) = L (fmap k fx)
  fmap k (R gx) = R (fmap k gx)
```

So, `Modify s :+: Yield` represents the choice between modifying and yielding. Any *signature* of simple commands (communicating with the world in terms of values rather than manipulating computations) can be turned into a functor this way. It's a bother that I have to do it by hand!

That gives me your monad: the free monad over the signature of modify and yield.

```
type Pause s = (:^*) (Modify s :+: Yield)
```

I can define the modify and yield commands as one-do-then-return. Apart from negotiating the dummy input for `yield`, that's just mechanical.

```
mutate :: (s -> s) -> Pause s ()
mutate f = Do (L (f :? Ret))

yield :: Pause s ()
yield = Do (R (() :? Ret))
```

The `step` function then gives a meaning to the strategy trees. It's a *control operator*, constructing one computation (maybe) from another.

```
step :: s -> Pause s () -> (s, Maybe (Pause s ()))
step s (Ret ())            = (s, Nothing)
step s (Do (L (f  :? k)))  = step (f s) (k ())
step s (Do (R (() :? k)))  = (s, Just (k ()))
```

The `step` function runs the strategy until either it finishes with a `Ret`, or it yields, mutating the state as it goes.

The general method goes like this: separate the *commands* (interacting in terms of values) from the *control operators* (manipulating computations); build the free monad of "strategy trees" over the signature of commands (cranking the handle); implement the control operators by recursion over the strategy trees.

5.6 Haskell monad return arbitrary data type

One way to define a `Monad` instance for this type is to treat it as a *free* monad. In effect, this takes `A a` to be a little syntax with one binary operator `C`, and variables represented by values of type `a` embedded by the `B` constructor. That makes `return` the `B` constructor, embedding variables, and >>= the operator which performs subsitution.

```
instance Monad A where
  return = B
  B x   >>= f = f x
  C l r >>= f = C (l >>= f) (r >>= f)
```

It's not hard to see that `(>>= B)` performs the identity substitution, and that composition of substitutions is associative.

Another, more "imperative" way to see this monad is that it captures the idea of computations that can flip coins (or read a bitstream or otherwise have some access to a sequence of binary choices).

```
data Coin = Heads | Tails
```

Any computation which can flip coins must either stop flipping and be a value (with `B`), or flip a coin and carry on (with `C`) in one way if the coin comes up `Heads` and another if `Tails`. The monadic operation which flips a coin and tells you what came up is

```
coin :: A Coin
coin = C (B Heads) (B Tails)
```

The >>= of `A` can now be seen as sequencing coin-flipping computations, allowing the choice of a subsequent computation to depend on the value delivered by an earlier computation.

If you have an infinite stream of coins, then (apart from your extraordinary good fortune) you're also lucky enough to be able to run any `A`-computation to its value, as follows

```
data Stream x = x :> Stream x   -- actually, I mean "codata"

flipping :: Stream Coin -> A v -> v
flipping _              (B v)    = v
flipping (Heads :> cs) (C h t)  = flipping cs h
flipping (Tails :> cs) (C h t)  = flipping cs t
```

The general pattern in this sort of monad is to have one constructor for returning a value (`B` here) and a bunch of others which represent the choice of possible operations and the different ways computations can continue given the result of an operation. Here `C` has no non-recursive parameters and two subtrees, so I could tell that there must be just one operation and that it must have just two possible outcomes, hence flipping a coin.

So, it's substitution for a syntax with variables and one binary operator, or it's a way of sequencing computations that flip coins. Which view is better? Well... they're two sides of the same coin.

5.7 Should I avoid using Monad fail?

Some monads have a sensible failure mechanism, e.g. the terminal monad:

```
data Fail x = Fail
```

Some monads don't have a sensible failure mechanism (`undefined` is not sensible), e.g. the initial monad:

```
data Return x = Return x
```

In that sense, it's clearly a wart to require all monads to have a `fail` method. If you're writing programs that abstract over monads `(Monad m) =>`, it's not very healthy to make use of that generic `m`'s `fail` method. That would result in a function you can instantiate with a monad where `fail` shouldn't really exist.

I see fewer objections to using `fail` (especially indirectly, by matching `Pat <- computation`) when working in a specific monad for which a good `fail` behaviour has been clearly specified. Such programs would hopefully survive a return to the old discipline where nontrivial pattern matching created a demand for `MonadZero` instead of just `Monad`.

One might argue that the better discipline is always to treat failure-cases explicitly. I object to this position on two counts: (1) that the point of monadic programming is to avoid such clutter, and (2) that the current notation for case analysis on the result of a monadic computation is so awful. The next release of SHE will support the notation (also found in other variants)

```
case <- computation of
  Pat_1 -> computation_1
  ...
  Pat_n -> computation_n
```

which might help a little.

But this whole situation is a sorry mess. It's often helpful to characterize monads by the operations which they support. You can see `fail`, `throw`, etc as operations supported by some monads but not others. Haskell makes it quite clumsy and expensive to support small localized changes in the set of operations available, introducing new operations by explaining how to handle them in terms of the old ones. If we seriously want to do a neater job here, we need to rethink how `catch` works, to make it a translator between *different* local error-handling mechanisms. I often want to bracket a computation which can fail uninformatively (e.g. by pattern match failure) with a handler that adds more contextual information before passing on the error. I can't help feeling that it's sometimes more difficult to do that than it should be.

So, this is a could-do-better issue, but at the very least, use `fail` only for specific monads which offer a sensible implementation, and handle the 'exceptions' properly.

5.8 Why isn't Kleisli an instance of Monoid?

In the business of library design, we face a choice point here, and we have chosen to be less than entirely consistent in our collective policy (or lack of it).

`Monoid` instances for `Monad` (or `Applicative`) type constructors can come about in a variety of ways. Pointwise lifting is always available, but we don't define

```
instance (Applicative f, Monoid x) => Monoid (f x) {- not really -} where
  mempty          = pure mempty
  mappend fa fb   = mappend <$> fa <*> fb
```

Note that the `instance Monoid (a -> b)` is just such a pointwise lifting, so the pointwise lifting for `(a -> m b)` does happen whenever the monoid instance for `m b` does pointwise lifting for the monoid on `b`.

We don't do pointwise lifting in general, not only because it would prevent other `Monoid` instances whose carriers happen to be applied types, but also because the structure of the `f` is often considered more significant than that of the `x`. A key case in point is the *free* monoid, better known as `[x]`, which is a `Monoid` by `[]` and `(++)`, rather than by pointwise lifting. The monoidal structure comes from the list wrapping, not from the elements wrapped.

My preferred rule of thumb is indeed to prioritise monoidal structure inherent in the type constructor over either pointwise lifting, or monoidal structure of specific instantiations of a type, like the composition monoid for `a -> a`. These can and do get `newtype` wrappings.

Arguments break out over whether `Monoid (m x)` should coincide with `MonadPlus m` whenever both exist (and similarly with `Alternative`). My sense is that the only good `MonadPlus` instance is a copy of a `Monoid` instance, but others differ. Still, the library is not consistent in this matter, especially not in the matter of (many readers will have seen this old bugbear of mine coming)...

...the monoid instance for `Maybe`, which ignores the fact that we routinely use `Maybe` to model possible failure and instead observes that that the same data type idea of chucking in an extra element can be used to give a semigroup a neutral element if it didn't already have one. The two constructions give rise to isomorphic types, but they are not conceptually cognate. (**Edit** To make matters worse, the idea is implemented awkwardly, giving instance a `Monoid` constraint, when only a `Semigroup` is needed. I'd like to see the `Semigroup`-extends-to-`Monoid` idea implemented, but *not* for `Maybe`.)

Getting back to `Kleisli` in particular, we have three obvious candidate instances:

1. `Monoid (Kleisli m a a)` with `return` and Kleisli composition
2. `MonadPlus m => Monoid (Kleisli m a b)` lifting `mzero` and `mplus` pointwise over `->`
3. `Monoid b => Monoid (Kleisli m a b)` lifting the monoid structure of `b` over `m` then `->`

I expect no choice has been made, just because it's not clear which choice to make. I hesitate to say so, but my vote would be for 2, prioritising the structure coming from `Kleisli m a` over the structure coming from `b`.

5.9 Monads at the prompt?

As things stand, the `IO`-specific behaviour relies on the way `IO` actions are a bit statelike and unretractable. So you can say things like

```
s <- readFile "foo.txt"
```

and get an actual value `s :: String`.

It's pretty clear that it takes more than just `Monad` structure to sustain that sort of interaction. It would not be so easy with

```
n <- [1, 2, 3]
```

to say what *value* n has.

One could certainly imagine adapting ghci to open a prompt allowing a monadic computation to be constructed `do`-style in multiple command-line interactions, delivering the whole computation when the prompt is closed. It's not clear what it would mean to inspect the intermediate values (other than to generate collections of printing computations of type `m (IO ())`, for the active monad `m`, of course).

But it would be interesting to ask whether what's special about `IO` that makes a nice interactive prompt behaviour possible can be isolated and generalized. I can't help sniffing a whiff of a comonadic value-in-context story about interaction at a prompt, but I haven't tracked it down yet. One might imagine addressing my list example by considering what it would mean to have a *cursor* into a space of possible values, the way `IO` has a cursor imposed on it by the here-and-now of the real world. Thank you for the food for thought.

5.10 Is this a case to use liftM?

`liftM` and friends serve the purpose of jacking up *pure* functions to a monadic setting, much as the `Applicative` combinators do.

```
liftM :: Monad m => (s -> t) -> m s -> m t
```

There are two issues with the code you tried. One is just a lack of parentheses.

```
liftM sendAllTo :: IO Socket -> IO (ByteString -> SockAddr -> IO ())
```

which is not what you meant. The other issue is that

```
sendAllTo :: Socket -> ByteString -> SockAddr -> IO ()
```

is a *monadic* operation, so lifting it will deliver two layers of `IO`. The usual method is to parenthesize the pure prefix of the application, like so

```
liftM (sendAllTo s datastring) :: IO SockAddr -> IO (IO ())
```

You can then build the argument with `liftM2`.

```
liftM2 SockAddrInet ioport (inet_adder host) :: IO SockAddr
```

That gives you

```
liftM (sendAllTo s datastring) (liftM2 SockAddrInet ioport (inet_adder host))
  :: IO (IO ())
```

which will achieve precisely nothing as it stands, because it explains how to compute an operation but doesn't actually invoke it! That's where you need

```
join (liftM (sendAllTo s datastring) (liftM2 SockAddrInet ioport (inet_addr host)))
  :: IO ()
```

or, more compactly

```
sendAllTo s datastring =<< liftM2 SockAddrInet ioport (inet_adder host)
```

Plug. The Strathclyde Haskell Enhancement supports idiom brackets, where `(|f a1 .. an|) :: m t` if `f :: s1 -> ... -> sn -> t` and `a1 :: m s1` ... `an :: m sn`.

These do the same job for `Applicative m` as the `liftM` family do for monads, treating `f` as a pure n-ary function and `a1`..`an` as effectful arguments. `Monad`s can and should be `Applicative` too, so

```
(|SockAddrInet ioprot (inet_addr host)|) :: IO SockAddr
```

and

```
(|(sendAllTo s datastring) (|SockAddrInet ioprot (inet_addr host)|)|) :: IO (IO ())
```

The notation then allows you to invoke computed monadic computations like the above, with a postfixed `@`.

```
(|(sendAllTo s datastring) (|SockAddrInet ioprot (inet_addr host)|) @|) :: IO ()
```

Note that I'm still parenthesizing the pure prefix of the application, so that the `f` of the template is the whole of `(sendAllTo s datastring)`. The notation allows you to mark pure arguments in any position with a `~`, so you can write this

```
(|sendAllTo ~s ~datastring (|SockAddrInet ioprot (inet_addr host)|) @|) :: IO ()
```

if the mood takes you.

Rant. We spend far too much energy on figuring out the right `liftM`, `join`, `=<<`, `do`, `(|...~...@|)` punctuation in order to explain how to cut up a type as a value-explaining kernel (`()` here) in an effect-explaining context (`IO` here). If this up-cutting were made more explicitly in types, we should need less noise in our programs to slap values and computations into alignment. I should much prefer the computer to infer where the `~` and `@` marks go, but as things stand, Haskell types are too ambiguous a document to make that feasible.

5.11 Zappy colists do *not* form a monad

I just thought I should clarify that the version with exercises and "Idioms" in the title is a rather *earlier* draft of the paper which eventually appeared in JFP. At that time, I mistakenly thought that colists (by which I mean possibly infinite, possibly finite lists) were a monad in a way which corresponds to zapp: there is a plausible candidate for the join (alluded to in other answers) but Jeremy Gibbons was kind enough to point out to us that it does not satisfy the monad laws. The counterexamples involve "ragged" lists of lists with varying finite lengths. Correspondingly, in the JFP article, we stood corrected. (We were rather happy about it, because we love to find applicative functors whose (<*>) is not the ap of a Monad.)

The necessarily infinite lists (i.e. *streams*), by ruling out the ragged cases, do indeed form a monad whose ap behaves like zapp. For a clue, note that Stream x is isomorphic to Nat -> x.

My apologies for the confusion. It's sometimes dangerous leaving old, unfinished drafts (replete with errors) lying (ha ha) around on the web.

5.12 Haskell io-streams and `forever` produces no output to stdout

The meaning of the `forever` in your code is that it takes for ever to choose *each* value in the stream. Its type

```
forever :: Monad m => m a -> m b
```

is a big clue that a computation built with `forever` never returns a value: the caller of `forever` gets to pick the type `b` arbitrarily, so no program can actually promise to deliver a value of that type. That's also why your program typechecks. The computation you pass to `forever` is repeatedly executed for its effect (in this case, choosing a random number), but no value is ever delivered, hence the stream never gets going.

You shouldn't need a `forever` to make a stream that keeps going. The behaviour of `makeInputStream` is to run its argument computation each time a value is demanded from the stream, so you've got your repetition there already.

Chapter 6

Differential Calculus for Types

6.1 Find the preceding element of an element in list

One of my favourite underappreciated utilities is rather handy for problems like this. Let me have backward lists, so I don't need to reverse my brain.

```
data Bwd x = B0 | Bwd x :< x  -- rightmost is nearest
```

Think of list elements as the beads on an abacus wire. Flick a few to the left and leave your finger resting on the next one. What have you? A list of beads to the left of your finger (with the rightmost nearest), a list of beads to the right of your finger (with the leftmost nearest), and the bead with your finger on it.

That is, a *one-hole element context* for lists is given by the pair of backward and forward lists either side of the hole.

```
type ListContext x = (Bwd x, [x])
```

Those who know my old songs recognize `ListContext` as the *derivative* of `[]`.

An element in focus (your finger on a bead) is

```
type ListFocus x = (ListContext x, x)
```

And there is a useful operation which decorates every list element with its context, putting it in focus.

```
focus :: [x] -> [ListFocus x]
focus = go B0 where
  go xz [] = []
  go xz (x : xs) = ((xz, xs), x) : go (xz :< x) xs
```

For example,

```
focus [1,2,3] = [((B0,[2,3]),1), ((B0 :< 1,[3]),2), ((B0 :< 1 :< 2,[]),3)]
```

and now it is very easy to answer all sorts of questions that concern an element and its immediate surroundings. You mightn't construct `focus` just to solve this problem, but it's the sort of thing I keep around because it solves lots of problems.

```
[p | ((_ :< p,_),q) <- focus xs, q == x]
```

computes all the values `p` which sit to the left of an `x` in `xs`. *As you can see.*

(By the way, this `focus` operation didn't come from nowhere. It arises from the differential structure of the datatype of lists. This answer (where it is called `picks`) tells the list story in more detail, and this answer develops the datatype generic story.)

6.2 Splitting a List

This answer is in two parts. The first part addresses the question directly. The second part goes off on a tangent (literally) in order to dig about in the mathematics behind the first part: it may thus prove to be difficult material of limited interest, but I thought a few extremists might enjoy it.

The answers I've seen so far make neat use of list comprehensions or their monadic equivalent, but they use *equality* to rule out duplicates, thus requiring an extra `Eq` constraint. Here's a solution which makes all the pairs of elements in two different *positions*.

Firstly, I write a handy function which decorates each element of a list with the list of elements in other positions: all the ways to "pick one and leave the others". It's very useful whenever lists are being used to collect stuff for selection-without-replacement, and it's something I find I use a lot.

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

Note that `map fst . picks = id`, so that the selected element in each position of the result is the element from that position in the original list: that's what I meant by "decorates".

Now it's easy to pick two, using the same list comprehension method as in the other answers. But instead of selecting the first component from the list itself, we can select from its `picks`, at the same time acquiring the list of candidates for the second component.

```
allPairs :: [x] -> [(x, x)]
allPairs xs = [(y, z) | (y, ys) <- picks xs, z <- ys]
```

It's just as easy to get hold of the triples, taking `picks` twice.

```
allTriples :: [x] -> [(x, x, x)]
allTriples ws = [(x, y, z) | (x, xs) <- picks ws, (y, ys) <- picks xs, z <- ys]
```

For uniformity, it's almost tempting to make the code slightly less efficient, writing `(z, _) <- picks ys` rather than `z <- ys` in both.

If the input list has no duplicates, you won't get any duplicating tuples in the output, because the tuples take their elements from different positions. However, you will get

```
Picks> allPairs ["cat", "cat"]
[("cat","cat"),("cat","cat")]
```

To avoid that, feel free to use `allPairs . nub`, which removes duplicates before selection and demands once more an `Eq` instance for the element type. ____ ***For extremists only: containers, calculus, comonads and combinatorics ahoy!***

`picks` is one instance of a more general construct, arising from the differential calculus. It's an amusing fact that for any given containery sort of a functor `f`, its mathematical derivative, ∂f, represents `f`-structures with one element removed. For example,

```
newtype Trio x = Trio (x, x, x)   -- x^3
```

has derivative

```
data DTrio x = Left3 ((), x, x) | Mid3 (x, (), x) | Right3 (x, x, ())  -- 3*x^2
```

A number of operations can be associated with this construction. Imagine we can really use ∂ (and we can kinda code it up using type families). We could then say

```
data InContext f x = (:-) {selected :: x, context :: ∂f x}
```

to give a type of selected elements decorated by context. We should certainly expect to have the operation

```
plug :: InContext f x -> f x   -- putting the element back in its place
```

This `plug` operation moves us towards the root if we're zippering about in a tree whose nodes are seen as containers of subtrees.

We should also expect `InContext f` to be a comonad, with

```
counit :: InContext f x -> x
counit = selected
```

projecting out the selected element and

```
cojoin :: InContext f x -> InContext f (InContext f x)
```

decorating every element with its context, showing all possible way you could *refocus*, selecting a different element.

The inestimable Peter Hancock once suggested to me that we should also expect to be able to move "down" (meaning "away from the root"), collecting all the possible ways to pick an element-in-context from a whole structure.

```
picks :: f x -> f (InContext f x)
```

should decorate every `x`-element in the input `f`-structure with its context. We should expect that

```
fmap selected . picks = id
```

which is the law we had earlier, but also

```
fmap plug (picks fx) = fmap (const fx) fx
```

telling us that every decorated element is a decomposition of the original data. We didn't have that law above. We had

```
picks :: [x] -> [(x, [x])]
```

decorating every element not quite with something a bit like its context: from just the list of other elements, you can't see where the "hole" is. In truth,

```
∂[] x = ([x], [x])
```

separating the list of elements before the hole from the elements after the hole. Arguably, I should have written

```
picks :: [x] -> [(x, ([x], [x]))]
picks [] = []
picks (x : xs) = (x, ([], xs)) : [(y, (x : ys, ys')) | (y, (ys, ys')) <- picks xs]
```

and that's certainly a very useful operation too.

But what's really going on is quite sensible, and only a slight abuse. In the code I originally wrote, I'm locally taking `[]` to represent *finite bags* or *unordered lists*. Bags are lists without a notion of specific position, so if you select one element, its context is just the bag of the remaining elements. Indeed

```
∂Bag = Bag   -- really? why?
```

so the right notion of `picks` is indeed

```
picks :: Bag x -> Bag (x, Bag x)
```

Represent `Bag` by `[]` and that's what we had. Moreover, for bags, `plug` is just `(:)` and, up to bag equality (i.e., permutation), the second law for `picks` *does* hold.

Another way of looking at bags is as a power series. A bag is a choice of tuples of any size, with all possible permutations (*n!* for size *n*) identified. So we can write it combinatorially as a big sum of powers quotiented by factorials, because you have to divide by x^n by n! to account for the fact that each of the n! orders in which you could have chosen the x's gives you the same bag.

```
 Bag x = 1 + x + x^2/2! + x^3/3! + ...
```

so

```
∂Bag x = 0 + 1 + x      + x^2/2! + ...
```

shifting the series sideways. Indeed, you may well have recognized the power series for `Bag` as being that for `exp` (or *e*^x), which is famous for being its own derivative.

So, phew! There you go. An operation naturally arising from the datatype interpretation of the exponential function, being its own derivative, is the handy piece of kit for solving problems based on selection-without-replacement.

6.3 nub as a List Comprehension

The comment from @amalloy that list comprehensions are confined to a "local" perspective is the key insight here. There is a sensible way to write `nub` as a list comprehension, but you first need to change your perspective.

An often useful function sadly omitted from the library is the function which decorates each element of a list with its context.

```
picks :: [x] -> [([x], x, [x])]
picks []       = []
picks (x : xs) = ([], x, xs) : [(x : bs, y, as) | (bs, y, as) <- picks xs]
```

So

```
picks [1,2,3] =
[([],1,[2,3]), ([1],2,[3]), ([1,2],3,[])]
```

Each element of the list is put in the middle of a triple, with the elements 'before' to its left and the elements 'after' to its right.

This answer of mine explains the deep structure which makes `picks` in some sense a "standard" operation, derivable from the structure of lists. But we don't need that background information to deploy it.

The `picks` function gives us exactly the contextual information we need to write `nub` as a list comprehension. All we need to do is pick out the elements which don't occur in their own 'before lists'.

```
myNub :: Eq x => [x] -> [x]
myNub xs = [x | (bs, x, as) <- picks xs, not (elem x bs)]
```

I make no promises as to the efficiency of this operation, but I do like the clarity that comes from combining list comprehensions with extra spatial context.

6.4 How to make a binary tree zipper an instance of Comonad?

In the differential calculus, Leibniz's notation causes less confusion than Newton's because it is explicit about the variable with respect to which we differentiate. Contexts in things are given by differentiation, so we must take care what is being contextualized. Here, there are two notions of "substructure" at work: *subtrees* and *elements*. They each have different (but related) notions of "context" and hence of "zipper", where a zipper is the pair of a thing and its context.

Your `BTZ` type is presented as the notion of zipper for *subtrees*. However, the zipper comonadic construction works on zippers for *elements*: `extract` means "give element here"; `duplicate` means "decorate each element with its context". So you need element contexts. Confusingly, for these binary trees, element zippers and subtree zippers are isomorphic, but that is for a very particular reason (namely that they form a cofree comonad).

Generally, element- and subtree-zippers differ, e.g., for lists. If we start by building the element-zipper comonad for lists, we are less likely to get lost when we come back to trees. Let me try also to fill in a bit more of the general picture, for others as well as yourself.

Sublist contexts

The *sublist*-contexts for `[a]` are just given by `[a]`, being the list of elements we pass by on the way out from the sublist to the whole list. The sublist context for `[3,4]` in `[1,2,3,4]` is `[2,1]`. Subnode contexts for recursive data are always lists representing what you see on the path from the node to the root. The type of each step is given by the partial derivative of the formula for one node of data with respect to the recursive variable. So here

```
[a] = t where              -- t is the recursive variable standing for [a]
  t = 1 + a*t              -- lists of a are either [] or an (a : t) pair
∂/∂t (1 + a*t) = a         -- that's one step on a path from node to root
sublist contexts are [a]   -- a list of such steps
```

So a sublist-zipper is a pair

```
data LinLZ a = LinLZ
  {  subListCtxt  :: [a]
  ,  subList      :: [a]
  }
```

We can write the function which plugs a sublist back into its context, reversing back up the path

```
plugLinLZ :: LinLZ a -> [a]
plugLinLZ (LinLZ { subListCtxt = [],      subList = ys})  = ys
plugLinLZ (LinLZ { subListCtxt = x : xs,  subList = ys})
  = plugLinLZ (LinLZ { subListCtxt = xs,  subList = x : ys})
```

But we can't make `LinLZ` a `Comonad`, because (for example) from

```
LinLZ { subListCtxt = [], subList = [] }
```

we can't `extract` an *element* (an `a` from `LinLZ a`), only a sublist.

List Element Contexts

A list *element* context is a pair of lists: the elements before the element in focus, and the elements after it. An element context in a recursive structure is always a pair: first give the subnode-context for the subnode where the element is stored, then give the context for the element in its node. We get the element-in-its-node context by differentiating the formula for a node with respect to the variable which stands for elements.

```
[a] = t where              -- t is the recursive variable standing for [a]
  t = 1 + a*t              -- lists of a are either [] or an (a : t) pair
∂/∂a (1 + a*t) = t = [a]   -- the context for the head element is the tail list
```

So an element context is given by a pair

```
type DL a =
  (  [a]     -- the sublist context for the node where the element is
  ,  [a]     -- the tail of the node where the element is
  )
```

and an element zipper is given by pairing such a context with the element "in the hole".

```
data ZL a = ZL
  {  this :: a
  ,  between :: DL a
  }  deriving (Show, Eq, Functor)
```

You can turn such a zipper back into a list (going "out" from an element) by first reconstituting the sublist where the element sits, giving us a sublist zipper, then plugging the sublist into its sublist-context.

```
outZL :: ZL a -> [a]
outZL (ZL { this = x, between = (zs, xs) })
  = plugLinLZ (LinLZ { subListCtxt = zs, subList = x : xs })
```

Putting each element into context

Given a list, we can pair each element up with its context. We get the list of ways we can "go into" one of the elements. We start like this,

```
into :: [a] -> [ZL a]
into xs = moreInto (LinLZ { subListCtxt = [], subList = xs })
```

but the real work is done by the helper function which works on a list-in-context.

```
moreInto :: LinLZ a -> [ZL a]
moreInto (LinLZ { subListCtxt = _,   subList = [] })      = []
moreInto (LinLZ { subListCtxt = zs,  subList = x : xs })
  =  ZL { this = x, between = (zs, xs) }
  :  moreInto (LinLZ { subListCtxt = x : zs,  subList = xs })
```

Notice that the output echoes the shape of the current `subList`. Also, the zipper in `x`'s place has `this = x`. Also, the generating zipper for decorating `xs` has `subList = xs` and the correct context, recording that we have moved past `x`. Testing,

```
into [1,2,3,4] =
  [  ZL {this = 1, between = ([],[2,3,4])}
  ,  ZL {this = 2, between = ([1],[3,4])}
  ,  ZL {this = 3, between = ([2,1],[4])}
  ,  ZL {this = 4, between = ([3,2,1],[])}
  ]
```

Comonadic structure for list element zippers

We've seen how to go out from an element, or into one of the available elements. The comonadic structure tells us how to move between elements, either staying where we are, or moving to one of the others.

```
instance Comonad ZL where
```

The `extract` gives us the element we're visiting.

```
  extract = this
```

To `duplicate` a zipper, we replace the current element `x` with the whole current zipper `zl` (whose `this = x`)...

```
  duplicate zl@(ZL { this = x, between = (zs, ys) }) = ZL
    {  this = zl
```

...and we work our way through the context, showing how to refocus at each element. Our existing `moreInto` lets us move inward, but we must also move `outward`...

```
    ,  between =
         (  outward (LinLZ { subListCtxt = zs, subList = x : ys })
         ,  moreInto (LinLZ { subListCtxt = x : zs, subList = ys })
         )
    }
```

...which involves travelling back along the context, moving elements into the sublist, as follows

```
    where
      outward (LinLZ { subListCtxt = [], subList = _ }) = []
      outward (LinLZ { subListCtxt = z : zs, subList = ys })
        =  ZL { this = z, between = (zs, ys) }
        :  outward (LinLZ { subListCtxt = zs, subList = z : ys })
```

So we get

```
duplicate ZL {this = 2, between = ([1],[3,4])}
  = ZL
  {  this = ZL {this = 2, between = ([1],[3,4])}
  ,  between =
     (  [  ZL {this = 1, between = ([],[2,3,4])}  ]
     ,  [  ZL {this = 3, between = ([2,1],[4])}
        ,  ZL {this = 4, between = ([3,2,1],[])}
        ]
     )
  }
```

where `this` is "staying at `2`" and we are `between` "moving to `1`" and "moving to `3` or moving to `4`".

So, the comonadic structure shows us how we can move between different *elements* located inside a list. The sublist structure plays a key role in finding the nodes where the elements are, but the zipper structure `duplicated` is an *element* zipper.

So what about trees?

Digression: labelled trees are comonads already

Let me refactor your type of binary trees to bring out some structure. Literally, let us pull the element which labels a leaf or a fork out as a common factor. Let us also isolate the functor (`TF`) which explains this leaf-or-fork subtree structure.

```
data TF t = Leaf | Fork (t, t) deriving (Show, Eq, Functor)
data BT a = a :& TF (BT a) deriving (Show, Eq, Functor)
```

That is, every tree node has a label, whether it is a leaf or a fork.

Wherever we have the structure that every node has a label and a blob of substructures, we have a comonad: the *cofree comonad*. Let me refactor a little more, abstracting out `TF`...

```
data CoFree f a = a :& f (CoFree f a) deriving (Functor)
```

...so we have a general `f` where we had `TF` before. We can recover our specific trees.

```
data TF t = Leaf | Fork (t, t) deriving (Show, Eq, Functor)
type BT = CoFree TF
deriving instance Show a => Show (BT a)
deriving instance Eq a => Eq (BT a)
```

And now, once for all, we can give the cofree comonad construction. As every subtree has a root element, every element can be decorated with the tree whose root it is.

```
instance Functor f => Comonad (CoFree f) where
  extract   (a :& _)     = a                          -- extract root element
  duplicate t@(a :& ft)  = t :& fmap duplicate ft     -- replace root element by whole t
```

Let's have an example

```
aTree =
  0 :& Fork
  (  1 :& Fork
     (  2 :& Leaf
     ,  3 :& Leaf
     )
  ,  4 :& Leaf
  )

duplicate aTree =
  (0 :& Fork (1 :& Fork (2 :& Leaf,3 :& Leaf),4 :& Leaf)) :& Fork
  (  (1 :& Fork (2 :& Leaf,3 :& Leaf)) :& Fork
     (  (2 :& Leaf) :& Leaf
     ,  (3 :& Leaf) :& Leaf
     )
  ,  (4 :& Leaf) :& Leaf
  )
```

See? Each element has been paired with its subtree!

Lists do not give rise to a cofree comonad, because not every node has an element: specifically, `[]` has no element. In a cofree comonad, there is always an element where you are, and you can see further down into the tree structure, *but not further up*.

In an element zipper comonad, there is always an element where you are, and you can see both up and down.

Subtree and element contexts in binary trees

Algebraically

```
d/dt (TF t) = d/dt (1 + t*t) = 0 + (1*t + t*1)
```

so we may define

```
type DTF t = Either ((), t) (t, ())
```

saying that a subtree inside the "blob of substructures" is either on the left or the right. We can check that "plugging in" works.

```
plugF :: t -> DTF t -> TF t
plugF  t  (Left   ((), r))  = Fork (t, r)
plugF  t  (Right  (l, ()))  = Fork (l, t)
```

If we instantiate `t` and pair up with the node label, we get one step of subtree context

```
type BTStep a = (a, DTF (BT a))
```

which is isomorphic to `Partial` in the question.

```
plugBTinBT :: BT a -> BTStep a -> BT a
plugBTinBT t (a, d) = a :& plugF t d
```

So, a *subtree*-context for one `BT a` inside another is given by `[BTStep a]`.

But what about an *element* context? Well, every element labels some subtree, so we should record both that subtree's context and the rest of the tree labelled by the element.

```
data DBT a = DBT
  {  below  :: TF (BT a)     -- the rest of the element's node
  ,  above  :: [BTStep a]    -- the subtree context of the element's node
  }  deriving (Show, Eq)
```

Annoyingly, I have to roll my own `Functor` instance.

```
instance Functor DBT where
  fmap f (DBT { above = a, below = b }) = DBT
    {  below = fmap (fmap f) b
    ,  above = fmap (f *** (either
         (Left   . (id *** fmap f))
         (Right  . (fmap f *** id)))) a
    }
```

Now I can say what an *element* zipper is.

```
data BTZ a = BTZ
  {  here  :: a
  ,  ctxt  :: DBT a
  }  deriving (Show, Eq, Functor)
```

If you're thinking "what's new?", you're right. We have a subtree context, `above`, together with a subtree given by `here` and `below`. And that's because the only elements are those which label nodes. Splitting a node up into an element and its context is the same as splitting it into its label and its blob of substructures. That is, we get this coincidence for cofree comonads, but not in general.

However, this coincidence is only a distraction! As we saw with lists, we don't need element-zippers to be the same as subnode-zippers to make element-zippers a comonad.

Following the same pattern as lists above, we can decorate every element with its context. The work is done by a helper function which accumulates the subtree context we are currently visiting.

```
down :: BT a -> BT (BTZ a)
down t = downIn t []

downIn :: BT a -> [BTStep a] -> BT (BTZ a)
downIn (a :& ft) ads =
  BTZ { here = a, ctxt = DBT { below = ft, above = ads } }
  :& furtherIn a ft ads
```

Note that `a` is replaced by a zipper focused on `a`. The subtrees are handled by another helper.

```
furtherIn :: a -> TF (BT a) -> [BTStep a] -> TF (BT (BTZ a))
furtherIn a Leaf           ads  = Leaf
furtherIn a (Fork (l, r))  ads  = Fork
  (  downIn l ((a, Left   ((), r)) : ads)
  ,  downIn r ((a, Right  (l, ())) : ads)
  )
```

See that `furtherIn` preserves the tree structure, but grows the subtree context suitably when it visits a subtree.

Let's double check.

```
down aTree =
  BTZ {  here  = 0, ctxt = DBT {
         below = Fork (1 :& Fork (2 :& Leaf,3 :& Leaf),4 :& Leaf),
         above = []}} :& Fork
  (  BTZ {  here = 1, ctxt = DBT {
            below = Fork (2 :& Leaf,3 :& Leaf),
            above = [(0,Left ((),4 :& Leaf))]}} :& Fork
     (  BTZ {  here = 2, ctxt = DBT {
               below = Leaf,
               above = [(1,Left ((),3 :& Leaf)),(0,Left ((),4 :& Leaf))]}} :& Leaf
     ,  BTZ {  here = 3, ctxt = DBT {
               below = Leaf,
               above = [(1,Right (2 :& Leaf,())),(0,Left ((),4 :& Leaf))]}} :& Leaf
     )
  ,  BTZ {  here = 4, ctxt = DBT {
            below = Leaf,
            above = [(0,Right (1 :& Fork (2 :& Leaf,3 :& Leaf),()))]}} :& Leaf)
```

See? Each element is decorated with its whole context, not just the tree below it.

Binary tree zippers form a Comonad

Now that we can decorate elements with their contexts, let us build the `Comonad` instance. As before…

```
instance Comonad BTZ where
  extract = here
```

…`extract` tells us the element in focus, and we can make use of our existing machinery to go further into the tree, but we need to build new kit to explore the ways we can move outwards.

```
  duplicate z@(BTZ { here = a, ctxt = DBT { below = ft, above = ads }}) = BTZ
    {  here = z
    ,  ctxt = DBT
         {  below = furtherIn a ft ads  -- move somewhere below a
         ,  above = go_a (a :& ft) ads  -- go above a
         }
    } where
```

To go outwards, as with lists, we must move back along the path towards the root. As with lists, each step on the path is a place we can visit.

```
    go_a t []          = []
    go_a t (ad : ads)  = go_ad t ad ads : go_a (plugBTinBT t ad) ads
    go_ad t (a, d) ads =
      (  BTZ { here = a, ctxt = DBT { below = plugF t d, above = ads } }  -- visit here
      ,  go_d t a d ads                                                   -- try other s
      )
```

Unlike with lists, there are alternative branches along that path to explore. Wherever the path stores an unvisited subtree, we must decorate *its* elements with *their* contexts.

```
    go_d t a (Left ((), r)) ads = Left ((), downIn r ((a, Right (t, ())) : ads))
    go_d t a (Right (l, ())) ads = Right (downIn l ((a, Left ((), t)) : ads), ())
```

So now we've explained how to refocus from any element position to any other.

Let's see. Here we were visiting `1`:

```
duplicate (BTZ {here = 1, ctxt = DBT {
                below = Fork (2 :& Leaf,3 :& Leaf),
                above = [(0,Left ((),4 :& Leaf))]}}) =
  BTZ {here = BTZ {here = 1, ctxt = DBT {
                   below = Fork (2 :& Leaf,3 :& Leaf),
                   above = [(0,Left ((),4 :& Leaf))]}}, ctxt = DBT {
       below = Fork (BTZ {here = 2, ctxt = DBT {
                          below = Leaf,
                          above = [(1,Left ((),3 :& Leaf)),(0,Left ((),4 :& Leaf))]}} :&
                    ,BTZ {here = 3, ctxt = DBT {
                          below = Leaf,
                          above = [(1,Right (2 :& Leaf,())),(0,Left ((),4 :& Leaf))]}}
                   ),
       above = [(BTZ {here = 0, ctxt = DBT {
                      below = Fork (1 :& Fork (2 :& Leaf,3 :& Leaf),4 :& Leaf),
                      above = []}}
                ,Left ((),BTZ {here = 4, ctxt = DBT {
                               below = Leaf,
                               above = [(0,Right (1 :& Fork (2 :& Leaf,3 :& Leaf),()))]
                )
               ]}}
```

By way of testing the comonad laws on a small sample of data, let us check:

```
fmap (\ z -> extract (duplicate z) == z) (down aTree)
  = True :& Fork (True :& Fork (True :& Leaf,True :& Leaf),True :& Leaf)
fmap (\ z -> fmap extract (duplicate z) == z) (down aTree)
  = True :& Fork (True :& Fork (True :& Leaf,True :& Leaf),True :& Leaf)
fmap (\ z -> fmap duplicate (duplicate z) == duplicate (duplicate z)) (down aTree)
  = True :& Fork (True :& Fork (True :& Leaf,True :& Leaf),True :& Leaf)
```

6.5 What's the `absurd` function in Data.Void useful for?

Consider this representation for lambda terms parametrized by their free variables. (See papers by Bellegarde and Hook 1994, Bird and Paterson 1999, Altenkirch and Reus 1999.)

```
data Tm a  = Var a
           | Tm a :$ Tm a
           | Lam (Tm (Maybe a))
```

You can certainly make this a `Functor`, capturing the notion of renaming, and a `Monad` capturing the notion of substitution.

```
instance Functor Tm where
  fmap rho (Var a)   = Var (rho a)
  fmap rho (f :$ s)  = fmap rho f :$ fmap rho s
```

```
  fmap rho (Lam t)   = Lam (fmap (fmap rho) t)

instance Monad Tm where
  return = Var
  Var a     >>= sig  = sig a
  (f :$ s)  >>= sig  = (f >>= sig) :$ (s >>= sig)
  Lam t     >>= sig  = Lam (t >>= maybe (Var Nothing) (fmap Just . sig))
```

Now consider the *closed* terms: these are the inhabitants of `Tm Void`. You should be able to embed the closed terms into terms with arbitrary free variables. How?

```
fmap absurd :: Tm Void -> Tm a
```

The catch, of course, is that this function will traverse the term doing precisely nothing. But it's a touch more honest than `unsafeCoerce`. And that's why `vacuous` was added to `Data.Void`...

Or write an evaluator. Here are values with free variables in `b`.

```
data Val b
  =  b :$$ [Val b]                                  -- a stuck application
  |  forall a. LV (a -> Val b) (Tm (Maybe a))       -- we have an incomplete environment
```

I've just represented lambdas as closures. The evaluator is parametrized by an environment mapping free variables in `a` to values over `b`.

```
eval :: (a -> Val b) -> Tm a -> Val b
eval g (Var a)   = g a
eval g (f :$ s)  = eval g f $$ eval g s where
  (b :$$ vs)  $$ v  = b :$$ (vs ++ [v])          -- stuck application gets longer
  LV g t      $$ v  = eval (maybe v g) t         -- an applied lambda gets unstuck
eval g (Lam t)   = LV g t
```

You guessed it. To evaluate a closed term at any target

```
eval absurd :: Tm Void -> Val b
```

More generally, `Void` is seldom used on its own, but is handy when you want to instantiate a type parameter in a way which indicates some sort of impossibility (e.g., here, using a free variable in a closed term). Often these parametrized types come with higher-order functions lifting operations on the parameters to operations on the whole type (e.g., here, `fmap`, `>>=`, `eval`). So you pass `absurd` as the general-purpose operation on `Void`.

For another example, imagine using `Either e v` to capture computations which hopefully give you a `v` but might raise an exception of type `e`. You might use this approach to document risk of bad behaviour uniformly. For perfectly well behaved computations in this setting, take `e` to be `Void`, then use

```
either absurd id :: Either Void v -> v
```

to run safely or

```
either absurd Right :: Either Void v -> Either e v
```

to embed safe components in an unsafe world.

Oh, and one last hurrah, handling a "can't happen". It shows up in the generic zipper construction, everywhere that the cursor can't be.

```
class Differentiable f where
  type D f :: * -> *              -- an f with a hole
  plug :: (D f x, x) -> f x       -- plugging a child in the hole

newtype K a     x  = K a          -- no children, just a label
newtype I       x  = I x          -- one child
data (f :+: g)  x  = L (f x)      -- choice
                   | R (g x)
data (f :*: g)  x  = f x :&: g x  -- pairing

instance Differentiable (K a) where
  type D (K a) = K Void           -- no children, so no way to make a hole
  plug (K v, x) = absurd v        -- can't reinvent the label, so deny the hole!
```

I decided not to delete the rest, even though it's not exactly relevant.

```
instance Differentiable I where
  type D I = K ()
  plug (K (), x) = I x

instance (Differentiable f, Differentiable g) => Differentiable (f :+: g) where
  type D (f :+: g) = D f :+: D g
  plug (L df, x) = L (plug (df, x))
  plug (R dg, x) = R (plug (dg, x))

instance (Differentiable f, Differentiable g) => Differentiable (f :*: g) where
  type D (f :*: g) = (D f :*: g) :+: (f :*: D g)
  plug (L (df :&: g), x) = plug (df, x) :&: g
  plug (R (f :&: dg), x) = f :&: plug (dg, x)
```

Actually, maybe it is relevant. If you're feeling adventurous, this unfinished article shows how to use `Void` to compress the representation of terms with free variables

```
data Term f x = Var x | Con (f (Term f x))   -- the Free monad, yet again
```

in any syntax generated freely from a `Differentiable` and `Traversable` functor `f`. We use `Term f Void` to represent regions with no free variables, and `[D f (Term f Void)]` to represent *tubes* tunnelling through regions with no free variables either to an isolated free variable, or to a junction in the paths to two or more free variables. Must finish that article sometime.

For a type with no values (or at least, none worth speaking of in polite company), `Void` is remarkably useful. And `absurd` is how you use it.

6.6 Writing cojoin or cobind for n-dimensional grids

Jagger/Richards: you can't always get what you want, but if you try sometime you just might find that you get what you need.

6.6.1 Cursors in Lists

Let me rebuild the components of your structure using snoc- and cons-lists to keep the spatial properties clear. I define

```
data Bwd x = B0 | Bwd x :< x deriving (Functor, Foldable, Traversable, Show)
data Fwd x = F0 | x :> Fwd x deriving (Functor, Foldable, Traversable, Show)
infixl 5 :<
infixr 5 :>

data Cursor x = Cur (Bwd x) x (Fwd x) deriving (Functor, Foldable, Traversable, Show)
```

Let's have comonads

```
class Functor f => Comonad f where
  counit  :: f x -> x
  cojoin  :: f x -> f (f x)
```

and let's make sure cursors are comonads

```
instance Comonad Cursor where
  counit (Cur _ x _) = x
  cojoin c = Cur (lefts c) c (rights c) where
    lefts (Cur B0 _ _) = B0
    lefts (Cur (xz :< x) y ys) = lefts c :< c where c = Cur xz x (y :> ys)
    rights (Cur _ _ F0) = F0
    rights (Cur xz x (y :> ys)) = c :> rights c where c = Cur (xz :< x) y ys
```

If you're turned on to this kind of stuff, you'll note that `Cursor` is a spatially pleasing variant of `InContext []`

```
InContext f x = (x, ∂f x)
```

where ∂ takes the formal derivative of a functor, giving its notion of one-hole context. `InContext f` is always a `Comonad`, as mentioned in this answer, and what we have here is just that `Comonad` induced by the differential structure, where `counit` extracts the element at the focus and `cojoin` decorates each element with its own context, effectively giving you a context full of refocused cursors and with an unmoved cursor at its focus. Let's have an example.

```
> cojoin (Cur (B0 :< 1) 2 (3 :> 4 :> F0))
Cur (B0 :< Cur B0 1 (2 :> 3 :> 4 :> F0))
    (Cur (B0 :< 1) 2 (3 :> 4 :> F0))
    (  Cur (B0 :< 1 :< 2) 3 (4 :> F0)
    :> Cur (B0 :< 1 :< 2 :< 3) 4 F0
    :> F0)
```

See? The 2 in focus has been decorated to become the cursor-at-2; to the left, we have the list of the cursor-at-1; to the right, the list of the cursor-at-3 and the cursor-at-4.

6.6.2 Composing Cursors, Transposing Cursors?

Now, the structure you're asking to be a `Comonad` is the n-fold composition of `Cursor`. Let's have

```
newtype (:.:) f g x = C {unC :: f (g x)} deriving Show
```

To persuade comonads `f` and `g` to compose, the `counit`s compose neatly, but you need a "distributive law"

```
transpose :: f (g x) -> g (f x)
```

so you can make the composite `cojoin` like this

```
f (g x)
  -(fmap cojoin)->
f (g (g x))
  -cojoin->
f (f (g (g x)))
  -(fmap transpose)->
f (g (f (g x)))
```

What laws should `transpose` satisfy? Probably something like

```
counit . transpose = fmap counit
cojoin . transpose = fmap transpose . transpose . fmap cojoin
```

or whatever it takes to ensure that any two ways to shoogle some sequence of f's and g's from one order to another give the same result.

Can we define a `transpose` for `Cursor` with itself? One way to get some sort of transposition cheaply is to note that `Bwd` and `Fwd` are *zippily* applicative, hence so is `Cursor`.

```
instance Applicative Bwd where
  pure x = pure x :< x
  (fz :< f) <*> (sz :< s) = (fz <*> sz) :< f s
  _ <*> _ = B0

instance Applicative Fwd where
  pure x = x :> pure x
  (f :> fs) <*> (s :> ss) = f s :> (fs <*> ss)
  _ <*> _ = F0

instance Applicative Cursor where
  pure x = Cur (pure x) x (pure x)
  Cur fz f fs <*> Cur sz s ss = Cur (fz <*> sz) (f s) (fs <*> ss)
```

And here you should begin to smell the rat. Shape mismatch results in *truncation*, and that's going to break the obviously desirable property that self-transpose is self-inverse. Any kind of raggedness will not survive. We do get a transposition operator: `sequenceA`, and for completely regular data, all is bright and beautiful.

```
> regularMatrixCursor
Cur (B0 :< Cur (B0 :< 1) 2 (3 :> F0))
           (Cur (B0 :< 4) 5 (6 :> F0))
           (Cur (B0 :< 7) 8 (9 :> F0) :> F0)
> sequenceA regularMatrixCursor
Cur (B0 :< Cur (B0 :< 1) 4 (7 :> F0))
           (Cur (B0 :< 2) 5 (8 :> F0))
           (Cur (B0 :< 3) 6 (9 :> F0) :> F0)
```

But even if I just move one of the inner cursors out of alignment (never mind making the sizes ragged), things go awry.

```
> raggedyMatrixCursor
Cur (B0 :< Cur ((B0 :< 1) :< 2) 3 F0)
           (Cur (B0 :< 4) 5 (6 :> F0))
           (Cur (B0 :< 7) 8 (9 :> F0) :> F0)
> sequenceA raggedyMatrixCursor
Cur (B0 :< Cur (B0 :< 2) 4 (7 :> F0))
           (Cur (B0 :< 3) 5 (8 :> F0))
           F0
```

When you have one outer cursor position and multiple inner cursor positions, there's no transposition which is going to behave well. Self-composing `Cursor` allows the inner structures to be ragged relative to one another, so no `transpose`, no `cojoin`. You can, and I did, define

```
instance (Comonad f, Traversable f, Comonad g, Applicative g) =>
  Comonad (f :.: g) where
    counit = counit . counit . unC
    cojoin = C . fmap (fmap C . sequenceA) . cojoin . fmap cojoin . unC
```

but it's an onus on us to make sure we keep the inner structures regular. If you're willing to accept that burden, then you can iterate, because `Applicative` and `Traversable` are readily closed under composition. Here are the bits and pieces

```
instance (Functor f, Functor g) => Functor (f :.: g) where
  fmap h (C fgx) = C (fmap (fmap h) fgx)

instance (Applicative f, Applicative g) => Applicative (f :.: g) where
  pure = C . pure . pure
  C f <*> C s = C (pure (<*>) <*> f <*> s)

instance (Functor f, Foldable f, Foldable g) => Foldable (f :.: g) where
  fold = fold . fmap fold . unC

instance (Traversable f, Traversable g) => Traversable (f :.: g) where
  traverse h (C fgx) = C <$> traverse (traverse h) fgx
```

Edit: for completeness, here's what it does when all is regular,

```
> cojoin (C regularMatrixCursor)
C {unC = Cur (B0 :< Cur (B0 :<
  C {unC = Cur B0 (Cur B0 1 (2 :> (3 :> F0))) (Cur B0 4 (5 :> (6 :> F0)) :> (Cur B0 7 (8
 (C {unC = Cur B0 (Cur (B0 :< 1) 2 (3 :> F0)) (Cur (B0 :< 4) 5 (6 :> F0) :> (Cur (B0 :<
 (C {unC = Cur B0 (Cur ((B0 :< 1) :< 2) 3 F0) (Cur ((B0 :< 4) :< 5) 6 F0 :> (Cur ((B0 :<
(Cur (B0 :<
  C {unC = Cur (B0 :< Cur B0 1 (2 :> (3 :> F0))) (Cur B0 4 (5 :> (6 :> F0))) (Cur B0 7
 (C {unC = Cur (B0 :< Cur (B0 :< 1) 2 (3 :> F0)) (Cur (B0 :< 4) 5 (6 :> F0)) (Cur (B0 :<
 (C {unC = Cur (B0 :< Cur ((B0 :< 1) :< 2) 3 F0) (Cur ((B0 :< 4) :< 5) 6 F0) (Cur ((B0
(Cur (B0 :<
  C {unC = Cur ((B0 :< Cur B0 1 (2 :> (3 :> F0))) :< Cur B0 4 (5 :> (6 :> F0))) (Cur B0
 (C {unC = Cur ((B0 :< Cur (B0 :< 1) 2 (3 :> F0)) :< Cur (B0 :< 4) 5 (6 :> F0)) (Cur (B0
 (C {unC = Cur ((B0 :< Cur ((B0 :< 1) :< 2) 3 F0) :< Cur ((B0 :< 4) :< 5) 6 F0) (Cur ((B
:> F0)}
```

6.6.3 Hancock's Tensor Product

For regularity, you need something stronger than composition. You need to be able to capture the notion of "an f-structure of g-structures-all-the-same-shape". This is what the inestimable Peter Hancock calls the "tensor product", which I'll write `f :><: g`: there's one "outer" f-shape and one "inner" g-shape common to all the inner g-structures, so transposition is readily definable and always self-inverse. Hancock's tensor is not conveniently definable in Haskell, but in a dependently typed setting, it's easy to formulate a notion of "container" which has this tensor.

To give you the idea, consider a degenerate notion of container

```
data (:<|) s p x = s :<| (p -> x)
```

where we say `s` is the type of "shapes" and `p` the type of "positions". A value consists of the choice of a shape and the storage of an `x` in each position. In the dependent case, the type of positions may depend on the choice of the shape (e.g., for lists, the shape is a number (the length), and you have that many positions). These containers have a tensor product

```
(s :<| p) :><: (s' :<| p')  =  (s, s') :<| (p, p')
```

which is like a generalised matrix: a pair of shapes give the dimensions, and then you have an element at each pair of positions. You can do this perfectly well when types `p` and `p'` depend on values in `s` and `s'`, and that is exactly Hancock's definition of the tensor product of containers.

6.6.4 InContext for Tensor Products

Now, as you may have learned in high school, ∂`(s :<| p) = (s, p) :<| (p-1)` where `p-1` is some type with one fewer element than `p`. Like $\partial(sx^p) = (s\mathrm{p})x^{(p-1)}$. *You select one position (recording it in the shape) and delete it. The snag is that* `p-1` *is tricky to get your hands on without dependent types. But* `InContext` *selects a position* without deleting it*.

```
InContext (s :<| p) ~= (s, p) :<| p
```

This works just as well for the dependent case, and we joyously acquire

```
InContext (f :><: g) ~= InContext f :><: InContext g
```

Now we know that `InContext f` is always a `Comonad`, and this tells us that tensor products of `InContext`s are comonadic because they are themselves `InContext`s. That's to say, you pick one position per dimension (and that gives you exactly one position in the whole thing), where before we had one outer position and lots of inner positions. With the tensor product replacing composition, everything works sweetly.

6.6.5 Naperian Functors

But there is a subclass of `Functor` for which the tensor product and the composition coincide. These are the `Functor`s `f` for which `f () ~ ()`: i.e., there is only one shape anyway, so raggedy values in compositions are ruled out in the first place. These `Functor`s are all isomorphic to `(p ->)` for some position set `p` which we can think of as the *logarithm* (the exponent to which `x` must be raised to give `f x`). Correspondingly, Hancock calls these `Naperian` functors after John Napier (whose ghost haunts the part of Edinburgh where Hancock lives).

```
class Applicative f => Naperian f where
  type Log f
  project :: f x -> Log f -> x
  positions :: f (Log f)
  --- project positions = id
```

A `Naperian` functor has a logarithm, inducing a `project`ion function mapping positions to the elements found there. `Naperian` functors are all zippily `Applicative`, with `pure` and `<*>` corresponding to the K and S combinators for the projections. It's also possible to construct a value where at each position is stored that very position's representation. Laws of logarithms which you might remember pop up pleasingly.

```
newtype Id x = Id {unId :: x} deriving Show

instance Naperian Id where
  type Log Id = ()
  project (Id x) () = x
```

```
  positions = Id ()

newtype (:*:) f g x = Pr (f x, g x) deriving Show

instance (Naperian f, Naperian g) => Naperian (f :*: g) where
  type Log (f :*: g) = Either (Log f) (Log g)
  project (Pr (fx, gx)) (Left p) = project fx p
  project (Pr (fx, gx)) (Right p) = project gx p
  positions = Pr (fmap Left positions, fmap Right positions)
```

Note that a fixed size array (a *vector*) is given by `(Id :*: Id :*: ... :*: Id :*: One)`, where `One` is the constant unit functor, whose logarithm is `Void`. So an array is `Naperian`. Now, we also have

```
instance (Naperian f, Naperian g) => Naperian (f :.: g) where
  type Log (f :.: g) = (Log f, Log g)
  project (C fgx) (p, q) = project (project fgx p) q
  positions = C $ fmap (\ p -> fmap (p ,) positions) positions
```

which means that multi-dimensional arrays are `Naperian`.

To construct a version of `InContext f` for `Naperian f`, just point to a position!

```
data Focused f x = f x :@ Log f

instance Functor f => Functor (Focused f) where
  fmap h (fx :@ p) = fmap h fx :@ p

instance Naperian f => Comonad (Focused f) where
  counit (fx :@ p) = project fx p
  cojoin (fx :@ p) = fmap (fx :@) positions :@ p
```

So, in particular, a `Focused` n-dimensional array will indeed be a comonad. A composition of vectors is a tensor product of n vectors, because vectors are `Naperian`. But the `Focused` n-dimensional array will be the n-fold tensor product, *not the composition*, of the n `Focused` vectors which determine its dimensions. To express this comonad in terms of zippers, we'll need to express them in a form which makes it possible to construct the tensor product. I'll leave that as an exercise for the future.

6.7 Zipper Comonads, Generically

Like the childcatcher in Chitty-Chitty-Bang-Bang luring kids into captivity with sweets and toys, recruiters to undergraduate Physics like to fool about with soap bubbles and boomerangs, but when the door clangs shut, it's "Right, children, time to learn about partial differentiation!". Me too. Don't say I didn't warn you.

Here's another warning: the following code needs {`-# LANGUAGE KitchenSink #-`}, or rather

```
{-# LANGUAGE TypeFamilies, FlexibleContexts, TupleSections, GADTs, DataKinds,
    TypeOperators, FlexibleInstances, RankNTypes, ScopedTypeVariables,
    StandaloneDeriving, UndecidableInstances #-}
```

in no particular order.

Differentiable functors give comonadic zippers

What is a differentiable functor, anyway?

```
class (Functor f, Functor (DF f)) => Diff1 f where
  type DF f :: * -> *
  upF      ::  ZF f x  ->  f x
  downF    ::  f x     ->  f (ZF f x)
  aroundF  ::  ZF f x  ->  ZF f (ZF f x)

data ZF f x = (:<-:) {cxF :: DF f x, elF :: x}
```

It's a functor which has a derivative, which is also a functor. The derivative represents a one-hole context for an *element*. The zipper type `ZF f x` represents the pair of a one-hole context and the element in the hole.

The operations for `Diff1` describe the kinds of navigation we can do on zippers (without any notion of "leftward" and "rightward", for which see my Clowns and Jokers paper). We can go "upward", reassembling the structure by plugging the element in its hole. We can go "downward", finding every way to visit an element in a give structure: we decorate every element with its context. We can go "around", taking an existing zipper and decorating each element with its context, so we find all the ways to refocus (and how to keep our current focus).

Now, the type of `aroundF` might remind some of you of

```
class Functor c => Comonad c where
  extract    :: c x -> x
  duplicate  :: c x -> c (c x)
```

and you're right to be reminded! We have, with a hop and a skip,

```
instance Diff1 f => Functor (ZF f) where
  fmap f (df :<-: x) = fmap f df :<-: f x

instance Diff1 f => Comonad (ZF f) where
  extract    = elF
  duplicate  = aroundF
```

and we insist that

```
extract . duplicate == id
fmap extract . duplicate == id
duplicate . duplicate == fmap duplicate . duplicate
```

We also need that

```
fmap extract (downF xs) == xs              -- downF decorates the element in position
fmap upF (downF xs) = fmap (const xs) xs   -- downF gives the correct context
```

Polynomial functors are differentiable

Constant functors are differentiable.

```
data KF a x = KF a
instance Functor (KF a) where
  fmap f (KF a) = KF a

instance Diff1 (KF a) where
  type DF (KF a) = KF Void
  upF (KF w :<-: _) = absurd w
  downF (KF a) = KF a
  aroundF (KF w :<-: _) = absurd w
```

There's nowhere to put an element, so it's impossible to form a context. There's nowhere to go `upF` or `downF` from, and we easily find all none of the ways to go `downF`.

The **identity** functor is differentiable.

```
data IF x = IF x
instance Functor IF where
  fmap f (IF x) = IF (f x)

instance Diff1 IF where
  type DF IF = KF ()
  upF (KF () :<-: x) = IF x
  downF (IF x) = IF (KF () :<-: x)
  aroundF z@(KF () :<-: x) = KF () :<-: z
```

There's one element in a trivial context, `downF` finds it, `upF` repacks it, and `aroundF` can only stay put.

Sum preserves differentiability.

```
data (f :+: g) x = LF (f x) | RF (g x)
instance (Functor f, Functor g) => Functor (f :+: g) where
  fmap h (LF f) = LF (fmap h f)
  fmap h (RF g) = RF (fmap h g)

instance (Diff1 f, Diff1 g) => Diff1 (f :+: g) where
  type DF (f :+: g) = DF f :+: DF g
  upF (LF f' :<-: x) = LF (upF (f' :<-: x))
  upF (RF g' :<-: x) = RF (upF (g' :<-: x))
```

The other bits and pieces are a bit more of a handful. To go `downF`, we must go `downF` inside the tagged component, then fix up the resulting zippers to show the tag in the context.

```
  downF (LF f) = LF (fmap (\ (f' :<-: x) -> LF f' :<-: x) (downF f))
  downF (RF g) = RF (fmap (\ (g' :<-: x) -> RF g' :<-: x) (downF g))
```

To go `aroundF`, we strip the tag, figure out how to go around the untagged thing, then restore the tag in all the resulting zippers. The element in focus, `x`, is replaced by its entire zipper, `z`.

```
  aroundF z@(LF f' :<-: (x :: x)) =
    LF (fmap (\ (f' :<-: x) -> LF f' :<-: x) . cxF $ aroundF (f' :<-: x :: ZF f x))
    :<-: z
  aroundF z@(RF g' :<-: (x :: x)) =
    RF (fmap (\ (g' :<-: x) -> RF g' :<-: x) . cxF $ aroundF (g' :<-: x :: ZF g x))
    :<-: z
```

Note that I had to use `ScopedTypeVariables` to disambiguate the recursive calls to `aroundF`. As a type function, `DF` is not injective, so the fact that `f' :: D f x` is not enough to force `f' :<-: x :: Z f x`.

Product preserves differentiability.

```
data (f :*: g) x = f x :*: g x
instance (Functor f, Functor g) => Functor (f :*: g) where
  fmap h (f :*: g) = fmap h f :*: fmap h g
```

To focus on an element in a pair, you either focus on the left and leave the right alone, or vice versa. Leibniz's famous product rule corresponds to a simple spatial intuition!

```
instance (Diff1 f, Diff1 g) => Diff1 (f :*: g) where
  type DF (f :*: g) = (DF f :*: g) :+: (f :*: DF g)
  upF (LF (f' :*: g) :<-: x) = upF (f' :<-: x) :*: g
  upF (RF (f :*: g') :<-: x) = f :*: upF (g' :<-: x)
```

Now, `downF` works similarly to the way it did for sums, except that we have to fix up the zipper context not only with a tag (to show which way we went) but also with the untouched other component.

```
  downF (f :*: g)
    =    fmap (\ (f' :<-: x) -> LF (f' :*: g) :<-: x) (downF f)
    :*:  fmap (\ (g' :<-: x) -> RF (f :*: g') :<-: x) (downF g)
```

But `aroundF` is a massive bag of laughs. Whichever side we are currently visiting, we have two choices:

1. Move `aroundF` on that side.
2. Move `upF` out of that side and `downF` into the other side.

Each case requires us to make use of the operations for the substructure, then fix up contexts.

```
  aroundF z@(LF (f' :*: g) :<-: (x :: x)) =
    LF (fmap (\ (f' :<-: x) -> LF (f' :*: g) :<-: x)
          (cxF $ aroundF (f' :<-: x :: ZF f x))
        :*: fmap (\ (g' :<-: x) -> RF (f :*: g') :<-: x) (downF g))
    :<-: z
    where f = upF (f' :<-: x)
  aroundF z@(RF (f :*: g') :<-: (x :: x)) =
    RF (fmap (\ (f' :<-: x) -> LF (f' :*: g) :<-: x) (downF f) :*:
        fmap (\ (g' :<-: x) -> RF (f :*: g') :<-: x)
          (cxF $ aroundF (g' :<-: x :: ZF g x)))
    :<-: z
    where g = upF (g' :<-: x)
```

Phew! The polynomials are all differentiable, and thus give us comonads.

Hmm. It's all a bit abstract. So I added `deriving Show` everywhere I could, and threw in

```
deriving instance (Show (DF f x), Show x) => Show (ZF f x)
```

which allowed the following interaction (tidied up by hand)

```
> downF (IF 1 :*: IF 2)
IF (LF (KF () :*: IF 2) :<-: 1) :*: IF (RF (IF 1 :*: KF ()) :<-: 2)

> fmap aroundF it
IF  (LF (KF () :*: IF (RF (IF 1 :*: KF ()) :<-: 2)) :<-: (LF (KF () :*: IF 2) :<-: 1))
:*:
IF  (RF (IF (LF (KF () :*: IF 2) :<-: 1) :*: KF ()) :<-: (RF (IF 1 :*: KF ()) :<-: 2))
```

Exercise Show that the composition of differentiable functors is differentiable, using the *chain rule*.

Sweet! Can we go home now? Of course not. We haven't differentiated any *recursive* structures yet.

Making recursive functors from bifunctors

A `Bifunctor`, as the existing literature on datatype generic programming (see work by Patrik Jansson and Johan Jeuring, or excellent lecture notes by Jeremy Gibbons) explains at length is a type constructor with two parameters, corresponding to two sorts of substructure. We should be able to "map" both.

```
class Bifunctor b where
  bimap :: (x -> x') -> (y -> y') -> b x y -> b x' y'
```

We can use `Bifunctor`s to give the node structure of recursive containers. Each node has *subnodes* and *elements*. These can just be the two sorts of substructure.

```
data Mu b y = In (b (Mu b y) y)
```

See? We "tie the recursive knot" in `b`'s first argument, and keep the parameter `y` in its second. Accordingly, we obtain once for all

```
instance Bifunctor b => Functor (Mu b) where
  fmap f (In b) = In (bimap (fmap f) f b)
```

To use this, we'll need a kit of `Bifunctor` instances.

The Bifunctor Kit

Constants are bifunctorial.

```
newtype K a x y = K a

instance Bifunctor (K a) where
  bimap f g (K a) = K a
```

You can tell I wrote this bit first, because the identifiers are shorter, but that's good because the code is longer.

Variables are bifunctorial.

We need the bifunctors corresponding to one parameter or the other, so I made a datatype to distinguish them, then defined a suitable GADT.

```
data Var = X | Y

data V :: Var -> * -> * -> * where
  XX :: x -> V X x y
  YY :: y -> V Y x y
```

That makes `V X x y` a copy of `x` and `V Y x y` a copy of `y`. Accordingly

```
instance Bifunctor (V v) where
  bimap f g (XX x) = XX (f x)
  bimap f g (YY y) = YY (g y)
```

Sums and **Products** of bifunctors are bifunctors

```
data (:++:) f g x y = L (f x y) | R (g x y) deriving Show

instance (Bifunctor b, Bifunctor c) => Bifunctor (b :++: c) where
  bimap f g (L b) = L (bimap f g b)
  bimap f g (R b) = R (bimap f g b)

data (:**:) f g x y = f x y :**: g x y deriving Show

instance (Bifunctor b, Bifunctor c) => Bifunctor (b :**: c) where
  bimap f g (b :**: c) = bimap f g b :**: bimap f g c
```

So far, so boilerplate, but now we can define things like

```
List = Mu (K () :++: (V Y :**: V X))

Bin = Mu (V Y :**: (K () :++: (V X :**: V X)))
```

If you want to use these types for actual data and not go blind in the pointilliste tradition of Georges Seurat, use *pattern synonyms*.

But what of zippers? How shall we show that `Mu b` is differentiable? We shall need to show that `b` is differentiable in *both* variables. Clang! It's time to learn about partial differentiation.

Partial derivatives of bifunctors

Because we have two variables, we shall need to be able to talk about them collectively sometimes and individually at other times. We shall need the singleton family:

```
data Vary :: Var -> * where
  VX :: Vary X
  VY :: Vary Y
```

Now we can say what it means for a Bifunctor to have partial derivatives at each variable, and give the corresponding notion of zipper.

```
class (Bifunctor b, Bifunctor (D b X), Bifunctor (D b Y)) => Diff2 b where
  type D b (v :: Var) :: * -> * -> *
  up      :: Vary v -> Z b v x y -> b x y
  down    :: b x y -> b (Z b X x y) (Z b Y x y)
  around  :: Vary v -> Z b v x y -> Z b v (Z b X x y) (Z b Y x y)

data Z b v x y = (:<-) {cxZ :: D b v x y, elZ :: V v x y}
```

This `D` operation needs to know which variable to target. The corresponding zipper `Z b v` tells us which variable `v` must be in focus. When we "decorate with context", we have to decorate `x`-elements with `X`-contexts and `y`-elements with `Y`-contexts. But otherwise, it's the same story.

We have two remaining tasks: firstly, to show that our bifunctor kit is differentiable; secondly, to show that `Diff2 b` allows us to establish `Diff1 (Mu b)`.

Differentiating the Bifunctor kit

I'm afraid this bit is fiddly rather than edifying. Feel free to skip along.

The constants are as before.

```
instance Diff2 (K a) where
  type D (K a) v = K Void
  up _ (K q :<- _) = absurd q
  down (K a) = K a
  around _ (K q :<- _) = absurd q
```

On this occasion, life is too short to develop the theory of the type level Kronecker-delta, so I just treated the variables separately.

```
instance Diff2 (V X) where
  type D (V X) X = K ()
  type D (V X) Y = K Void
  up VX (K () :<- XX x)  = XX x
  up VY (K q :<- _)      = absurd q
  down (XX x) = XX (K () :<- XX x)
  around VX z@(K () :<- XX x)  = K () :<- XX z
  around VY (K q :<- _)        = absurd q

instance Diff2 (V Y) where
```

```
  type D (V Y) X = K Void
  type D (V Y) Y = K ()
  up VX (K q :<- _)       = absurd q
  up VY (K () :<- YY y)   = YY y
  down (YY y) = YY (K () :<- YY y)
  around VX (K q :<- _)         = absurd q
  around VY z@(K () :<- YY y)   = K () :<- YY z
```

For the structural cases, I found it useful to introduce a helper allowing me to treat variables uniformly.

```
vV :: Vary v -> Z b v x y -> V v (Z b X x y) (Z b Y x y)
vV VX z = XX z
vV VY z = YY z
```

I then built gadgets to facilitate the kind of "retagging" we need for `down` and `around`. (Of course, I saw which gadgets I needed as I was working.)

```
zimap :: (Bifunctor c) => (forall v. Vary v -> D b v x y -> D b' v x y) ->
         c (Z b X x y) (Z b Y x y) -> c (Z b' X x y) (Z b' Y x y)
zimap f = bimap
  (\ (d :<- XX x) -> f VX d :<- XX x)
  (\ (d :<- YY y) -> f VY d :<- YY y)

dzimap :: (Bifunctor (D c X), Bifunctor (D c Y)) =>
          (forall v. Vary v -> D b v x y -> D b' v x y) ->
          Vary v -> Z c v (Z b X x y) (Z b Y x y) -> D c v (Z b' X x y) (Z b' Y x y)
dzimap f VX (d :<- _) = bimap
  (\ (d :<- XX x) -> f VX d :<- XX x)
  (\ (d :<- YY y) -> f VY d :<- YY y)
  d
dzimap f VY (d :<- _) = bimap
  (\ (d :<- XX x) -> f VX d :<- XX x)
  (\ (d :<- YY y) -> f VY d :<- YY y)
  d
```

And with that lot ready to go, we can grind out the details. Sums are easy.

```
instance (Diff2 b, Diff2 c) => Diff2 (b :++: c) where
  type D (b :++: c) v = D b v :++: D c v
  up v (L b' :<- vv) = L (up v (b' :<- vv))
  down (L b) = L (zimap (const L) (down b))
  down (R c) = R (zimap (const R) (down c))
  around v z@(L b' :<- vv :: Z (b :++: c) v x y)
    = L (dzimap (const L) v ba) :<- vV v z
    where ba = around v (b' :<- vv :: Z b v x y)
  around v z@(R c' :<- vv :: Z (b :++: c) v x y)
    = R (dzimap (const R) v ca) :<- vV v z
    where ca = around v (c' :<- vv :: Z c v x y)
```

Products are hard work, which is why I'm a mathematician rather than an engineer.

```
instance (Diff2 b, Diff2 c) => Diff2 (b :**: c) where
  type D (b :**: c) v = (D b v :**: c) :++: (b :**: D c v)
  up v (L (b' :**: c) :<- vv) = up v (b' :<- vv) :**: c
```

```
  up v (R (b :**: c') :<- vv) = b :**: up v (c' :<- vv)
  down (b :**: c) =
    zimap (const (L . (:**: c))) (down b) :**: zimap (const (R . (b :**:))) (down c)
  around v z@(L (b' :**: c) :<- vv :: Z (b :**: c) v x y)
    = L (dzimap (const (L . (:**: c))) v ba :**:
        zimap (const (R . (b :**:))) (down c))
      :<- vV v z where
      b = up v (b' :<- vv :: Z b v x y)
      ba = around v (b' :<- vv :: Z b v x y)
  around v z@(R (b :**: c') :<- vv :: Z (b :**: c) v x y)
    = R (zimap (const (L . (:**: c))) (down b):**:
        dzimap (const (R . (b :**:))) v ca)
      :<- vV v z where
      c = up v (c' :<- vv :: Z c v x y)
      ca = around v (c' :<- vv :: Z c v x y)
```

Conceptually, it's just as before, but with more bureaucracy. I built these using pre-type-hole technology, using `undefined` as a stub in places I wasn't ready to work, and introducing a deliberate type error in the one place (at any given time) where I wanted a useful hint from the typechecker. You too can have the typechecking as videogame experience, even in Haskell.

Subnode zippers for recursive containers

The partial derivative of `b` with respect to `X` tells us how to find a subnode one step inside a node, so we get the conventional notion of zipper.

```
data MuZpr b y = MuZpr
  {  aboveMu  :: [D b X (Mu b y) y]
  ,  hereMu   :: Mu b y
  }
```

We can zoom all the way up to the root by repeated plugging in `X` positions.

```
muUp :: Diff2 b => MuZpr b y -> Mu b y
muUp (MuZpr {aboveMu = [], hereMu = t}) = t
muUp (MuZpr {aboveMu = (dX : dXs), hereMu = t}) =
  muUp (MuZpr {aboveMu = dXs, hereMu = In (up VX (dX :<- XX t))})
```

But we need *element*-zippers.

Element-zippers for fixpoints of bifunctors

Each element is somewhere inside a node. That node is sitting under a stack of `X`-derivatives. But the position of the element in that node is given by a `Y`-derivative. We get

```
data MuCx b y = MuCx
  {  aboveY  :: [D b X (Mu b y) y]
  ,  belowY  :: D b Y (Mu b y) y
  }

instance Diff2 b => Functor (MuCx b) where
  fmap f (MuCx { aboveY = dXs, belowY = dY }) = MuCx
    {  aboveY  = map (bimap (fmap f) f) dXs
    ,  belowY  = bimap (fmap f) f dY
    }
```

Boldly, I claim

```
instance Diff2 b => Diff1 (Mu b) where
  type DF (Mu b) = MuCx b
```

but before I develop the operations, I'll need some bits and pieces.

I can trade data between functor-zippers and bifunctor-zippers as follows:

```
zAboveY :: ZF (Mu b) y -> [D b X (Mu b y) y]  -- the stack of `X`-derivatives above me
zAboveY (d :<-: y) = aboveY d

zZipY :: ZF (Mu b) y -> Z b Y (Mu b y) y      -- the `Y`-zipper where I am
zZipY (d :<-: y) = belowY d :<- YY y
```

That's enough to let me define:

```
  upF z  = muUp (MuZpr {aboveMu = zAboveY z, hereMu = In (up VY (zZipY z))})
```

That is, we go up by first reassembling the node where the element is, turning an element-zipper into a subnode-zipper, then zooming all the way out, as above.

Next, I say

```
  downF  = yOnDown []
```

to go down starting with the empty stack, and define the helper function which goes `down` repeatedly from below any stack:

```
yOnDown :: Diff2 b => [D b X (Mu b y) y] -> Mu b y -> Mu b (ZF (Mu b) y)
yOnDown dXs (In b) = In (contextualize dXs (down b))
```

Now, `down b` only takes us inside the node. The zippers we need must also carry the node's context. That's what `contextualise` does:

```
contextualize :: (Bifunctor c, Diff2 b) =>
  [D b X (Mu b y) y] ->
  c (Z b X (Mu b y) y) (Z b Y (Mu b y) y) ->
  c (Mu b (ZF (Mu b) y)) (ZF (Mu b) y)
contextualize dXs = bimap
  (\ (dX :<- XX t) -> yOnDown (dX : dXs) t)
  (\ (dY :<- YY y) -> MuCx {aboveY = dXs, belowY = dY} :<-: y)
```

For every `Y`-position, we must give an element-zipper, so it is good we know the whole context `dXs` back to the root, as well as the `dY` which describes how the element sits in its node. For every `X`-position, there is a further subtree to explore, so we grow the stack and keep going!

That leaves only the business of shifting focus. We might stay put, or go down from where we are, or go up, or go up and then down some other path. Here goes.

```
  aroundF z@(MuCx {aboveY = dXs, belowY = dY} :<-: _) = MuCx
    {  aboveY = yOnUp dXs (In (up VY (zZipY z)))
    ,  belowY = contextualize dXs (cxZ $ around VY (zZipY z))
    }  :<-: z
```

As ever, the existing element is replaced by its entire zipper. For the `belowY` part, we look where else we can go in the existing node: we will find either alternative element `Y`-positions or further `X`-subnodes to explore, so we `contextualise` them. For the `aboveY` part, we must work our way back up the stack of `X`-derivatives after reassembling the node we were visiting.

```
yOnUp :: Diff2 b => [D b X (Mu b y) y] -> Mu b y ->
         [D b X (Mu b (ZF (Mu b) y)) (ZF (Mu b) y)]
yOnUp [] t = []
yOnUp (dX : dXs) (t :: Mu b y)
  =  contextualize dXs (cxZ $ around VX (dX :<- XX t))
  :  yOnUp dXs (In (up VX (dX :<- XX t)))
```

At each step of the way, we can either turn somewhere else that's `around`, or keep going up.

And that's it! I haven't given a formal proof of the laws, but it looks to me as if the operations carefully maintain the context correctly as they crawl the structure.

What have we learned?

Differentiability induces notions of thing-in-its-context, inducing a comonadic structure where `extract` gives you the thing and `duplicate` explores the context looking for other things to contextualise. If we have the appropriate differential structure for nodes, we can develop differential structure for whole trees.

Oh, and treating each individual arity of type constructor separately is blatantly horrendous. The better way is to work with functors between indexed sets

```
f :: (i -> *) -> (o -> *)
```

where we make `o` different sorts of structure storing `i` different sorts of element. These are *closed* under the Jacobian construction

```
J f :: (i -> *) -> ((o, i) -> *)
```

where each of the resulting `(o, i)`-structures is a partial derivative, telling you how to make an `i`-element-hole in an `o`-structure. But that's dependently typed fun, for another time.

6.8 Traversable and zippers: necessity and sufficiency

Every Traversable functor is a container with finitely many positions for elements. In order to combine the effects for computations at each element, there must only be finitely many. So, for example, the infinite `Stream` functor is not Traversable, because there is no way to deliver a reliable function which pulls `Maybe` through. We'd need

```
sequence :: Stream (Maybe x) -> Maybe (Stream x)
```

but if you want to check that everything in the stream succeeds, you'll have a long wait.

Zippers correspond to the ability to identify a particular element position (which further gives rise to a connection with derivatives, but that's another story). To be able to plug an element back in its hole, you need an effective way to decide equality on positions. If you have only finitely many positions, that's bound to be true (in the absence of information-hiding). So being Traversable is certainly sufficient for having a Zipper.

It's not necessary, however. `Stream` has a perfectly sensible Zipper

```
type StreamContext x = ([x], Stream x)
type StreamZipper x = (StreamContext x, x)
```

which represents the context as a finite (ok, ok, add a bang or two) list before the selected element and an infinite stream after it.

The positions in an infinite `Stream` are natural numbers. Natural numbers have a decidable equality, but there are infintely many of them.

tl;dr finite implies countable, but not vice versa.

6.9 How to write this (funny filter) function idiomatically?

You can assemble this function from pieces which either are standard or should be. The accepted answer has the right clue about zippers. My answer about differentiation and comonads gives a general treatment of the relevant operations, but let me be specific here.

I define the type of "lists with one element hole" as follows:

```
data Bwd x = B0 | Bwd x :< x deriving Show
type HoleyList x = (Bwd x, [x])
```

Strictly speaking, I don't need to introduce backward lists to do that, but I get so easily confused if I have to reverse things in my head. (It so happens that `HoleyList` is the formal derivative of `[]`.)

I can now define what it is to be a list element in its context.

```
type InContext x = (HoleyList x, x)
```

The idea is that the second component of the pair belongs in between the backward list and the forward list. I can define the function which plugs the list back together (Called `upF` in the generic treatment.)

```
plug :: InContext x -> [x]
plug ((B0, xs), y)      = y : xs
plug ((xz :< x, xs), y) = plug ((xz, y : xs), x)
```

I can also define the function that gives all the ways to take a list apart (`downF`, generically).

```
selections :: [x] -> [InContext x]
selections = go B0 where
  go xz [] = []
  go xz (x : xs) = ((xz, xs), x) : go (xz :< x) xs
```

Note that

```
map snd  (selections xs) = xs
map plug (selections xs) = map (const xs) xs
```

And now we're good to follow Bartek's recipe.

```
selectModify :: (a -> Bool) -> (a -> a) -> [a] -> [[a]]
selectModify p f = map (plug . (id *** f)) . filter (p . snd) . selections
```

That is: filter the selections by the test, apply the function to the element in focus, then plug back together. If you have the zipper equipment lying about, it's a one-liner, and it should work for any differentiable functor, not just lists! Job done!

```
> selectModify ((1 ==) . (`mod` 2)) (2*) [1..10]
[[2,2,3,4,5,6,7,8,9,10]
,[1,2,6,4,5,6,7,8,9,10]
,[1,2,3,4,10,6,7,8,9,10]
,[1,2,3,4,5,6,14,8,9,10]
,[1,2,3,4,5,6,7,8,18,10]]
```

6.10 Computing a term of a list depending on all previous terms

Statutory Calculus Warning. The basic answer to this question involves specialising a standard recursion scheme. But I got a bit carried away pulling at the thread of it. Things take a more abstract turn as I seek to apply the same method to structures other than lists. I end up reaching for Isaac Newton and Ralph Fox, and in the process devise the *alopegmorphism*, which may be something new.

But anyway, something of the sort ought to exist. It looks like a special case of the *anamorphism* or "unfold". Let's start with what's called `unfoldr` in the library.

```
unfoldr :: (seed -> Maybe (value, seed)) -> seed -> [value]
```

It shows how to grow a list of values from a seed, repeatedly using a function called a *coalgebra*. At each step, the coalgebra says whether to stop with `[]` or to carry on by consing a value onto a list grown from a new seed.

```
unfoldr coalg s = case coalg s of
  Nothing       -> []
  Just (v, s')  -> v : unfoldr coalg s'
```

Here, the *seed* type can be whatever you like — whatever local state is appropriate to the unfolding process. One entirely sensible notion of seed is simply "the list so far", perhaps in reverse order, so that the most recently added elements are nearest.

```
growList :: ([value] -> Maybe value) -> [value]
growList g = unfoldr coalg B0 where
  coalg vz = case g vz of   -- I say "vz", not "vs" to remember it's reversed
    Nothing  -> Nothing
    Just v   -> Just (v, v : vz)
```

At each step, our `g` operation looks at the context of values we already have and decides whether to add another: if so, the new value becomes both the head of the list and the most recent value in the new context.

So, this `growList` hands you at each step your list of previous results, ready for `zipWith (*)`. The reversal is rather handy for the convolution, so perhaps we're looking at something like

```
ps = growList $ \ pz -> Just (sum (zipWith (*) sigmas pz) `div` (length pz + 1))
sigmas = [sigma j | j <- [1..]]
```

perhaps?

A recursion scheme? For lists, we have a special case of the anamorphism, where the seed is the context of what we've built so far, and once we've said how to build a bit more, we know how to grow the context by the same token. It's not hard to see how that works for lists. But how does it work for anamorphisms in general? **Here's where things get hairy.**

We build up possibly infinite values whose node shape is given by some functor `f` (whose parameter turns out to be "substructures" when we "tie the knot").

```
newtype Nu f = In (f (Nu f))
```

In an anamorphism, the coalgebra uses the seed to choose a shape for the outermost node, populated with seeds for the substructures. (Co)recursively, we map the anamorphism across, growing those seeds into substructures.

```
ana :: Functor f => (seed -> f seed) -> seed -> Nu f
ana coalg s = In (fmap (ana coalg) (coalg s))
```

Let's reconstruct `unfoldr` from `ana`. We can build lots of ordinary recursive structures from `Nu` and a few simple parts: the *polynomial Functor kit*.

```
newtype  K1 a      x  = K1 a                  -- constants (labels)
newtype  I         x  = I x                   -- substructure places
data     (f :+: g) x  = L1 (f x) | R1 (g x)   -- choice (like Either)
data     (f :*: g) x  = f x :*: g x           -- pairing (like (,))
```

with `Functor` instances

```
instance Functor (K1 a) where fmap f (K1 a) = K1 a
instance Functor I      where fmap f (I s) = I (f s)
instance (Functor f, Functor g) => Functor (f :+: g) where
  fmap h (L1 fs) = L1 (fmap h fs)
  fmap h (R1 gs) = R1 (fmap h gs)
instance (Functor f, Functor g) => Functor (f :*: g) where
  fmap h (fx :*: gx) = fmap h fx :*: fmap h gx
```

For lists of `value`, the node shape functor is

```
type ListF value = K1 () :+: (K1 value :*: I)
```

meaning "either a boring label (for nil) or the (cons) pair of a `value` label and a sublist". The type of a `ListF value` coalgebra becomes

```
seed -> (K1 () :+: (K1 value :*: I)) seed
```

which is isomorphic (by "evaluating" the polynomial `ListF value` at `seed`) to

```
seed -> Either () (value, seed)
```

which is but a hair's breadth from the

```
seed -> Maybe (value, seed)
```

that `unfoldr` expects. You can recover an ordinary list like so

```
list :: Nu (ListF a) -> [a]
list (In (L1 _))                = []
list (In (R1 (K1 a :*: I as)))  = a : list as
```

Now, how do we grow some general `Nu f`? A good start is to choose the *shape* for the outermost node. A value of type `f ()` gives just the shape of a node, with trivial stubs in the substructure positions. Indeed, to grow our trees, we basically need some way to choose the "next" node shape given some idea where we've got to and what we've done so far. We should expect

```
grow :: (..where I am in a Nu f under construction.. -> f ()) -> Nu f
```

Note that for growing lists, our step function returns a `ListF value ()`, which is isomorphic to `Maybe value`.

But how do we express where we are in a `Nu f` so far? We're going to be so-many-nodes-in from the root of the structure, so we should expect a stack of layers. Each layer should tell us (1) its shape, (2) which position we're currently at, and (3) the structures already built to the left of that position, but we expect still to have stubs in the positions at which we have not yet arrived. In other words, it's an example of the *dissection* structure from my 2008 POPL paper about Clowns and Jokers.

The dissection operator turns a functor `f` (seen as a container of elements) into a bifunctor `Diss f` with two different sorts of elements, those on the left (clowns) and those on the right (jokers) of a "cursor position" within an `f` structure. First, let's have the `Bifunctor` class and some instances.

```
class Bifunctor b where
  bimap :: (c -> c') -> (j -> j') -> b c j -> b c' j'

newtype K2 a       c j = K2 a
data    (f :++: g) c j = L2 (f c j) | R2 (g c j)
```

```
data    (f :**: g) c j = f c j :**: g c j
newtype Clowns f   c j = Clowns (f c)
newtype Jokers f   c j = Jokers (f j)

instance Bifunctor (K2 a) where
  bimap h k (K2 a) = K2 a
instance (Bifunctor f, Bifunctor g) => Bifunctor (f :++: g) where
  bimap h k (L2 fcj) = L2 (bimap h k fcj)
  bimap h k (R2 gcj) = R2 (bimap h k gcj)
instance (Bifunctor f, Bifunctor g) => Bifunctor (f :**: g) where
  bimap h k (fcj :**: gcj) = bimap h k fcj :**: bimap h k gcj
instance Functor f => Bifunctor (Clowns f) where
  bimap h k (Clowns fc) = Clowns (fmap h fc)
instance Functor f => Bifunctor (Jokers f) where
  bimap h k (Jokers fj) = Jokers (fmap k fj)
```

Note that `Clowns f` is the bifunctor which amounts to an `f` structure containing only clowns, whilst `Jokers f` has only jokers. If you feel bothered by the repetition of all the `Functor` paraphernalia to get the `Bifunctor` paraphernalis, you're right to be bothered: it gets less laborious if we abstract away the arity and work with functors between *indexed* sets, but that's a whole other story.

Let's define *dissection* as a class which associates a bifunctor with a functor.

```
class (Functor f, Bifunctor (Diss f)) => Dissectable f where
  type Diss f :: * -> * -> *
  rightward   ::  Either (f j) (Diss f c j, c) ->
                  Either (j, Diss f c j) (f c)
```

The type `Diss f c j` represents an `f`-structure with a "hole" or "cursor position" at one element position, and in the positions to the left of the hole we have "clowns" in `c`, and to the right we have "jokers" in `j`. (The terminology is lifted from the *Stealer's Wheel* song "Stuck in the Middle with You".)

The key operation in the class is the isomorphism `rightward` which tells us how to move one place to the right, starting from either

- left of a whole structure full of jokers, or
- a hole in the structure, together with a clown to put in the hole

and arriving at either

- a hole in the structure, together with the joker which came out of it, or
- right of a whole structure full of clowns.

Isaac Newton was fond of dissections, but he called them *divided differences* and defined them on real-valued functions to get the slope between two points on a curve, thus

```
divDiff f c j  =  (f c - f j) / (c - j)
```

and he used them to make best polynomial approximations to any old functions, and the like. Multiply up and multiply out

```
divDiff f c j * c - j * divDiff f c j  =  f c - f j
```

then get rid of the subtraction by adding to both sides

```
f j + divDiff f c j * c  =  f c + j * divDiff f c j
```

and you've got the `rightward` isomorphism.

We might build a bit more intuition for these things if we look at the instances, and then we can get back to our original problem.

A boring old constant has zero as its divided difference.

```
instance Dissectable (K1 a) where
  type Diss (K1 a) = K2 Void
  rightward (Left (K1 a)) = (Right (K1 a))
  rightward (Right (K2 v, _)) = absurd v
```

If we start to the left and go to the right, we jump over the whole structure, because there are no element positions. If we start in an element position, someone is lying!

The identity functor has just *one* position.

```
instance Dissectable I where
  type Diss I = K2 ()
  rightward (Left (I j))       = Left (j, K2 ())
  rightward (Right (K2 (), c)) = Right (I c)
```

If we start to the left, we arrive in the position and out pops the joker; push in a clown and we finish on the right.

For sums, the structure is inherited: we just have to get the detagging and retagging correct.

```
instance (Dissectable f, Dissectable g) => Dissectable (f :+: g) where
  type Diss (f :+: g) = Diss f :++: Diss g
  rightward x = case x of
      Left (L1 fj)      -> ll (rightward (Left fj))
      Right (L2 df, c)  -> ll (rightward (Right (df, c)))
      Left (R1 gj)      -> rr (rightward (Left gj))
      Right (R2 dg, c)  -> rr (rightward (Right (dg, c)))
    where
      ll (Left (j, df)) = Left (j, L2 df)
      ll (Right fc)     = Right (L1 fc)
      rr (Left (j, dg)) = Left (j, R2 dg)
      rr (Right gc)     = Right (R1 gc)
```

For products, we must be somewhere in a pair of structures: either we're on the left between clowns and jokers with the right structure all jokers, or the left structure is all clowns and we're on the right between clowns and jokers.

```
instance (Dissectable f, Dissectable g) => Dissectable (f :*: g) where
  type Diss (f :*: g) = (Diss f :**: Jokers g) :++: (Clowns f :**: Diss g)
  rightward x = case x of
      Left (fj :*: gj) -> ll (rightward (Left fj)) gj
      Right (L2 (df :**: Jokers gj), c) -> ll (rightward (Right (df, c))) gj
      Right (R2 (Clowns fc :**: dg), c) -> rr fc (rightward (Right (dg, c)))
    where
      ll (Left (j, df)) gj = Left (j, L2 (df :**: Jokers gj))
      ll (Right fc)     gj = rr fc (rightward (Left gj))  -- (!)
      rr fc (Left (j, dg)) = Left (j, R2 (Clowns fc :**: dg))
      rr fc (Right gc)     = Right (fc :*: gc)
```

The `rightward` logic ensures that we work our way through the left structure, then once we're done with it, we start work on the right. The line marked `(!)` is the key moment in the middle, where we emerge from the right of the left structure and then enter the left of the right structure.

Huet's notion of "left" and "right" cursor movements in data structures arise from dissectability (if you complete the `rightward` isomorphism with its `leftward` counterpart). The *derivative* of `f` is just the limit when the difference between clowns and jokers tend to zero, or for us, what you get when you have the same sort of stuff either side of the cursor.

Moreover, if you take clowns to be zero, you get

```
rightward :: Either (f x) (Diss f Void x, Void) -> Either (x, Diss f Void x) (f Void)
```

but we can remove the impossible input case to get

```
type Quotient f x = Diss f Void x
leftmost :: f x -> Either (x, Quotient f x) (f Void)
leftmost = rightward . Left
```

which tells us that every `f` structure either has a leftmost element or none at all, a result we learn in school as the "Remainder Theorem". The multivariate version of the `Quotient` operator is the "derivative" which Brzozowski applied to regular expressions.

But *our* special case is Fox's derivative (about which I learned from Dan Piponi):

```
type Fox f x = Diss f x ()
```

That's the type of `f`-structures with stubs to the right of a cursor. *Now* we can give the type of our general `grow` operator.

```
grow :: Dissectable f => ([Fox f (Nu f)] -> f ()) -> Nu f
```

Our "context" is a stack of layers, each of which has fully grown data to the left and stubs to the right. We can implement `grow` directly as follows:

```
grow g = go [] where
  go stk = In (walk (rightward (Left (g stk)))) where
    walk (Left ((), df)) = walk (rightward (Right (df, go (df : stk))))
    walk (Right fm)      = fm
```

As we arrive at each position, the joker we extract is just a stub, but its context tells us how to extend the stack in order to grow a substructure of the tree, which gives us the clown that we need to move right. Once we've filled in all the stubs with trees, we're done!

But here's the twist: `grow` is not so easy to express as an anamorphism. It's easy to give the "seed" for the leftmost child of each node, because we have only stubs to the right of us. But to give the next seed to the right, we need more than the leftmost seed — we need the tree that grows from it! The anamorphism pattern requires us to give all the seeds for substructures before growing any of them. Our `growList` is an anamorphism only because list nodes have *at most one* child.

So it's something new, after all, growing from nothing, but allowing later growth at a given layer to depend on earlier trees, with the Fox derivative capturing the idea of "stubs where we have yet to work". Perhaps we should call it an *alopegmorphism*, from the Greek $\alpha\lambda\omega\pi\eta\xi$ for "fox".

6.11 Reasonable Comonad implementations (for nonempty lists)

Nonempty lists arise as two distinct comonads by two standard constructions.

Firstly, the *cofree comonad* is given thus.

```
data Cofree f x = x :& f (Cofree f x)  -- every node is labelled with an x

instance Functor f => Functor (Cofree f) where
  fmap f (x :& fcx) = f x :& fmap (fmap f) fcx

instance Functor f => Comonad (Cofree f) where
  extract (x :& _) = x   -- get the label of the top node
  duplicate cx@(_ :& fcx) = cx :& fmap duplicate fcx
```

Nonempty lists can be given as

```
type Nellist1 = Cofree Maybe
```

and are thus automatically comonadic. That gives you the "tails" comonad.

Meanwhile, the decomposition of a structure as an "element zipper" induces comonadic structure. As I explained at great length,

Differentiability amounts to this bunch of operations on zippers (individual elements picked out of their context and put "in focus")

```
class (Functor f, Functor (DF f)) => Diff1 f where
  type DF f :: * -> *
  upF      ::  ZF f x  ->  f x           -- defocus
  downF    ::  f x     ->  f (ZF f x)    -- find all ways to focus
  aroundF  ::  ZF f x  ->  ZF f (ZF f x) -- find all ways to *re*focus

data ZF f x = (:<-:) {cxF :: DF f x, elF :: x}
```

so we get a functor and a comonad

```
instance Diff1 f => Functor (ZF f) where
  fmap f (df :<-: x) = fmap f df :<-: f x

instance Diff1 f => Comonad (ZF f) where
  extract    = elF
  duplicate  = aroundF
```

In principle, nonempty lists arise by this construction, too. The trouble is that the functor being differentiated is not so easy to express in Haskell, even though the derivative is sensible. Let's go nuts…

Nonempty lists amount to `ZF thingy x` where `DF thingy = []`. Can we integrate lists? Fooling about algebraically might give us a clue

```
[x] = Either () (x, [x]) = 1 + x * [x]
```

so as a power series, we get

```
[x] = Sum(n :: Nat). x^n
```

and we can integrate power series

```
Integral [x] dx = Sum(n :: Nat). x^(n+1)/(n+1)
```

which means we get some sort of arbitrary tuples of size (n+1), but we have to identify them up to some relation where the equivalence classes have size (n+1). One way to do that is to identify tuples up to rotation, so you don't know which of the (n+1) positions is "first".

That is, lists are the derivative of nonempty cycles. Think about a bunch of people at a round table playing cards (possibly solitaire). Rotate the table and you get the same bunch of people

playing cards. But once you designate the *dealer*, you fix the list of other players in order, clockwise starting left of the dealer.

Two standard constructions; two comonads for the same functor.

(In my comment earlier, I remarked about the possibility of multiple monads. It's a bit involved, but here's a starting point. Every monad `m` is also applicative, and the applicative laws make `m ()` a monoid. Correspondingly, every monoid structure for `m ()` at least gives a candidate for a monad structure on `m`. In the case of writer monads `(,) s`, we get that the candidates for monads are the monoids on `(s, ())` which are just the same as the monoids on `s`.)

Edit Nonempty lists are also *monadic* in at least two distinct ways.

I define the identity and pairing for functors, as follows.

```
newtype I          x = I x
data    (f :*: g) x = (:&:) {lll :: f x, rrr :: g x}
```

Now, I can introduce nonempty lists as follows, then define concatenation.

```
newtype Ne x = Ne ((I :*: []) x)

cat :: Ne x -> Ne x -> Ne x
cat (Ne (I x :&: xs)) (Ne (I y :&: ys)) = Ne (I x :&: (xs ++ y : ys))
```

These are monadic just the way possibly empty lists are:

```
instance Monad Ne where
  return x = Ne (I x :&: [])
  Ne (I x :&: xs) >>= k = foldl cat (k x) (map k xs)
```

However, `I` is a monad:

```
instance Monad I where
  return = I
  I a >>= k = k a
```

Moreover, monads are closed under pairing:

```
instance (Monad f, Monad g) => Monad (f :*: g) where
  return x = return x :&: return x
  (fa :&: ga) >>= k = (fa >>= (lll . k)) :&: (ga >>= (rrr . k))
```

So we could just have written

```
newtype Ne x = Ne ((I :*: []) x) deriving (Monad, Applicative, Functor)
```

but the `return` for that monad gives us double vision.

```
return x = Ne (I x :&: [x])
```

So there you are: nonempty lists are comonadic two ways, monadic two ways, applicative six ways,...

(Lots more to say about this, but I have to stop somewhere.)

6.12 Representable (or Naperian) Functors

> And Noah said unto the animals "Go forth and multiply!", but the snakes said "We cannot multiply, for we are adders.", so Noah took wood from the Ark and, shaping it, said "I am building you a table of logs.".

Representable functors are sometimes also called "Naperian" functors (it's Peter Hancock's term: Hank's a denizen of the same part of Edinburgh as John Napier, of logarithmic fame) because when `F x ~= T -> x`, and remembering that, combinatorially, `T -> x` is "`x` to the power `T`", we see that `T` is in some sense `Log F`.

The first thing to note is that `F () ~= T -> () ~= ()`. That tells us *there is only one shape*. Functors which offer us a choice of shape cannot be Naperian, because they don't give a uniform presentation of the positions for data. That means `[]` is not Naperian, because different-length lists have positions represented by different types. However, an infinite `Stream` has positions given by the natural numbers.

Correspondingly, given any two `F` structures, their shapes are bound to match, so they have a sensible `zip`, giving us the basis for an `Applicative F` instance.

Indeed, we have

```
          a  -> p x
====================
  (Log p, a) ->   x
```

making `p` a right adjoint, so `p` preserves all limits, hence unit and products in particular, making it a monoidal functor, not just a *lax* monoidal functor. That is, the alternative presentation of `Applicative` has operations which are isomorphisms.

```
unit  :: ()          ~= p ()
mult  :: (p x, p y) ~= p (x, y)
```

Let's have a type class for the things. I cook it a bit differently from the `Representable` class.

```
class Applicative p => Naperian p where
  type Log p
  logTable  :: p (Log p)
  project   :: p x -> Log p -> x
  tabulate  :: (Log p -> x) -> p x
  tabulate f = fmap f logTable
  -- LAW1: project logTable = id
  -- LAW2: project px <$> logTable = px
```

We have a type `Log f`, representing at least some of the positions inside an `f`; we have a `logTable`, storing in each position the representative of that position, acting like a 'map of an `f`' with placenames in each place; we have a `project` function extracting the data stored at a given position.

The first law tells us that the `logTable` is accurate for all the positions which are represented. The second law tells us that we have represented *all* the positions. We may deduce that

```
tabulate (project px)
  = {definition}
fmap (project px) logTable
  = {LAW2}
px
```

and that

```
project (tabulate f)
  = {definition}
project (fmap f logTable)
  = {free theorem for project}
f . project logTable
  = {LAW1}
f . id
  = {composition absorbs identity}
f
```

We could imagine a generic instance for `Applicative`

```
instance Naperian p => Applicative p where
  pure x    = fmap (pure x)                    logTable
  pf <$> px = fmap (project pf <*> project ps) logTable
```

which is as much as to say that `p` inherits its own K and S combinators from the usual K and S for functions.

Of course, we have

```
instance Naperian ((->) r) where
  type Log ((->) r) = r  -- log_x (x^r) = r
  logTable = id
  project = ($)
```

Now, all the limit-like constructions preserve Naperianity. `Log` maps limity things to colimity things: it *calculates* left adjoints.

We have the terminal object and products.

```
data K1       x = K1
instance Applicative K1 where
  pure x    = K1
  K1 <*> K1 = K1
instance Functor K1 where fmap = (<*>) . pure

instance Naperian K1 where
  type Log K1 = Void -- "log of 1 is 0"
  logTable = K1
  project K1 nonsense = absurd nonsense

data (p * q)  x = p x :*: q x
instance (Applicative p, Applicative q) => Applicative (p * q) where
  pure x = pure x :*: pure x
  (pf :*: qf) <*> (ps :*: qs) = (pf <*> ps) :*: (qf <*> qs)
instance (Functor p, Functor q) => Functor (p * q) where
  fmap f (px :*: qx) = fmap f px :*: fmap f qx

instance (Naperian p, Naperian q) => Naperian (p * q) where
  type Log (p * q) = Either (Log p) (Log q)  -- log (p * q) = log p + log q
  logTable = fmap Left logTable :*: fmap Right logTable
  project (px :*: qx) (Left i)  = project px i
  project (px :*: qx) (Right i) = project qx i
```

We have identity and composition.

```
data I        x = I x
instance Applicative I where
  pure x = I x
  I f <*> I s = I (f s)
instance Functor I where fmap = (<*>) . pure

instance Naperian I where
  type Log I = ()    -- log_x x = 1
  logTable = I ()
  project (I x) () = x

data (p << q) x = C (p (q x))
instance (Applicative p, Applicative q) => Applicative (p << q) where
  pure x = C (pure (pure x))
  C pqf <*> C pqs = C (pure (<*>) <*> pqf <*> pqs)
instance (Functor p, Functor q) => Functor (p << q) where
  fmap f (C pqx) = C (fmap (fmap f) pqx)

instance (Naperian p, Naperian q) => Naperian (p << q) where
  type Log (p << q) = (Log p, Log q)  -- log (q ^ log p) = log p * log q
  logTable = C (fmap (\ i -> fmap (i ,) logTable) logTable)
  project (C pqx) (i, j) = project (project pqx i) j
```

Naperian functors are closed under *greatest* fixpoints, with their logarithms being the corresponding *least* fixpoints. E.g., for streams, we have

```
log_x (Stream x)
  =
log_x (nu y. x * y)
  =
mu log_xy. log_x (x * y)
  =
mu log_xy. log_x x + log_x y
  =
mu log_xy. 1 + log_xy
  =
Nat
```

It's a bit fiddly to render that in Haskell without introducing Naperian *bifunctors* (which have two sets of positions for two sorts of things), or (better) Naperian functors on indexed types (which have indexed positions for indexed things). What's easy, though, and hopefully gives the idea, is the cofree comonad.

```
data{-codata-} CoFree p x = x :- p (CoFree p x)
  -- i.e., (I * (p << CoFree p)) x
instance Applicative p => Applicative (CoFree p) where
  pure x = x :- pure (pure x)
  (f :- pcf) <*> (s :- pcs) = f s :- (pure (<*>) <*> pcf <*> pcs)
instance Functor p => Functor (CoFree p) where
  fmap f (x :- pcx) = f x :- fmap (fmap f) pcx

instance Naperian p => Naperian (CoFree p) where
  type Log (CoFree p) = [Log p]  -- meaning finite lists only
  logTable = [] :- fmap (\ i -> fmap (i :) logTable) logTable
  project (x :- pcx) []       = x
  project (x :- pcx) (i : is) = project (project pcx i) is
```

We may take `Stream = CoFree I`, giving

```
Log Stream = [Log I] = [()] ~= Nat
```

Now, the derivative `D p` of a functor gives its type of one-hole context, telling us i) the shape of a `p`, ii) the position of the hole, iii) the data that are not in the hole. If `p` is Naperian, there is no choice of shape, so putting trivial data in the non-hole positions, we find that we just get the position of the hole.

```
D p () ~= Log p
```

More on that connection can be found in this answer of mine about tries.

Anyhow, Naperian is indeed a funny local Scottish name for Representable, which are the things for which you can build a table of logs: they are the constructions characterized entirely by projection, offering no choice of 'shape'.

6.13 Tries as Naperian Functors; Matching via their Derivatives

Edit: I remembered a very helpful fact about logarithms and derivatives which I discovered whilst disgustingly hung over on a friend's sofa. Sadly, that friend (the late great Kostas Tourlas) is no longer with us, but I commemorate him by being disgustingly hung over on a different friend's sofa.

Let's remind ourselves about tries. (Lots of my mates were working on these structures in the early noughties: Ralf Hinze, Thorsten Altenkirch and Peter Hancock spring instantly to mind in that regard.) What's really going on is that we're computing the exponential of a type `t`, remembering that `t -> x` is a way of writing `x ^ t`.

That is, we expect to equip a type `t` with a functor `Expo t` such that `Expo t x` represents `t -> x`. We should further expect `Expo t` to be applicative (zippily). **Edit:** Hancock calls such functors "Naperian", because they have logarithms, and they're applicative in the same way as functions, with `pure` being the K combinator and `<*>` being S. It is immediate that `Expo t ()` must be isomorphic with `()`, with `const (pure ())` and `const ()` doing the (not much) work.

```
class Applicative (Expo t) => EXPO t where
  type Expo t :: * -> *
  appl  :: Expo t x -> (t -> x)       -- trie lookup
  abst  :: (t -> x) -> Expo t x       -- trie construction
```

Another way of putting it is that `t` is the *logarithm* of `Expo t`.

(I nearly forgot: fans of calculus should check that `t` is isomorphic to ∂ `(Expo t) ()`. This isomorphism might actually be rather useful. **Edit:** it's extremely useful, and we shall add it to `EXPO` later.)

We'll need some functor kit stuff. The identity functor is zippiy applicative...

```
data I     ::                          (* -> *) where
  I   :: x -> I x
  deriving (Show, Eq, Functor, Foldable, Traversable)

instance Applicative I where
  pure x = I x
  I f <*> I s = I (f s)
```

...and its logarithm is the unit type

```
instance EXPO () where
  type Expo () = I
  appl (I x) () = x
  abst f        = I (f ())
```

Products of zippy applicatives are zippily applicative…

```
data (:*:) :: (* -> *) -> (* -> *) -> (* -> *) where
  (:*:) :: f x -> g x -> (f :*: g) x
  deriving (Show, Eq, Functor, Foldable, Traversable)

instance (Applicative p, Applicative q) => Applicative (p :*: q) where
  pure x = pure x :*: pure x
  (pf :*: qf) <*> (ps :*: qs) = (pf <*> ps) :*: (qf <*> qs)
```

…and their logarithms are sums.

```
instance (EXPO s, EXPO t) => EXPO (Either s t) where
  type Expo (Either s t) = Expo s :*: Expo t
  appl (sf :*: tf) (Left s)  = appl sf s
  appl (sf :*: tf) (Right t) = appl tf t
  abst f = abst (f . Left) :*: abst (f . Right)
```

Compositions of zippy applicatives are zippily applicative…

```
data (:<:) :: (* -> *) -> (* -> *) -> (* -> *) where
  C :: f (g x) -> (f :<: g) x
  deriving (Show, Eq, Functor, Foldable, Traversable)

instance (Applicative p, Applicative q) => Applicative (p :<: q) where
  pure x          = C (pure (pure x))
  C pqf <*> C pqs = C (pure (<*>) <*> pqf <*> pqs)
```

and their logarithms are products.

```
instance (EXPO s, EXPO t) => EXPO (s, t) where
  type Expo (s, t) = Expo s :<: Expo t
  appl (C stf) (s, t) = appl (appl stf s) t
  abst f = C (abst $ \ s -> abst $ \ t -> f (s, t))
```

If we switch on enough stuff, we may now write

```
newtype Tree    = Tree (Either () (Tree, Tree))
  deriving (Show, Eq)
pattern Leaf     = Tree (Left ())
pattern Node l r = Tree (Right (l, r))

newtype ExpoTree x = ExpoTree (Expo (Either () (Tree, Tree)) x)
  deriving (Show, Eq, Functor, Applicative)

instance EXPO Tree where
  type Expo Tree = ExpoTree
  appl (ExpoTree f) (Tree t) = appl f t
  abst f = ExpoTree (abst (f . Tree))
```

The `TreeMap a` type in the question, being

```
data TreeMap a
    = TreeMap {
        tm_leaf :: Maybe a,
        tm_node :: TreeMap (TreeMap a)
      }
```

is exactly `Expo Tree (Maybe a)`, with `lookupTreeMap` being `flip appl`.

Now, given that `Tree` and `Tree -> x` are rather different things, it strikes me as odd to want code to work "on both". The tree equality test is a special case of the lookup only in that the tree equality test is any old function which acts on a tree. There is a coincidence coincidence, however: to test equality, we must turn each tree into own self-recognizer. **Edit:** that's exactly what the log-diff iso does.

The structure which gives rise to an equality test is some notion of *matching*. Like this:

```
class Matching a b where
  type Matched a b :: *
  matched :: Matched a b -> (a, b)
  match   :: a -> b -> Maybe (Matched a b)
```

That is, we expect `Matched a b` to represent somehow a pair of an `a` and a `b` which match. We should be able to extract the pair (forgetting that they match), and we should be able to take any pair and try to match them.

Unsurprisingly, we can do this for the unit type, quite successfully.

```
instance Matching () () where
  type Matched () () = ()
  matched () = ((), ())
  match () () = Just ()
```

For products, we work componentwise, with component mismatch being the only danger.

```
instance (Matching s s', Matching t t') => Matching (s, t) (s', t') where
  type Matched (s, t) (s', t') = (Matched s s', Matched t t')
  matched (ss', tt') = ((s, t), (s', t')) where
    (s, s') = matched ss'
    (t, t') = matched tt'
  match (s, t) (s', t') = (,) <$> match s s' <*> match t t'
```

Sums offer some chance of mismatch.

```
instance (Matching s s', Matching t t') =>
    Matching (Either s t) (Either s' t') where
  type Matched (Either s t) (Either s' t')
    = Either (Matched s s') (Matched t t')
  matched (Left  ss') = (Left s,  Left s')  where (s, s') = matched ss'
  matched (Right tt') = (Right t, Right t') where (t, t') = matched tt'
  match (Left s)  (Left s')  = Left  <$> match s s'
  match (Right t) (Right t') = Right <$> match t t'
  match _         _          = Nothing
```

Amusingly, we can obtain an equality test for trees now as easily as

```
instance Matching Tree Tree where
  type Matched Tree Tree = Tree
  matched t = (t, t)
  match (Tree t1) (Tree t2) = Tree <$> match t1 t2
```

(Incidentally, the `Functor` subclass that captures a notion of matching, being

```
class HalfZippable f where  -- "half zip" comes from Roland Backhouse
  halfZip :: (f a, f b) -> Maybe (f (a, b))
```

is sadly neglected. Morally, for each such `f`, we should have

```
Matched (f a) (f b) = f (Matched a b)
```

A fun exercise is to show that if `(Traversable f, HalfZippable f)`, then the free monad on `f` has a first-order unification algorithm.)

I suppose we can build "singleton association lists" like this:

```
mapOne :: forall a. (Tree, a) -> Expo Tree (Maybe a)
mapOne (t, a) = abst f where
  f :: Tree -> Maybe a
  f u = pure a <* match t u
```

And we could try combining them with this gadget, exploiting the zippiness of all the `Expo`ts...

```
instance Monoid x => Monoid (ExpoTree x) where
  mempty = pure mempty
  mappend t u = mappend <$> t <*> u
```

...but, yet again, the utter stupidity of the `Monoid` instance for `Maybe x` continues to frustrate clean design.

We can at least manage

```
instance Alternative m => Alternative (ExpoTree :<: m) where
  empty = C (pure empty)
  C f <|> C g = C ((<|>) <$> f <*> g)
```

An amusing exercise is to fuse `abst` with `match`, and perhaps that's what the question is really driving at. Let's refactor `Matching`.

```
class EXPO b => Matching a b where
  type Matched a b :: *
  matched :: Matched a b -> (a, b)
  match'  :: a -> Proxy b -> Expo b (Maybe (Matched a b))

data Proxy x = Poxy  -- I'm not on GHC 8 yet, and Simon needs a hand here
```

For `()`, what's new is

```
instance Matching () () where
  -- skip old stuff
  match' () (Poxy :: Proxy ()) = I (Just ())
```

For sums, we need to tag successful matches, and fill in the unsuccessful parts with a magnificently Glaswegian `pure Nothing`.

```
instance (Matching s s', Matching t t') =>
    Matching (Either s t) (Either s' t') where
  -- skip old stuff
  match' (Left s) (Poxy :: Proxy (Either s' t')) =
    ((Left <$>) <$> match' s (Poxy :: Proxy s')) :*: pure Nothing
  match' (Right t) (Poxy :: Proxy (Either s' t')) =
    pure Nothing :*: ((Right <$>) <$> match' t (Poxy :: Proxy t'))
```

For pairs, we need to build matching in sequence, dropping out early if the first component fails.

```
instance (Matching s s', Matching t t') => Matching (s, t) (s', t') where
  -- skip old stuff
  match' (s, t) (Poxy :: Proxy (s', t'))
    = C (more <$> match' s (Poxy :: Proxy s')) where
    more Nothing  = pure Nothing
    more (Just s) = ((,) s <$>) <$> match' t (Poxy :: Proxy t')
```

So we can see that there is a connection between a constructor and the trie for its matcher.

Homework: fuse `abst` with `match'`, effectively tabulating the entire process.

Edit: writing `match'`, we parked each sub-matcher in the position of the trie corresponding to the sub-structure. And when you think of things in particular positions, you should think of zippers and differential calculus. Let me remind you.

We'll need functorial constants and coproducts to manage choice of "where the hole is".

```
data K     :: * ->                    (* -> *) where
  K :: a -> K a x
  deriving (Show, Eq, Functor, Foldable, Traversable)

data (:+:) :: (* -> *) -> (* -> *) -> (* -> *) where
  Inl :: f x -> (f :+: g) x
  Inr :: g x -> (f :+: g) x
  deriving (Show, Eq, Functor, Foldable, Traversable)
```

And now we may define

```
class (Functor f, Functor (D f)) => Differentiable f where
  type D f :: (* -> *)
  plug :: (D f :*: I) x -> f x
  -- there should be other methods, but plug will do for now
```

The usual laws of calculus apply, with composition giving a spatial interpretation to the *chain rule*.

```
instance Differentiable (K a) where
  type D (K a) = K Void
  plug (K bad :*: I x) = K (absurd bad)

instance Differentiable I where
  type D I = K ()
  plug (K () :*: I x) = I x

instance (Differentiable f, Differentiable g) => Differentiable (f :+: g) where
  type D (f :+: g) = D f :+: D g
  plug (Inl f' :*: I x) = Inl (plug (f' :*: I x))
  plug (Inr g' :*: I x) = Inr (plug (g' :*: I x))

instance (Differentiable f, Differentiable g) => Differentiable (f :*: g) where
  type D (f :*: g) = (D f :*: g) :+: (f :*: D g)
  plug (Inl (f' :*: g) :*: I x) = plug (f' :*: I x) :*: g
  plug (Inr (f :*: g') :*: I x) = f :*: plug (g' :*: I x)

instance (Differentiable f, Differentiable g) => Differentiable (f :<: g) where
  type D (f :<: g) = (D f :<: g) :*: D g
  plug ((C f'g :*: g') :*: I x) = C (plug (f'g :*: I (plug (g' :*: I x))))
```

It will not harm us to insist that `Expo t` is differentiable, so let us extend the `EXPO` class. What's a "trie with a hole"? It's a trie which is missing the output entry for exactly one of the possible inputs. And that's the key.

```
class (Differentiable (Expo t), Applicative (Expo t)) => EXPO t where
  type Expo t :: * -> *
  appl  :: Expo t x -> t -> x
  abst  :: (t -> x) -> Expo t x
  hole  :: t -> D (Expo t) ()
  eloh  :: D (Expo t) () -> t
```

Now, `hole` and `eloh` will witness the isomorphism.

```
instance EXPO () where
  type Expo () = I
  -- skip old stuff
  hole ()     = K ()
  eloh (K ()) = ()
```

The unit case wasn't very exciting, but the sum case begins to show structure:

```
instance (EXPO s, EXPO t) => EXPO (Either s t) where
  type Expo (Either s t) = Expo s :*: Expo t
  hole (Left s)  = Inl (hole s  :*: pure ())
  hole (Right t) = Inr (pure () :*: hole t)
  eloh (Inl (f' :*: _)) = Left (eloh f')
  eloh (Inr (_ :*: g')) = Right (eloh g')
```

See? A `Left` is mapped to a trie with a hole on the left; a `Right` is mapped to a trie with a hole on the right.

Now for products.

```
instance (EXPO s, EXPO t) => EXPO (s, t) where
  type Expo (s, t) = Expo s :<: Expo t
  hole (s, t) = C (const (pure ()) <$> hole s) :*: hole t
  eloh (C f' :*: g') = (eloh (const () <$> f'), eloh g')
```

A trie for a pair is a right trie stuffed inside a left trie, so the hole for a particular pair is found by making a hole for the right element in the particular subtrie for the left element.

For trees, we make another wrapper.

```
newtype DExpoTree x = DExpoTree (D (Expo (Either () (Tree, Tree))) x)
  deriving (Show, Eq, Functor)
```

So, how do we turn a tree into its trie recognizer? First, we grab its "everyone but me" trie, and we fill in all those outputs with `False`, then we plug in `True` for the missing entry.

```
matchMe :: EXPO t => t -> Expo t Bool
matchMe t = plug ((const False <$> hole t) :*: I True)
```

Homework hint: `D f :*: I` is a comonad.

Absent friends!

Chapter 7

Dependently Typed Haskell

7.1 Dependently typed language best suited to "real world" programming?

The accepted answer contains misinformation. Typechecking in Agda is decidable unless you switch off positivity/termination/universe checking. Moreover, infinite processes are programmable in Agda, just as IO processes are programmable in Haskell: the only restriction is that infinite processes cannot be unfolded indefinitely when being executed *in the course of typechecking*. You can implement a Turing Machine simulator in Agda: you just can't tell the lie that it's guaranteed to terminate or persuade the typechecker to run it in an unbounded way.

I do, however, agree that dependently typed languages are still at the experimental stage when it comes to "real world" programming. We can't yet support heavy duty development, but we can sustain a significant hobby amongst those with an eye to the future, rather like functional languages in the old days.

Idris, as suggested by Twey, is the closest candidate to a "real world" dependently typed language. It's much more focused on getting stuff done than Agda is. I'd recommend Agda as the better vehicle for getting to grips with the ideas behind dependently typed programming, but Idris is the more practical option.

It is, I'm pleased to say, worth considering recent releases of Haskell as a candidate in this discussion. Since GHC 7.4, Haskell has started supporting a useful notion of type level data, and with at least the *singleton* technique (a kludge though that is), we can really have types depending on run-time values (by making them depend on static variables constrained to equal run-time values). Haskell is thus a real "real world" language in the early phase of experimenting with dependent types.

7.2 Why not be dependently typed?

Dependently Typed Haskell, Now?

Haskell is, to a small extent, a dependently typed language. There is a notion of type-level data, now more sensibly typed thanks to `DataKinds`, and there is some means (`GADTs`) to give a run-time representation to type-level data. Hence, **values of run-time stuff effectively show up in types**, which is what it means for a language to be dependently typed.

Simple datatypes are *promoted* to the kind level, so that the values they contain can be used in types. Hence the archetypal example

```
data Nat = Z | S Nat

data Vec :: Nat -> * -> * where
```

```
  VNil  :: Vec Z x
  VCons :: x -> Vec n x -> Vec (S n) x
```

becomes possible, and with it, definitions such as

```
vApply :: Vec n (s -> t) -> Vec n s -> Vec n t
vApply VNil         VNil         = VNil
vApply (VCons f fs) (VCons s ss) = VCons (f s) (vApply fs ss)
```

which is nice. Note that the length `n` is a purely static thing in that function, ensuring that the input and output vectors have the same length, even though that length plays no role in the execution of `vApply`. By contrast, it's much trickier (i.e., impossible) to implement the function which makes `n` copies of a given `x` (which would be `pure` to `vApply`'s `<*>`)

```
vReplicate :: x -> Vec n x
```

because it's vital to know how many copies to make at run-time. Enter singletons.

```
data Natty :: Nat -> * where
  Zy :: Natty Z
  Sy :: Natty n -> Natty (S n)
```

For any promotable type, we can build the singleton family, indexed over the promoted type, inhabited by run-time duplicates of its values. `Natty n` is the type of run-time copies of the type-level `n :: Nat`. We can now write

```
vReplicate :: Natty n -> x -> Vec n x
vReplicate Zy     x = VNil
vReplicate (Sy n) x = VCons x (vReplicate n x)
```

So there you have a type-level value yoked to a run-time value: inspecting the run-time copy refines static knowledge of the type-level value. Even though terms and types are separated, we can work in a dependently typed way by using the singleton construction as a kind of epoxy resin, creating bonds between the phases. That's a long way from allowing arbitrary run-time expressions in types, but it ain't nothing.

What's Nasty? What's Missing?

Let's put a bit of pressure on this technology and see what starts wobbling. We might get the idea that singletons should be manageable a bit more implicitly

```
class Nattily (n :: Nat) where
  natty :: Natty n
instance Nattily Z where
  natty = Zy
instance Nattily n => Nattily (S n) where
  natty = Sy natty
```

allowing us to write, say,

```
instance Nattily n => Applicative (Vec n) where
  pure = vReplicate natty
  (<*>) = vApply
```

That works, but it now means that our original `Nat` type has spawned three copies: a kind, a singleton family and a singleton class. We have a rather clunky process for exchanging explicit `Natty n` values and `Nattily n` dictionaries. Moreover, `Natty` is not `Nat`: we have some sort of dependency on run-time values, but not at the type we first thought of. No fully dependently typed language makes dependent types this complicated!

Meanwhile, although `Nat` can be promoted, `Vec` cannot. You can't index by an indexed type. Full on dependently typed languages impose no such restriction, and in my career as a dependently typed show-off, I've learned to include examples of two-layer indexing in my talks, just to teach folks who've made one-layer indexing difficult-but-possible not to expect me to fold up like a house of cards. What's the problem? Equality. GADTs work by translating the constraints you achieve implicitly when you give a constructor a specific return type into explicit equational demands. Like this.

```
data Vec (n :: Nat) (x :: *)
  = n ~ Z => VNil
  | forall m. n ~ S m => VCons x (Vec m x)
```

In each of our two equations, both sides have kind `Nat`.

Now try the same translation for something indexed over vectors.

```
data InVec :: x -> Vec n x -> * where
  Here :: InVec z (VCons z zs)
  After :: InVec z ys -> InVec z (VCons y ys)
```

becomes

```
data InVec (a :: x) (as :: Vec n x)
  = forall m z (zs :: Vec x m). (n ~ S m, as ~ VCons z zs) => Here
  | forall m y z (ys :: Vec x m). (n ~ S m, as ~ VCons y ys) => After (InVec z ys)
```

and now we form equational constraints between `as :: Vec n x` and `VCons z zs :: Vec (S m) x` where the two sides have syntactically distinct (but provably equal) kinds. GHC core is not currently equipped for such a concept!

What else is missing? Well, **most of Haskell** is missing from the type level. The language of terms which you can promote has just variables and non-GADT constructors, really. Once you have those, the `type family` machinery allows you to write type-level programs: some of those might be quite like functions you would consider writing at the term level (e.g., equipping `Nat` with addition, so you can give a good type to append for `Vec`), but that's just a coincidence!

Another thing missing, in practice, is a *library* which makes use of our new abilities to index types by values. What do `Functor` and `Monad` become in this brave new world? I'm thinking about it, but there's a lot still to do.

Running Type-Level Programs

Haskell, like most dependently typed programming languages, has *two* operational semanticses. There's the way the run-time system runs programs (closed expressions only, after type erasure, highly optimised) and then there's the way the typechecker runs programs (your type families, your "type class Prolog", with open expressions). For Haskell, you don't normally mix the two up, because the programs being executed are in different languages. Dependently typed languages have separate run-time and static execution models for the *same* language of programs, but don't worry, the run-time model still lets you do type erasure and, indeed, proof erasure: that's what Coq's *extraction* mechanism gives you; that's at least what Edwin Brady's compiler does (although Edwin erases unnecessarily duplicated values, as well as types and proofs). The phase distinction may not be a distinction of *syntactic category* any longer, but it's alive and well.

Dependently typed languages, being total, allow the typechecker to run programs free from the fear of anything worse than a long wait. As Haskell becomes more dependently typed, we

face the question of what its static execution model should be? One approach might be to restrict static execution to total functions, which would allow us the same freedom to run, but might force us to make distinctions (at least for type-level code) between data and codata, so that we can tell whether to enforce termination or productivity. But that's not the only approach. We are free to choose a much weaker execution model which is reluctant to run programs, at the cost of making fewer equations come out just by computation. And in effect, that's what GHC actually does. The typing rules for GHC core make no mention of *running* programs, but only for checking evidence for equations. When translating to the core, GHC's constraint solver tries to run your type-level programs, generating a little silvery trail of evidence that a given expression equals its normal form. This evidence-generation method is a little unpredictable and inevitably incomplete: it fights shy of scary-looking recursion, for example, and that's probably wise. One thing we don't need to worry about is the execution of `IO` computations in the typechecker: remember that the typechecker doesn't have to give `launchMissiles` the same meaning that the run-time system does!

Hindley-Milner Culture

The Hindley-Milner type system achieves the truly awesome coincidence of four distinct distinctions, with the unfortunate cultural side-effect that many people cannot see the distinction between the distinctions and assume the coincidence is inevitable! What am I talking about?

- terms *vs* types
- explicitly written things *vs* implicitly written things
- presence at run-time *vs* erasure before run-time
- non-dependent abstraction *vs* dependent quantification

We're used to writing terms and leaving types to be inferred... and then erased. We're used to quantifying over type variables with the corresponding type abstraction and application happening silently and statically.

You don't have to veer too far from vanilla Hindley-Milner before these distinctions come out of alignment, and that's *no bad thing*. For a start, we can have more interesting types if we're willing to write them in a few places. Meanwhile, we don't have to write type class dictionaries when we use overloaded functions, but those dictionaries are certainly present (or inlined) at run-time. In dependently typed languages, we expect to erase more than just types at run-time, but (as with type classes) that some implicitly inferred values will not be erased. E.g., `vReplicate`'s numeric argument is often inferable from the type of the desired vector, but we still need to know it at run-time.

Which language design choices should we review because these coincidences no longer hold? E.g., is it right that Haskell provides no way to instantiate a `forall x. t` quantifier explicitly? If the typechecker can't guess `x` by unifiying `t`, we have no other way to say what `x` must be.

More broadly, we cannot treat "type inference" as a monolithic concept that we have either all or nothing of. For a start, we need to split off the "generalisation" aspect (Milner's "let" rule), which relies heavily on restricting which types exist to ensure that a stupid machine can guess one, from the "specialisation" aspect (Milner's "var" rule) which is as effective as your constraint solver. We can expect that top-level types will become harder to infer, but that internal type information will remain fairly easy to propagate.

Next Steps For Haskell

We're seeing the type and kind levels grow very similar (and they already share an internal representation in GHC). We might as well merge them. It would be fun to take `* :: *` if we can: we lost *logical* soundness long ago, when we allowed bottom, but *type* soundness is usually a weaker requirement. We must check. If we must have distinct type, kind, etc levels, we can at least make sure everything at the type level and above can always be promoted. It would

be great just to re-use the polymorphism we already have for types, rather than re-inventing polymorphism at the kind level.

We should simplify and generalise the current system of constraints by allowing *heterogeneous* equations `a ~ b` where the kinds of `a` and `b` are not syntactically identical (but can be proven equal). It's an old technique (in my thesis, last century) which makes dependency much easier to cope with. We'd be able to express constraints on expressions in GADTs, and thus relax restrictions on what can be promoted.

We should eliminate the need for the singleton construction by introducing a dependent function type, `pi x :: s -> t`. A function with such a type could be applied *explicitly* to any expression of type `s` which lives in the *intersection* of the type and term languages (so, variables, constructors, with more to come later). The corresponding lambda and application would not be erased at run-time, so we'd be able to write

```
vReplicate :: pi n :: Nat -> x -> Vec n x
vReplicate Z     x = VNil
vReplicate (S n) x = VCons x (vReplicate n x)
```

without replacing `Nat` by `Natty`. The domain of `pi` can be any promotable type, so if GADTs can be promoted, we can write dependent quantifier sequences (or "telescopes" as de Briuijn called them)

```
pi n :: Nat -> pi xs :: Vec n x -> ...
```

to whatever length we need.

The point of these steps is to *eliminate complexity* by working directly with more general tools, instead of making do with weak tools and clunky encodings. The current partial buy-in makes the benefits of Haskell's sort-of dependent types more expensive than they need to be.

Too Hard?

Dependent types make a lot of people nervous. They make me nervous, but I like being nervous, or at least I find it hard not to be nervous anyway. But it doesn't help that there's quite such a fog of ignorance around the topic. Some of that's due to the fact that we all still have a lot to learn. But proponents of less radical approaches have been known to stoke fear of dependent types without always making sure the facts are wholly with them. I won't name names. These "undecidable typechecking", "Turing incomplete", "no phase distinction", "no type erasure", "proofs everywhere", etc, myths persist, even though they're rubbish.

It's certainly not the case that dependently typed programs must always be proven correct. One can improve the basic hygiene of one's programs, enforcing additional invariants in types without going all the way to a full specification. Small steps in this direction quite often result in much stronger guarantees with few or no additional proof obligations. It is not true that dependently typed programs are inevitably *full* of proofs, indeed I usually take the presence of any proofs in my code as the cue to *question my definitions*.

For, as with any increase in articulacy, we become free to say foul new things as well as fair. E.g., there are plenty of crummy ways to define binary search trees, but that doesn't mean there isn't a good way. It's important not to presume that bad experiences cannot be bettered, even if it dents the ego to admit it. Design of dependent definitions is a new skill which takes learning, and being a Haskell programmer does not automatically make you an expert! And even if some programs are foul, why would you deny others the freedom to be fair?

Why Still Bother With Haskell?

I really enjoy dependent types, but most of my hacking projects are still in Haskell. Why? Haskell has type classes. Haskell has useful libraries. Haskell has a workable (although far from ideal)

treatment of programming with effects. Haskell has an industrial strength compiler. The dependently typed languages are at a much earlier stage in growing community and infrastructure, but we'll get there, with a real generational shift in what's possible, e.g., by way of metaprogramming and datatype generics. But you just have to look around at what people are doing as a result of Haskell's steps towards dependent types to see that there's a lot of benefit to be gained by pushing the present generation of languages forwards, too.

7.3 Simple dependent type example in Haskell for Dummies. How are they useful in practice in Haskell? Why should I care about dependent types ?

Late to the party, this answer is basically a shameless plug.

Sam Lindley and I wrote a paper about Hasochism, the pleasure and pain of dependently typed programming in Haskell. It gives plenty of examples of what's possible *now* in Haskell and draws points of comparison (favourable as well as not) with the Agda/Idris generation of dependently typed languages.

Although it is an academic paper, it is about actual programs, and you can grab the code from Sam's repo. We have lots of little examples (e.g. orderedness of mergesort output) but we end up with a text editor example, where we use indexing by width and height to manage screen geometry: we make sure that components are regular rectangles (vectors of vectors, not ragged lists of lists) and that they fit together exactly.

The key power of dependent types is to maintain consistency between separate data components (e.g., the head vector in a matrix and every vector in its tail must all have the same length). That's never more important than when writing conditional code. The situation (which will one day come to be seen as having been ridiculously naïve) is that the following are all type-preserving rewrites

- `if b then t else e => if b then e else t`
- `if b then t else e => t`
- `if b then t else e => e`

Although we are presumably testing `b` because it gives us some useful insight into what would be appropriate (or even safe) to do next, none of that insight is mediated via the type system: the idea that `b`'s truth justifies `t` and its falsity justifies `e` is missing, despite being critical.

Plain old Hindley-Milner does give us one means to ensure some consistency. Whenever we have a polymorphic function

```
f :: forall a. r[a] -> s[a] -> t[a]
```

we must instantiate `a` consistently: however the first argument fixes `a`, the second argument must play along, and we learn something useful about the result while we are at it. Allowing data at the type level is useful because some forms of consistency (e.g. lengths of things) are more readily expressed in terms of data (numbers).

But the real breakthrough is GADT pattern matching, where the type of a pattern can *refine* the type of the argument it matches. You have a vector of length `n`; you look to see whether it's nil or cons; now you know whether `n` is zero or not. This is a form of testing where the type of the code in each case is more specific than the type of the whole, because in each case something which has been *learned* is reflected at the type level. It is **learning by testing** which makes a language dependently typed, at least to some extent.

Here's a silly game to play, whatever typed language you use. Replace every type variable and every primitive type in your type expressions with 1 and evaluate types numerically (sum the sums, multiply the products, `s -> t` means `t`-to-the-`s`) and see what you get: if you get 0, you're a logician; if you get 1, you're a software engineer; if you get a power of 2, you're an

electronic engineer; if you get infinity, you're a programmer. What's going on in this game is a crude attempt to measure the information we're managing and the choices our code must make. Our usual type systems are good at managing the "software engineering" aspects of coding: unpacking and plugging together components. But as soon as a choice has been made, there is no way for types to observe it, and as soon as there are choices to make, there is no way for types to guide us: non-dependent type systems approximate all values in a given type as the same. That's a pretty serious limitation on their use in bug prevention.

7.4 Haskell singletons: What do we gain with SNat?

What do we gain? Hmm. The status of singletons is that of *awkward but currently necessary workaround*, and the sooner we can do away with them, the better.

Let me see if I can clarify the picture. We have a data type `Nat`:

```
data Nat = Zero | Suc Nat
```

(wars have been started over even more trivial issues than the number of 'c's in `Suc`)

The type `Nat` has run-time values which are indistinguishable at the type level. The Haskell type system currently has the *replacement* property, which means that in any well typed program, you may replace any well typed subexpression by an alternative subexpression with the same scope and type, and the program will continue to be well typed. For example, you can rewrite every occurrence of

```
if <b> then <t> else <e>
```

to

```
if <b> then <e> else <t>
```

and you can be sure that nothing will go wrong... with the outcome of checking your program's type.

The replacement property is an embarrassment. It's clear proof that your type system gives up at the very moment that meaning starts to matter.

Now, by being a data type for run-time values, `Nat` also becomes a type of type-level values `'Zero` and `'Suc`. The latter live only in Haskell's type language and have no run-time presence at all. Please note that although `'Zero` and `'Suc` exist at the type level, it is unhelpful to refer to them as "types" and the people who currently do that should desist. They do not have type `*` and can thus not *classify values* which is what types worthy of the name do.

There is no direct means of exchange between run-time and type-level `Nat`s, which can be a nuisance. The paradigmatic example concerns a key operation on *vectors*:

```
data Vec :: Nat -> * -> * where
  VNil   :: Vec 'Zero x
  VCons  :: x -> Vec n x -> Vec ('Suc n) x
```

We might like to compute a vector of copies of a given element (perhaps as part of an `Applicative` instance). It might look like a good idea to give the type

```
vec :: forall (n :: Nat) (x :: *). x -> Vec n x
```

but can that possibly work? In order to make `n` copies of something, we need to know `n` at run time: a program has to decide whether to deploy `VNil` and stop or to deploy `VCons` and keep going, and it needs some data to do that. A good clue is the `forall` quantifier, which is *parametric*: it indicates thats the quantified information is available only to types and is erased by run time.

Haskell currently enforces an entirely spurious coincidence between dependent quantification (what `forall` does) and erasure for run time. It does *not* support a dependent but not erased quantifier, which we often call `pi`. The type and implementation of `vec` should be something like

```
vec :: pi (n :: Nat) -> forall (x :: *). Vec n x
vec 'Zero    x = VNil
vec ('Suc n) x = VCons x (vec n x)
```

where arguments in `pi`-positions are written in the type language, but the data are available at run time.

So what do we do instead? We use singletons to capture indirectly what it means to be a *run-time copy of type-level data*.

```
data SNat :: Nat -> * where
  SZero :: SNat Zero
  SSuc  :: SNat n -> SNat (Suc n)
```

Now, `SZero` and `SSuc` make run-time data. `SNat` is not isomorphic to `Nat`: the former has type `Nat -> *`, while the latter has type `*`, so it is a type error to try to make them isomorphic. There are many run-time values in `Nat`, and the type system does not distinguish them; there is exactly one run-time value (worth speaking of) in each different `SNat n`, so the fact that the type system cannot distinguish them is beside the point. The point is that each `SNat n` is a different type for each different `n`, and that GADT pattern matching (where a pattern can be of a more specific instance of the GADT type it is known to be matching) can refine our knowledge of `n`.

We may now write

```
vec :: forall (n :: Nat). SNat n -> forall (x :: *). x -> Vec n x
vec SZero    x = VNil
vec (SSuc n) x = VCons x (vec n x)
```

Singletons allow us to bridge the gap between run time and type-level data, by exploiting the only form of run-time analysis that allows the refinement of type information. It's quite sensible to wonder if they're really necessary, and they presently are, only because that gap has not yet been eliminated.

7.5 Motivation for limitation on data kind promotion

An interesting thing happens if you promote types indexed by promoted types. Imagine we build

```
data Nat = Ze | Su Nat
```

and then

```
data Vec :: Nat -> * -> * where
  VNil   :: Vec Ze x
  VCons  :: x -> Vec n x -> Vec (Su n) x
```

Behind the scenes, the *internal* types of the constructors represent the instantiated return indices by constraints, as if we had written

```
data Vec (n :: Nat) (a :: *)
  =            n ~ Ze    => VNil
  | forall k.  n ~ Su k  => VCons a (Vec k a)
```

Now if we were allowed something like

```
data Elem :: forall n a. a -> Vec n a -> * where
  Top :: Elem x (VCons x xs)
  Pop :: Elem x xs -> Elem x (VCons y xs)
```

the translation to internal form would have to be something like

```
data Elem (x :: a) (zs :: Vec n a)
  = forall (k :: Nat), (xs :: Vec k a).            (n ~ Su k, zs ~ VCons x xs) =>
      Top
  | forall (k :: Nat), (xs :: Vec k s), (y :: a).  (n ~ Su k, zs ~ VCons y xs) =>
      Pop (Elem x xs)
```

but look at the second constraint in each case! We have

```
zs :: Vec n a
```

but

```
VCons x xs, VCons y xs :: Vec (Su k) a
```

But in System FC as then defined, equality constraints must have types of the same kind on both sides, so this example is not inconsiderably problematic.

One fix is use the evidence for the first constraint to fix up the second, but then we'd need dependent constraints

```
(q1 :: n ~ Su k, zs |> q1 ~ VCons x xs)
```

Another fix is just to allow heterogeneous equations, as I did in dependent type theory fifteen years ago. There will inevitably be equations between things whose kinds are equal in ways which are not syntactically obvious.

It's the latter plan that is currently favoured. As far as I understand, the policy you mention was adopted as a holding position, until the design for a core language with heterogeneous equality (as proposed by Weirich and colleagues) has matured to implementation. We live in interesting times.

7.6 What is an indexed monad?

As ever, the terminology people use is not entirely consistent. There's a variety of inspired-by-monads-but-strictly-speaking-isn't-quite notions. The term "indexed monad" is one of a number (including "monadish" and "parameterised monad" (Atkey's name for them)) of terms used to characterize one such notion. (Another such notion, if you're interested, is Katsumata's "parametric effect monad", indexed by a monoid, where return is indexed neutrally and bind accumulates in its index.)

First of all, let's check kinds.

```
IxMonad (m :: state -> state -> * -> *)
```

That is, the type of a "computation" (or "action", if you prefer, but I'll stick with "computation"), looks like

```
m before after value
```

where `before, after :: state` and `value :: *`. The idea is to capture the means to interact safely with an external system that has some *predictable* notion of state. A computation's type tells you what the state must be `before` it runs, what the state will be `after` it runs and (like with regular monads over `*`) what type of `value`s the computation produces.

The usual bits and pieces are `*`-wise like a monad and `state`-wise like playing dominoes.

```
ireturn  ::  a -> m i i a    -- returning a pure value preserves state
ibind    ::  m i j a ->      -- we can go from i to j and get an a, thence
             (a -> m j k b)  -- we can go from j to k and get a b, therefore
             -> m i k b      -- we can indeed go from i to k and get a b
```

The notion of "Kleisli arrow" (function which yields computation) thus generated is

```
a -> m i j b   -- values a in, b out; state transition i to j
```

and we get a composition

```
icomp :: IxMonad m => (b -> m j k c) -> (a -> m i j b) -> a -> m i k c
icomp f g = \ a -> ibind (g a) f
```

and, as ever, the laws exactly ensure that `ireturn` and `icomp` give us a category

```
      ireturn `icomp` g = g
      f `icomp` ireturn = f
(f `icomp` g) `icomp` h = f `icomp` (g `icomp` h)
```

or, in comedy fake C/Java/whatever,

```
      g(); skip = g()
      skip; f() = f()
{h(); g()}; f() = h(); {g(); f()}
```

Why bother? To model "rules" of interaction. For example, you can't eject a dvd if there isn't one in the drive, and you can't put a dvd into the drive if there's one already in it. So

```
data DVDDrive :: Bool -> Bool -> * -> * where  -- Bool is "drive full?"
  DReturn :: a -> DVDDrive i i a
  DInsert :: DVD ->                   -- you have a DVD
             DVDDrive True k a ->     -- you know how to continue full
             DVDDrive False k a       -- so you can insert from empty
  DEject  :: (DVD ->                  -- once you receive a DVD
              DVDDrive False k a) ->  -- you know how to continue empty
             DVDDrive True k a        -- so you can eject when full

instance IxMonad DVDDrive where  -- put these methods where they need to go
  ireturn = DReturn              -- so this goes somewhere else
  ibind (DReturn a)     k  = k a
  ibind (DInsert dvd j) k  = DInsert dvd (ibind j k)
  ibind (DEject j)      k  = DEject j $ \ dvd -> ibind (j dvd) k
```

With this in place, we can define the "primitive" commands

```
dInsert :: DVD -> DVDDrive False True ()
dInsert dvd = DInsert dvd $ DReturn ()

dEject :: DVDrive True False DVD
dEject = DEject $ \ dvd -> DReturn dvd
```

from which others are assembled with `ireturn` and `ibind`. Now, I can write (borrowing `do`-notation)

```
discSwap :: DVD -> DVDDrive True True DVD
discSwap dvd = do dvd' <- dEject; dInsert dvd ; ireturn dvd'
```

but not the physically impossible

```
discSwap :: DVD -> DVDDrive True True DVD
discSwap dvd = do dInsert dvd; dEject      -- ouch!
```

Alternatively, one can define one's primitive commands directly

```
data DVDCommand :: Bool -> Bool -> * -> * where
  InsertC  :: DVD -> DVDCommand False True ()
  EjectC   :: DVDCommand True False DVD
```

and then instantiate the generic template

```
data CommandIxMonad :: (state -> state -> * -> *) ->
                       state -> state -> * -> * where
  CReturn  :: a -> CommandIxMonad c i i a
  (:?)     :: c i j a -> (a -> CommandIxMonad c j k b) ->
              CommandIxMonad c i k b

instance IxMonad (CommandIxMonad c) where
  ireturn = CReturn
  ibind (CReturn a) k  = k a
  ibind (c :? j)    k  = c :? \ a -> ibind (j a) k
```

In effect, we've said what the primitive Kleisli arrows are (what one "domino" is), then built a suitable notion of "computation sequence" over them.

Note that for every indexed monad `m`, the "no change diagonal" `m i i` is a monad, but in general, `m i j` is not. Moreover, values are not indexed but computations are indexed, so an indexed monad is not just the usual idea of monad instantiated for some other category.

Now, look again at the type of a Kleisli arrow

```
a -> m i j b
```

We know we must be in state `i` to start, and we predict that any continuation will start from state `j`. We know a lot about this system! This isn't a risky operation! When we put the dvd in the drive, it goes in! The dvd drive doesn't get any say in what the state is after each command.

But that's not true in general, when interacting with the world. Sometimes you might need to give away some control and let the world do what it likes. For example, if you are a server, you might offer your client a choice, and your session state will depend on what they choose. The server's "offer choice" operation does not determine the resulting state, but the server should be able to carry on anyway. It's not a "primitive command" in the above sense, so indexed monads are not such a good tool to model the *unpredictable* scenario.

What's a better tool?

```
type f :-> g = forall state. f state -> g state

class MonadIx (m :: (state -> *) -> (state -> *)) where
  returnIx    :: x :-> m x
  flipBindIx  :: (a :-> m b) -> (m a :-> m b)  -- tidier than bindIx
```

Scary biscuits? Not really, for two reasons. One, it looks rather more like what a monad is, because it *is* a monad, but over `(state -> *)` rather than `*`. Two, if you look at the type of a Kleisli arrow,

```
a :-> m b   =   forall state. a state -> m b state
```

you get the type of computations with a *precondition* `a` and postcondition `b`, just like in Good Old Hoare Logic. Assertions in program logics have taken under half a century to cross the Curry-Howard correspondence and become Haskell types. The type of `returnIx` says "you can achieve any postcondition which holds, just by doing nothing", which is the Hoare Logic rule for "skip". The corresponding composition is the Hoare Logic rule for ";".

Let's finish by looking at the type of `bindIx`, putting all the quantifiers in.

```
bindIx :: forall i. m a i -> (forall j. a j -> m b j) -> m b i
```

These `forall`s have opposite polarity. We choose initial state `i`, and a computation which can start at `i`, with postcondition `a`. The world chooses any intermediate state `j` it likes, but it must give us the evidence that postcondition `b` holds, and from any such state, we can carry on to make `b` hold. So, in sequence, we can achieve condition `b` from state `i`. By releasing our grip on the "after" states, we can model *unpredictable* computations.

Both `IxMonad` and `MonadIx` are useful. Both model validity of interactive computations with respect to changing state, predictable and unpredictable, respectively. Predictability is valuable when you can get it, but unpredictability is sometimes a fact of life. Hopefully, then, this answer gives some indication of what indexed monads are, predicting both when they start to be useful and when they stop.

7.7 Fixpoints of functors on indexed sets

Quite right to construct a recursive functor by taking the fixpoint of a bifunctor, because 1 + 1 = 2. The list node structure is given as a container with 2 sorts of substructure: "elements" and "sublists".

It can be troubling that we need a whole other notion of `Functor` (which captures a rather specific variety of functor, despite its rather general name), to construct a `Functor` as a fixpoint. We can, however (as a bit of a stunt), shift to a slightly more general notion of functor which is *closed* under fixpoints.

```
type p -:> q = forall i. p i -> q i

class FunctorIx (f :: (i -> *) -> (o -> *)) where
  mapIx :: (p -:> q) -> f p -:> f q
```

These are the functors on *indexed sets*, so the names are not just gratuitous homages to Goscinny and Uderzo. You can think of `o` as "sorts of structure" and `i` as "sorts of substructure". Here's an example, based on the fact that 1 + 1 = 2.

```
data ListF :: (Either () () -> *) -> (() -> *) where
  Nil  :: ListF p '()
  Cons :: p (Left '()) -> p (Right '()) -> ListF p '()

instance FunctorIx ListF where
  mapIx f Nil        = Nil
  mapIx f (Cons a b) = Cons (f a) (f b)
```

To exploit the choice of substructure sort, we'll need a kind of type-level case analysis. We can't get away with a type function, as

1. we need it to be partially applied, and that's not allowed;
2. we need a bit at run time to tell us which sort is present.

```
data Case :: (i -> *) -> (j -> *) -> (Either i j -> *)  where
  CaseL :: p i -> Case p q (Left i)
  CaseR :: q j -> Case p q (Right j)

caseMap :: (p -:> p') -> (q -:> q') -> Case p q -:> Case p' q'
caseMap f g (CaseL p) = CaseL (f p)
caseMap f g (CaseR q) = CaseR (g q)
```

And now we can take the fixpoint:

```
data Mu :: ((Either i j -> *) -> (j -> *)) ->
           ((i -> *) -> (j -> *)) where
  In :: f (Case p (Mu f p)) j -> Mu f p j
```

In each substructure position, we do a case split to see whether we should have a `p`-element or a `Mu f p` substructure. And we get its functoriality.

```
instance FunctorIx f => FunctorIx (Mu f) where
  mapIx f (In fpr) = In (mapIx (caseMap f (mapIx f)) fpr)
```

To build lists from these things, we need to juggle between `*` and `() -> *`.

```
newtype K a i = K {unK :: a}

type List a = Mu ListF (K a) '()
pattern NilP :: List a
pattern NilP       = In Nil
pattern ConsP :: a -> List a -> List a
pattern ConsP a as = In (Cons (CaseL (K a)) (CaseR as))
```

Now, for lists, we get

```
map' :: (a -> b) -> List a -> List b
map' f = mapIx (K . f . unK)
```

7.8 Why is the type system refusing my seemingly valid program?

If your program *really* seemed valid to you, then you would be able to write the type of `get` that does the job you want in Haskell, not in handwave. Let me help you improve your handwave and uncover the reason you are asking for the moon on a stick.

> What I want to express is: `get :: (Convert a_contained_by_D b) => D -> b`, which seems impossible.

As stated, that's not quite as precise as you would need. Indeed, it's what Haskell gives you now, in that

```
get :: (Convert A b, Convert B b, Convert C b) => D -> b
```

any `a` which can be contained by `D` is required, one at a time, to be convertible to that `b`. And that's why you're getting classic sysadmin logic: no `D` is allowed to be gotten unless they all can `b`.

The problem is that you need to know the status not of the type which might be contained in *any* old `D`, but rather the type contained in the particular `D` that you receive as the input. Right? You want

```
print (get (DB B) :: A)  -- this to work
print (get (DC C) :: A)  -- this to fail
```

but `DB B` and `DC C` are just two different elements of `D`, and as far as the Haskell type system is concerned, within each type **everything different is the same**. If you want to discriminate between elements of `D`, then you need a `D`-pendent type. Here's how I'd write it in handwave.

```
DInner :: D -> *
DInner (DA a) = A
DInner (DB b) = B
DInner (DC c) = C

get :: forall x. pi (d :: D) -> (Convert (DInner d) x) => x
get (DA x) = convert x
get (DB x) = convert x
get (DC x) = convert x
```

where `pi` is the binding form for data which are passed at run time (unlike `forall`) but on which types may depend (unlike `->`). Now the constraint is talking not about arbitrary `D`s but the very `d :: D` in your hand, and the constraint can compute exactly what is needed by inspecting its `DInner`.

There is nothing you can say that will make it go away but my `pi`.

Sadly, whilst `pi` is rapidly descending from the sky, it has not yet landed. None the less, unlike the moon, it can be reached with a stick. No doubt you will complain that I am changing the setup, but really I am just translating your program from Haskell in approximately 2017 to Haskell in 2015. You'll `get` it back, one day, with the very type I handwaved.

There is nothing you can say, but you can *sing*.

Step 1. Switch on `DataKinds` and `KindSignatures` and build the singletons for your types (or get Richard Eisenberg to do it for you).

```
data A = A deriving Show
data Aey :: A -> * where  -- think of "-ey" as an adjectival suffix
  Aey :: Aey 'A           -- as in "tomatoey"

data B = B deriving Show
data Bey :: B -> * where
  Bey :: Bey 'B

data C = C deriving Show
data Cey :: C -> * where
  Cey :: Cey 'C

data D = DA A | DB B | DC C deriving Show
data Dey :: D -> * where
  DAey :: Aey a -> Dey (DA a)
  DBey :: Bey b -> Dey (DB b)
  DCey :: Cey c -> Dey (DC c)
```

The idea is (i) that datatypes become kinds, and (ii) that singletons characterize the type-level data which have a run time presentation. So type level `DA a` exists at run time provided `a` does, etc.

Step 2. Guess who's coming to `DInner`. Switch on `TypeFamilies`.

```
type family DInner (d :: D) :: * where
  DInner (DA a) = A
  DInner (DB b) = B
  DInner (DC c) = C
```

Step 3. Get you some `RankNTypes`, and now you can write

```
get :: forall x. forall d. Dey d -> (Convert (DInner d) x) => x
--                ^^^^^^^^^^^^^^^^^^
-- this is a plausible fake of pi (d :: D) ->
```

Step 4. Try to write `get` and screw up. We have to match on the run time evidence that the type level `d` is representable. We need that to get the type level `d` specialised in the computation of `DInner`. If we had proper `pi`, we could match on a `D` value that serves double duty, but for now, match on `Dey d` instead.

```
get (DAey x) = convert x   -- have x :: Aey a, need x :: A
get (DBey x) = convert x   -- and so on
get (DCey x) = convert x   -- and so forth
```

Maddeningly, our `x`es are now singletons, where, to `convert`, we need the underlying data. We need more of the singleton apparatus.

Step 5. Introduce and instantiate the singleton class, to "demote" type level values (as long as we know their run time representatives). Again, Richard Eisenberg's `singletons` library can Template-Haskell the boilerplate out of this, but let's see what's going on

```
class Sing (s :: k -> *) where   -- s is the singleton family for some k
  type Sung s :: *               -- Sung s is the type-level version of k
  sung :: s x -> Sung s          -- sung is the demotion function

instance Sing Aey where
  type Sung Aey = A
  sung Aey = A

instance Sing Bey where
  type Sung Bey = B
  sung Bey = B

instance Sing Cey where
  type Sung Cey = C
  sung Cey = C

instance Sing Dey where
  type Sung Dey = D
  sung (DAey aey) = DA (sung aey)
  sung (DBey bey) = DB (sung bey)
  sung (DCey cey) = DC (sung cey)
```

Step 6. Do it.

```
get :: forall x. forall d. Dey d -> (Convert (DInner d) x) => x
get (DAey x) = convert (sung x)
get (DBey x) = convert (sung x)
get (DCey x) = convert (sung x)
```

Be assured, when we have proper `pi`, those `DAey`s will be actual `DA`s and those `x`s will no longer need to be `sung`. My handwave type for `get` will be Haskell, and your code for `get` will be fine. But in the meantime

```
main = do
  print (get (DCey Cey) :: B)
  print (get (DBey Bey) :: A)
```

typechecks just fine. That's to say, your program (plus `DInner` and the correct type for `get`) seems like valid Dependent Haskell, and we're nearly there.

7.9 Is it possible to program and check invarants in Haskell?

The following is a stunt, but it's quite a safe stunt so do try it at home. It uses some of the entertaining new toys to bake *order* invariants into mergeSort.

```
{-# LANGUAGE GADTs, PolyKinds, KindSignatures, MultiParamTypeClasses,
    FlexibleInstances, RankNTypes, FlexibleContexts #-}
```

I'll have natural numbers, just to keep things simple.

```
data Nat = Z | S Nat deriving (Show, Eq, Ord)
```

But I'll define $<=$ in type class Prolog, so the typechecker can try to figure order out implicitly.

```
class LeN (m :: Nat) (n :: Nat) where
instance             LeN Z n where
instance LeN m n =>  LeN (S m) (S n) where
```

In order to sort numbers, I need to know that any two numbers can be ordered *one way or the other*. Let's say what it means for two numbers to be so orderable.

```
data OWOTO :: Nat -> Nat -> * where
  LE :: LeN x y => OWOTO x y
  GE :: LeN y x => OWOTO x y
```

We'd like to know that every two numbers are indeed orderable, provided we have a runtime representation of them. These days, we get that by building the *singleton family* for `Nat`. `Natty n` is the type of runtime copies of `n`.

```
data Natty :: Nat -> * where
  Zy :: Natty Z
  Sy :: Natty n -> Natty (S n)
```

Testing which way around the numbers are is quite a lot like the usual Boolean version, except with evidence. The step case requires unpacking and repacking because the types change. Instance inference is good for the logic involved.

```
owoto :: forall m n. Natty m -> Natty n -> OWOTO m n
owoto Zy      n       = LE
owoto (Sy m)  Zy      = GE
owoto (Sy m)  (Sy n)  = case owoto m n of
  LE -> LE
  GE -> GE
```

Now we know how to put numbers in order, let's see how to make ordered lists. The plan is to describe what it is to be in order *between loose bounds*. Of course, we don't want to exclude any elements from being sortable, so the type of *bounds* extends the element type with bottom and top elements.

```
data Bound x = Bot | Val x | Top deriving (Show, Eq, Ord)
```

I extend the notion of <= accordingly, so the typechecker can do bound checking.

```
class LeB (a :: Bound Nat)(b :: Bound Nat) where
instance             LeB Bot     b        where
instance LeN x y =>  LeB (Val x) (Val y)  where
instance             LeB (Val x) Top      where
instance             LeB Top     Top      where
```

And here are ordered lists of numbers: an `OList l u` is a sequence `x1 :< x2 :< ... :< xn :< ONil` such that `l <= x1 <= x2 <= ... <= xn <= u`. The `x :<` checks that `x` is above the lower bound, then imposes `x` as the lower bound on the tail.

```
data OList :: Bound Nat -> Bound Nat -> * where
  ONil :: LeB l u => OList l u
  (:<) :: forall l x u. LeB l (Val x) =>
          Natty x -> OList (Val x) u -> OList l u
```

We can write `merge` for ordered lists just the same way we would if they were ordinary. The key invariant is that if both lists share the same bounds, so does their merge.

```
merge :: OList l u -> OList l u -> OList l u
merge ONil      lu         = lu
merge lu        ONil       = lu
merge (x :< xu) (y :< yu)  = case owoto x y of
  LE  -> x :< merge xu (y :< yu)
  GE  -> y :< merge (x :< xu) yu
```

The branches of the case analysis extend what is already known from the inputs with just enough ordering information to satisfy the requirements for the results. Instance inference acts as a basic theorem prover: fortunately (or rather, with a bit of practice) the proof obligations are easy enough.

Let's seal the deal. We need to construct runtime witnesses for numbers in order to sort them this way.

```
data NATTY :: * where
  Nat :: Natty n -> NATTY

natty :: Nat -> NATTY
natty Z      =                            Nat Zy
natty (S n)  = case natty n of Nat n ->  Nat (Sy n)
```

We need to trust that this translation gives us the `NATTY` that corresponds to the `Nat` we want to sort. This interplay between `Nat`, `Natty` and `NATTY` is a bit frustrating, but that's what it takes in Haskell just now. Once we've got that, we can build `sort` in the usual divide-and-conquer way.

```
deal :: [x] -> ([x], [x])
deal []        = ([], [])
deal (x : xs)  = (x : zs, ys) where (ys, zs) = deal xs
```

```
sort :: [Nat] -> OList Bot Top
sort []   = ONil
sort [n]  = case natty n of Nat n -> n :< ONil
sort xs   = merge (sort ys) (sort zs) where (ys, zs) = deal xs
```

I'm often surprised by how many programs that make sense to us can make just as much sense to a typechecker.

[Here's some spare kit I built to see what was happening.

```
instance Show (Natty n) where
  show Zy = "Zy"
  show (Sy n) = "(Sy " ++ show n ++ ")"
instance Show (OList l u) where
  show ONil = "ONil"
  show (x :< xs) = show x ++ " :< " ++ show xs
ni :: Int -> Nat
ni 0 = Z
ni x = S (ni (x - 1))
```

And nothing was hidden.]

7.10 Why is typecase a bad thing?

It's really odd that people think pattern matching on types is bad. We get a lot of mileage out of pattern matching on data which *encode* types, whenever we do a universe construction. If you take the approach that Thorsten Altenkirch and I pioneered (and which my comrades and I began to engineer), the types do form a *closed* universe, so you don't even need to solve the (frankly worth solving) problem of computing with open datatypes to treat types as data. If we could pattern match on types directly, we wouldn't need a decoding function to map type codes to their meanings, which at worst reduces clutter, and at best reduces the need to prove and coerce by equational laws about the behaviour of the decoding function. I have every intention of building a no-middleman closed type theory this way. Of course, you need that level 0 types inhabit a level 1 datatype. That happens as a matter of course when you build an inductive-recursive universe hierarchy.

But what about parametricity, I hear you ask?

Firstly, I don't want parametricity when I'm trying to write type-generic code. Don't force parametricity on me.

Secondly, why should types be the only things we're parametric in? Why shouldn't we *sometimes* be parametric in other stuff, e.g., perfectly ordinary type indices which inhabit datatypes but which we'd prefer not to have at run time? It's a real nuisance that quantities which play a part only in *specification* are, just because of their type, forced to be present.

The type of a domain has *nothing whatsoever* to do with whether quantification over it should be parametric.

Let's have (e.g. as proposed by Bernardy and friends) a discipline where both parametric/erasable and non-parametric/matchable quantification are distinct and both available. Then types can be data and we can still say what we mean.

7.11 Why can't I pattern match on a type family?

When you declare

```
leftthing :: a -> Leftthing a
```

you are saying that the *caller* of `leftthing` gets to choose what `a` is.

When you then write

```
leftthing (Twothings a b) = leftthing a
```

you are *presuming* that they have chosen a `Twothings` type, and as that is not necessarily the case, your program is rejected.

You may have thought that you were *testing whether* they had chosen a `Twothings` type, but no! Type information is erased before run time, so there is no way to make such a test.

You *can* try to restore the necessary run time information. First let me fix the inconsistency between your `Leftthing` and `leftthing`.

```
type family Leftthing a where
  Leftthing (Twothings a b) = Leftthing{-you forgot the recursion!-} a
  Leftthing a = a
```

Now we can define the GADT of witnesses to `Twothing`ness.

```
data IsItTwothings :: * -> * where
  YesItIs   :: IsItTwothings a -> IsItTwothings (Twothings a b)
  NoItIsn't :: Leftthing a ~ a => IsItTwothings a
            -- ^^^^^^^^^^^^^^^ this constraint will hold for any type
            -- which is *definitely not* a Twothings type
```

And then we can pass the witness as an argument:

```
leftthing :: IsItTwothings a -> a -> Leftthing a
leftthing (YesItIs r) (Twothings a b) = leftthing r a
leftthing NoItIsn't   b               = b
```

In effect, the witness is the unary encoding of the number of left-nested `Twothings`es at the root of your type. That's enough information to determine at run time the correct amount of unpacking to do.

```
> leftthing (YesItIs (YesItIs NoItIsn't)) (Twothings (Twothings True 11) (Twothings "sti
True
```

To sum up, you can't find out a type by pattern matching on a value. Rather, you need to know the type to do pattern matching (because the type determines the memory layout, and there are no run time type tags). You can't pattern match on types directly (because they're just not there to be matched on). You can construct data types which act as run time evidence of type structure and match on those instead.

Perhaps, one day, your program will work if you give it the type

```
leftthing :: pi a. a -> Leftthing a
```

where `pi` is the dependent quantifier, indicating that the hidden type argument is not erased, but rather passed and matched on at run time. That day has not yet come, but I think it will.

7.12 Positive integer type

I would be failing in my duty as his supervisor if I failed to plug Adam Gundry's Inch preprocessor, which manages integer constraints for Haskell.

Smart constructors and abstraction barriers are all very well, but they push too much testing to run time and don't allow for the possibility that you might actually know what you're doing in a way that checks out statically, with no need for `Maybe` padding. (A pedant writes. The author

of another answer appears to suggest that 0 is positive, which some might consider contentious. Of course, the truth is that we have uses for a variety of lower bounds, 0 and 1 both occurring often. We also have some use for upper bounds.)

In the tradition of Xi's DML, Adam's preprocessor adds an extra layer of precision on top of what Haskell natively offers but the resulting code erases to Haskell as is. It would be great if what he's done could be better integrated with GHC, in coordination with the work on type level natural numbers that Iavor Diatchki has been doing. We're keen to figure out what's possible.

To return to the general point, Haskell is currently not sufficiently dependently typed to allow the construction of subtypes by comprehension (e.g., elements of Integer greater than 0), but you can often refactor the types to a more indexed version which admits static constraint. Currently, the *singleton* type construction is the cleanest of the available unpleasant ways to achieve this. You'd need a *kind* of "static" integers, then inhabitants of kind `Integer -> *` capture properties of particular integers such as "having a dynamic representation" (that's the singleton construction, giving each static thing a unique dynamic counterpart) but also more specific things like "being positive".

Inch represents an imagining of what it would be like if you didn't need to bother with the singleton construction in order to work with some reasonably well behaved subsets of the integers. Dependently typed programming is often possible in Haskell, but is currently more complicated than necessary. The appropriate sentiment toward this situation is embarrassment, and I for one feel it most keenly.

7.13 Test if a value matches a constructor

Tags of tagged unions ought to be first-class values, and with a wee bit of effort, they are.

Jiggery-pokery alert:

```
{-# LANGUAGE GADTs, DataKinds, KindSignatures,
    TypeFamilies, PolyKinds, FlexibleInstances,
    PatternSynonyms
#-}
```

Step one: define type-level versions of the tags.

```
data TagType = EmptyTag | SingleTag | PairTag | LotsTag
```

Step two: define value-level witnesses for the representability of the type-level tags. Richard Eisenberg's Singletons library will do this for you. I mean something like this:

```
data Tag :: TagType -> * where
  EmptyT   :: Tag EmptyTag
  SingleT  :: Tag SingleTag
  PairT    :: Tag PairTag
  LotsT    :: Tag LotsTag
```

And now we can say what stuff we expect to find associated with a given tag.

```
type family Stuff (t :: TagType) :: * where
  Stuff EmptyTag   = ()
  Stuff SingleTag  = Int
  Stuff PairTag    = (Int, Int)
  Stuff LotsTag    = [Int]
```

So we can refactor the type you first thought of

```
data NumCol :: * where
  (:&) :: Tag t -> Stuff t -> NumCol
```

and use `PatternSynonyms` to recover the behaviour you had in mind:

```
pattern Empty        = EmptyT   :&  ()
pattern Single  i    = SingleT  :&  i
pattern Pair    i j  = PairT    :&  (i, j)
pattern Lots    is   = LotsT    :&  is
```

So what's happened is that each constructor for `NumCol` has turned into a tag indexed by the kind of tag it's for. That is, constructor tags now live separately from the rest of the data, synchronized by a common index which ensures that the stuff associated with a tag matches the tag itself.

But we can talk about tags alone.

```
data Ex :: (k -> *) -> * where  -- wish I could say newtype here
  Witness :: p x -> Ex p
```

Now, `Ex Tag`, is the type of "runtime tags with a type level counterpart". It has an `Eq` instance

```
instance Eq (Ex Tag) where
  Witness EmptyT   ==  Witness EmptyT   = True
  Witness SingleT  ==  Witness SingleT  = True
  Witness PairT    ==  Witness PairT    = True
  Witness LotsT    ==  Witness LotsT    = True
  _                ==  _                = False
```

Moreover, we can easily extract the tag of a `NumCol`.

```
numColTag :: NumCol -> Ex Tag
numColTag (n :& _) = Witness n
```

And that allows us to match your specification.

```
filter ((Witness PairT ==) . numColTag) :: [NumCol] -> [NumCol]
```

Which raises the question of whether your specification is actually what you need. The point is that detecting a tag entitles you an expectation of that tag's stuff. The output type `[NumCol]` doesn't do justice to the fact that you know you have just the pairs.

How might you tighten the type of your function and still deliver it?

7.14 Difference between Haskell and Idris: Reflection of Runtime/Compiletime in the type universes

Yes, you're right to observe that the types versus values distinction in Idris does not align with the compiletime-only versus runtime-and-compiletime distinction. That's a good thing. It is useful to have values which exist only at compiletime, just as in program logics we have "ghost variables" used only in specifications. It is useful also to have type representations at runtime, allowing datatype generic programming.

In Haskell, `DataKinds` (and `PolyKinds`) let us write

```
type family Cond (b :: Bool)(t :: k)(e :: k) :: k where
  Cond 'True  t e = t
  Cond 'False t e = e
```

and in the not too distant future, we shall be able to write

```
item :: pi (b :: Bool) -> Cond b Int [Int]
item True  = 42
item False = [1,2,3]
```

but until that technology is implemented, we have to make do with singleton forgeries of dependent function types, like this:

```
data Booly :: Bool -> * where
  Truey  :: Booly 'True
  Falsey :: Booly 'False

item :: forall b. Booly b -> Cond b Int [Int]
item Truey  = 42
item Falsey = [1,2,3]
```

You can get quite far with such fakery, but it would all get a lot easier if we just had the real thing.

Crucially, the plan for Haskell is to maintain and separate `forall` and `pi`, supporting parametric and ad hoc polymorphism, respectively. The lambdas and applications that go with `forall` can still be erased in runtime code generation, just as now, but those for `pi` are retained. It would also make sense to have runtime type abstractions `pi x :: * -> ...` and throw the rats' nest of complexity that is `Data.Typeable` into the dustbin.

7.15 Why GADT/existential data constructors cannot be used in lazy patterns?

Consider

```
data EQ a b where
  Refl :: EQ a a
```

If we could define

```
transport :: Eq a b -> a -> b
transport ~Refl a = a
```

then we could have

```
transport undefined :: a -> b
```

and thus acquire

```
transport undefined True = True :: Int -> Int
```

and then a crash, rather than the more graceful failure you get when trying to head-normalise the `undefined`.

GADT data represent evidence *about* types, so bogus GADT data threaten type safety. It is necessary to be strict with them to validate that evidence: you can't trust unevaluated computations in the presence of bottom.

7.16 Can GADTs be used to prove type inequalities in GHC?

Here's a shorter version of Philip JF's solution, which is the way dependent type theorists have been refuting equations for years.

```
type family Discriminate x
type instance Discriminate Int  = ()
type instance Discriminate Char = Void

transport :: Equal a b -> Discriminate a -> Discriminate b
transport Refl d = d

refute :: Equal Int Char -> Void
refute q = transport q ()
```

In order to show that things are different, you have to catch them *behaving differently* by providing a computational context which results in distinct observations. `Discriminate` provides exactly such a context: a type-level program which treats the two types differently.

It is not necessary to resort to `undefined` to solve this problem. Total programming sometimes involves rejecting impossible inputs. Even where `undefined` is available, I would recommend not using it where a total method suffices: the total method *explains* why something is impossible and the typechecker confirms; `undefined` merely documents *your promise*. Indeed, this method of refutation is how Epigram dispenses with "impossible cases" whilst ensuring that a case analysis covers its domain.

As for computational behaviour, note that `refute`, via `transport` is necessarily strict in `q` and that `q` cannot compute to head normal form in the empty context, simply because no such head normal form exists (and because computation preserves type, of course). In a total setting, we'd be sure that `refute` would never be invoked at run time. In Haskell, we're at least certain that its argument will diverge or throw an exception before we're obliged to respond to it. A *lazy* version, such as

```
absurdEquality e = error "you have a type error likely to cause big problems"
```

will ignore the toxicity of `e` and tell you that you have a type error when you don't. I prefer

```
absurdEquality e = e `seq` error "sue me if this happens"
```

if the honest refutation is too much like hard work.

7.17 Implementing a zipper for length-indexed lists

Alignment

Dependently typed programming is like doing two jigsaws which some rogue has glued together. Less metaphorically, we express simultaneous computations at the value level and at the type level, and we must ensure their compatibility. Of course, we are each our own rogue, so if we can arrange for the jigsaws to be glued in alignment, we shall have an easier time of it. When you see proof obligations for type repair, you might be tempted to ask

> Do I need to add some sort of proof objects (`data Refl a b where Refl :: Refl a a` et al.) or is there some way to make this work with just adding more explicit type signatures?

But you might first consider in what way the value- and type-level computations are out of alignment, and whether there is any hope to bring them closer.

A Solution

The question here is how to compute the vector (length-indexed list) of selections from a vector. So we'd like something with type

```
List (Succ n) a -> List (Succ n) (a, List n a)
```

where the element in each input position gets decorated with the one-shorter vector of its siblings. The proposed method is to scan left-to-right, accumulating the elder siblings in a list which grows on the right, then concatenate with the younger siblings at each position. Growing lists on the right is always a worry, especially when the `Succ` for the length is aligned to the `Cons` on the left. The need for concatenation necessitates type-level addition, but the arithmetic resulting from right-ended activity is out of alignment with the computation rules for addition. I'll get back to this style in a bit, but let's try thinking it out again.

Before we get into any accumulator-based solution, let's just try bog standard structural recursion. We have the "one" case and the "more" case.

```
picks (Cons x xs@Nil)         = Cons (x, xs) Nil
picks (Cons x xs@(Cons _ _))  = Cons (x, xs) (undefined (picks xs))
```

In both cases, we put the first decomposition at the front. In the second case, we have checked that the tail is nonempty, so we can ask for its selections. We have

```
x         :: a
xs        :: List (Succ n) a
picks xs  :: List (Succ n) (a, List n a)
```

and we want

```
Cons (x, xs) (undefined (picks xs))  :: List (Succ (Succ n)) (a, List (Succ n) a)
              undefined (picks xs)   :: List (Succ n) (a, List (Succ n) a)
```

so the `undefined` needs to be a function which grows all the sibling lists by reattaching `x` at the left end (and left-endedness is good). So, I define the `Functor` instance for `List n`

```
instance Functor (List n) where
  fmap f Nil          = Nil
  fmap f (Cons x xs)  = Cons (f x) (fmap f xs)
```

and I curse the `Prelude` and

```
import Control.Arrow((***))
```

so that I can write

```
picks (Cons x xs@Nil)         = Cons (x, xs) Nil
picks (Cons x xs@(Cons _ _))  = Cons (x, xs) (fmap (id *** Cons x) (picks xs))
```

which does the job with not a hint of addition, let alone a proof about it.

Variations

I got annoyed about doing the same thing in both lines, so I tried to wriggle out of it:

```
picks :: m ~ Succ n => List m a -> List m (a, List n a)  -- DOESN'T TYPECHECK
picks Nil          = Nil
picks (Cons x xs)  = Cons (x, xs) (fmap (id *** (Cons x)) (picks xs))
```

But GHC solves the constraint aggressively and refuses to allow `Nil` as a pattern. And it's correct to do so: we really shouldn't be computing in a situation where we know statically that `Zero ~ Succ n`, as we can easily construct some segfaulting thing. The trouble is just that I put my constraint in a place with too global a scope.

Instead, I can declare a wrapper for the result type.

```
data Pick :: Nat -> * -> * where
  Pick :: {unpick :: (a, List n a)} -> Pick (Succ n) a
```

The `Succ n` return index means the nonemptiness constraint is *local* to a `Pick`. A helper function does the left-end extension,

```
pCons :: a -> Pick n a -> Pick (Succ n) a
pCons b (Pick (a, as)) = Pick (a, Cons b as)
```

leaving us with

```
picks' :: List m a -> List m (Pick m a)
picks' Nil          = Nil
picks' (Cons x xs)  = Cons (Pick (x, xs)) (fmap (pCons x) (picks' xs))
```

and if we want

```
picks = fmap unpick . picks'
```

That's perhaps overkill, but it might be worth it if we want to separate older and younger siblings, splitting lists in three, like this:

```
data Pick3 :: Nat -> * -> * where
  Pick3 :: List m a -> a -> List n a -> Pick3 (Succ (m + n)) a

pCons3 :: a -> Pick3 n a -> Pick3 (Succ n) a
pCons3 b (Pick3 bs x as) = Pick3 (Cons b bs) x as

picks3 :: List m a -> List m (Pick3 m a)
picks3 Nil          = Nil
picks3 (Cons x xs)  = Cons (Pick3 Nil x xs) (fmap (pCons3 x) (picks3 xs))
```

Again, all the action is left-ended, so we're fitting nicely with the computational behaviour of `+`.

Accumulating

If we want to keep the style of the original attempt, accumulating the elder siblings as we go, we could do worse than to keep them *zipper-style*, storing the closest element in the most accessible place. That is, we can store the elder siblings in reverse order, so that at each step we need only `Cons`, rather than concatenating. When we want to build the full sibling list in each place, we need to use reverse-concatenation (really, plugging a sublist into a list zipper). You can type `revCat` easily for vectors if you deploy the *abacus-style* addition:

```
type family (+/) (a :: Nat) (b :: Nat) :: Nat
type instance (+/) Zero     n  =  n
type instance (+/) (Succ m) n  =  m +/ Succ n
```

That's the addition which is in alignment with the value-level computation in `revCat`, defined thus:

```
revCat :: List m a -> List n a -> List (m +/ n) a
revCat Nil         ys  =  ys
revCat (Cons x xs) ys  =  revCat xs (Cons x ys)
```

We acquire a zipperized `go` version

```
picksr :: List (Succ n) a -> List (Succ n) (a, List n a)
picksr = go Nil where
  go :: List p a -> List (Succ q) a -> List (Succ q) (a, List (p +/ q) a)
  go p (Cons x xs@Nil)         =  Cons (x, revCat p xs) Nil
  go p (Cons x xs@(Cons _ _))  =  Cons (x, revCat p xs) (go (Cons x p) xs)
```

and nobody proved anything.

Conclusion

Leopold Kronecker should have said

> God made the natural numbers *to perplex us*: all the rest is the work of man.

One `Succ` looks very like another, so it is very easy to write down expressions which give the size of things in a way which is out of alignment with their structure. Of course, we can and should (and are about to) equip GHC's constraint solver with improved kit for type-level numerical reasoning. But before that kicks in, it's worth just conspiring to align the `Cons`es with the `Succ`s.

7.18 Monoid for integers modulo

Expanding on my comment, here's a first crack. The modulus is enforced by type, but not the canonical choice of representative: that's just done by computation, so would necessitate an abstraction barrier. Types of bounded numbers are also available, but they take a bit more work.

Enter, `{-# LANGUAGE KitchenSink #-}`. I mean (actually the not too bad)

```
{-# LANGUAGE DataKinds, GADTs, KindSignatures, FlexibleInstances #-}
```

and let's get cracking.

Firstly, just by reflex, I introduce the Hasochistic natural numbers:

```
data Nat = Z | S Nat              -- type-level numbers
data Natty :: Nat -> * where      -- value-level representation of Nat
  Zy :: Natty Z
  Sy :: Natty n -> Natty (S n)
class NATTY n where               -- value-level representability
  natty :: Natty n
instance NATTY Z where
  natty = Zy
instance NATTY n => NATTY (S n) where
  natty = Sy natty
```

To my mind, that's just what you do when you want to declare a datatype and then allow other types to depend on its values. Richard Eisenberg's "singletons" library automates the construction.

(If the example goes on to use numbers to index vectors, some people point out that vectors of `()` can also serve as singletons for `Nat`. They're technically correct, of course, but misguided.

When we think of `Natty` and `NATTY` as systematically generated from `Nat`, they're an entitlement we can exploit or not as we see fit, not an extra to justify. This example does not involve vectors, and it would be perverse to introduce vectors just to have singletons for `Nat`.)

I hand-roll a bunch of conversion functions and `Show` instances, so we can see what we're doing, apart from anything else.

```
int :: Nat -> Integer
int Z = 0
int (S n) = 1 + int n

instance Show Nat where
  show = show . int

nat :: Natty n -> Nat
nat Zy = Z
nat (Sy n) = S (nat n)

instance Show (Natty n) where
  show = show . nat
```

Now we're ready to declare `Mod`.

```
data Mod :: Nat -> * where
  (:%) :: Integer -> Natty n -> Mod (S n)
```

The type carries the modulus. The values carry an unnormalized representative of the equivalence class, but we had better figure out how to normalize it. Division for unary numbers is a peculiar sport which I learned as a child.

```
remainder :: Natty n   -- predecessor of modulus
          -> Integer   -- any representative
          -> Integer   -- canonical representative
  -- if candidate negative, add the modulus
remainder n x | x < 0 = remainder n (int (nat (Sy n)) + x)
  -- otherwise get dividing
remainder n x = go (Sy n) x x where
  go :: Natty m  -- divisor countdown (initially the modulus)
     -> Integer  -- our current guess at the representative
     -> Integer  -- dividend countdown
     -> Integer  -- the canonical representative
    -- when we run out of dividend the guessed representative is canonical
  go _      c 0 = c
    -- when we run out of divisor but not dividend,
    --   the current dividend countdown is a better guess at the rep,
    --   but perhaps still too big, so start again, counting down
    --   from the modulus (conveniently still in scope)
  go Zy     _ y = go (Sy n) y y
    -- otherwise, decrement both countdowns
  go (Sy m) c y = go m c (y - 1)
```

Now we can make a smart constructor.

```
rep :: NATTY n                  -- we pluck the modulus rep from thin air
    => Integer -> Mod (S n)     -- when we see the modulus we want
rep x = remainder n x :% n where n = natty
```

And then the `Monoid` instance is easy:

```
instance NATTY n => Monoid (Mod (S n)) where
  mempty                    = rep 0
  mappend (x :% _) (y :% _) = rep (x + y)
```

I chucked in some other things, too:

```
instance Show (Mod n) where
  show (x :% n) = concat ["(", show (remainder n x), " :% ", show (Sy n), ")"]
instance Eq (Mod n) where
  (x :% n) == (y :% _) = remainder n x == remainder n y
```

With a little convenience...

```
type Four = S (S (S (S Z)))
```

we get

```
> foldMap rep [1..5] :: Mod Four
(3 :% 4)
```

So yes, you do need dependent types, but Haskell is dependently typed enough.

7.19 Are there non-trivial Foldable or Traversable instances that don't look like containers?

Every valid `Traversable f` is isomorphic to `Normal s` for some `s :: Nat -> *` where

```
data Normal (s :: Nat -> *) (x :: *) where  -- Normal is Girard's terminology
  (:-) :: s n -> Vec n x -> Normal s x

data Nat = Zero | Suc Nat

data Vec (n :: Nat) (x :: *) where
  Nil   :: Vec Zero n
  (:::) :: x -> Vec n x -> Vec (Suc n) x
```

but it's not at all trivial to implement the iso in Haskell (but it's worth a go with full dependent types). Morally, the `s` you pick is

```
data {- not really -} ShapeSize (f :: * -> *) (n :: Nat) where
  Sized :: pi (xs :: f ()) -> ShapeSize f (length xs)
```

and the two directions of the iso separate and recombine shape and contents. The shape of a thing is given just by `fmap (const ())`, and the key point is that the length of the shape of an `f x` is the length of the `f x` itself.

Vectors are traversable in the visit-each-once-left-to-right sense. Normals are traversable exactly in by preserving the shape (hence the size) and traversing the vector of elements. To be traversable is to have finitely many element positions arranged in a linear order: isomorphism to a normal functor exactly exposes the elements in their linear order. Correspondingly, every `Traversable` structure is a (finitary) container: they have a set of shapes-with-size and a corresponding notion of position given by the initial segment of the natural numbers strictly less than the size.

The `Foldable` things are also finitary and they keep things in an order (there is a sensible `toList`), but they are not guaranteed to be `Functor`s, so they don't have such a crisp notion of *shape*. In that sense (the sense of "container" defined by my colleagues Abbott, Altenkirch and Ghani), they do not necessarily admit a shapes-and-positions characterization and are thus not containers. If you're lucky, some of them may be containers upto some quotient. Indeed `Foldable` exists to allow processing of structures like `Set` whose internal structure is intended to be a secret, and certainly depends on ordering information about the elements which is not necessarily respected by traversing operations. Exactly what constitutes a well behaved `Foldable` is rather a moot point, however: I won't quibble with the pragmatic benefits of that library design choice, but I could wish for a clearer specification.

7.20 Are GHC's Type Famlies An Example of System F-omega?

System F-omega allows universal quantification, abstraction and application at *higher kinds*, so not only over types (at kind `*`), but also at kinds `k1 -> k2`, where `k1` and `k2` are themselves kinds generated from `*` and `->`. Hence, the type level itself becomes a simply typed lambda-calculus.

Haskell delivers slightly less than F-omega, in that the type system allows quantification and application at higher kinds, but not abstraction. Quantification at higher kinds is how we have types like

```
fmap :: forall f, s, t. Functor f => (s -> t) -> f s -> f t
```

with `f :: * -> *`. Correspondingly, variables like `f` can be instantiated with higher-kinded type expressions, such as `Either String`. The lack of abstraction makes it possible to solve unification problems in type expressions by the standard first-order techniques which underpin the Hindley-Milner type system.

However, *type families* are not really another means to introduce higher-kinded types, nor a replacement for the missing type-level lambda. Crucially, they must be *fully applied*. So your example,

```
type family Foo a
type instance Foo Int = Int
type instance Foo Float = ...
....
```

should not be considered as introducing some

```
Foo :: * -> * -- this is not what's happening
```

because `Foo` on its own is not a meaningful type expression. We have only the weaker rule that `Foo t :: *` whenever `t :: *`.

Type families do, however, act as a distinct type-level computation mechanism beyond F-omega, in that they introduce *equations* between type expressions. The extension of System F with equations is what gives us the "System Fc" which GHC uses today. Equations `s ~ t` between type expressions of kind `*` induce coercions transporting values from `s` to `t`. Computation is done by deducing equations from the rules you give when you define type families.

Moreover, you *can* give type families a higher-kinded return type, as in

```
type family Hoo a
type instance Hoo Int = Maybe
type instance Hoo Float = IO
...
```

so that `Hoo t :: * -> *` whenever `t :: *`, but still we cannot let `Hoo` stand alone.

The trick we sometimes use to get around this restriction is `newtype` wrapping:

```
newtype Noo i = TheNoo {theNoo :: Foo i}
```

which does indeed give us

```
Noo :: * -> *
```

but means that we have to apply the projection to make computation happen, so `Noo Int` and `Int` are provably distinct types, but

```
theNoo :: Noo Int -> Int
```

So it's a bit clunky, but we can kind of compensate for the fact that type families do not directly correspond to type operators in the F-omega sense.

7.21 How to derive Eq for a GADT with a non-* kinded phantom type parameter

As others have identified, the key to the problem is the existentially quantified `tag` in the type of `Con3`. When you're trying to define

```
Con3 s == Con3 t = ???
```

there's no reason why `s` and `t` should be expressions with the same `tag`.

But perhaps you don't care? You can perfectly well define the *heterogeneous* equality test which is happy to compare `Expr`s structurally, regardless of tags.

```
instance Eq (Expr tag) where
  (==) = heq where
    heq :: Expr a -> Expr b -> Bool
    heq (Con1 i) (Con1 j) = i == j
    heq (Con2 s) (Con2 t) = heq s t
    heq (Con3 s) (Con3 t) = heq s t
```

If you do care, then you might be well advised to equip `Con3` with a run-time witness to the existentially quantified `tag`. The standard way to do this is with the *singleton* construction.

```
data SingExprTag (tag :: ExprTag) where
  SingTag1 :: SingExprTag Tag1
  SingTag2 :: SingExprTag Tag2
```

Case analysis on a value in `SingExprTag tag` will exactly determine what `tag` is. We can slip this extra piece of information into `Con3` as follows:

```
data Expr' (tag :: ExprTag) where
  Con1' :: Int -> Expr' tag
  Con2' :: Expr' tag -> Expr' tag
  Con3' :: SingExprTag tag -> Expr' tag -> Expr' Tag2
```

Now we can check whether the tags match. We could write a heterogeneous equality for tag singletons like this…

```
heqTagBoo :: SingExprTag a -> SingExprTag b -> Bool
heqTagBoo SingTag1 SingTag1 = True
heqTagBoo SingTag2 SingTag2 = True
heqTagBoo _        _        = False
```

...but to do so would be perfectly useless, as it only gives us a value of type `Bool`, with no idea what that value means nor to what its truth might entitle us. Knowing that `heqTagBoo a b = True` does not tell the typechecker anything useful about the tags which `a` and `b` witness. **A Boolean is a bit uninformative.**

We can write essentially the same test, but delivering in the positive case some *evidence* that the tags are equal.

```
data x :=: y where
  Refl :: x :=: x

singExprTagEq :: SingExprTag a -> SingExprTag b -> Maybe (a :=: b)
singExprTagEq SingTag1 SingTag1  = Just Refl
singExprTagEq SingTag2 SingTag2  = Just Refl
singExprTagEq _        _         = Nothing
```

Now we're cooking with gas! We can implement an instance of `Eq` for `Expr'` which uses `ExprTag` comparison to justify a recursive call on two `Con3'` children when the tags have been shown to match.

```
instance Eq (Expr' tag) where
  Con1' i    ==  Con1' j    = i == j
  Con2' s    ==  Con2' t    = s == t
  Con3' a s  ==  Con3' b t  = case singExprTagEq a b of
    Just Refl -> s == t
    Nothing -> False
```

The general situation is that promoted types need their associated singleton types (at least until we get proper Π-types), and we need evidence-producing heterogeneous equality tests for those singleton families, so that we can compare two singletons and gain type-level knowledge when they witness the same type-level values. Then as long as your GADTs carry singleton witnesses for any existentials, you can test equality homogeneously, ensuring that positive results from singleton tests give the bonus of unifying types for the other tests.

7.22 Do all Type Classes in Haskell Have a Category Theoretic Analogue?

When interpreted sufficiently pedantically, the answer to all of these questions is "yes", but for uninformatively trivial reasons.

Every category C restricts to a discrete subcategory |C| with the same objects as C but only identity morphisms (and hence no interesting structure). At the very least, operations on Haskell types can be boringly interpreted as operations on the discrete category $|*|$. The recent "roles" story amounts to (but is not spun as) an attempt to acknowledge that the morphisms matter, not just the objects. The "nominal" role for types amounts to working in $|*|$ rather than $*$.

(Note, I dislike the use of "Hask" as the name of the "category of Haskell types and functions": I fear that labelling one category as *the* Haskell category has the unfortunate side-effect of blinding us to the wealth of *other* categorical structure in Haskell programming. It's a trap.)

Being differently pedantic, I'd note that you can make up any old crap as a typeclass over any old kind, with no interesting structure whatsoever (but with trivial structure that can still be talked about categorically, if one must). However, the classes you find in the library are very

often structure-rich. Classes over `* -> *` are often, by design, subclasses of `Functor`, requiring the existence of certain natural transformations in addition to `fmap`.

For question 2. Yes, of course a class over `*` gives a subcategory of `*`. It's no problem to chuck objects out of a category, because the categorical requirement that identities and composites exist require *morphisms* to exist, given objects, but make no demands about which *objects* exist. The fact that it's boringly possible makes it a boring fact. However, many Haskell typeclasses over `*` give rise to much more interesting categories than those arising just as subcategories of `*`. E.g., the `Monoid` class gives us a category where the objects are instances of `Monoid` and the arrows are *monoid homomorphisms*: not just any old function `f` from one `Monoid` to another, but one which preserves the structure: `f mempty = mempty` and `f (mappend x y) = mappend (f x) (f y)`.

For question 3, well, in that there's a ton of categorical structure lurking everywhere, there's certainly a ton of categorical structure available (possibly but not necessarily) at higher kinds. I'm particularly fond of functors between indexed families of sets.

```
type (s :: k -> *) :-> (t :: k -> *) = forall x. s x -> t x

class FunctorIx (f :: (i -> *) -> (j -> *)) where
  mapIx :: (s :-> t) -> (f s :-> f t)
```

When `i` and `j` coincide, it becomes sensible to ask when such an `f` is a monad. The usual categorical definition suffices, even though we've left `* -> *` behind.

The message is this: nothing about being a typeclass inherently induces *interesting* categorical structure; there is plenty of interesting categorical structure which can usefully be presented via type classes over all manner of kinds. There are most certainly interesting functors from `*` (sets and functions) to `* -> *` (functors and natural transformations). Don't be blinded by careless talk about "Hask" to the richness of categorical structure in Haskell.

7.23 Haskell type resolution in Type Classes (Generator, Comonad)

Reid's right in his comment. When you write

```
class Generator g where
  next :: State g a
```

you're really saying

```
class Generator g where
  next :: forall a. State g a
```

so that from a given state in `g`, your clients can generate an element of whatever type `a` they wish for, rather than whatever type is being supplied by the state in `g`.

There are three sensible ways to fix this problem. I'll sketch them in the order I'd prefer them.

Plan A is to recognize that any generator of things is in some sense a container of them, so presentable as a type constructor rather than a type. It should certainly be a `Functor` and with high probability a `Comonad`. So

```
class Comonad f => Generator f where
  move :: forall x. f x -> f x
  next :: forall x. State (f x) x
  next = state $ \ g -> (extract g, move g)
  -- laws
  -- move . duplicate = duplicate . move

instance Generator [] where
  move = tail
```

If that's all Greek to you, maybe now is your opportunity to learn some new structure on a need-to-know basis!

Plan B is to ignore the comonadic structure and add an *associated type*.

```
class Generator g where
  type From g
  next :: State g (From g)

instance Generator [a] where
  type From [a] = a
  next = state $ \ (a : as) -> (a, as)
```

Plan C is the "functional dependencies" version, which is rather like MonadSupply, as suggested by Cirdec.

```
class Generator g a | g -> a where
  next :: State g a

instance Generator [a] a where
  next = state $ \ (a : as) -> (a, as)
```

What all of these plans have in common is that the functional relationship between `g` and `a` is somehow acknowledged. Without that, there's nothing doing.

7.24 Prove idempotency of type-level disjunction

The thing that you ask for is not possible, but something quite like it might do instead. It's not possible because the proof requires a case analysis on type level Booleans, but you have no data which enables you to make such an event occur. The fix is to include just such information via a singleton.

First up, let me note that your type for `idemp` is a little obfuscated. The constraint `a ~ b` just names the same thing twice. The following typechecks:

```
idemq :: p (Or b b) -> p b
idemq = undefined
idemp :: a ~ b => p (Or a b) -> p a
idemp = idemq
```

(If you have a constraint `a ~ t` where `t` does not contain `a`, it's usually good to substitute `t` for `a`. The exception is in `instance` declarations: an `a` in an instance head will match anything, hence the instance will fire even if that thing has not yet obviously become `t`. But I digress.)

I claim `idemq` is undefinable because we have no useful information about `b`. The only data available inhabit `p`-of-something, and we don't know what `p` is.

We need to reason by cases on `b`. Bear in mind that with general recursive type families, we can define type level Booleans which are neither `True` nor `False`. If I switch on `UndecidableInstances`, I can define

```
type family Loop (b :: Bool) :: Bool
type instance Loop True = Loop False
type instance Loop False = Loop True
```

so `Loop True` cannot be reduced to `True` or `False`, and locally worse, there is no way to show that

```
Or (Loop True) (Loop True) ~ Loop True     -- this ain't so
```

There's no way out of it. We need run time evidence that our `b` is one of the well behaved Booleans that computes somehow to a value. Let us therefore *sing*

```
data Booly :: Bool -> * where
  Truey   :: Booly True
  Falsey  :: Booly False
```

If we know `Booly b`, we can do a case analysis which will tell us what `b` is. Each case will then go through nicely. Here's how I'd play it, using an equality type defined with `PolyKinds` to pack up the facts, rather than abstracting over uses `p b`.

```
data (:=:) a b where
  Refl :: a :=: a
```

Our key fact is now plainly stated and proven:

```
orIdem :: Booly b -> Or b b :=: b
orIdem Truey   = Refl
orIdem Falsey  = Refl
```

And we can deploy this fact by strict case analysis:

```
idemp :: Booly b -> p (Or b b) -> p b
idemp b p = case orIdem b of Refl -> p
```

The case analysis must be strict, to check that the evidence is not some loopy lie, but rather an honest to goodness `Refl` silently packing up just the proof of `Or b b ~ b` that's needed to fix up the types.

If you don't want to sling all these singleton values around explicitly, you can, as kosmikus suggests, hide them in a dictionary and extract them just when you need them.

Richard Eisenberg and Stephanie Weirich have a Template Haskell library which mills these families and classes for you. SHE can build them too and lets you write

```
orIdem pi b :: Bool. Or b b :=: b
orIdem {True}   = Refl
orIdem {False}  = Refl
```

where `pi b :: Bool.` expands to `forall b :: Bool. Booly b ->`.

But it's such a palaver. That's why my gang are working on adding an actual `pi` to Haskell, being a non-parametric quantifier (distinct from `forall` and `->`) which can be instantiated by stuff in the now nontrivial intersection between Haskell's type and term languages. This `pi` could also have an "implicit" variant, where the argument is by default kept hidden. The two respectively correspond to using singleton families and classes, but there's no need to define datatypes three times over to get the additional kit.

It might be worth mentioning that in a total type theory, it is *not* needed to pass the extra copy of the Boolean `b` at run time. The thing is, `b` is used only to make the proof that data may be transported from `p (Or b b)` to `p b`, not necessarily to make the data being transported. We don't compute under binders at run time, so there's no way to cook up a dishonest proof of the equation, hence we can erase the proof component and the copy of `b` that delivers it. As Randy Pollack says, *the best thing about working in a strongly normalizing calculus is not having to normalize things*.

7.25 Recursively defined instances and constraints

The definition of `pure` is indeed at the heart of the problem. What should its type be, fully quantified and qualified?

```
pure :: forall (n :: Nat) (x :: *). x -> Vector n x            -- (X)
```

won't do, as there is no information available at run-time to determine whether `pure` should emit `VNil` or `VCons`. Correspondingly, as things stand, you can't just have

```
instance Applicative (Vector n)                                -- (X)
```

What can you do? Well, working with the Strathclyde Haskell Enhancement, in the Vec.lhs example file, I define a precursor to `pure`

```
vec :: forall x. pi (n :: Nat). x -> Vector {n} x
vec {Zero}    x = VNil
vec {Succ n}  x = VCons x (vec n x)
```

with a `pi` type, requiring that a copy of `n` be passed at runtime. This `pi (n :: Nat).` desugars as

```
forall n. Natty n ->
```

where `Natty`, with a more prosaic name in real life, is the singleton GADT given by

```
data Natty n where
  Zeroy :: Natty Zero
  Succy :: Natty n -> Natty (Succ n)
```

and the curly braces in the equations for `vec` just translate `Nat` constructors to `Natty` constructors. I then define the following diabolical instance (switching off the default Functor instance)

```
instance {:n :: Nat:} => Applicative (Vec {n}) where
  hiding instance Functor
  pure = vec {:n :: Nat:} where
  (<*>) = vapp where
    vapp :: Vec {m} (s -> t) -> Vec {m} s -> Vec {m} t
    vapp  VNil          VNil          = VNil
    vapp  (VCons f fs)  (VCons s ss)  = VCons (f s) vapp fs ss
```

which demands further technology, still. The constraint `{:n :: Nat:}` desugars to something which requires that a `Natty n` witness exists, and by the power of scoped type variables, the same `{:n :: Nat:}` subpoenas that witness explicitly. Longhand, that's

```
class HasNatty n where
  natty :: Natty n
instance HasNatty Zero where
  natty = Zeroy
instance HasNatty n => HasNatty (Succ n) where
  natty = Succy natty
```

and we replace the constraint `{:n :: Nat:}` with `HasNatty n` and the corresponding term with `(natty :: Natty n)`. Doing this construction systematically amounts to writing a fragment of a Haskell typechecker in type class Prolog, which is not my idea of joy so I use a computer.

Note that the `Traversable` instance (pardon my idiom brackets and my silent default Functor and Foldable instances) requires no such jiggery pokery

```
instance Traversable (Vector n) where
  traverse f VNil         = (|VNil|)
  traverse f (VCons x xs) = (|VCons (f x) (traverse f xs)|)
```

That's all the structure you need to get matrix multiplication without further explicit recursion.

TL;DR Use the singleton construction and its associated type class to collapse all of the recursively defined instances into the existence of a runtime witness for the type-level data, from which you can compute by explicit recursion.

What are the design implications?

GHC 7.4 has the type promotion technology but SHE still has the singleton construction `pi`-types to offer. One clearly important thing about promoted datatypes is that they're *closed*, but that isn't really showing up cleanly yet: the constructability of singleton witnesses is the manifestation of that closedness. Somehow, if you have `forall (n :: Nat).` then it's always reasonable to demand a singleton as well, but to do so makes a difference to the generated code: whether it's explicit as in my `pi` construct, or implicit as in the dictionary for `{:n :: Nat:}`, there is extra runtime information to sling around, and a correspondingly weaker free theorem.

An open design question for future versions of GHC is how to manage this distinction between the presence and absence of runtime witnesses to type-level data. On the one hand, we need them in constraints. On the other hand, we need to pattern-match on them. E.g., should `pi (n :: Nat).` mean the explicit

```
forall (n :: Nat). Natty n ->
```

or the implicit

```
forall (n :: Nat). {:n :: Nat:} =>
```

? Of course, languages like Agda and Coq have both forms, so maybe Haskell should follow suit. There is certainly room to make progress, and we're working on it!

7.26 How can I get the length of dependently typed interval?

Here's my version of your program. I'm using

```
{-# LANGUAGE GADTs, DataKinds, KindSignatures, TypeFamilies #-}
```

and I've got `Nat` and its singleton

```
data Nat = Z | S Nat

data SNat :: Nat -> * where
  ZZ :: SNat Z
  SS :: SNat n -> SNat (S n)
```

Your `Interval` type is more familiar to me as the "suffix" definition of "less-than-or-equal": "suffix" because if you upgraded from numbers to lists and labelled each `S` with an element, you'd have the definition of a list suffix.

```
data Le :: Nat -> Nat -> * where
  Len :: SNat n -> Le n n
  Les :: Le m n -> Le m (S n)
```

Here's addition.

```
type family Plus (x :: Nat) (y :: Nat) :: Nat
type instance Plus Z     y  = y
type instance Plus (S x) y  = S (Plus x y)
```

Now, your puzzle is to count the `Les` constructors in some `Le`-value, extracting the singleton for the difference between its indices. Rather than *assuming* that we're working with some `Le n (Plus m n)` and trying to compute a `SNat m`, I'm going to write a function which computes the difference between *arbitrary* `Le m o`-indices and *establishes* the connection with `Plus`.

Here's the additive definition of `Le`, with singletons supplied.

```
data AddOn :: Nat -> Nat -> * where
  AddOn :: SNat n -> SNat m -> AddOn n (Plus m n)
```

We can easily establish that `Le` implies `AddOn`. Pattern matching on some `AddOn n o` reveals `o` to be `Plus m n` for some `m` and hands us the singletons we wanted.

```
leAddOn :: Le m o -> AddOn m o
leAddOn (Len n) = AddOn n ZZ
leAddOn (Les p) = case leAddOn p of AddOn n m -> AddOn n (SS m)
```

More generally, I'd advise formulating dependently typed programming problems minimizing the presence of defined functions in the indices of types over which you plan to match. This avoids complicated unification. (Epigram used to colour such functions green, hence the advice **"Don't touch the green slime!"**.) `Le n o`, it turns out (for that is the point of `leAddOn`), is no less informative a type than `Le n (Plus m n)`, but it is rather easier to match on.

Yet more generally, it is quite a normal experience to set up a dependent datatype which captures the logic of your problem but is absolutely ghastly to work with. This does not mean that all datatypes which capture the correct logic will be absolutely ghastly to work with, just that you need think harder about the ergonomics of your definition. Getting these definitions neat is not a skill that very many people pick up in their ordinary Functional Programming learning experience, so expect to climb a new learning curve.

7.27 How to make catamorphisms work with parameterized/indexed types?

I wrote a talk on this topic called "Slicing It" in 2009. It certainly points to the work by my Strathclyde colleagues, Johann and Ghani, on initial algebra semantics for GADTs. I used the notation which SHE provides for writing data-indexed types, but that has pleasingly been superseded by the "promotion" story.

The key point of the talk is, as per my comment, to be systematic about using exactly one index, but to exploit the fact that its kind can vary.

So indeed, we have (using my current preferred "Goscinny and Uderzo" names)

```
type s :-> t = forall i. s i -> t i

class FunctorIx f where
  mapIx :: (s :-> t) -> (f s :-> f t)
```

Now you can show `FunctorIx` is *closed* under fixpoints. The key is to combine two indexed sets into a one that offers a choice of index.

```
data Case (f :: i -> *) (g :: j -> *) (b :: Either i j) :: * where
  L :: f i -> Case f g (Left i)
  R :: g j -> Case f g (Right j)
```

```
(<?>) :: (f :-> f') -> (g :-> g') -> Case f g :-> Case f' g'
(f <?> g) (L x) = L (f x)
(f <?> g) (R x) = R (g x)
```

Now we can now take fixpoints of functors whose "contained elements" stand for either "payload" or "recursive substructures".

```
data MuIx (f :: (Either i j -> *) -> j -> *) :: (i -> *) -> j -> * where
  InIx :: f (Case x (MuIx f x)) j -> MuIx f x j
```

As a result, we can `mapIx` over "payload"...

```
instance FunctorIx f => FunctorIx (MuIx f) where
  mapIx f (InIx xs) = InIx (mapIx (f <?> mapIx f) xs)
```

...or write a catamorphism over the "recursive substructures"...

```
foldIx :: FunctorIx f => (f (Case x t) :-> t) -> MuIx f x :-> t
foldIx f (InIx xs) = f (mapIx (id <?> foldIx f) xs)
```

...or both at once.

```
mapFoldIx :: FunctorIx f => (x :-> y) -> (f (Case y t) :-> t) -> MuIx f x :-> t
mapFoldIx e f (InIx xs) = f (mapIx (e <?> mapFoldIx e f) xs)
```

The joy of `FunctorIx` is that it has such splendid closure properties, thanks to the ability to vary the indexing kinds. `MuIx` allows for notions of payload, and can be iterated. There is thus an incentive to work with structured indices rather than multiple indices.

7.28 Constraining Constructors in a Signature

You *can* do this sort of thing with GADTs. Far be it from me to judge whether what results is a rabbit hole, but let me at least show the recipe. I'm using the new `PolyKinds` extension, but you can manage with less.

First, decide what sorts of stuff you will need, and define a datatype of those sorts.

```
data Sort = Base | Compound
```

Next, define your data indexed by their sorts. It's like building a little typed language.

```
data WeaponPart :: Sort -> * where
  WInt    :: Int ->                                    WeaponPart Base
  WHash   :: Map.Map String Int ->                     WeaponPart Base
  WNull   ::                                           WeaponPart Base
  WTrans  :: (Some WeaponPart -> Some WeaponPart) ->   WeaponPart Compound
```

You can represent 'data of any sort' via existential quantification, as follows:

```
data Some p where
  Wit :: p x -> Some p
```

Note that the `x` does not escape, but we can still inspect the 'evidence' that `x` 'satisfies' `p`. Note that `Some` must be a `data`type, not a `newtype` as GHC objects to existential `newtype`s.

You are now free to write `Sort`-generic operations. If you have generic inputs, you can just use polymorphism, effectively currying `Some p -> ...` as `forall x. p x -> ...`.

```
instance Show (WeaponPart x) where
  show (WInt x)    = "WInt " ++ (show x)
  show (WHash x)   = "WHash " ++ (show x)
  show (WTrans _)  = "WTrans"
  show WNull       = "WNull"
```

The existential is needed for `Sort`-generic outputs: here I use it for input and output.

```
cold :: Some WeaponPart -> Some WeaponPart
cold (Wit (WInt x))    = Wit (WHash (Map.singleton "frost" x))
cold (Wit (WHash x))   = Wit (WHash $ Map.insertWith (+) "frost" 5 x)
cold (Wit (WTrans x))  = cold $ x (Wit (WInt 5))
cold (Wit WNull)       = cold $ Wit (WInt 5)
```

I had to add the occasional touch of `Wit` about the place, but it's the same program.

Meanwhile, we can now write

```
ofTheAbyss :: WeaponPart Compound -> Some WeaponPart
ofTheAbyss (WTrans x) = x (Wit (WTrans x))
```

So it's not horrendous to work with embedded type systems. Sometimes there is a cost: if you want your embedded language to have *subsorting*, you may find you do extra computation just to change the index of some data's type, making no difference to the data themselves. If you don't need subsorting, the extra discipline can often be a real friend.

7.29 Standard name of a sum type like Either but for 3 cases?

In recent Haskell, I'd switch on a bit of kitchen sink.

```
{-# LANGUAGE PolyKinds, DataKinds, GADTs, KindSignatures,
    TypeOperators, PatternSynonyms #-}
```

Then I'd define type-level list membership

```
data (:>) :: [x] -> x -> * where
  Ze ::             (x ': xs) :> x
  Su :: xs :> x -> (y ': xs) :> x
```

and now I have all the finite sums, without cranking out a whole raft of OneOfN type definitions:

```
data Sum :: [*] -> * where
  (:-) :: xs :> x -> x -> Sum xs
```

But, to address Tomas's issue about readability, I'd make use of pattern synonyms. Indeed, this sort of thing is the reason I've been banging on about pattern synonyms for years.

You can have a funny version of `Maybe`:

```
type MAYBE x = Sum '[(), x]

pattern NOTHING :: MAYBE x
pattern NOTHING = Ze :- ()

pattern JUST :: x -> MAYBE x
pattern JUST x = Su Ze :- x
```

and you can even use `newtype` to build recursive sums.

```
newtype Tm x = Tm (Sum '[x, (Tm x, Tm x), Tm (Maybe x)])

pattern VAR :: x -> Tm x
pattern VAR x = Tm (Ze :- x)

pattern APP :: Tm x -> Tm x -> Tm x
pattern APP f s = Tm (Su Ze :- (f, s))

pattern LAM :: Tm (Maybe x) -> Tm x
pattern LAM b = Tm (Su (Su Ze) :- b)
```

The newtype wrapper also lets you make `instance` declaration for types built that way.

You can, of course, also use pattern synonyms to hide an iterated `Either` nicely.

This technique is not exclusive to sums: you can do it for products, too, and that's pretty much what happens in de Vries and Löh's Generics-SOP library.

The big win from such an encoding is that the description of data is itself (type-level) data, allowing you to cook up lots of `deriving`-style functionality without hacking the compiler.

In the future (if I have my way), all datatypes will be *defined*, not *declared*, with datatype descriptions made of data specifiying both the algebraic structure (allowing generic equipment to be computed) of the data and its appearance (so you can see what you're doing when working with a specific type).

But the future is sort of here already.

7.30 How to specify the type for a heterogenous collection in a GADT formulated AST?

One way to approach this problem is to tag values with run-time type representatives. I'm channelling Stephanie Weirich, here. Let's have a small example. First, give a representation to some types. That's typically done with a *singleton* construction.

```
data Type :: * -> * where
  Int   :: Type Int
  Char  :: Type Char
  List  :: Type x -> Type [x]
```

So `Type Int` contains one value, which I've also called `Int`, because it acts as the run-time representative of the type `Int`. If you can see colour even in monochrome things, the `Int` left of the `::` is red, and the `Int` after `Type` is blue.

Now we can do existential packaging, preserving utility.

```
data Cell :: * where
 (:::) :: x -> Type x -> Cell
```

A `Cell` is a value tagged with a run-time representative of its type. You can recover the utility of the value by reading its type tag. Indeed, as types are first-order structures, we can check them for equality in a useful way.

```
data EQ :: k -> k -> * where
  Refl :: EQ x x

typeEQ :: Type x -> Type y -> Maybe (EQ x y)
typeEQ Int Int = Just Refl
```

```
typeEQ Char Char = Just Refl
typeEQ (List s) (List t) = case typeEQ s t of
  Just Refl -> Just Refl
  Nothing -> Nothing
typeEQ _ _ = Nothing
```

A Boolean equality on type representatives is no use: we need the equality test to construct the *evidence* that the represented types can be unified. With the evidence-producing test, we can write

```
gimme :: Type x -> Cell -> Maybe x
gimme t (x ::: s) = case typeEQ s t of
  Just Refl -> Just x
  Nothing   -> Nothing
```

Of course, writing the type tags is a nuisance. But why keep a dog and bark yourself?

```
class TypeMe x where
  myType :: Type x

instance TypeMe Int where
  myType = Int

instance TypeMe Char where
  myType = Char

instance TypeMe x => TypeMe [x] where
  myType = List myType

cell :: TypeMe x => x -> Cell
cell x = x ::: myType
```

And now we can do things like

```
myCells :: [Cell]
myCells = [cell (length "foo"), cell "foo"]
```

and then get

```
> gimme Int (head myCells)
Just 3
```

Of course, it would all be so much tidier if we didn't have to do the singleton construction and could just pattern-match on such types as we might choose to retain at run-time. I expect we'll get there when the mythical `pi` quantifier becomes less mythical.

7.31 Type-threaded heterogenous lists and defaulting(?) with type families?

Is there some particular reason why the *Kleene star* GADT won't do this job?

```
data Star r a b where
  Nil   :: Star r a a
  Cons  :: r a b -> Star r b c -> Star r a c
```

```
compose :: Star (->) a b -> a -> b
compose Nil          = id
compose (Cons f fs)  = compose fs . f
```

But if you need a type class approach, I wouldn't interfere.

7.32 Constructor that lifts (via DataKinds) to $* \to A$

At the moment, I'm afraid not. I haven't spotted an obvious workaround, either.

This ticket documents the prospects for the declaration of data kinds, born kind, rather than being data types with kindness thrust upon them. It would be entirely reasonable for the constructors of such things to pack up types as you propose. We're not there yet, but it doesn't look all that problematic.

My eyes are on a greater prize. I would like * to be perfectly sensible type of runtime values, so that the kind you want could exist by promotion as we have it today. Combine that with the mooted notion of `pi`-type (non-parametric abstraction over the portion of the language that's effectively shared by types and values) and we might get a more direct way to make ad hoc type abstractions than we have with `Data.Typeable`. The usual `forall` would remain parametric.

7.33 How should the general type of a "lemma" function be understood?

We've had some excellent answers, but as the perpetrator, I thought I'd offer some remarks.

Yes, there are multiple equivalent presentations of these lemmas. The presentation I use is one of them, and the choice is largely a pragmatic one. These days (in a more recent codebase), I go as far as to define

```
-- Holds :: Constraint -> *
type Holds c = forall t . (c => t) -> t
```

This is an example of an *eliminator type*: it abstracts over what it delivers (the *motive* of the elimination) and it requires you to construct zero or more *methods* (one, here) of achieving the motive under more specific circumstances. The way to read it is *backwards*. It says

> If you have a problem (to inhabit any motive type `t`), and nobody else can help, maybe you can make progress by assuming constraint `c` in your method.

Given that the language of constraints admits conjunction (aka tupling), we acquire the means to write lemmas of the form

```
lemma :: forall x1 .. xn. (p1[x1 .. xn],.. pm[x1 .. xn])        -- premises
                       => t1[x1 .. xn] -> .. tl[x1 .. xn]       -- targets
                       -> Holds (c1[x1 .. xn],.. ck[x1 .. xn])  -- conclusions
```

and it might even be that some constraint, a premise `p` or a conclusion `c`, has the form of an equation

```
l[x1 .. xn] ~ r[x1 .. cn]
```

Now, to deploy such a `lemma`, consider the problem of filling a hole

```
_ :: Problem
```

Refine this `_` by the elimination `lemma`, specifying the *targets*. The *motive* comes from the problem at hand. The *method* (singular in the case of `Holds`) remains open.

```
lemma target1 .. targetl $ _
```

and the method hole will not have changed type

```
_ :: Problem
```

but GHC will know a bunch more stuff and thus be more likely to believe your solution.

Sometimes, there's a constraint-versus-data choice to make for what's a (constraint) premise and what's a (data) target. I tend to pick these to avoid ambiguity (Simon likes to guess the `x1 .. xn`, but sometimes needs a hint) and to facilitate *proof by induction*, which is much easier on targets (often the singletons for type level data) than on premises.

As to deployment, for equations, you can certainly switch to a datatype presentation and break out a case analysis

```
case dataLemma target1 .. targetl of Refl -> method
```

and indeed, if you equip yourself with the `Dict` existential

```
data Dict (c :: Constraint) :: * where
  Dict :: c => Dict c
```

you can do a bunch at once

```
case multiLemma blah blah blah of (Refl, Dict, Dict, Refl) -> method
```

but the eliminator form is more compact and readable *when there is at most one method*. Indeed, we can chain multiple lemmas without sliding ever rightward

```
lemma1 ..   $
...
lemmaj ..   $
method
```

If you have such an eliminator with two or more cases, I think it's often better to wrap it up as a GADT, so that usage sites helpfully tag each case with a constructor label.

Anyhow, yes, the point is to choose the presentation of the facts which most compactly enables us to extend the reach of GHC's constraint solving machinery so that more stuff just typechecks. If you're in a scrap with Simon, it's often a good strategy to explain yourself to Dimitrios next door.

Chapter 8

Type Theory

8.1 What is the combinatory logic equivalent of intuitionistic type theory?

So I thought about it a bit more and made some progress. Here's a first stab at encoding Martin-Löf's delightfully simple (but inconsistent) `Set : Set` system in a combinatory style. It's not a good way to finish, but it's the easiest place to get started. The syntax of this type theory is just lambda-calculus with type annotations, Pi-types, and a universe Set.

The Target Type Theory

For completeness' sake, I'll present the rules. Context validity just says you can build contexts from empty by adjoining fresh variables inhabiting `Set`s.

```
                    G |- valid   G |- S : Set
--------------     ---------------------------- x fresh for G
  . |- valid        G, x:S |- valid
```

And now we can say how to synthesize types for terms in any given context, and how to change the type of something up to the computational behaviour of the terms it contains.

```
  G |- valid             G |- S : Set   G |- T : Pi S \ x:S -> Set
------------------     ------------------------------------------
  G |- Set : Set         G |- Pi S T : Set

  G |- S : Set   G, x:S |- t : T x         G |- f : Pi S T   G |- s : S
-----------------------------------     --------------------------------
  G |- \ x:S -> t : Pi S T                G |- f s : T s

  G |- valid                  G |- s : S   G |- T : Set
-------------- x:S in G     ---------------------------- S ={beta} T
  G |- x : S                  G |- s : T
```

In a small variation from the original, I've made lambda the only binding operator, so the second argument of Pi should be a function computing the way the return type depends on the input. By convention (e.g. in Agda, but sadly not in Haskell), scope of lambda extends rightwards as far as possible, so you can often leave abstractions unbracketed when they're the last argument of a higher-order operator: you can see I did that with Pi. Your Agda type `(x : S) -> T` becomes `Pi S \ x:S -> T`.

(*Digression*. Type annotations on lambda are necessary if you want to be able to *synthesize* the type of abstractions. If you switch to type *checking* as your modus operandi, you still need annotations to check a beta-redex like `(\ x -> t) s`, as you have no way to guess the types of the parts from that of the whole. I advise modern designers to check types and exclude beta-redexes from the very syntax.)

(*Digression*. This system is inconsistent as `Set : Set` allows the encoding of a variety of "liar paradoxes". When Martin-Löf proposed this theory, Girard sent him an encoding of it in his own inconsistent System U. The subsequent paradox due to Hurkens is the neatest toxic construction we know.)

Combinator Syntax and Normalization

Anyhow, we have two extra symbols, Pi and Set, so we might perhaps manage a combinatory translation with S, K and two extra symbols: I chose U for the universe and P for the product.

Now we can define the untyped combinatory syntax (with free variables):

```
data SKUP = S | K | U | P deriving (Show, Eq)

data Unty a
  = C SKUP
  | Unty a :. Unty a
  | V a
  deriving (Functor, Eq)
infixl 4 :.
```

Note that I've included the means to include free variables represented by type `a` in this syntax. Apart from being a reflex on my part (every syntax worthy of the name is a free monad with `return` embedding variables and >>= perfoming substitution), it'll be handy to represent intermediate stages in the process of converting terms with binding to their combinatory form.

Here's normalization:

```
norm :: Unty a -> Unty a
norm (f :. a)  = norm f $. a
norm c         = c

($.) :: Unty a -> Unty a -> Unty a        -- requires first arg in normal form
C S :. f :. a $. g  = f $. g $. (a :. g)  -- S f a g = f g (a g)   share environment
C K :. a $. g       = a                   -- K a g = a            drop environment
n $. g              = n :. norm g         -- guarantees output in normal form
infixl 4 $.
```

(An exercise for the reader is to define a type for exactly the normal forms and sharpen the types of these operations.)

Representing Type Theory

We can now define a syntax for our type theory.

```
data Tm a
  = Var a
  | Lam (Tm a) (Tm (Su a))    -- Lam is the only place where binding happens
  | Tm a :$ Tm a
  | Pi (Tm a) (Tm a)          -- the second arg of Pi is a function computing a Set
  | Set
  deriving (Show, Functor)
```

```
infixl 4 :$

data Ze
magic :: Ze -> a
magic x = x `seq` error "Tragic!"

data Su a = Ze | Su a deriving (Show, Functor, Eq)
```

I use a de Bruijn index representation in the Bellegarde and Hook manner (as popularised by Bird and Paterson). The type `Su a` has one more element than `a`, and we use it as the type of free variables under a binder, with `Ze` as the newly bound variable and `Su x` being the shifted representation of the old free variable `x`.

Translating Terms to Combinators

And with that done, we acquire the usual translation, based on *bracket abstraction*.

```
tm :: Tm a -> Unty a
tm (Var a)    = V a
tm (Lam _ b)  = bra (tm b)
tm (f :$ a)   = tm f :. tm a
tm (Pi a b)   = C P :. tm a :. tm b
tm Set        = C U

bra :: Unty (Su a) -> Unty a              -- binds a variable, building a function
bra (V Ze)      = C S :. C K :. C K       -- the variable itself yields the identity
bra (V (Su x))  = C K :. V x              -- free variables become constants
bra (C c)       = C K :. C c              -- combinators become constant
bra (f :. a)    = C S :. bra f :. bra a   -- S is exactly lifted application
```

Typing the Combinators

The translation shows the way we use the combinators, which gives us quite a clue about what their types should be. `U` and `P` are just set constructors, so, writing untranslated types and allowing "Agda notation" for Pi, we should have

```
U : Set
P : (A : Set) -> (B : (a : A) -> Set) -> Set
```

The `K` combinator is used to lift a value of some type `A` to a constant function over some other type `G`.

```
  G : Set   A : Set
------------------------------
  K : (a : A) -> (g : G) -> A
```

The `S` combinator is used to lift applications over a type, upon which all of the parts may depend.

```
  G : Set
  A : (g : G) -> Set
  B : (g : G) -> (a : A g) -> Set
-------------------------------------------------
  S : (f : (g : G) ->    (a : A g) -> B g a   ) ->
      (a : (g : G) ->    A g                  ) ->
           (g : G) ->    B g (a g)
```

If you look at the type of `S`, you'll see that it exactly states the *contextualised* application rule of the type theory, so that's what makes it suitable to reflect the application construct. That's its job!

We then have application only for closed things

```
  f : Pi A B
  a : A
--------------
  f a : B a
```

But there's a snag. I've written the types of the combinators in ordinary type theory, not combinatory type theory. Fortunately, I have a machine that will make the translation.

A Combinatory Type System

```
---------
  U : U

------------------------------------------------------------
  P : PU(S(S(KP)(S(S(KP)(SKK))(S(KK)(KU))))(S(KK)(KU)))

  G : U
  A : U
----------------------------------------
  K : P[A](S(S(KP)(K[G]))(S(KK)(K[A])))

  G : U
  A : P[G](KU)
  B : P[G](S(S(KP)(S(K[A])(SKK)))(S(KK)(KU)))
-------------------------------------------------------------------------------------------
  S : P(P[G](S(S(KP)(S(K[A])(SKK)))(S(S(KS)(S(S(KS)(S(KK)(K[B])))(S(KK)(SKK))))
      (S(S(KS)(KK))(KK)))))(S(S(KP)(S(S(KP)(K[G]))(S(S(KS)(S(KK)(K[A]))))
      (S(S(KS)(KK))(KK)))))(S(S(KS)(S(S(KS)(S(KK)(KP)))(S(KK)(K[G])))))
      (S(S(KS)(S(S(KS)(S(KK)(KS)))(S(S(KS)(S(S(KS)(S(KK)(KS)))
      (S(S(KS)(S(KK)(KK)))(S(KK)(K[B])))))(S(S(KS)(S(S(KS)(S(KK)(KS)))(S(KK)(KK))))
      (S(KK)(KK)))))))(S(S(KS)(S(S(KS)(S(KK)(KS)))(S(S(KS)(S(KK)(KK)))
      (S(S(KS)(KK))(KK)))))(S(S(KS)(S(S(KS)(S(KK)(KS)))(S(KK)(KK))))(S(KK)(KK)))))))

  M : A   B : U
----------------- A ={norm} B
  M : B
```

So there you have it, in all its unreadable glory: a combinatory presentation of `Set : Set`!

There's still a bit of a problem. The syntax of the system gives you no way to guess the `G`, `A` and `B` parameters for `S` and similarly for `K`, just from the terms. Correspondingly, we can verify *typing derivations* algorithmically, but we can't just typecheck combinator terms as we could with the original system. What might work is to require the input to the typechecker to bear type annotations on uses of S and K, effectively recording the derivation. But that's another can of worms...

This is a good place to stop, if you've been keen enough to start. The rest is "behind the scenes" stuff.

Generating the Types of the Combinators

I generated those combinatory types using the bracket abstraction translation from the relevant type theory terms. To show how I did it, and make this post not entirely pointless, let me offer

my equipment.

I can write the types of the combinators, fully abstracted over their parameters, as follows. I make use of my handy `pil` function, which combines Pi and lambda to avoid repeating the domain type, and rather helpfully allows me to use Haskell's function space to bind variables. Perhaps you can almost read the following!

```
pTy :: Tm a
pTy = fmap magic $
  pil Set $ \ _A -> pil (pil _A $ \ _ -> Set) $ \ _B -> Set

kTy :: Tm a
kTy = fmap magic $
  pil Set $ \ _G -> pil Set $ \ _A -> pil _A $ \ a -> pil _G $ \ g -> _A

sTy :: Tm a
sTy = fmap magic $
  pil Set $ \ _G ->
  pil (pil _G $ \ g -> Set) $ \ _A ->
  pil (pil _G $ \ g -> pil (_A :$ g) $ \ _ -> Set) $ \ _B ->
  pil (pil _G $ \ g -> pil (_A :$ g) $ \ a -> _B :$ g :$ a) $ \ f ->
  pil (pil _G $ \ g -> _A :$ g) $ \ a ->
  pil _G $ \ g -> _B :$ g :$ (a :$ g)
```

With these defined, I extracted the relevant *open* subterms and ran them through the translation.

A de Bruijn Encoding Toolkit

Here's how to build `pil`. Firstly, I define a class of `Fin`ite sets, used for variables. Every such set has a constructor-preserving `emb`edding into the set above, plus a new `top` element, and you can tell them apart: the `embd` function tells you if a value is in the image of `emb`.

```
class Fin x where
  top :: Su x
  emb :: x -> Su x
  embd :: Su x -> Maybe x
```

We can, of course, instantiate `Fin` for `Ze` and `Suc`

```
instance Fin Ze where
  top = Ze              -- Ze is the only, so the highest
  emb = magic
  embd _ = Nothing      -- there was nothing to embed

instance Fin x => Fin (Su x) where
  top = Su top          -- the highest is one higher
  emb Ze     = Ze            -- emb preserves Ze
  emb (Su x) = Su (emb x)    -- and Su
  embd Ze      = Just Ze           -- Ze is definitely embedded
  embd (Su x)  = fmap Su (embd x)  -- otherwise, wait and see
```

Now I can define less-or-equals, with a *weakening* operation.

```
class (Fin x, Fin y) => Le x y where
  wk :: x -> y
```

The `wk` function should embed the elements of `x` as the *largest* elements of `y`, so that the extra things in `y` are smaller, and thus in de Bruijn index terms, bound more locally.

```
instance Fin y => Le Ze y where
  wk = magic    -- nothing to embed

instance Le x y => Le (Su x) (Su y) where
  wk x = case embd x of
    Nothing  -> top          -- top maps to top
    Just y   -> emb (wk y)   -- embedded gets weakened and embedded
```

And once you've got that sorted out, a bit of rank-n skullduggery does the rest.

```
lam :: forall x. Tm x -> ((forall y. Le (Su x) y => Tm y) -> Tm (Su x)) -> Tm x
lam s f = Lam s (f (Var (wk (Ze :: Su x))))
pil :: forall x. Tm x -> ((forall y . Le (Su x) y => Tm y) -> Tm (Su x)) -> Tm x
pil s f = Pi s (lam s f)
```

The higher-order function doesn't just give you a term representing the variable, it gives you an *overloaded* thing which becomes the correct representation of the variable in any scope where the variable is visible. That is, the fact that I go to the trouble of distinguishing the different scopes *by type* gives the Haskell typechecker enough information to compute the shifting required for the translation to de Bruijn representation. Why keep a dog and bark yourself?

8.2 Do Hask or Agda have equalisers?

tl;dr the proposed candidate is not quite an equaliser, but its irrelevant counterpart is

The candidate for an equaliser in Agda looks good. So let's just try it. We'll need some basic kit. Here are my refusenik ASCII dependent pair type and homogeneous intensional equality.

```
record Sg (S : Set)(T : S -> Set) : Set where
  constructor _,_
  field
    fst : S
    snd : T fst
open Sg

data _==_ {X : Set}(x : X) : X -> Set where
  refl : x == x
```

Here's your candidate for an equaliser for two functions

```
Q : {S T : Set}(f g : S -> T) -> Set
Q {S}{T} f g = Sg S \ s -> f s == g s
```

with the `fst` projection sending `Q f g` into `S`.

What it says: an element of `Q f g` is an element `s` of the source type, together with a proof that `f s == g s`. But is this an equaliser? Let's try to make it so.

To say what an equaliser is, I should define function composition.

```
_o_ : {R S T : Set} -> (S -> T) -> (R -> S) -> R -> T
(f o g) x = f (g x)
```

FIXME: `http://i.stack.imgur.com/odrtv.jpg`

Figure 8.1: equaliser diagram

So now I need to show that any `h : R -> S` which identifies `f o h` and `g o h` must factor through the candidate `fst : Q f g -> S`. I need to deliver both the other component, `u : R -> Q f g` and the proof that indeed `h` factors as `fst o u`. Here's the picture: `(Q f g , fst)` is an equalizer if whenever the diagram commutes without `u`, there is a unique way to add `u` with the diagram still commuting.

Here goes existence of the mediating `u`.

```
mediator : {R S T : Set}(f g : S -> T)(h : R -> S) ->
           (q : (f o h) == (g o h)) ->
           Sg (R -> Q f g) \ u -> h == (fst o u)
```

Clearly, I should pick the same element of `S` that `h` picks.

```
mediator f g h q = (\ r -> (h r , ?0)) , ?1
```

leaving me with two proof obligations

```
?0 : f (h r) == g (h r)
?1 : h == (\ r -> h r)
```

Now, `?1` can just be `refl` as Agda's definitional equality has the eta-law for functions. For `?0`, we are blessed by `q`. Equal functions respect application

```
funq : {S T : Set}{f g : S -> T} -> f == g -> (s : S) -> f s == g s
funq refl s = refl
```

so we may take `?0 = funq q r`.

But let us not celebrate prematurely, for the existence of a mediating morphism is not sufficient. We require also its uniqueness. And here the wheel is likely to go wonky, because == is *intensional*, so uniqueness means there's only ever one way to *implement* the mediating map. But then, our assumptions are also intensional...

Here's our proof obligation. We must show that any other mediating morphism is equal to the one chosen by `mediator`.

```
mediatorUnique :
  {R S T : Set}(f g : S -> T)(h : R -> S) ->
  (qh : (f o h) == (g o h)) ->
  (m : R -> Q f g) ->
  (qm : h == (fst o m)) ->
  m == fst (mediator f g h qh)
```

We can immediately substitute via `qm` and get

```
mediatorUnique f g .(fst o m) qh m refl = ?

? :  m == (\ r -> (fst (m r) , funq qh r))
```

which looks good, because Agda has eta laws for records, so we know that

```
m == (\ r -> (fst (m r) , snd (m r)))
```

but when we try to make `? = refl`, we get the complaint

```
snd (m _) != funq qh _ of type f (fst (m _)) == g (fst (m _))
```

which is annoying, because identity proofs are unique (in the standard configuration). Now, you can get out of this by postulating extensionality and using a few other facts about equality

```
postulate ext : {S T : Set}{f g : S -> T} -> ((s : S) -> f s == g s) -> f == g

sndq : {S : Set}{T : S -> Set}{s : S}{t t' : T s} ->
       t == t' -> _==_ {Sg S T} (s , t) (s , t')
sndq refl = refl

uip : {X : Set}{x y : X}{q q' : x == y} -> q == q'
uip {q = refl}{q' = refl} = refl

? = ext (\ s -> sndq uip)
```

but that's overkill, because the only problem is the annoying equality proof mismatch: the computable parts of the implementations match on the nose. So the fix is to work with *irrelevance*. I replace `Sg` by the `Ex`istential quantifier, whose second component is marked as irrelevant with a dot. Now it matters not which proof we use that the witness is good.

```
record Ex (S : Set)(T : S -> Set) : Set where
  constructor _,_
  field
    fst : S
    .snd : T fst
open Ex
```

and the new candidate equaliser is

```
Q : {S T : Set}(f g : S -> T) -> Set
Q {S}{T} f g = Ex S \ s -> f s == g s
```

The entire construction goes through as before, except that in the last obligation

```
? = refl
```

is accepted!

So yes, even in the intensional setting, eta laws and the ability to mark fields as irrelevant give us equalisers.

No undecidable typechecking was involved in this construction.

8.3 Why do we need containers?

To my mind, the value of containers (as in container theory) is their *uniformity*. That uniformity gives considerable scope to use container representations as the basis for executable specifications, and perhaps even machine-assisted program derivation.

Containers: a theoretical tool, not a good run-time data representation strategy

I would *not* recommend fixpoints of (normalized) containers as a good general purpose way to implement recursive data structures. That is, it is helpful to know that a given functor has (up to iso) a presentation as a container, because it tells you that generic container functionality (which is easily implemented, once for all, thanks to the uniformity) can be instantiated to your particular functor, and what behaviour you should expect. But that's not to say that a container implementation will be efficient in any practical way. Indeed, I generally prefer first-order encodings (tags and tuples, rather than functions) of first-order data.

To fix terminology, let us say that the type `Cont` of containers (on `Set`, but other categories are available) is given by a constructor $<|$ packing two fields, shapes and positions

```
S : Set
P : S -> Set
```

(This is the same signature of data which is used to determine a Sigma type, or a Pi type, or a W type, but that does not mean that containers are the same as any of these things, or that these things are the same as each other.)

The interpretation of such a thing as a functor is given by

```
[_]C : Cont -> Set -> Set
[ S <| P ]C X = Sg S \ s -> P s -> X  -- I'd prefer (s : S) * (P s -> X)
mapC : (C : Cont){X Y : Set} -> (X -> Y) -> [ C ]C X -> [ C ]C Y
mapC (S <| P) f (s , k) = (s , f o k)  -- o is composition
```

And we're already winning. That's `map` implemented once for all. What's more, the functor laws hold by computation alone. There is no need for recursion on the structure of types to construct the operation, or to prove the laws.

Descriptions are denormalized containers

Nobody is attempting to claim that, operationally, `Nat <| Fin` gives an *efficient* implementation of lists, just that by making that identification we learn something useful about the structure of lists.

Let me say something about *descriptions*. For the benefit of lazy readers, let me reconstruct them.

```
data Desc : Set1 where
  var : Desc
  sg pi  : (A : Set)(D : A -> Desc) -> Desc
  one : Desc                    -- could be Pi with A = Zero
  _*_ : Desc -> Desc -> Desc    -- could be Pi with A = Bool

con : Set -> Desc   -- constant descriptions as singleton tuples
con A = sg A \ _ -> one

_+_ : Desc -> Desc -> Desc   -- disjoint sums by pairing with a tag
S + T = sg Two \ { true -> S ; false -> T }
```

Values in `Desc` describe functors whose fixpoints give datatypes. Which functors do they describe?

```
[_]D : Desc -> Set -> Set
[ var    ]D X = X
[ sg A D ]D X = Sg A \ a -> [ D a ]D X
[ pi A D ]D X = (a : A) -> [ D a ]D X
[ one    ]D X = One
[ D * D' ]D X = Sg ([ D ]D X) \ _ -> [ D' ]D X

mapD : (D : Desc){X Y : Set} -> (X -> Y) -> [ D ]D X -> [ D ]D Y
mapD var      f x        = f x
mapD (sg A D) f (a , d)  = (a , mapD (D a) f d)
mapD (pi A D) f g        = \ a -> mapD (D a) f (g a)
mapD one      f <>       = <>
mapD (D * D') f (d , d') = (mapD D f d , mapD D' f d')
```

We inevitably have to work by recursion over descriptions, so it's harder work. The functor laws, too, do not come for free. We get a better representation of the data, operationally, because we don't need to resort to functional encodings when concrete tuples will do. But we have to work harder to write programs.

Note that every container has a description:

```
sg S \ s -> pi (P s) \ _ -> var
```

But it's also true that every description has a *presentation* as an isomorphic container.

```
ShD  : Desc -> Set
ShD D = [ D ]D One

PosD : (D : Desc) -> ShD D -> Set
PosD var      <>       = One
PosD (sg A D) (a , d)  = PosD (D a) d
PosD (pi A D) f        = Sg A \ a -> PosD (D a) (f a)
PosD one      <>       = Zero
PosD (D * D') (d , d') = PosD D d + PosD D' d'

ContD : Desc -> Cont
ContD D = ShD D <| PosD D
```

That's to say, containers are a normal form for descriptions. It's an exercise to show that `[ D ]D X` is naturally isomorphic to `[ ContD D ]C X`. That makes life easier, because to say what to do for descriptions, it's enough in principle to say what to do for their normal forms, containers. The above `mapD` operation could, in principle, be obtained by fusing the isomorphisms to the uniform definition of `mapC`.

Differential structure: containers show the way

Similarly, if we have a notion of equality, we can say what one-hole contexts are for containers *uniformly*

```
_-[_] : (X : Set) -> X -> Set
X -[ x ] = Sg X \ x' -> (x == x') -> Zero

dC : Cont -> Cont
dC (S <| P) = (Sg S P) <| (\ { (s , p) -> P s -[ p ] })
```

That is, the shape of a one-hole context in a container is the pair of the shape of the original container and the position of the hole; the positions are the original positions apart from that of the hole. That's the proof-relevant version of "multiply by the index, decrement the index" when differentiating power series.

This uniform treatment gives us the specification from which we can derive the centuries-old program to compute the derivative of a polynomial.

```
dD : Desc -> Desc
dD var = one
dD (sg A D) = sg A \ a -> dD (D a)
dD (pi A D) = sg A \ a -> (pi (A -[ a ]) \ { (a' , _) -> D a' }) * dD (D a)
dD one      = con Zero
dD (D * D') = (dD D * D') + (D * dD D')
```

How can I check that my derivative operator for descriptions is correct? By checking it against the derivative of containers!

Don't fall into the trap of thinking that just because a presentation of some idea is not operationally helpful that it cannot be conceptually helpful.

On "Freer" monads

One last thing. The `Freer` trick amounts to rearranging an arbitrary functor in a particular way (switching to Haskell)

```
data Obfuncscate f x where
  (:<) :: forall p. f p -> (p -> x) -> Obfuncscate f x
```

but this is not an *alternative* to containers. This is a slight currying of a container presentation. If we had *strong* existentials and dependent types, we could write

```
data Obfuncscate f x where
  (:<) :: pi (s :: exists p. f p) -> (fst s -> x) -> Obfuncscate f x
```

so that `(exists p. f p)` represents shapes (where you can choose your representation of positions, then mark each place with its position), and `fst` picks out the existential witness from a shape (the position representation you chose). It has the merit of being obviously strictly positive *exactly* because it's a container presentation.

In Haskell, of course, you have to curry out the existential, which fortunately leaves a dependency only on the type projection. It's the weakness of the existential which justifies the equivalence of `Obfuncscate f` and `f`. If you try the same trick in a dependent type theory with strong existentials, the encoding loses its uniqueness because you can project and tell apart different choices of representation for positions. That is, I can represent `Just 3` by

```
Just () :< const 3
```

or by

```
Just True :< \ b -> if b then 3 else 5
```

and in Coq, say, these are provably distinct.

Challenge: characterizing polymorphic functions

Every polymorphic function between container types is given in a particular way. There's that uniformity working to clarify our understanding again.

If you have some

```
f : {X : Set} -> [ S <| T ]C X -> [ S' <| T' ]C X
```

it is (extensionally) given by the following data, which make no mention of elements whatsoever:

```
toS    : S -> S'
fromP  : (s : S) -> P' (toS s) -> P s

f (s , k) = (toS s , k o fromP s)
```

That is, the only way to define a polymorphic function between containers is to say how to translate input shapes to output shapes, then say how to fill output positions from input positions.

For your preferred representation of strictly positive functors, give a similarly tight characterisation of the polymorphic functions which eliminates abstraction over the element type. (For descriptions, I would use exactly their reducability to containers.)

Challenge: capturing "transposability"

Given two functors, `f` and `g`, it is easy to say what their composition `f o g` is: `(f o g) x` wraps up things in `f (g x)`, giving us "`f`-structures of `g`-structures". But can you readily impose the extra condition that all of the `g` structures stored in the `f` structure have the same shape?

Let's say that `f >< g` captures the *transposable* fragment of `f o g`, where all the `g` shapes are the same, so that we can just as well turn the thing into a `g`-structure of `f`-structures. E.g., while `[] o []` gives *ragged* lists of lists, `[] >< []` gives *rectangular* matrices; `[] >< Maybe` gives lists which are either all `Nothing` or all `Just`.

Give `><` for your preferred representation of strictly positive functors. For containers, it's this easy.

```
(S <| P) >< (S' <| P') = (S * S') <| \ { (s , s') -> P s * P' s' }
```

Conclusion

Containers, in their normalized Sigma-then-Pi form, are not intended to be an efficient machine representation of data. But the knowledge that a given functor, implemented however, has a presentation as a container helps us understand its structure and give it useful equipment. Many useful constructions can be given abstractly for containers, once for all, when they must be given case-by-case for other presentations. So, yes, it is a good idea to learn about containers, if only to grasp the rationale behind the more specific constructions you actually implement.

8.4 To what extent are Applicative/Monad instances uniquely determined?

Given that every `Applicative` has a `Backwards` counterpart,

```
newtype Backwards f x = Backwards {backwards :: f x}
instance Applicative f => Applicative (Backwards f) where
  pure x = Backwards (pure x)
  Backwards ff <*> Backwards fs = Backwards (flip ($) <$> fs <*> ff)
```

it's *unusual* for `Applicative` to be uniquely determined, just as (and this is very far from unrelated) many sets extend to monoids in multiple ways.

In this answer, I set the exercise of finding at least four distinct valid `Applicative` instances for nonempty lists: I won't spoil it here, but I will give a big hint on how to hunt.

Meanwhile, in some wonderful recent work (which I saw at a summer school a few months ago), Tarmo Uustalu showed a rather neat way to get a handle on this problem, at least when the underlying functor is a *container*, in the sense of Abbott, Altenkirch and Ghani.

Warning: Dependent types ahead!

What is a container? If you have dependent types to hand, you can present container-like functors F uniformly, as being determined by two components

1. a set of shapes, S : Set
2. an S-indexed set of positions, P : S -> Set

Up to isomorphism, container data structures in F X are given by the dependent pair of some shape s : S, and some function e : P s -> X, which tells you the element located at each position. That is, we define the extension of a container

```
(S <| P) X = (s : S) * (P s -> X)
```

(which, by the way, looks a lot like a generalized power series if you read -> as reversed exponentiation). The triangle is supposed to remind you of a tree node sideways, with an element s : S labelling the apex, and the baseline representing the position set P s. We say that some functor is a container if it is isomorphic to some `S <| P`.

In Haskell, you can easily take `S = F ()`, but constructing `P` can take quite a bit of type-hackery. But that *is* something you can try at home. You'll find that containers are closed under all the usual polynomial type-forming operations, as well as identity,

```
Id ~= () <| \ _ -> ()
```

composition, where a whole shape is made from just one outer shape and an inner shape for each outer position,

```
(S0 <| P0) . (S1 <| P1)  ~=  ((S0 <| P0) S1) <| \ (s0, e0) -> (p0 : P0, P1 (e0 p0))
```

and some other things, notably the *tensor*, where there is one outer and one inner shape (so "outer" and "inner" are interchangeable)

```
(S0 <| P0) (X) (S1 <| P1)   =   ((S0, S1) <| \ (s0, s1) -> (P0 s0, P1 s1))
```

so that `F (X) G` means "`F`-structures of `G`-structures-all-the-same-shape", e.g., `[] (X) []` means *rectangular* lists-of-lists. But I digress

Polymorphic functions between containers Every polymorphic function

```
m : forall X. (S0 <| P0) X -> (S1 <| P1) X
```

can be implemented by a *container morphism*, constructed from two components in a very particular way.

1. a function `f : S0 -> S1` mapping input shapes to output shapes;
2. a function `g : (s0 : S0) -> P1 (f s0) -> P0 s0` mapping output positions to input positions.

Our polymorphic function is then

```
\ (s0, e0) -> (f s0, e0 . g s0)
```

where the output shape is computed from the input shape, then the output positions are filled up by picking elements from input positions.

(If you're Peter Hancock, you have a whole other metaphor for what's going on. Shapes are Commands; Positions are Responses; a container morphism is a *device driver*, translating commands one way, then responses the other.)

Every container morphism gives you a polymorphic function, but the reverse is also true. Given such an m, we may take

```
(f s, g s) = m (s, id)
```

That is, we have a *representation theorem*, saying that every polymorphic function between two containers is given by such an `f`, `g`-pair.

What about `Applicative`? We kind of got a bit lost along the way, building all this machinery. But it *has* been worth it. When the underlying functors for monads and applicatives are containers, the polymorphic functions `pure` and `<*>`, `return` and `join` must be representable by the relevant notion of container morphism.

Let's take applicatives first, using their monoidal presentation. We need

```
unit : () -> (S <| P) ()
mult : forall X, Y. ((S <| P) X, (S <| P) Y) -> (S <| P) (X, Y)
```

The left-to-right maps for shapes require us to deliver

```
unitS : () -> S
multS : (S, S) -> S
```

so it looks like we might need a monoid. And when you check that the applicative laws, you find we need *exactly* a monoid. Equipping a container with applicative structure is *exactly* refining the monoid structures on its shapes with suitable position-respecting operations. There's nothing to do for `unit` (because there is no chocie of source position), but for `mult`, we need that whenenver

```
multS (s0, s1) = s
```

we have

```
multP (s0, s1) : P s -> (P s0, P s1)
```

satisfying appropriate identity and associativity conditions. If we switch to Hancock's interpretation, we're defining a monoid (skip, semicolon) for commands, where there is no way to look at the response to the first command before choosing the second, like commands are a deck of punch cards. We have to be able to chop up responses to combined commands into the individual responses to the individual commands.

So, every monoid on the shapes gives us a potential applicative structure. For lists, shapes are numbers (lengths), and there are a great many monoids from which to choose. Even if shapes live in `Bool`, we have quite a bit of choice.

What about `Monad`? Meanwhile, for monads `M` with `M ~= S <| P`. We need

```
return : Id -> M
join   : M . M -> M
```

Looking at shapes first, that means we need a sort-of lopsided monoid.

```
return_f : () -> S
join_f   : (S <| P) S -> S  --  (s : S, P s -> S) -> S
```

It's lopsided because we get a bunch of shapes on the right, not just one. If we switch to Hancock's interpretation, we're defining a kind of sequential composition for commands, where we do let the second command be chosen on the basis of the first response, like we're interacting at a teletype. More geometrically, we're explaining how to glom two layers of a tree into one. It would be very surprising if such compositions were unique.

Again, for the positions, we have to map single output positions to pairs in a coherent way. This is trickier for monads: we first choose an outer position (response), then we have to choose an inner position(response) appropriate to the shape (command) found at the first position (chosen after the first response).

I'd love to link to Tarmo's work for the details, but it doesn't seem to have hit the streets yet. He has actually used this analysis to enumerate all possible monad structures for several choices of underlying container. I'm looking forward to the paper!

Edit. By way of doing honour to the other answer, I should observe that when everywhere `P s = ()`, then `(S <| P) X ~= (S, X)` and the monad/applicative structures coincide exactly with each other and with the monoid structures on `S`. That is, for writer monads, we need only choose the shape-level operations, because there is exactly one position for a value in every case.

8.5 Pattern matching in Observational Type Theory

I guess I'll field this one. I find it a strange question, but that's because of my own particular journey. The short answer is: don't do pattern matching in OTT, or in any kernel type theory. Which is not the same thing as to not do pattern matching ever.

The long answer is basically my PhD thesis.

In my PhD thesis, I show how to elaborate high-level programs written in a pattern matching style into a kernel type theory which has only the induction principles for inductive datatypes and a suitable treatment of propositional equality. The elaboration of pattern matching introduces propositional equations on datatype indices, then solves them by unification. Back then, I was using an intensional equality, but observational equality gives you at least the same power. That is: my technology for elaborating pattern matching (and thus keeping it out of the kernel theory), hiding all the equational piggery-jokery, predates the upgrade to observational equality. The ghastly vlookup you've used to illustrate your point might correspond to the output of the elaboration process, but the input need not be that bad. The nice definition

```
vlookup : Fin n -> Vec X n -> X
vlookup fz     (vcons x xs) = x
vlookup (fs i) (vcons x xs) = vlookup i xs
```

elaborates just fine. The equation-solving that happens along the way is just the same equation-solving that Agda does at the meta-level when checking a definition by pattern matching, or that Haskell does. Don't be fooled by programs like

```
f :: a ~ b => a -> b
f x = x
```

In *kernel* Haskell, that elaborates to some sort of

```
f {q} x = coerce q x
```

but it's not in your face. And it's not in compiled code, either. OTT equality proofs, like Haskell equality proofs, can be erased before computing with *closed* terms.

Digression. To be clear about the status of equality data in Haskell, the GADT

```
data Eq :: k -> k -> * where
  Refl :: Eq x x
```

really gives you

```
Refl :: x ~ y -> Eq x y
```

but because the type system is not logically sound, type safety relies on strict pattern matching on that type: you can't erase `Refl` and you really must compute it and match it at run time, but you *can* erase the data corresponding to the proof of `x~y`. In OTT, the entire propositional fragment is proof-irrelevant for open terms and erasable for closed computation. *End of digression.*

The decidability of equality on this or that datatype is not especially relevant (at least, not if you have uniqueness of identity proofs; if you don't always have UIP, decidability is one way to get it sometimes). The equational problems which show up in pattern matching are on arbitrary *open* expressions. That's a lot of rope. But a machine can certainly decide the fragment which consists of first-order expressions built from variables and fully applied constructors (and that's what Agda does when you split cases: if the constraints are too weird, the thing just barfs). OTT should allow us to push a bit further into the decidable fragments of higher-order unification. If you know `(forall x. f x = t[x])` for unknown `f`, that's equivalent to `f = \ x -> t[x]`.

So, "no pattern matching in OTT" has always been a deliberate design choice, as we always intended it to be an elaboration target for a translation we already knew how to do. Rather, it's a strict upgrade in kernel theory power.

8.6 Provable coherence in OTT

Firstly, thanks for asking about Observational Type Theory. Secondly, what you've done here *does* seem to hang together, even though it has things in different places from where Thorsten Altenkirch, Wouter Swierstra and I put them in our version of the story. Thirdly, it's no surprise (at least not to me) that coherence is derivable, leaving reflexivity the only postulate. That's true of our OTT as well, and Wouter did the proofs in Agda 1, back when we wrote that paper. Proof irrelevance and the shortness of life meant I didn't port his proofs to Agda 2.

If you've missed anything, it's lurking in your remark

> We still need to postulate something to define `subst` and other stuff.

If you have some `P : X -> Set`, some `a, b : X` and some `q : a = b`, you expect to get a function in `P a -> P b`. The "equal functions take equal inputs to equal outputs" formulation gives you that, as `refl P : P = P`, so from `q`, we can deduce `P a = P b`. Your "equal functions take a given input to equal outputs" formulation does not allow you to let `q` bridge the gap from `a` to `b`.

In the presence of `refl` and `subst`, "two equal inputs" amounts to the same thing as "one input used in two places". It seems to me that you've moved the work into whatever else you need to get `subst`. Depending on how lazy your definition of `coerce` is (and *that*'s how you get proof irrelevance), you will need only a postulate.

With your particular formulation, you might even get away with a *homogeneous* value equality. If you're fixing type gaps with coercions rather than equations, you might save yourself some trouble (and maybe get rid of that equation on the domain type in function equality). Of course, in that case, you'd need to think about how to replace the statement of coherence.

We tried quite hard to keep coercion out of the definition of equality, to retain some sort of symmetry, and to keep type equations out of value equations, mostly to have less to think about at one go. It's interesting to see that at least some parts of the construction might get easier with "a thing and its coercion" replacing "two equal things".

8.7 How to solve goals with invalid type equalities in Coq?

tl;dr Cardinality arguments are the only way to show types unequal. You can certainly automate cardinality arguments more effectively with a bit of reflection. If you want to go further, give your types a syntactic representation by constructing a universe, ensuring your proof obligations are framed as syntactic inequality of representations rather than semantic inequality of types.

Isomorphism as Equality

It's widely believed (and there may even be a proof of it somewhere) that Coq's logic is consistent with the axiom that *isomorphic* sets are *propositionally equal*. Indeed, this is a consequence of the Univalence Axiom from Vladimir Voevodsky, which people are having so much fun with at the moment. I must say, it seems very plausible that it is consistent (in the absence of typecase), and that a computational interpretation can be constructed which somehow transports values between equal types by inserting whichever component of the isomorphism is needed at any given moment.

If we assume that such an axiom is consistent, we discover that type inequality in the logic as it stands can hold by only refuting the existence of type isomorphism. As a result, your partial solution is, at least in principle, where it's at. Enumerability is rather key to showing non-isomorphism. I'm not sure what the status of `nat = (nat -> nat)` might be, but it is clear *from outside the system* that every inhabitant of `nat -> nat` has a normal form, and that there are countably many normal forms: it's at least plausible that there are consistent axioms or reflection principles which make the logic more *intensional* and which validate that hypothesis.

Automating Cardinality Arguments

I can see two steps you might take to improve on the present situation. The less radical step is to improve your generic technology for making these cardinality arguments by better use of reflection. You're ideally placed to do so, because in general, you're looking to show that a finite set is distinct from some larger set. Suppose we have some notion of `DList A`, a list of distinct elements of `A`. If you can construct an *exhaustive* `DList A` and a *longer* `DList B`, then you can disprove `A = B`.

There's a lovely definition of DList by *induction-recursion*, but Coq doesn't have induction-recursion. Fortunately, it's one of those definitions we can simulate by careful use of indexing. Forgive my informal syntax, but let's have

```
Parameters
  A   : Set
  d   : A -> A -> bool
  dok : forall x y, d x y = true -> x = y -> False
```

That's `d` for "distinct". If a set already has decidable equality, you can equip it with `d` very easily. A large set can be equipped with an adequate `d` for our purposes with not much work. And actually, that's the crucial step: following the wisdom of the SSReflect team, we take advantage of the smallness of our domain by working with `bool` rather than `Prop`, and make the computer do the heavy lifting.

Now, let us have

```
DListBody : (A -> bool) -> Set
```

where the index is the *freshness test* for the list

```
dnil  : DListBody (const true)       (* any element is fresh for the empty list *)
dsnoc : forall f, (xs : DListBody f) -> (x : A) -> is_true (f x) ->
          DListBody (fun y => f y /\ d x y)
```

And if you like, you can define `DList` wrapping `DListBody` existentially. Perhaps that's actually hiding information we want, though, because to show such a thing exhaustive goes like this:

```
Exhaustive (f : A -> bool)(mylist : DListBody f) = forall x : A, is_false (f x)
```

So if you can write down a DListBody for a finite enumeration, you can prove it exhaustive just by a case analysis with trivial subgoals.

You then need only make the pigeonholing argument once. When you want to disprove an equality between types (assuming you already have suitable candidates for `d`), you exhaustively enumerate the smaller and exhibit a longer list from the larger, and that's it.

Working in a Universe

The more radical alternative is to question why you are getting these goals in the first place, and whether they really mean what you want them to. What are types supposed to be, really? There are multiple possible answers to that question, but it is at least open that they are in some sense "cardinalities". If you want to think of types as being more concrete and syntactic, distinct if they are built by distinct constructions, then you may need to equip types with a more concrete representation by working in a *universe*. You define an inductive datatype of "names" for types, together with the means to decode names as types, then you reframe your development in terms of names. You should find that inequality of names follows by ordinary constructor discrimination.

The snag is that universe constructions can be a bit tricky in Coq, again because induction-recursion is unsupported. It depends heavily on what types you need to consider. Maybe you can define inductively some `U : Set` then implement a recursive decoder `T : U -> Set`. That's certainly plausible for universes of simple types. If you want a universe of dependent types, things get a bit sweatier. You can at least do this much

```
U : Type   (* note that we've gone up a size *)
NAT : U
PI : forall (A : Set), (A -> U) -> U

T : U -> Set
T NAT = nat
T (PI A B) = forall (a : A), T (B a)
```

but note that the domain of `PI` is unencoded in `Set`, not in `U`. The inductive-recursive Agdans can get over this, defining `U` and `T` simultaneously

```
U : Set   (* nice and small *)
NAT : U
PI : forall (A : U), (T A -> U) -> U   (* note the use of T *)

T : U -> Set
T NAT = nat
T (PI A B) = forall (a : T A), T (B a)
```

but Coq won't have that. Again, the workaround is to use indexing. Here the cost is that `U` is inevitably large.

```
U : Set -> Type
NAT : U nat
PI : forall (A : Set)(B : A -> Set),
       U A -> (forall a, U (B a)) -> U (forall a, B a)
```

But you can still get a lot of stuff done with a universe built that way. For example, one can equip such a universe with a computationally effective extensional equality.

8.8 Is it possible to express the type of balanced untagged binary trees on the calculus of constructions?

I don't know the details of Morte, but I have some clues about what's possible in typed lambda-calculi more broadly.

If `Nat` is defined impredicatively, it might be possible to define these trees by iteration.

```
Nat : *
Nat = (x : *) -> (x -> x) -> x -> x
Pair : * -> * -> *
Pair x y = (z : *) -> (x -> y -> z) -> z
Tree : * -> Nat -> *
Tree a n = n * (\ t -> Pair t t) a
```

Of course, to get away with that, I need a *large* elimination. Here, I've casually just taken `* : *`, but that's not safe in general. Inductive definitions admit large eliminations unproblematically: impredicatively encoded datatypes, not so.

But, above, I exploit the fact that the indexing structure of the trees happens to be compatible with that of the `Nat`s which index them, and there is no reason why that should be the case in general. Indices vary in all sorts of wacky ways: it's only those that characterize some sort of "size" that get smaller as we go inward.

Indexed structures do admit a Church encoded presentation. It's just that instead of iterating over a set, we iterate over an indexed set. Here's one way to express it.

```
Tree : * -> Nat -> *
Tree a n = (x : Nat -> *) ->
           (a -> x Z) ->
           ((n : Nat) -> x n -> x n -> x (S n)) ->
           x n
```

It's easy to write some things like

```
leftmost : (a : *) -> (n : Nat) -> Tree a n -> a
leftmost a n t = t (\ _ -> a) (\ a -> a) (\ _ l _ -> l)
```

but

```
leftChild : (a : *) -> (n : Nat) -> Tree a (S n) -> Tree a n
```

is a taller order, requiring some way to inspect or constrain numbers. That's why GHC Haskell has all this stuff about equality, ~.

8.9 Agda-like programming in Coq/Proof General?

Let me teach you one weird trick. It may not be the answer to all your concerns, but it might help, at least conceptually.

Let's implement addition for natural numbers, the latter being given by

```
Inductive nat : Set :=
  | zero : nat
  | suc : nat -> nat.
```

You can try to write addition by tactics, but this happens.

```
Theorem plus' : nat -> nat -> nat.
Proof.
  induction 1.

plus' < 2 subgoals

  ============================
   nat -> nat

subgoal 2 is:
 nat -> nat
```

you can't see what you're doing.

The trick is to look more closely at what you're doing. We can introduce a gratuitously dependent type, cloning `nat`.

```
Inductive PLUS (x y : nat) : Set :=
  | defPLUS : nat -> PLUS x y.
```

The idea is that `PLUS x y` is the type of "the way to compute `plus x y`". We'll need a projection, allowing us to extract the result of such a computation.

```
Theorem usePLUS : forall x y, PLUS x y -> nat.
Proof.
  induction 1.
    exact n.
Defined.
```

Now we're ready to program by proving.

```
Theorem mkPLUS : forall x y, PLUS x y.
Proof.

mkPLUS < 1 subgoal

  ============================
   forall x y : nat, PLUS x y
```

The conclusion of the goal shows us our current left-hand side and context. The analogue of `C-c C-c` in Agda is...

```
  induction x.

mkPLUS < 2 subgoals

  ============================
   forall y : nat, PLUS zero y

subgoal 2 is:
 forall y : nat, PLUS (suc x) y
```

And you can see it has done a case split! Let's knock off the base case.

```
    intros y.
      exact (defPLUS zero y    y).
```

Invoking the constructor of PLUS is like writing an equation. Imagine an = sign before its third argument. For the step case, we need to make a recursive call.

```
    intros y.
      eapply (fun h => (defPLUS (suc x) y    (suc (usePLUS x y h)))).
```

To make the recursive call, we invoke `usePLUS` with the arguments we want, here `x` and `y`, but we abstract over the third argument, which is the explanation of how actually to compute it. We are left with just that subgoal, effectively the termination check.

```
mkPLUS < 1 subgoal

  x : nat
  IHx : forall y : nat, PLUS x y
  y : nat
  ============================
   PLUS x y
```

And now, rather than using Coq's guardedness check, you use...

```
        auto.
```

...which checks that the inductive hypotheses cover the recursive call. We're

```
Defined.
```

We have a worker, but we need a wrapper.

```
Theorem plus : nat -> nat -> nat.
Proof.
  intros x y.
    exact (usePLUS x y (mkPLUS x y)).
Defined.
```

And we're ready to go.

```
Eval compute in (plus (suc (suc zero)) (suc (suc zero))).

Coq <      = suc (suc (suc (suc zero)))
     : nat
```

You *have* an interactive construction tool. You *can* game it to show you the pertinent details of the problem you're solving by making types more informative. The resulting proof script...

```
Theorem mkPLUS : forall x y, PLUS x y.
Proof.
  induction x.
    intros y.
      exact             (defPLUS zero    y    y).
    intros y.
      eapply (fun h =>  (defPLUS (suc x) y    (suc (usePLUS x y h)))).
        auto.
Defined.
```

...is explicit about the program it constructs. You can see that's defining addition.

If you automate this setup for program construction, then layer on an interface showing you the program you're building and the key problem-simplifying tactics, you get a funny little programming language called Epigram 1.