Functional Pearl: Getting a Quick Fix on Comonads

Kenneth Foner
University of Pennsylvania, USA [*]
kfoner@seas.upenn.edu

Abstract

A piece of functional programming folklore due to Piponi provides Löb's theorem from modal provability logic with a computational interpretation as an unusual fixed point. Interpreting modal necessity as an arbitrary Functor in Haskell, the "type" of Löb's theorem is inhabited by a fixed point function allowing each part of a structure to refer to the whole.

However, Functor's logical interpretation may be used to prove Löb's theorem only by relying on its implicit functorial strength, an axiom not available in the provability modality. As a result, the well known loeb fixed point "cheats" by using functorial strength to implement its recursion.

Rather than Functor, a closer Curry analogue to modal logic's Howard inspiration is a closed (semi-)comonad, of which Haskell's ComonadApply typeclass provides analogous structure. Its computational interpretation permits the definition of a novel fixed point function allowing each part of a structure to refer to its own context within the whole. This construction further guarantees maximal sharing and asymptotic efficiency superior to loeb for locally contextual computations upon a large class of structures. With the addition of a distributive law, closed comonads may be composed into spaces of arbitrary dimensionality while preserving the performance guarantees of this new fixed point.

From these elements, we construct a small embedded domain-specific language to elegantly express and evaluate multidimensional "spreadsheet-like" recurrences for a variety of cellular automata.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; F.3.3 [*Studies of Program Constructs*]: Functional Constructs; F.4.1 [*Mathematical Logic and Formal Languages*]: Modal Logic

Keywords Haskell, closed comonads, modal logic, spreadsheets

1. Introduction

In 2006, Dan Piponi wrote a blog post, "From Löb's Theorem to Spreadsheet Evaluation," [18] the result of which has become a curious piece of folklore in the Haskell community. He writes that one way to write code is to "pick a theorem, find the corresponding type, and find a function of that type." As an exercise in this style of Curry-Howard spelunking, he picks Löb's theorem from the modal logic of provability and attempts to derive a Haskell program to which it corresponds.

Several years later, I came across his loeb function myself and was fascinated equally by its claimed Curry-Howard connection as by its "almost perverse" reliance on laziness (to borrow a phrase from Done [5]). As I explored loeb's world, however, something increasingly seemed to be amiss. Though the term Piponi constructs has a type which mirrors the statement of Löb's theorem, the program inhabiting that type (that is, the proof witnessing the theorem) is built from a collection of pieces which don't necessarily correspond to the available and necessary logical components used to prove Löb's theorem in the land of Howard.

Piponi is clear that he isn't sure if his loeb term accurately witnesses Löb's theorem. I take his initial exploration as an opportunity to embark on a quest for a closer translation of Löb's theorem, from the foreign languages of modal logic to the familiar tongue of my homeland: Haskell. This journey will lead us to find a new efficient comonadic fixed point which inhabits a more accurate computational translation of Löb's theorem (§2–11). When combined with the machinery of comonadic composition, we'll find that this fixed point enables us to concisely express "spreadsheet-like" recurrences (§13) which can be generalized to apply to an n-dimensional infinite Cartesian grid (§14–15). Using these pieces, we can build a flexible embedded language for concisely describing such multi-dimensional recurrences (§16–18) such as the Fibonacci sequence, Pascal's triangle, and Conway's game of life (§19).

2. Modal Logic, Squished to Fit in a Small Box

The first step of the journey is to learn some of the language we're attempting to translate: modal logic. Modal logics extend ordinary classical or intuitionistic logic with an additional operator, $\Box$, which behaves according to certain extra axioms. The choice of these axioms permits the definition of many different modal systems.

Martin Hugo Löb's eponymous theorem is the reason this paper exists at all. In the language of modal logic, it reads:

$$\Box(\Box P \to P) \to \Box P$$

If we read $\Box$ as "is provable," this statement tells us that, for some statement P, if it's provable that P's provability implies P's truth, then P itself is provable.

Conventionally, Löb's theorem is taken as an additional axiom in some modal logic [22], but in some logics which permit a *modal fixed point* operation, it may be derived rather than taken as axiomatic [14]. In particular, Löb's theorem is provable in the K4 system of modal logic extended with modal fixed points [15]. We'll return to this proof later. First, we need to understand Piponi's derivation for a Löb-inspired function: it's fun and useful, but further, understanding what gets lost in translation brings us closer to a more faithful translation of the theorem into Haskell.

[*] The work in this paper was performed in part at Brandeis University, USA.

3. A Bridge Built on a Functor

To carry Löb's theorem across the river between logic and computation, we first need to build a bridge: an interpretation of the theorem as a type. If we read implication as Haskell's function arrow, the propositional variable P must necessarily translate to a type variable of kind $*$ and the modal $\Box$ operator must translate to a type of kind $* \to *$. Thus, the type signature for Löb's computational equivalent must take the form:

$$\mathsf{loeb} :: (\_\ f) \Rightarrow f\ (f\ a \to a) \to f\ a$$

There is an unspecified constraint $(\_\ f)$ in this signature because we're looking for something of maximal generality, so we want to leave f and a polymorphic – but then we need to choose some constraint if we want any computational traction to build such a term.

Piponi fills the unknown constraint by specifying that f be a Functor. Under this assumption, he constructs the loeb function.[1]

$$\begin{aligned}&\mathsf{loeb} :: \mathsf{Functor}\ f \Rightarrow f\ (f\ a \to a) \to f\ a\\ &\mathsf{loeb}\ \mathit{ffs} = \mathsf{fix}\ (\lambda \mathit{fa} \to \mathsf{fmap}\ (\$\ \mathit{fa})\ \mathit{ffs})\end{aligned}$$

It's easiest to get an intuition for this function when f is taken to be a "container-like" Functor. In this case, loeb takes a container of functions, each from a whole container of as to a single a, and returns a container of as where each element is the result of the corresponding function from the input container applied to the whole resultant container. More succinctly: loeb finds the unique fixed point (if it exists) of a system of recurrences which refer to each others' solutions by index within the whole set of solutions. Even more succinctly: loeb looks like evaluating a spreadsheet.

Instantiating f as $[\,]$, we find:

$$\mathsf{loeb}\ [\mathsf{length}, (!!\ 0), \lambda x \to x\ !!\ 0 + x\ !!\ 1] \equiv [3, 3, 6]$$

As expected: the first element is the length of the whole list (which may be computed without forcing any of its elements); the second element is equal to the first element; the last element is equal to the sum of the first two. Not every such input has a unique fixed point, though. For instance:

$$\mathsf{loeb}\ [\mathsf{length}, \mathsf{sum}] \equiv [2, \bot]$$

We cannot compute the second element because it is defined in terms of itself, strictly. Any part of loeb's input which requires an element at its own position to be strictly evaluated will return $\bot$. More generally, any elements which participate in such a cycle of any length will return $\bot$ when evaluated with loeb. This is necessary, as no unique solution exists to such a recurrence.

Instantiating loeb with other functors yields other similar fixed points. In particular, when f is $(\to)\ r$, the "reader" functor, loeb is equivalent to a flipped version of the ordinary fixed point function:

$$\begin{aligned}&\mathsf{loeb}_{(\to)\ r} :: (r \to (r \to a) \to a) \to r \to a\\ &\mathsf{loeb}_{(\to)\ r} \equiv \mathsf{fix} \circ \mathsf{flip}\end{aligned}$$

(See Appendix A for elaboration.)

4. Is the Box Strong Enough?

Despite its intriguing functionality, the claim that loeb embodies a constructive interpretation of Löb's theorem – now relatively widespread – doesn't hold up to more detailed scrutiny. In particular, the implementation of loeb uses f implicitly as a *strong functor*, and $\Box$ emphatically is not one.

In order to see why that's a problem, let's take a closer look at the rules for the use of $\Box$. The K4 system of modal logic, as mentioned earlier, allows us to prove Löb's theorem once we add to it *modal* fixed points. This ought to ring some Curry-Howard bells, as we're looking for a typeclass to represent $\Box$ which, in conjunction with *computational* fixed points, will give us "Löb's program." The axioms for the K4 system are:

$$\begin{aligned}&\vdash \Box A \to \Box\Box A && \text{axiom (4)}\\ &\vdash \Box(A \to B) \to \Box A \to \Box B && \text{distribution axiom}\end{aligned}$$

Informally, axiom (4) means that if A is provable with no assumptions then it's provably provable, and the distribution axiom tells us that we're allowed to use *modus ponens* inside of the $\Box$ operator.

Additionally, all modal logics have the "necessitation" rule of inference that $\vdash A$ lets us conclude that $\vdash \Box A$ [8]. An important thing to notice about this rule is that it *doesn't* allow us to derive $\vdash A \to \Box A$, though it might seem like it. Significantly, there are no assumptions to the left of the turnstile in $\vdash A$, so we can lift A into the $\Box$ modality only if we can prove it under no assumptions. If we try to derive $\vdash A \to \Box A$, we might use the variable introduction (axiom) rule to get $A \vdash A$, but then get stuck, because we don't have the empty context required by the necessitation rule to then lift A into the $\Box$.

Nevertheless, the necessitation and distribution axioms are sufficient to show that $\Box$ is a functor: if we have some theorem $\vdash A \to B$, we can lift it to $\vdash \Box(A \to B)$ by the necessitation rule, and then distribute the implication to get $\vdash \Box A \to \Box B$. Thus, in modal logic, if we have $\vdash A \to B$ then we have $\vdash \Box A \to \Box B$, and so whatever our translation of $\Box$, it should be *at least* a Functor.

That being said, loeb is secretly using its Functor as more than a mere functor – it uses it as a *strong* functor. In category theory, a functor F is strong with respect to some product operation $(\times)$ if we can define the natural transformation $F(a) \times b \to F(a \times b)$ [12]. In Haskell, *every* Functor is strong because we can form closures over variables in scope to embed them in the function argument to fmap. As such, we may flex our functorial muscles any time we please:[2,3]

$$\begin{aligned}&\mathsf{flex} :: \mathsf{Functor}\ f \Rightarrow f\ a \to b \to f\ (a, b)\\ &\mathsf{flex}\ \mathit{fa}\ b = \mathsf{fmap}\ (, b)\ \mathit{fa}\end{aligned}$$

We can rephrase loeb in a partially point-free style to elucidate how it depends upon functorial strength:

$$\begin{aligned}&\mathsf{loeb} :: \mathsf{Functor}\ f \Rightarrow f\ (f\ a \to a) \to f\ a\\ &\mathsf{loeb}\ \mathit{ffs} \equiv \mathsf{fix}\ (\mathsf{fmap}\ (\mathsf{uncurry}\ (\$)) \circ \mathsf{flex}\ \mathit{ffs})\end{aligned}$$

(See Appendix B for elaboration.)

While the loeb function needs to flex, K4's $\Box$ modality does not have the strength to match it. We're not permitted to sneak an arbitrary "unboxed" value into an existing box – and nor should we be. Given a single thing in a box, $\Box A$, and a weakening law, $\Box(A \times B) \to \Box B$, functorial strength lets us fill the $\Box$ with whatever we please – and thus $\Box$ with functorial strength is no longer a modality of provability, for it now proves everything which is true. In Haskell, this operation constitutes "filling" an a-filled functor with an arbitrary other value:

$$\begin{aligned}&\mathsf{fill} :: \mathsf{Functor}\ f \Rightarrow f\ a \to b \to f\ b\\ &\mathsf{fill}\ \mathit{fa} = \mathsf{fmap}\ \mathsf{snd} \circ \mathsf{flex}\ \mathit{fa}\end{aligned}$$

You might know fill by another name: Data.Functor's ($>).

[1] For pedagogical reasons, we rephrase this using an explicit fixpoint; Piponi's equivalent original is: $\mathsf{loeb}\ \mathit{ffs} = \mathit{fa}$ **where** $\mathit{fa} = \mathsf{fmap}\ (\$\ \mathit{fa})\ \mathit{ffs}$.

[2] This definition of flex is curried, but since the product we're interested in is $(,)$, uncurry flex matches the categorical presentation exactly. We present it curried as this aids in the idiomatic presentation of later material.

[3] In the definition of flex, we make use of GHC's TupleSections syntactic extension to clarify the presentation. In this notation, $(a,) \equiv \lambda b \to (a, b)$ and $(, b) \equiv \lambda a \to (a, b)$.

While making a true constructive interpretation of Löb's theorem in Haskell, we have to practice a kind of asceticism: though functorial strength is constantly available to us, we must take care to eschew it from our terms. To avoid using strength, we need to make sure that every argument to fmap is a *closed term*. This restriction is equivalent to the necessitation rule's stipulation that its argument be provable under no assumptions. Haskell's type system can't easily enforce such a limitation; we must bear that burden of proof ourselves.[4]

Piponi chooses to translate $\Box$ to Functor as an initial guess. "But what should $\Box$ become in Haskell?" he asks. "We'll defer that decision until later and assume as little as possible," he decides, assigning the unassuming Functor typeclass to the modal $\Box$. Given that Functor isn't expressive enough to keep us from needing its forbidden strength, a closer translation of the theorem requires us to find a different typeclass for $\Box$ – one with a little more *oomph*.

5. Contextual Computations, Considered

Recall that loeb computes the solution to a system of recurrences where each part of the system can refer to the solution to the whole system (e.g. each element of the list can be lazily defined in terms of the whole list). Specifically, individual functions $f\ a \to a$ receive via loeb a view upon the "solved" structure $f\ a$ which is the same for each such viewing function.

A structure often described as capturing computations in some context is the comonad, the monad's less ostentatious dual. In Haskell, we can define comonads as a typeclass:

```
class Functor w ⇒ Comonad w where
  extract   :: w a → a                    -- dual to return
  duplicate :: w a → w (w a)              -- dual to join
  extend    :: (w a → b) → w a → w b      -- dual to (=<<)
```

For a Comonad w, given a $w\ a$, we can extract an a. (Contrast this with a Monad m, where we can return an a into an $m\ a$.) We can also duplicate any $w\ a$ to yield a doubly-nested $w\ (w\ a)$. (Contrast this with a Monad m, where we can join a doubly-nested $m\ (m\ a)$ into an $m\ a$).

Comonads also follow laws dual to those of monads:

$$\text{extract} \circ \text{duplicate} \equiv \text{id} \quad (1)$$

$$\text{fmap extract} \circ \text{duplicate} \equiv \text{id} \quad (2)$$

$$\text{duplicate} \circ \text{duplicate} \equiv \text{fmap duplicate} \circ \text{duplicate} \quad (3)$$

Since duplicate has type $w\ a \to w\ (w\ a)$, these laws tell us: we can eliminate the *outer* layer (1) or the *inner* layer (2) from the doubly-nested result of duplicate, and what we get back must be the same thing which was originally duplicated; also, if we duplicate the result of duplicate, this final value cannot depend upon whether we call duplicate for a second time on the *outer* or *inner* layer resulting from the first call to duplicate.[5]

The monadic operations ($\gg=$) and join may be defined in terms of each other and fmap; the same is true of their duals, extend and duplicate. In Control.Comonad, we have the following default definitions, so defining a Comonad requires specifying only extract and one of duplicate or extend:

```
duplicate = extend id
extend f  = fmap f ∘ duplicate
```

[4]The typing rules for Cloud Haskell's static special form depend on whether a term is closed or not – only certain sorts of closed terms may be serialized over the network – but these rules are specific to this particular feature and cannot presently be used for other purposes [6].

[5]These laws also show us that just as a monad is a monoid in the category of endofunctors, a comonad is a comonoid in the category of endofunctors: (1) and (2) are left and right identities, and (3) is co-associativity.

6. A Curious Comonadic Connection

From now, let's call the hypothetical computational Löb's theorem "lfix" (for Löb-fix). We'll only use the already-popularized "loeb" to refer to the function due to Piponi.

Comonads seem like a likely place to search for lfix, not only because their intuition in terms of contextual computations evokes the behavior of loeb, but also because the type of the comonadic duplicate matches axiom (4) from K4 modal logic.

$$\vdash \Box P \to \Box\Box P \ \ \text{axiom (4)} \ \cong \ \text{duplicate} :: w\ a \to w\ (w\ a)$$

Before going on, we'd do well to notice that the operations from Comonad are not a perfect match for K4 necessity. In particular, Comonad has nothing to model the distribution axiom, and extract has a type which does not correspond to a derivable theorem in K4. We'll reconcile these discrepancies shortly in §10.

That being said, if we browse Control.Comonad, we can find two fixed points with eerily similar types to the object of our quest.

```
lfix ::            (_ w) ⇒ w (w a → a) → w a
cfix :: Comonad w  ⇒       (w a → a) → w a
wfix :: Comonad w  ⇒     w (w a → a) →   a
```

The first of these, cfix, comes from Dominic Orchard [17].

```
cfix :: Comonad w ⇒ (w a → a) → w a
cfix f = extend f (cfix f)
```

It's close to the Löb signature, but not close enough: it doesn't take as input a w of anything; it starts with a naked function, and there's no easy way to wrangle our way past that.

The second, wfix, comes from David Menendez [16].

```
wfix :: Comonad w ⇒ w (w a → a) → a
wfix f = extract f (extend wfix f)
```

It, like cfix, is missing a w somewhere as compared with the type of lfix, but it's missing a w on the type of its *result* – and we can work with that. Specifically, using extend :: $(w\ a \to b) \to w\ a \to w\ b$, we can define:

```
possibility :: Comonad w ⇒ w (w a → a) → w a
possibility = extend wfix
```

In order to test out this possibility, we first need to build a comonad with which to experiment.

7. Taping Up the Box

In the original discussion of loeb, the most intuitive and instructive example was that of the list functor. This exact example won't work for lfix: lists don't form a comonad because there's no way to extract from the empty list.

Non-empty lists do form a comonad where extract = head and duplicate = init ∘ tails, but this type is too limiting. Because extend f = fmap f ∘ duplicate, the context seen by any extended function in the non-empty-list comonad only contains a rightwards view of the structure. This means that all references made in our computation would have to point rightwards – a step back in expressiveness from Piponi's loeb, where references can point anywhere in the input Functor.

From Gérard Huet comes the *zipper* structure, a family of data types each of which represent a single "focus" element and its context within some (co)recursive algebraic data type [9]. Not by coincidence, every zipper induces a comonad [1]. A systematic way of giving a comonadic interface to a zipper is to let extract yield the value currently in focus and let duplicate "diagonalize" the zipper, such that each element of the duplicated result is equal to the "view" from its position of the whole of the original zipper.

This construction is justified by the second comonad law: fmap extract ∘ duplicate ≡ id. When we diagonalize a zipper, each inner-zipper element of the duplicated zipper is focused on the element which the original zipper held at that position. Thus, when we fmap extract, we get back something identical to the original.

A particular zipper of interest is the *list zipper* (sometimes known as the *pointed list*), which is composed of a focus and a context within a list – that is, a pair of lists representing the elements to the left and right of the focus.

Though list zippers are genuine comonads with total comonadic operations, there are other desirable functions which are impossible to define for them in a total manner. The possible finitude of the context lists means that we have to explicitly handle the "edge case" where we've reached the edge of the context.

To be lazy, in both senses of the word, we can eliminate edges entirely by placing ourselves amidst an endless space, replacing the possibly-finite lists in the zipper's context with definitely-infinite streams. The Stream datatype is isomorphic to a list without the *nil* constructor, thus making its infinite extent (if we ignore $\bot$) certain.[6]

```
data Stream a = Cons a (Stream a)
```

By crossing two Streams and a focus element, we get a zipper into a both-ways-infinite stream. We'll call this datatype a Tape after the bidirectionally infinite tape of a Turing machine.[7]

```
data Tape a = Tape {viewL :: Stream a
                   , focus  :: a
                   , viewR :: Stream a
                   } deriving Functor
```

We can move left and right in the Tape by grabbing onto an element from the direction we want to move and pulling ourselves toward it, like climbing a rope.

```
moveL, moveR :: Tape a → Tape a
moveL (Tape (Cons l ls) c rs)  = Tape ls l (Cons c rs)
moveR (Tape ls c (Cons r rs)) = Tape (Cons c ls) r rs
```

Notice that moveL and moveR are total, in contrast to their equivalents on pointed finite lists.

Tape forms a Comonad, whose instance we can define using the functionality outlined above, as well as an iteration function for building Tapes.

```
iterate :: (a → a) → (a → a) → a → Tape a
iterate prev next seed =
  Tape (Stream.iterate prev (prev seed))
       seed
       (Stream.iterate next (next seed))

instance Comonad Tape where
  extract   = focus
  duplicate = iterate moveL moveR
```

As with other comonads derived from zippers, duplicate for Tapes is a kind of diagonalization.

8. Taking possibility for a Test Drive

With our newly minted Tape comonad in hand, it's time to kick the tires on this new possibility. To start, let's first try something really simple: counting to 10000.

```
tenKilo = print ∘ Stream.take 10000 ∘ viewR ∘ possibility $
  Tape (Stream.repeat (const 0))        -- zero left of origin
       (const 0)                        -- zero at origin
       (Stream.repeat                   -- right of origin:
         (succ ∘ extract ∘ moveL))      -- 1 + leftward value
```

Notice that the pattern of recursion in the definition of this Tape would be impossible using the non-empty-list comonad, as elements extending rightward look to the left to determine their values.

Enough talk – let's run it!. . . . .
. . . . Unfortunately, that sets our test drive off to quite the slow start. On my machine,[8] this program takes a full 13 seconds to print the number 10000. To understand better how abysmally slow that is, let's compare it to a naïve list-based version of what we'd hope is roughly the same program:

```
listTenKilo = print $ Prelude.take 10000 [1 ..]
```

This program takes almost no time at all – as well it should! The Tape-based counting program is slower than the list-based one by a factor of around 650. Your results may vary a bit of course, but not dramatically enough to call this kind of performance acceptable for all but the most minuscule applications.

This empirical sluggishness was how I first discovered the inadequacy of possibility for efficient computation, but we can rigorously justify why it necessarily must be so slow and mathematically characterize just how slow that is.

9. Laziness *sans* Sharing ≅ Molasses in January

Recall the definitions of wfix and possibility:

```
wfix :: Comonad w ⇒ w (w a → a) → a
wfix w = extract w (extend wfix w)

possibility :: Comonad w ⇒ w (w a → a) → w a
possibility = extend wfix
```

Rephrasing wfix in terms of an explicit fixed-point and subsequently inlining the definitions of fix and extend, we see that

```
wfix ≡ wf where wf =
         λw → extract w (fmap wf (duplicate w))
```

In this version of the definition, it's more clear that wfix does not share the eventual fixed point it computes in recursive calls to itself. The only sharing is of wfix itself as a function.

In practical terms, this means that when evaluating possibility, every time a particular function $w\ a \to a$ contained in the input $w\ (w\ a \to a)$ makes a reference to another part of the result structure $w\ a$, the entire part of the result demanded by that function must be recomputed. In the counting benchmark above, this translates into an extra linear factor of time complexity in what should be a linear algorithm.

Worse still, in recurrences with a branching factor higher than one, this lack of sharing translates not merely into a linear cost, but into an exponential one. For example, each number in the Fibonacci sequence depends upon its *two* predecessors, so the following program runs in time exponential to the size of its output.[9]

[6] This datatype is defined in the library Data.Stream.

[7] For brevity, we use the GHC extension DeriveFunctor to derive the canonical Functor instance for Tape.

[8] Compiled using GHC 7.8.3 with -O2 optimization, run on OS X 10.10 with a 2 GHz quad-core processor.

[9] The following code uses **do**-notation with the reader monad $(\to)\ r$ to express the function summing the two elements left of a Tape's focus.

```
slowFibs = print ∘ Stream.take n ∘ viewR ∘ possibility $
  Tape (Stream.repeat (const 0))
       (const 1)
       (Stream.repeat $ do
         a ← extract ∘ moveL
         b ← extract ∘ moveL ∘ moveL
         return $ a + b)
  where n = 30   -- increase this number at your own peril
```

This is just as exponentially slow as a naïve Fibonacci calculation in terms of explicit recursion with no memoization.

10. Finding Some Closure

It would be disappointing if the intractably slow possibility were truly the most efficient computational translation of Löb's theorem. In order to do better, let's first take a step back to revisit the guiding Curry-Howard intuition which brought us here.

Since Löb's theorem is provable in K4 modal logic augmented with modal fixed points, we tried to find a typeclass which mirrored K4's axioms, which we could combine with the fix combinator to derive our lfix. We identified Comonad's duplicate with axiom (4) of K4 logic and derived a fixed point which used this, the comonadic extract, and the functoriality (without needing strength) of $\Box$. As mentioned before, there's something fishy about this construction.

Firstly, K4 modal logic has nothing which corresponds to the comonadic extract :: Comonad $w \Rightarrow w\ a \rightarrow a$. When we add to K4 the axiom $\Box A \rightarrow A$ (usually called (T) in the literature) to which extract corresponds, we get a different logic; namely, S4 modal logic. Unfortunately, axiom (T) when combined with Löb's theorem leads to inconsistency: necessitation upon (T) gives us $\forall A. \Box(\Box A \rightarrow A)$, then Löb's theorem gives us $\forall A. \Box A$, and finally, applying (T) yields $\forall A. A$: a logical inconsistency.

As a result, no good translation of Löb's theorem can use extract or anything derived from it, as a proof using inconsistent axioms is no proof at all. Notably, the wfix we used in defining possibility must be derived in part from extract. If its slow performance didn't already dissuade us, this realization certainly disqualifies possibility from our search.[10]

The second mismatch between K4 and Comonad is the latter's absence of something akin to the distribution axiom: Comonad gives us no equivalent of $\Box(A \rightarrow B) \rightarrow \Box A \rightarrow \Box B$.

The distribution axiom should look familiar to Haskellers. Squint slightly, and it looks identical to the signature of the Applicative "splat" operator $(\circledast)$.[11] Compare:

$$(\circledast) :: \mathsf{Applicative}\ f \Rightarrow f\ (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b$$
$$\text{distribution axiom:} \quad \Box(A \rightarrow B) \rightarrow \Box A \rightarrow \Box B$$

At first glance, this seems quite promising – we might hastily conclude that the constraint matching the modal $\Box$ is that of an Applicative Comonad. But while $(\circledast)$ is all well and good, the other method of Applicative ruins the deal: pure has exactly the type we can't allow into the $\Box$: Applicative $f \Rightarrow a \rightarrow f\ a$, which corresponds to the implication $A \rightarrow \Box A$, which we've been trying to escape from the start!

Hope need not be lost: another algebraic structure fits the bill perfectly: the *closed comonad*. In Haskell, the ComonadApply typeclass models the closed comonad structure (at least up to values containing $\bot$).[12]

```
class Comonad w ⇒ ComonadApply w where
  (⊛) :: w (a → b) → w a → w b
```

ComonadApply lacks a unit like Applicative's pure, thus freeing us from unwanted pointedness.

A brief detour into abstract nonsense: the structure implied by ComonadApply is, to quote the documentation, "a strong lax symmetric semi-monoidal comonad on the category Hask of Haskell types," [10]. A monoidal comonad is equivalent to a closed comonad in a Cartesian-closed category (such as that of Haskell types), and this structure is exactly what's necessary to categorically model intuitionistic S4 modal logic [3][2]. If we avoid using extract, we find ourselves where we wanted to be: in the world of intuitionistic K4 modal logic.

11. Putting the Pieces Together

Now that we've identified ComonadApply as a slightly-overpowered match for the $\Box$ modality, it's time to put all the pieces together. Because Functor is a superclass of Comonad, which itself is a superclass of ComonadApply, we have all the methods of these three classes at our disposal. Somehow, we need to assemble our function lfix from a combination of these specific parts:[13]

```
fix       :: (a → a) → a
fmap      :: Functor w ⇒ (a → b) → w a → w b
duplicate :: Comonad w ⇒ w a → w (w a)
(⊛)       :: ComonadApply w ⇒ w (a → b) → w a → w b
```

One promising place to start is with fix. We know from experience with wfix that we'll want to share the result of lfix in some recursive position, or we'll pay for it asymptotically later. If we set up our function as below, we can guarantee that we do so.

```
lfix :: ComonadApply w ⇒ w (w a → a) → w a
lfix f = fix _
```

GHC will gladly infer for us that the hole (_) above has type $w\ a \rightarrow w\ a$.[14] If we didn't have typed holes, we could certainly see why for ourselves by "manual type inference": fix, specialized from $(a \rightarrow a) \rightarrow a$ to return a result of type $w\ a$, must take as input something of type $w\ a \rightarrow w\ a$. We can elide such chains of reasoning below – I prefer to drive my type inference on automatic.

We might guess that, like in cfix, we have to duplicate our input in each recursive call:

```
lfix f = fix (_ ∘ duplicate)
```

Further, since we haven't yet used f in the right hand side of our definition, a good guess for its location is as an argument to the unknown (_).

```
lfix f = fix (_ f ∘ duplicate)
```

This new hole is of the type $w\ (w\ a \rightarrow a) \rightarrow w\ (w\ a) \rightarrow w\ a$, a specialization of the type of the ComonadApply operation, (⊛). Plugging it in finally yields the lfix for which we've been searching.

```
lfix :: ComonadApply w ⇒ w (w a → a) → w a
lfix f = fix ((f ⊛) ∘ duplicate)
```

[10] What we really need is a semi-comonad – that is, a comonad without extract. The programming convenience of extract, however, makes it worthwhile to stick with Comonad, but with a firm resolution not to use extract in deriving our lfix. We discuss this more comprehensively in §12.

[11] What is typeset here as $(\circledast)$ is spelled in ASCII Haskell as (<*>).

[12] What is typeset here as (⊛) is spelled in ASCII Haskell as (<@>).

[13] Note the conspicuous absence of extract, as our faithfulness to K4 prevents us from using it in our translation of the proof.

[14] This "typed hole" inference is present in GHC versions 7.8 and greater.

12. Consulting a Road Map

The term lfix checks many boxes on the list of criteria for a computational interpretation of Löb's theorem. Comparing it against a genuine proof of the theorem in its native habitat due to Mendelson [15] shows that lfix makes use of a nearly isomorphic set of prerequisites: duplicate (axiom 4), $(\circledast)$ (distribution axiom), and fix, which as used here roughly corresponds to the role of modal fixed points in the proof.[15]

Mendelson doesn't prove Löb's theorem in the exact form we've seen; rather, he proves $(\Box P \to P) \to P$. This statement, though, implies the truth of the version we've seen: to raise everything "up a box," we need to apply the necessitation and distribution rules to derive our version, $\Box(\Box P \to P) \to \Box P$.

A quick summary of modal fixed points: If a logic has modal fixed points, that means we can take any formula with one free variable $F(x)$ and find some other formula Ψ such that $F(\Box\Psi) \iff \Psi$. The particular modal fixed point required by Mendelson's proof is of the form $(\Box\mathscr{L} \to \mathscr{C}) \iff \mathscr{L}$, for some fixed $\mathscr{C}$.

There is a slight discrepancy between modal fixed points and Haskell's value-level fixed points. The existence of a modal fixed point is an assumption about recursive propositions – and thus, corresponds to the existence of a certain kind of recursive type. By contrast, the Haskell term fix expresses recursion at the value level, not the type level. This mismatch is, however, only superficial. By Curry's fixed point theorem, we know that value-level general recursion is derivable from the existence of recursive types. Similarly, Mendelson's proof makes use of a restricted family of recursive propositions (propositions with modal fixed points) to give a restricted kind of "modal recursion." By using Haskell's fix rather than deriving analogous functionality from a type-level fixed point, we've used a mild sleight of hand to streamline this step in the proof. Consider the following term (and its type), which when applied to $(f \circledast)$ yields our lfix:

```
λg → fix (g ∘ duplicate)
    :: Comonad w ⇒ (w (w a) → w a) → w a
```

Because our lfix is lifted up by a box from Mendelson's proof, the fixed point we end up taking matches the form $\Box\mathscr{L} \iff (\Box\Box\mathscr{L} \to \Box\mathscr{L})$, which lines up with the type of the term above.

In Haskell, not every expressible (modal) fixed point has a (co)terminating solution. Just as in the case of loeb, it's possible to use lfix to construct cyclic references, and just as in the case of loeb, any cycles result in $\bot$. This is exactly as we would expect for a constructive proof of Löb's theorem: if the fixed point isn't uniquely defined, the premises of the proof (which include the existence of a fixed point) are bogus, and thus we can prove falsity ($\bot$).

Another way of seeing this: the hypothesis of Löb's theorem $\Box(\Box P \to P)$ is effectively an assumption that axiom (T) holds not in general, but just for this one proposition P. Indeed, the inputs to lfix which yield fully productive results are precisely those for which we can extract a non-bottom result from any location (i.e. those for which axiom (T) always locally holds). Any way to introduce a cycle to such a recurrence must involve extract or some specialization of it – without it, functions within the recurrence can only refer to properties of the whole "container" (such as length) and never to specific other elements of the result. Just as we can (mis)use Functor as a strong functor (noted in §4), we can (mis)use Comonad's extract to introduce non-termination. In both cases, the type system does not prevent us from stepping outside the laws we've given ourselves; it's our responsibility to ensure some kinds of safety – and if we do, lfix's logical analogy does hold. Vitally, the "proof" witnessed by lfix is itself consistent: it does not use extract. Inconsistency is only (potentially) introduced when the programmer chooses to externally combine lfix and extract.

[15]Notably, we don't need to use the necessitation rule in this proof, which means that we can get away with a *semi*-monoidal comonad ala ComonadApply.

13. A Zippier Fix

As a result of dissecting Mendelson's proof, we've much greater confidence this time around in our candidate term lfix and its closer fidelity to the modality. In order to try it out on the previous example, we first need to give an instance of ComonadApply for the Tape comonad. But what should that instance be?

The laws which come with the ComonadApply class, those of a symmetric semi-monoidal comonad, are as follows:

$$(\circ) \mathbin{\$} u \circledast v \circledast w \equiv u \circledast (v \circledast w) \quad (1)$$

$$\text{extract } (p \circledast q) \equiv \text{extract } p \text{ (extract } q) \quad (2)$$

$$\text{duplicate } (p \circledast q) \equiv (\circledast) \mathbin{\$} \text{duplicate } p \circledast \text{duplicate } q \quad (3)$$

In the above, we use the infix fmap operator: $(\$) = \text{fmap}$.[16]

Of particular interest is the third law, which enforces a certain structure on the $(\circledast)$ operation. Specifically, the third law is the embodiment for ComonadApply of the *symmetric* in "symmetric semi-monoidal comonad." It enforces a distribution law which can only be upheld if $(\circledast)$ is idempotent with respect to cardinality and shape: if some r is the same shape as $p \circledast q$, then $r \circledast p \circledast q$ must be the same shape as well [19]. For instance, the implementation of Applicative's $(\circledast)$ for lists – a computation with the shape of a Cartesian product and thus an inherent asymmetry – would fail the third ComonadApply law if we used it to implement $(\circledast)$ for the (non-empty) list comonad.[17]

We functional programmers have a word for operations like this: $(\circledast)$ must have the structure of a *zip*. It's for this reason that Uustalu and Vene define a ComonadZip class, deriving the equivalent of ComonadApply from its singular method czip :: (ComonadZip w) $\Rightarrow w\ a \to w\ b \to w\ (a, b)$ [20]. The czip and $(\circledast)$ operations may be defined only in terms of one another and fmap – their two respective classes are isomorphic. Instances of ComonadApply generally have fewer possible law-abiding implementations than do those of Applicative because they are thus constrained to be "zippy."

This intuition gives us the tools we need to write a proper instance of ComonadApply first for Streams. . .[18]

```
instance ComonadApply Stream where
  Cons x xs ⊛ Cons x′ xs′ =
    Cons (x x′) (xs ⊛ xs′)
```

. . . and then for Tapes, relying on the Stream instance we just defined to properly zip the component Streams of the Tape.

```
instance ComonadApply Tape where
  Tape ls c rs ⊛ Tape ls′ c′ rs′ =
    Tape (ls ⊛ ls′) (c c′) (rs ⊛ rs′)
```

With these instances in hand (or at least, in scope), we can run the trivial count-to-10000 benchmark again. This time, it runs in linear time with only a small constant overhead beyond the naïve list-based version.

In this new world, recurrences with a higher branching factor than one no longer exhibit an exponential slowdown, instead running quickly by exploiting the sharing lfix guarantees on container-like

[16]What is typeset here as $(\$)$ is spelled in ASCII Haskell as `(<$>)`.

[17]For a type instantiating both Applicative and ComonadApply, the equivalence $(\circledast) \equiv (\circledast)$ should always hold [10].

[18]We rely here on a Comonad instance for Streams analogous to that for non-empty lists given in §7: extract = head and duplicate = tails.

functors. If we rewrite the Fibonacci example mentioned earlier to use lfix instead of possibility, it becomes extremely quick – on my machine, it computes the 10000th element of the sequence in 0.75 seconds.

A memoizing Fibonacci sequence can be computed by a fixed point over a list:

```
listFibs =
  fix $ λfibs → 0 : 1 : zipWith (+) fibs (tail fibs)
```

The Fibonacci sequence in terms of lfix is *extensionally* equivalent to the well-known one-liner above; moreover, they are also *asymptotically* equivalent – and in practice, the constant factor overhead for the lfix-derived version is relatively small.[19]

To summarize: the "zip" operation enabled by ComonadApply is the source of lfix's computational "zippiness." By allowing the eventual future value of the comonadic result $w\ a$ to be shared by every recursive reference, lfix ensures that every element of its result is computed at most once.[20]

14. Building a Nest in Deep Space

The only comonad we've examined in depth so far is the one-dimensional Tape zipper, but there is a whole world of comonads out there. Piponi's original post is titled, "From Löb's Theorem to Spreadsheet Evaluation," and as we've seen, loeb on the list functor looks just like evaluating a one-dimensional spreadsheet with absolute references. Likewise, lfix on the Tape comonad looks just like evaluating a one-dimensional spreadsheet with relative references.

It's worth emphasizing that loeb and lfix compute *different things*. The "proof" embodied by lfix contains as its computational content a totally distinct pattern of (co)recursion from the simpler one of loeb. Their results will coincide only on very particular inputs.[21]

So far, all we've seen are analogues to one-dimensional "spreadsheets" – but spreadsheets traditionally have *two* dimensions. We could build a two-dimensional Tape to represent two-dimensional spreadsheets – nest a Tape within another Tape and we'd have made a two-dimensional space to explore – but this is an unsatisfactory special case of a more general pattern. As Willem van der Poel apocryphally put it, we'd do well to heed the "zero, one, infinity" rule. Why make a special case for two dimensions when we can talk about $n \in \mathbb{N}$ dimensions?

The Compose type[22] represents the composition of two types f and g, each of kind $* \to *$:

```
newtype Compose f g a =
  Compose {getCompose :: f (g a)}
```

We can generalize Compose using a GADT indexed by a type-level snoc-list of the composed types it wraps:[23]

```
data Flat (only :: * → *)
data Nest (outsides :: *) (innermost :: * → *)
data Nested fs a where
  Flat :: f a → Nested (Flat f) a
  Nest :: Nested fs (f a) → Nested (Nest fs f) a
```

To find an intuition for how Nested operates, observe how the type changes as we add Nest constructors:

```
            [Just True]  ::                        [Maybe  Bool]
       Flat [Just True]  :: Nested        (Flat []) (Maybe  Bool)
Nest (Flat [Just True])  :: Nested (Nest (Flat [])  Maybe) Bool
```

The only way to initially make some type Nested is to first call the Flat constructor on it. Subsequently, further outer layers of structure may be unwrapped from the type using the Nest constructor, which moves a layer of type application from the second parameter to the snoc-list in Nested's first parameter.

In this way, while Compose canonically represents the composition of types only up to associativity (for any f, g, and h, Compose f (Compose $g\ h$) $\cong$ Compose (Compose $f\ g$) h), the Nested type gives us a single canonical representation by removing the choice of left-associativity.

It's trivial to define a getCompose accessor for the Compose type, but in order to provide the same functionality for Nested, we need to use a closed type family to describe the type resulting from unwrapping one of Nested's constructors.

```
type family UnNest x where
  UnNest (Nested (Flat f)      a) = f a
  UnNest (Nested (Nest fs f) a) = Nested fs (f a)
unNest :: Nested fs a → UnNest (Nested fs a)
unNest (Flat x) = x
unNest (Nest x) = x
```

Having wrought this new type, we can put it to use by defining once and for all how to evaluate an arbitrarily deep stack of comonadic types as a higher-dimensional spreadsheet.

15. Introducing Inductive Instances

The Nested type enables us to encapsulate inductive patterns in typeclass definitions for composed types. By giving two instances for a given typeclass – the base case for Nested (Flat f), and the inductive case for Nested (Nest fs f) – we can instantiate a typeclass for *all* Nested types, no matter how deeply composed.

The general design pattern illustrated here is a powerful technique to write pseudo-dependently-typed Haskell code: model the inductive structure of some complex family of types in a GADT, and then instantiate typeclass instances which perform "ad-hoc" polymorphic recursion on that datatype.

For Nested's Functor instance, the base case Nested (Flat f) relies upon f to be a Functor; the inductive case, Nested (Nest fs f) relies on f as well as Nested fs to be Functors.

```
instance Functor f ⇒ Functor (Nested (Flat f)) where
  fmap g = Flat ∘ fmap g ∘ unNest
instance (Functor f, Functor (Nested fs)) ⇒
  Functor (Nested (Nest fs f)) where
  fmap g = Nest ∘ fmap (fmap g) ∘ unNest
```

With that, arbitrarily large compositions of Functors may now themselves be Functors – as we know to be true.

The Nested type will only give us n-dimensional "spreadsheets" if we provide a bit more than Functor, though – we also need inductive instances for Comonad and ComonadApply.

---

[19] This constant factor is around 1.3x in my own testing.

[20] Notably, lfix exhibits this memoization only on "structural" comonads. Haskell does not automatically memoize the results of functions but does memoize thunks computing elements of data structures. As a result, the comonad **newtype** ZTape a = ZTape (Integer → a), though isomorphic to Tape, does not get the full benefit of lfix's performance boost, while Tape does.

[21] In particular, lfix and loeb coincide over inputs that contain no functions depending on locally contextual information.

[22] This type is canonically located in Data.Functor.Compose.

[23] Rather than letting Flat and Nest inhabit a new kind (via a lifted sum type), we let them be uninhabited types of kind $*$ so we can alias the type and value constructor names. If GHC had kind-declarations without associated types, we could have increased kind-safety as well as this kind of punning between the type and value levels.

As with Functor, defining the base-case Comonad instance – that for Nested (Flat f) – is relatively straightforward; we lean on the Comonad instance of the wrapped type, unwrapping and rewrapping as necessary.

```
instance Comonad w ⇒ Comonad (Nested (Flat w)) where
  extract   = extract ∘ unNest
  duplicate = fmap Flat ∘ Flat
              ∘ duplicate
              ∘ unNest
```

The inductive case is trickier. To duplicate something of type Nested (Nest *fs* f) a, we need to create something of type Nested (Nest *fs* f) (Nested (Nest *fs* f) a).

Our first step must be to unNest, as we can't do much without looking inside the Nested thing. This gives us a Nested *fs* (f a). If f is a Comonad, we can now fmap duplicate to duplicate the newly revealed *inner* layer, giving us a Nested *fs* (f (f a)). If (inductively) Nested *fs* is a Comonad, we can also duplicate the *outer* layer to get a Nested *fs* (Nested *fs* (f (f a))).

Here is where, with merely our inductive Comonad constraints in hand, we get stuck. We need to distribute f over Nested *fs*, but we have no way to do so with only the power we've given ourselves.

In order to compose comonads, we need to add another precondition to what we've seen so far: distributivity.[24]

```
class Functor g ⇒ Distributive g where
  distribute :: Functor f ⇒ f (g a) → g (f a)
```

Haskellers might recognize this as the almost-dual to the more familiar Traversable class, where distribute is the dual to sequenceA. Both Traversable and Distributive enable us to define a function of the form f (g a) $\to$ g (f a), but a Traversable f constraint means we can push an f beneath an arbitrary Applicative, while a Distributive g constraint means we can pull a g from underneath an arbitrary Functor.

With only a Comonad constraint on all the types in a Nested, we can duplicate the inner and outer layers as above. To finish deriving the inductive-case instance for Nested comonads, we need only distribute to swap the middle two layers of what have become four, and re-wrap the results in Nested's constructors to inject the result back into the Nested type:

```
instance (Comonad w, Comonad (Nested ws),
          Functor (Nested (Nest ws w)),
          Distributive w) ⇒
  Comonad (Nested (Nest ws w)) where
  extract   = extract ∘ extract ∘ unNest
  duplicate = fmap Nest ∘ Nest
              ∘ fmap distribute
              ∘ duplicate
              ∘ fmap duplicate
              ∘ unNest
```

After that, it's smooth sailing to define ComonadApply in both the base and inductive cases for Nested:

```
instance ComonadApply w ⇒
  ComonadApply (Nested (Flat w)) where
  Flat g ⊛ Flat x = Flat (g ⊛ x)
instance (ComonadApply w,
          ComonadApply (Nested ws),
          Distributive w) ⇒
  ComonadApply (Nested (Nest ws w)) where
  Nest g ⊛ Nest x = Nest ((⊛) <$> g ⊛ x)
```

[24] A version of this class may be found in Data.Distributive.

Along these lines, we can instantiate many other inductive instances for Nested types, including Applicative, Alternative, Foldable, Traversable, and Distributive.

Of course, we can't very well *use* such instances without Distributive instances for the base types we intend to Nest. To elucidate how to distribute, let's turn again to the Tape.

Formally, a Distributive instance for a functor g witnesses the property that g is a representable functor preserving all limits – that is, it's isomorphic to $(\to)$ r for some r [11]. We know that any Tape a, representing a bidirectionally infinite sequence of as, is isomorphic to functions of type Integer $\to$ a (though with potentially better performance for certain operations). Therefore, Tapes must be Distributive – but we haven't concluded this in a particularly *constructive* way. How can we build such an instance?

In order to distribute Tapes, we first need to learn how to unfold them. Given the standard unfold over Streams...

```
Stream.unfold :: (c → (a, c)) → c → Stream a
```

...we can build an unfold for Tapes:

```
unfold :: (c → (a, c))   -- leftwards unfolding function
       → (c → a)         -- function from seed to focus value
       → (c → (a, c))    -- rightwards unfolding function
       → c               -- seed
       → Tape a
unfold prev center next seed =
  Tape (Stream.unfold prev seed)
       (center seed)
       (Stream.unfold next seed)
```

With this unfold, we can define a distribute for Tapes.[25]

```
instance Distributive Tape where
  distribute =
    unfold (fmap (extract ∘ moveL) &&& fmap moveL)
           (fmap extract)
           (fmap (extract ∘ moveR) &&& fmap moveR)
```

This definition of distribute unfolds a new Tape outside the Functor, eliminating the inner Tape within it by fmapping movement and extraction functions through the Functor layer. Notably, the shape of the outer Tape we had to create could not possibly depend on information from the inner Tape locked up inside of the f (Tape a). This is true in general: in order for us to be able to distribute, every possible value of any Distributive functor must have a fixed shape and no extra information to replicate beyond its payload of a values [11].

16. Asking for Directions in Higher Dimensions

Although we now have the type language to describe arbitrary-dimensional closed comonads, we don't yet have a good way to talk about movement within these dimensions. The final pieces to the puzzle are those we need to refer to relative positions within these nested spaces.

We'll represent an n-dimensional relative reference as a list of coordinates indexed by its length n using the conventional construction for length-indexed vectors via GADTs.

```
data Nat = S Nat | Z
infixr :::
data Vec (n :: Nat) (a :: *) where
  Nil   :: Vec Z a
  (:::) :: a → Vec n a → Vec (S n) a
```

[25] We define the "fanout" operator as f &&& $g = \lambda x \to (f\ x, g\ x)$. This is a specialization of a more general function from Control.Arrow.

A single relative reference in one dimension is a wrapped Int...

```
newtype Ref = Ref {getRef :: Int}
```

...and an n-dimensional relative coordinate is an n-vector of these:

```
type Coord n = Vec n Ref
```

We can combine together two different Coords of potentially different lengths by adding the corresponding components and letting the remainder of the longer dangle off the end. This preserves the understanding that an n-dimensional vector can be considered as an m-dimensional vector ($n \leq m$) where the last $(m - n)$ of its components are zero.

In order to define this heterogeneous vector addition function $(\&)$, we need to compute the length of its resulting vector in terms of a type-level maximum operation over natural numbers.

```
type family Max n m where
  Max (S n) (S m) = S (Max n m)
  Max    n     Z  = n
  Max    Z     m  = m
```

```
(&) :: Coord n → Coord m → Coord (Max n m)
(Ref q ::: qs) & (Ref r ::: rs) = Ref (q + r) ::: (qs & rs)
qs             & Nil            = qs
Nil            & rs             = rs
```

The way the zip operation in $(\&)$ handles extra elements in the longer list means that we should consider the first element of a Coord to be the distance in the first dimension. Since we always combine from the front of a vector, adding a dimension constitutes *tacking* another coordinate onto the end of a Coord.

Keeping this in mind, we can build convenience functions for constructing relative references in those dimensions we care about.

```
type Sheet1 = Nested (Flat Tape)
rightBy, leftBy :: Int → Coord (S Z)
rightBy x  = Ref x ::: Nil
leftBy     = rightBy ∘ negate
type Sheet2 = Nested (Nest (Flat Tape) Tape)
belowBy, aboveBy :: Int → Coord (S (S Z))
belowBy x = Ref 0 ::: Ref x ::: Nil
aboveBy   = belowBy ∘ negate
type Sheet3 = Nested (Nest (Nest (Flat Tape) Tape) Tape)
outwardBy, inwardBy :: Int → Coord (S (S (S Z)))
outwardBy x = Ref 0 ::: Ref 0 ::: Ref x ::: Nil
inwardBy    = outwardBy ∘ negate
    ⋮
```

We can continue this pattern *ad infinitum* (or at least, *ad* some very large *finitum*), and the pattern could easily be automated via Template Haskell, should we desire.

We choose here the coordinate convention that the positive directions, in increasing order of dimension number, are right, below, and inward; the negative directions are left, above, and outward. These names are further defined to refer to unit vectors in their respective directions (that is, right = rightBy 1 and so forth).

An example of using this tiny language: the coordinate specified by right & aboveBy 3 :: Coord (S (S Z)) refers to the selfsame relative position indicated by reading it aloud as English.

A common technique when designing a domain specific language is to separate its abstract syntax from its implementation. This is exactly what we've done – we've defined how to *describe* relative positions in n-space; what we have yet to do is *interpret* those coordinates.

17. Following Directions in Higher Dimensions

Moving about in one dimension requires us to either move left or right by the absolute value of a reference, as determined by its sign:

```
tapeGo :: Ref → Tape a → Tape a
tapeGo (Ref r) =
  foldr (∘) id $
    replicate (abs r) (if r < 0 then moveL else moveR)
```

Going somewhere based on an n-dimensional coordinate means taking that coordinate and some Tape of at least that number of dimensions, and moving around in it appropriately.

```
class Go n t where
  go :: Coord n → t a → t a
```

We can go nowhere no matter where we are:

```
instance Go Z (Nested ts) where go = const id
```

Going somewhere in a one-dimensional Tape reduces to calling the underlying tapeGo function:

```
instance Go (S Z) (Nested (Flat Tape)) where
  go (r ::: _) (Flat t) = Flat (tapeGo r t)
```

As usual, it's the inductive case which requires more consideration. If we can move in $n-1$ dimensions in an $(n-1)$-dimensional Tape, then we can move in n dimensions in an n-dimensional Tape:

```
instance (Go    n  (Nested ts), Functor (Nested ts)) ⇒
          Go (S n) (Nested (Nest ts Tape)) where
  go (r ::: rs) t =
    Nest ∘ go rs ∘ fmap (tapeGo r) ∘ unNest $ t
```

Notice how this (polymorphically) recursive definition treats the structure of the nested Tapes: the innermost Tape always corresponds to the first dimension, and successive dimensions correspond to Tapes nested outside of it. In each recursive call to go, we unwrap one layer of the Nested type, revealing another outer layer of the contained type, to be accessed via fmap (tapeGo r).

In an analogous manner, we can also use relative coordinates to specify how to slice out a section of an n-dimensionally Nested Tape, starting at our current coordinates. The only twist is that we need to use an associated type family to represent the type of the resultant n-nested list.

```
tapeTake :: Ref → Tape a → [a]
tapeTake (Ref r) t =
  focus t : Stream.take (abs r) (view t)
  where view = if r < 0 then viewL else viewR
class Take n t where
  type ListFrom t a
  take :: Coord n → t a → ListFrom t a
instance Take (S Z) (Nested (Flat Tape)) where
  type ListFrom (Nested (Flat Tape)) a = [a]
  take (r ::: _) (Flat t) = tapeTake r t
instance (Functor (Nested ts), Take n (Nested ts)) ⇒
         Take (S n) (Nested (Nest ts Tape)) where
  type ListFrom (Nested (Nest ts Tape)) a =
    ListFrom (Nested ts) [a]
  take (r ::: rs) t =
    take rs ∘ fmap (tapeTake r) ∘ unNest $ t
```

18. New Construction in Higher Dimensions

To use an analogy to more low-level programming terminology, we've now defined how to *peek* at Nested Tapes, but we don't yet know how to *poke* them. To modify such structures, we can again use an inductive typeclass construction. The interface we want should take an n-dimensionally nested container of some kind, and insert its contents into a given nested Tape of the same dimensionality. In other words:

```
class InsertNested ls ts where
  insertNested :: Nested ls a → Nested ts a → Nested ts a
```

It's not fixed from the outset which base types have a reasonable semantics for insertion into an n-dimensionally nested space. So that we can easily add new insertable types, we'll split out one-dimensional insertion into its own typeclass, InsertBase, and define InsertNested's base case in terms of InsertBase.

```
class InsertBase l t where
  insertBase :: l a → t a → t a
```

An instance of InsertBase for some type l means that we know how to take an $l\ a$ and insert it into a $t\ a$ to give us a new $t\ a$. Its instance for one-directionally extending types is right-biased by convention.

```
instance InsertBase [ ] Tape where
  insertBase [ ] t = t
  insertBase (x : xs) (Tape ls c rs) =
    Tape ls x (Stream.prefix xs (Cons c rs))
instance InsertBase Stream Tape where
  insertBase (Cons x xs) (Tape ls _ _) = Tape ls x xs
instance InsertBase Tape Tape where
  insertBase t _ = t
```

Now we're in a better position to define the dimensional induction necessary to insert n-nested things into n-nested Tapes. The base case relies on insertBase, as expected:

```
instance InsertBase l t ⇒
  InsertNested (Flat l) (Flat t) where
  insertNested (Flat l) (Flat t) = Flat (insertBase l t)
```

The trick in the recursive case is to generate a Nested structure full of functions, each of which knows how to insert the relevant elements at their own position into the pre-existing structure there – and then to use (◈) to zip together this space of modifying functions with the space of arguments to which they apply.

```
instance (InsertBase l t, InsertNested ls ts,
          Functor (Nested ls), Functor (Nested ts),
          ComonadApply (Nested ts)) ⇒
  InsertNested (Nest ls l) (Nest ts t) where
  insertNested (Nest l) (Nest t) =
    Nest $ insertNested (insertBase ◈ l) (fill t id) ◈ t
```

Note that although the above instance uses the fill function to generate a nested structure filled with the identity function, it is not abusing functorial strength in so doing, as id is a closed term.

Using some mildly "Hasochistic" type hackery (ala [13]) we can take a nested structure which is not yet Nested – such as a triply-nested list [[[Int]]] – and lift it into a Nested type – such as Nested (Nest (Flat [ ]) [ ]) [Int]. The asNestedAs function has the type

```
asNestedAs :: NestedAs x y ⇒ x → y → AsNestedAs x y
```

where the AsNestedAs $x\ y$ type family application describes the type resulting from wrapping x in as many constructors of Nested as are wrapped around y, and the NestedAs typeclass witnesses that this operation is possible. (See Appendix C for a full definition of this function.)

With asNestedAs, we can define an insert function which does not require the inserted thing to be already wrapped in a Nested type. This automatic lifting requires knowledge of the type into which we're inserting values, which means that insert and functions based upon it may require some light type annotation to infer properly.

```
insert :: (InsertNested l t, NestedAs x (Nested t a),
           AsNestedAs x (Nested t a) ∼ Nested l a) ⇒
          x → Nested t a → Nested t a
insert l t = insertNested (asNestedAs l t) t
```

With that in place, we can define the high level interface to our "spreadsheet" library. Borrowing from the nomenclature of spreadsheets, we define two "cell accessor" functions – one to dereference an individual cell, and another to dereference a Traversable collection of cells:

```
cell :: (Comonad w, Go n w) ⇒ Coord n → w a → a
cell = (extract ∘) ∘ go
cells :: (Traversable t, Comonad w, Go n w) ⇒
         t (Coord n) → w a → t a
cells = traverse cell
```

We may use the sheet function to construct a sheet with a default background into which some other values have been inserted:

```
sheet :: (InsertNested l ts,
          ComonadApply (Nested ts),
          Applicative (Nested ts),
          NestedAs x (Nested ts a),
          AsNestedAs x (Nested ts a) ∼ Nested l a) ⇒
         a → x → Nested ts a
sheet background list = insert list (pure background)
```

Because sheet has to invent an entire infinite space of values, we need to rely on pure from the Applicative class to generate the *background* space into which it may insert its *list* argument. This is not an issue for Tapes, as they are indeed Applicative, where (⊛) = (◈) and pure = tapeOf, tapeOf being the function which makes a Tape by repeating a single value. This constraint doesn't mean we can't build and manipulate "spreadsheets" with non-Applicative layers; it merely means we can't as easily manufacture them *ex nihilo*. The sheet function is purely a pleasing convenience, not a necessity.

19. Conclusion: Zippy Comonadic Computations in Infinite *n*-Dimensional Boxes

This journey has taken us from the general and abstract to the specific and concrete. By transliterating and composing the axioms of the $\Box$ modality, we found a significantly more accurate translation of Löb's theorem into Haskell, which turned out to embody a maximally efficient fixed point operation over closed comonads. Noticing that closed comonads can compose with the addition of a distributive law, we lifted this new fixed-point into spaces composed of arbitrarily nested comonads. On top of this framework, we layered typeclass instances which perform induction over dimensionality to provide an interpretation for a small domain-specific language of relative references into one family of representable comonads.

Now, where can we go?

To start, we can construct the Fibonacci sequence with optimal memoization, using syntax which looks a lot nicer than before:[26]

[26] In the following examples, we make use of a Num instance for functions so that $f + g = \lambda a \to f\ a + g\ a$, and likewise for the other arithmetic operators. These instances may be found in Data.Numeric.Function.

```
fibonacci :: Sheet1 Integer
fibonacci = lfix ∘ sheet 1 $
  repeat $ cell (leftBy 2) + cell left
```

If we take a peek at it, it's as we'd expect:

```
take (rightBy 15) ∘ go (leftBy 2) $ fibonacci
 ≡ [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987]
```

Moving up into two dimensions, we can define the infinite grid representing Pascal's triangle…[27]

```
pascal :: Sheet2 Integer
pascal = lfix ∘ sheet 0 $
  repeat 1 ⟐ repeat (1 ⟐ pascalRow)
  where pascalRow = repeat $ cell above + cell left
```

…which looks like this:

```
take (belowBy 4 & rightBy 4) pascal
 ≡ [[1, 1, 1,  1,  1 ],
    [1, 2, 3,  4,  5 ],
    [1, 3, 6,  10, 15],
    [1, 4, 10, 20, 35],
    [1, 5, 15, 35, 70]]
```

Like fibonacci, pascal evaluates each part of its result at most once.

Conway's game of life [7] is a nontrivial comonadic computation [4]. We can represent a cell in the life universe as either X (dead) or O (alive):

```
data Cell = X | O deriving Eq
```

More interestingly, we define a Universe as a *three*-dimensional space where two axes map to the spatial axes of Conway's game, but the third represents *time*.

```
type Universe = Sheet3 Cell
```

We can easily parameterize by the "rules of the game" – which neighbor counts trigger cell birth and death, respectively:

```
type Ruleset = ([Int], [Int])
```

To compute the evolution of a game from a two-dimensional list of cells "seeding" the system, we insert that seed into a "blank" Universe where each cell not part of the seed is defined in terms of the action of the rules upon its previous-timestep neighbors. We can then evaluate this "function space" with lfix.[28]

```
life :: Ruleset → [[Cell]] → Universe
life ruleset seed =
  lfix $ insert [map (map const) seed] blank
  where
    blank =
      sheet (const X) (repeat ∘ tapeOf ∘ tapeOf $ rule)
    rule place =
      case onBoth (neighbors place ∈) ruleset of
          (True, _) → O
          (_, True) → cell inward place
          _         → X
    neighbors = length ∘ filter (O ==) ∘ cells bordering
    bordering = map (inward &) (diag ++ vert ++ horz)
    diag  = (&) ⟨$⟩ horz ⊛ vert
    vert  = [above, below]
    horz  = map to2D [right, left]
    to2D = (belowBy 0 &)
```

[27] What is typeset here as (⟐) is spelled in ASCII Haskell as (<:>), and is defined to be Stream's Cons constructor in infix form.

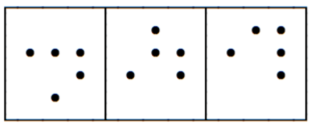
[28] In the definition of life, we let onBoth $f\ (x, y) = (f\ x, f\ y)$.

Conway's original game is instantiated by applying the more general life function to his defining rule set:

```
conway :: [[Cell]] → Sheet3 Cell
conway = life ([3], [2, 3])
```

After defining a pretty-printer for Universes (elided here due to space), we can watch the flight of a glider as it soars off to infinity.

```
glider :: Sheet3 Cell
glider = conway [[O, O, O],
                 [X, X, O],
                 [X, O, X]]
```

printLife glider ≡  …

20. Further Work

I have a strong suspicion that lfix and the slow possibility from earlier in the paper are extensionally equivalent despite their differing asymptotic characteristics. A precise characterization of the criteria for their equivalence would be useful and enlightening.

In this paper, I focused on a particular set of zipper topologies: n-dimensional infinite Cartesian grids. Exploring the space of expressible computations with lfix on more diverse topologies may lend new insight to a variety of dynamic programming algorithms.

I suspect that it is impossible to thread dynamic cycle-detection through lfix while preserving its performance characteristics and without resorting to unsafe language features. A resolution to this suspicion likely involves theoretical work on heterogeneous compositions of monads and comonads. Further, it could be worth exploring how to use unsafe language features to implement dynamic cycle detection in lfix while still presenting a pure API.

Static cycle detection in lfix-like recurrences is undecidable in full generality. Nevertheless, we might present a restricted interface within which it is decidable. The interaction between laziness and decidable cyclicity complicates this approach: in general, whether a reference is evaluated depends on arbitrary computation. Other approaches to this problem might include the use of full dependent types or a refinement type system like LiquidHaskell [21]. In particular, it seems that many of the constraints we were unable to encode in the Haskell type system might be adequately expressed in a dependent type system augmented with linear or affine types.

Acknowledgments

Thank you firstly to Harry Mairson, my advisor at Brandeis, for his advice and camaraderie. A huge thanks to Dominic Orchard for his help with proof-reading, equational reasoning techniques, and numerous pieces of clarifying insight. Without Dan Piponi and his outstanding blog, I wouldn't even have set out on this journey.

Many others have given me ideas, questions, and pointers to great papers; among them are Edward Kmett, Jeremy Gibbons, Tarmo Uustalu, Getty Ritter, Kyle Carter, and Gershom Bazerman. Thanks also to an anonymous reviewer for noticing an important technical point which I'd missed.

Finally: thank you to Caleb Marcus for putting up with enough comonadic lunch-table conversation to fulfill a lifetime quota, to Eden Zik for standing outside in the cold to talk about spreadsheets, and to Isabel Ballan for continually reminding me that in the end, it's usually to do with Wittgenstein.

References

[1] D. Ahmen, J. Chapman, and T. Uustalu. When is a Container a Comonad? *Logical Methods in Computer Science*, 10(3), 2014.

[2] N. Alechina, M. Mendler, V. D. Paiva, and E. Ritter. Categorical and Kripke Semantics for Constructive S4 Modal Logic. *Computer Science Logic*, pages 292–307, 2001.

[3] G. M. Beirman and V. C. V. D. Paiva. Intuitionistic Necessity Revisited. Technical report, University of Birmingham, School of Computer Science, 1996.

[4] S. Capobianco and T. Uustalu. A Categorical Outlook on Cellular Automata. *arXiv preprint arXiv:1012.1220*, 2010.

[5] C. Done. Twitter Waterflow Problem and Loeb, 2014. URL `chrisdone.com/posts/twitter-problem-loeb`.

[6] J. Epstein, A. Black, and S. Peyton-Jones. Towards Haskell in the Cloud. In *ACM SIGPLAN Notices*, volume 46, pages 118–129. ACM, 2011.

[7] M. Gardner. The Fantastic Combinations of John Conway's New Solitaire Game "Life". *Scientific American*, 223(4): 120–123, October 1970.

[8] J. Garson. Modal Logic. *Stanford Encyclopedia of Philosophy*, 2014. URL `plato.stanford.edu/entries/logic-modal`.

[9] G. Huet. Functional Pearl: The Zipper. *Journal of Functional Programming*, 7(5):549–554, September 1997.

[10] E. Kmett. The Comonad Package, July 2014. URL `hackage.haskell.org/package/comonad`.

[11] E. Kmett. The Distributive Package, May 2014. URL `hackage.haskell.org/package/distributive`.

[12] A. Kock. Strong Functors and Monoidal Monads. *Archiv der Mathematik*, 23(1):113–120, 1972.

[13] S. Lindley and C. McBride. Hasochism: the Pleasure and Pain of Dependently Typed Haskell Programming. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell*, pages 81–92, 2013.

[14] P. Lindstrom. Notes on Some Fixed Point Constructions in Provability Logic. *Journal of Philosophical Logic*, 35(3): 225–230, 2006.

[15] E. Mendelson. *Introduction to Mathematical Logic*. CRC Press, 4 edition, 1997.

[16] D. Menendez. Paper: The Essence of Dataflow Programming. Haskell mailing list archive. URL `mail.haskell.org/pipermail/haskell/2005-September/016502.html`.

[17] D. Orchard. Programming Contextual Computations. Technical Report UCAM-CL-TR-854, University of Cambridge, Computer Laboratory, May 2014.

[18] D. Piponi. From Löb's Theorem to Spreadsheet Evaluation, November 2006. URL `blog.sigfpe.com/2006/11/from-l-theorem-to-spreadsheet.html`.

[19] T. Uustalu and V. Vene. Comonadic Notions of Computation. *Electronic Notes in Theoretical Computer Science*, 203(5): 263–284, 2008.

[20] T. Uustalu and V. Vene. The Essence of Dataflow Programming. *Lecture Notes in Computer Science*, 4164: 135–167, November 2006.

[21] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones. Refinement Types for Haskell. *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, pages 269–282, 2014.

[22] R. Verbrugge. Provability Logic. *Stanford Encyclopedia of Philosophy*, 2010. URL `plato.stanford.edu/entries/logic-provability`.

A. Proof: $\mathsf{loeb}_{(\to)\,r} \equiv \mathsf{fix} \circ \mathsf{flip}$

```
loeb_(→) r :: (r → (r → a) → a) → r → a
loeb_(→) r = λf → fix (λx → fmap ($ x) f)
           {- fmap_(→) r = (∘) -}
           ≡ λf → fix (λx → (∘) ($ x) f)
           {- inline (∘); β-reduce; definition of flip -}
           ≡ λf → fix (λx y → flip f x y))
           {- η-reduce inner λ-expression -}
           ≡ λf → fix (flip f)
           {- η-reduce outer λ-expression -}
           ≡ fix ∘ flip
```

B. Proof: loeb Uses Functorial Strength

```
loeb :: Functor f ⇒ f (f a → a) → f a
loeb f = fix (λx → fmap ($ x) f)
       {- meaning of section notation -}
       ≡ fix (λx → fmap (flip ($) x) f)
       {- curry/uncurry inverses -}
       ≡ fix (λx → fmap (uncurry (flip ($)) ∘ (,) x) f)
       {- uncurry (flip x) ∘ (,) y ≡ uncurry x ∘ flip (,) y -}
       ≡ fix (λx → fmap (uncurry ($) ∘ flip (,) x) f)
       {- fmap (f ∘ g) ≡ fmap f ∘ fmap g (functor law) -}
       ≡ fix (λx → (fmap (uncurry ($)) ∘ fmap (flip (,) x)) f)
       {- inline flip; β-reduce; use tuple section notation -}
       ≡ fix (λx → (fmap (uncurry ($)) ∘ fmap (,x)) f)
       {- definition of flex; η-reduce -}
       ≡ fix (fmap (uncurry ($)) ∘ flex f)
```

C. Full Listing of the Function asNestedAs

```
type family AddNest x where
  AddNest (Nested fs (f x)) = Nested (Nest fs f) x
type family AsNestedAs x y where
  AsNestedAs (f x) (Nested (Flat g) b) = Nested (Flat f) x
  AsNestedAs x y = AddNest (AsNestedAs x (UnNest y))
class NestedAs x y where
  asNestedAs :: x → y → AsNestedAs x y
instance (AsNestedAs (f a) (Nested (Flat g) b)
           ~ Nested (Flat f) a) ⇒
  NestedAs (f a) (Nested (Flat g) b) where
  asNestedAs x _ = Flat x
instance (AsNestedAs (f a)
             (UnNest (Nested (Nest g h) b))
           ~ Nested fs (f′ a′),
          AddNest (Nested fs (f′ a′))
           ~ Nested (Nest fs f′) a′,
          NestedAs (f a) (UnNest (Nested (Nest g h) b)))
    ⇒ NestedAs (f a) (Nested (Nest g h) b) where
  asNestedAs x y = Nest (asNestedAs x (unNest y))
```

D. Haskell Dialect

The code in this paper relies on a number of language extensions available in GHC 7.8 and later: DataKinds, GADTs, MultiParamTypeClasses, FlexibleContexts, FlexibleInstances, TypeFamilies, UndecidableInstances, DeriveFunctor, and TupleSections.