Getting a Quick Fix on Comonads

A quest to `extract` computation and not `duplicate` work

Kenneth Foner

Brandeis University / Galois, Inc.

June 30, 2014

A tale of two blog articles

Dan Piponi, November 2006 (`blog.sigfpe.com`):

"From Löb's Theorem to Spreadsheet Evaluation"

```
loeb :: Functor f => f (f a -> a) -> f a
loeb fs = xs where xs = fmap ($ xs) fs
```

A tale of two blog articles

Dan Piponi, November 2006 (`blog.sigfpe.com`):

"From Löb's Theorem to Spreadsheet Evaluation"

```
loeb :: Functor f => f (f a -> a) -> f a
loeb fs = xs where xs = fmap ($ xs) fs
```

Example:

```
> loeb [length, (!! 0), \x -> x !! 0 + x !! 1]
```

A tale of two blog articles

Dan Piponi, November 2006 (`blog.sigfpe.com`):

"From Löb's Theorem to Spreadsheet Evaluation"

```
loeb :: Functor f => f (f a -> a) -> f a
loeb fs = xs where xs = fmap ($ xs) fs
```

Example:

```
> loeb [length, (!! 0), \x -> x !! 0 + x !! 1]
[3,3,6]
```

A tale of two blog articles

Dan Piponi, November 2006 (`blog.sigfpe.com`):

"From Löb's Theorem to Spreadsheet Evaluation"

```
loeb :: Functor f => f (f a -> a) -> f a
loeb fs = xs where xs = fmap ($ xs) fs
```

Example:

```
> loeb [length, sum]
```

A tale of two blog articles

Dan Piponi, November 2006 (`blog.sigfpe.com`):

"From Löb's Theorem to Spreadsheet Evaluation"

```
loeb :: Functor f => f (f a -> a) -> f a
loeb fs = xs where xs = fmap ($ xs) fs
```

Example:

```
> loeb [length, sum]
⊥
```

A tale of two blog articles

Dan Piponi, December 2006 (`blog.sigfpe.com`):

"Evaluating Cellular Automata is Comonadic"

> *I want to work on 'universes' that extend to infinity in both directions. And I want this universe to be constructed lazily on demand.*
>
> *We can think of a universe with the cursor pointing at a particular element as being an element with a neighbourhood on each side.*

An unexpected journey

```haskell
loeb :: Functor f => f (f a -> a) -> f a
loeb fs = xs where xs = fmap ($ xs) fs
```

- `loeb`: each element refers to *absolute positions* in a structure
- comonads: computations in context of *relative position* in a structure

These are almost the same thing!

(Co)monads: a brief summary

Monads:

Most Haskellers define monads via `return` and `(>>=)`. Today, we'll use `return` and `join`. *Note:* `x >>= f == join (fmap f x)`.

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

(Co)monads: a brief summary

Monads:

Most Haskellers define monads via `return` and `(>>=)`. Today, we'll use `return` and `join`. *Note:* `x >>= f == join (fmap f x)`.

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

Comonads:

```
class Functor w => Comonad w where
  extract   :: w a -> a       -- a.k.a. coreturn
  duplicate :: w a -> w (w a) -- a.k.a. cojoin
```

(from Edward Kmett's `Control.Comonad`)

A particular flavor of comonad

```
data Stream a = Cons a (Stream a) -- no nil!
```

(from Wouter Swierstra's `Data.Stream`)

A particular flavor of comonad

```
data Stream a = Cons a (Stream a) -- no nil!
```

(from Wouter Swierstra's `Data.Stream`)

```
head :: Stream a -> a
head (Cons x _) = x

tail :: Stream a -> Stream a
tail (Cons _ xs) = xs

iterate :: (a -> a) -> a -> Stream a
iterate f x = Cons x (iterate f (f x))
```

A particular flavor of comonad

```
data Tape a = (Stream a) a (Stream a)
```

A particular flavor of comonad

```
data Tape a = (Stream a) a (Stream a)

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
```

A particular flavor of comonad

```
data Tape a = (Stream a) a (Stream a)

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

iterate :: (a -> a) -> (a -> a) -> a -> Tape a
iterate prev next x =
   Tape (Stream.iterate prev x) x (Stream.iterate next x)
```

A particular flavor of comonad

```haskell
instance Comonad Tape where
   extract (Tape _ c _) = c
   duplicate = iterate moveL moveR
```

A particular flavor of comonad

```
instance Comonad Tape where
   extract (Tape _ c _) = c
   duplicate = iterate moveL moveR
```

Duplicate is "diagonalization." Movement and duplication commute:

```
moveL . duplicate == duplicate . moveL
moveR . duplicate == duplicate . moveR
```

Back to Piponi's `loeb`

Löb's theorem: $\Box(\Box P \to P) \to \Box P$

> *I'm going to take that as my theorem from which I'll derive a type. But what should $\Box$ become in Haskell?*
>
> *We'll defer that decision until later and assume as little as possible. Let's represent $\Box$ by a type that is a Functor.*

(Piponi, 2006)

Back to Piponi's `loeb`

Löb's theorem: $\Box(\Box P \to P) \to \Box P$

> *I'm going to take that as my theorem from which I'll derive a type. But what should $\Box$ become in Haskell?*
>
> *We'll defer that decision until later and assume as little as possible. Let's represent $\Box$ by a type that is a Functor.*

(Piponi, 2006)

```
loeb :: Functor f => f (f a -> a) -> f a
```

Back to Piponi's `loeb`

Löb's theorem: $\Box(\Box P \to P) \to \Box P$

> *I'm going to take that as my theorem from which I'll derive a type. But what should $\Box$ become in Haskell?*
>
> *We'll defer that decision until later and assume as little as possible. Let's represent $\Box$ by a type that is a Functor.*

(Piponi, 2006)

```
loeb :: Functor f => f (f a -> a) -> f a
```

But $\Box$ could also have more structure...

Fixed that for you

```haskell
loeb :: Functor f => f (f a -> a) -> f a
loeb fs = xs where xs = fmap ($ xs) fs
```

Fixed that for you

```haskell
loeb :: Functor f => f (f a -> a) -> f a
loeb fs = xs where xs = fmap ($ xs) fs

fix :: (a -> a) -> a
fix f = let x = f x in x
```

Fixed that for you

```haskell
loeb :: Functor f => f (f a -> a) -> f a
loeb fs = xs where xs = fmap ($ xs) fs
```

```haskell
fix :: (a -> a) -> a
fix f = let x = f x in x
```

We can redefine Piponi's `loeb` in terms of `fix`:

```haskell
loeb :: Functor f => f (f a -> a) -> f a
loeb fs = fix $ \xs -> fmap ($ xs) fs
```

I'll use this one from now on.

Fixed that for you

```
loeb :: Functor f => f (f a -> a) -> f a
loeb fs = fix $ \xs -> fmap ($ xs) fs
```

We want to find:

```
???? :: Comonad w => w (w a -> a) -> w a
```

Fixed that for you

```
loeb :: Functor f => f (f a -> a) -> f a
loeb fs = fix $ \xs -> fmap ($ xs) fs
```

We want to find:

```
???? :: Comonad w => w (w a -> a) -> w a

cfix :: Comonad w =>   (w a -> a) -> w a
```

Fixed that for you

```
loeb :: Functor f => f (f a -> a) -> f a
loeb fs = fix $ \xs -> fmap ($ xs) fs
```

We want to find:

```
???? :: Comonad w => w (w a -> a) -> w a

cfix :: Comonad w =>   (w a -> a) -> w a


wfix :: Comonad w => w (w a -> a) ->   a
```

Fixed that for you

```
loeb :: Functor f => f (f a -> a) -> f a
loeb fs = fix $ \xs -> fmap ($ xs) fs
```

We want to find:

```
???? :: Comonad w => w (w a -> a) -> w a

cfix :: Comonad w =>   (w a -> a) -> w a
cfix f = fix (fmap f . duplicate)

wfix :: Comonad w => w (w a -> a) ->   a
wfix w = extract w (fmap wfix (duplicate w))
```

Is this our fix?

```
possibility :: Comonad w => w (w a -> a) -> w a
possibility = fmap wfix . duplicate
```

Is this our fix?

```
possibility :: Comonad w => w (w a -> a) -> w a
possibility = fmap wfix . duplicate
```

It type-checks, so it has to be right! Right?

Well, sort of. . .

Well, sort of...

Let's try to count to 10000!

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

(This syntax gets more elegant later.)

Well, sort of. . .

```
$ time ./possibility
```

Well, sort of. . .

```
$ time ./possibility

[0,1,2,3,4 ... some time later ... 9998, 9999, 10000]
       39.49 real         38.87 user          0.38 sys
```

Well, sort of. . .

```
$ time ./possibility

[0,1,2,3,4 ... some time later ... 9998, 9999, 10000]
       39.49 real          38.87 user           0.38 sys
```

256 increment operations per second.

(And this gets worse—it's not linear. . .)

Sharing is caring (as well as polynomial complexity)

```haskell
wfix :: Comonad w => w (w a -> a) -> a
wfix w = extract w (fmap wfix (duplicate w))

possibility :: Comonad w => w (w a -> a) -> w a
possibility = fmap wfix . duplicate
```

Sharing is caring (as well as polynomial complexity)

```haskell
wfix :: Comonad w => w (w a -> a) -> a
wfix w = extract w (fmap wfix (duplicate w))

possibility :: Comonad w => w (w a -> a) -> w a
possibility = fmap wfix . duplicate
```

- No sharing: computation is shaped like a tree, not a DAG

Sharing is caring (as well as polynomial complexity)

```haskell
wfix :: Comonad w => w (w a -> a) -> a
wfix w = extract w (fmap wfix (duplicate w))

possibility :: Comonad w => w (w a -> a) -> w a
possibility = fmap wfix . duplicate
```

- No sharing: computation is shaped like a tree, not a DAG
- Count all the way up from zero for each number, so $O(n^2)$

Sharing is caring (as well as polynomial complexity)

```
wfix :: Comonad w => w (w a -> a) -> a
wfix w = extract w (fmap wfix (duplicate w))

possibility :: Comonad w => w (w a -> a) -> w a
possibility = fmap wfix . duplicate
```

- No sharing: computation is shaped like a tree, not a DAG
- Count all the way up from zero for each number, so $O(n^2)$
- In a higher-dimensional space with > 1 reference per cell, would be exponential or worse.

Sharing is caring (as well as polynomial complexity)

```haskell
wfix :: Comonad w => w (w a -> a) -> a
wfix w = extract w (fmap wfix (duplicate w))

possibility :: Comonad w => w (w a -> a) -> w a
possibility = fmap wfix . duplicate
```

- No sharing: computation is shaped like a tree, not a DAG
- Count all the way up from zero for each number, so $O(n^2)$
- In a higher-dimensional space with > 1 reference per cell, would be exponential or worse.

Sharing is caring (as well as polynomial complexity)

```
wfix :: Comonad w => w (w a -> a) -> a
wfix w = extract w (fmap wfix (duplicate w))

possibility :: Comonad w => w (w a -> a) -> w a
possibility = fmap wfix . duplicate
```

- No sharing: computation is shaped like a tree, not a DAG
- Count all the way up from zero for each number, so $O(n^2)$
- In a higher-dimensional space with > 1 reference per cell, would be exponential or worse.

That really succs.

Sharing is caring (as well as polynomial complexity)

```
wfix :: Comonad w => w (w a -> a) -> a
wfix w = extract w (fmap wfix (duplicate w))
```

The root of the problem: `wfix` can't be expressed in terms of `fix`.

Sharing is caring (as well as polynomial complexity)

```haskell
wfix :: Comonad w => w (w a -> a) -> a
wfix w = extract w (fmap wfix (duplicate w))
```

The root of the problem: `wfix` can't be expressed in terms of `fix`.

```haskell
notWhatI'mTalkingAbout :: Comonad w => w (w a -> a) -> a
notWhatI'mTalkingAbout =
  fix $ \wfix ->
    \w -> extract w (fmap wfix (duplicate w))
```

Holding on to the future

```haskell
wfix :: Comonad w => w (w a -> a) -> a
wfix w = extract w (fmap wfix (duplicate w))
```

More specifically: `wfix` is inexpressible in terms of `fix` on its *argument*.

Why does this mean it's inefficient?

Holding on to the future

```haskell
wfix :: Comonad w => w (w a -> a) -> a
wfix w = extract w (fmap wfix (duplicate w))
```

More specifically: `wfix` is inexpressible in terms of `fix` on its *argument*.

Why does this mean it's inefficient?

No single reference to the eventual future of the computation.

Holding on to the future

Epiphany: Any efficient "evaluation" function looks like:

```
evaluate :: Comonad w => w (w a -> a) -> w a
evaluate fs = fix $ _
```

Filling in the holes

```
evaluate :: Comonad w => w (w a -> a) -> w a
evaluate fs = fix $ _
```

Filling in the holes

```
evaluate :: Comonad w => w (w a -> a) -> w a
evaluate fs = fix $ _
```

```
Found hole with type:  w a -> w a
```

(Error messages have been cleaned for your viewing enjoyment.)

Filling in the holes

```
evaluate :: Comonad w => w (w a -> a) -> w a
evaluate fs = fix $ _ . duplicate
```

Filling in the holes

```
evaluate :: Comonad w => w (w a -> a) -> w a
evaluate fs = fix $ _ . duplicate

Found hole with type:  w (w a) -> w a
```

Filling in the holes

```
evaluate :: Comonad w => w (w a -> a) -> w a
evaluate fs = fix $ _ fs . duplicate
```

Filling in the holes

```
evaluate :: Comonad w => w (w a -> a) -> w a
evaluate fs = fix $ _ fs . duplicate

Found hole with type:  w (w a -> a) -> w (w a) -> w a
```

Filling in the holes

```haskell
evaluate :: Comonad w => w (w a -> a) -> w a
evaluate fs = fix $ (fs <@>) . duplicate

(<@>) :: ComonadApply w => w (a -> b) -> w a -> w b
```

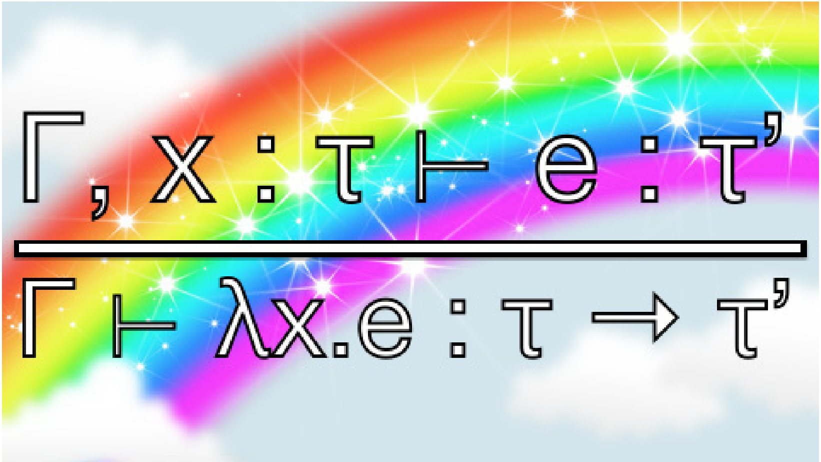
Filling in the holes

```
evaluate :: Comonad w => w (w a -> a) -> w a
evaluate fs = fix $ (fs <@>) . duplicate

(<@>) :: ComonadApply w => w (a -> b) -> w a -> w b
```

```
Could not deduce (ComonadApply w)
   arising from a use of ‘<@>’
   Possible fix:
      add (ComonadApply w) to the context
      of the type signature for evaluate.
```

Filling in the holes

```
evaluate :: ComonadApply w => w (w a -> a) -> w a
evaluate fs = fix $ (fs <@>) . duplicate

(<@>) :: ComonadApply w => w (a -> b) -> w a -> w b
```


$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \to \tau'}$$

Will it blend?

Let's try to count to 10000... again!

```haskell
evaluate :: ComonadApply w => w (w a -> a) -> w a
evaluate fs = fix $ (fs <@>) . duplicate

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

Will it blend?

```
$ time ./evaluate
```

Will it blend?

```
$ time ./evaluate

[0,1,2,3,4 ... a blur on the screen ... 9999, 10000]
       0.01 real         0.00 user         0.00 sys
```

Will it blend?

```
$ time ./evaluate
```

```
[0,1,2,3,4 ... a blur on the screen ... 9999, 10000]
       0.01 real         0.00 user          0.00 sys
```

Still very slightly slower than `take 10000 [1..]`, almost certainly because GHC fuses away the intermediate list.

Aside: list fusion in `evaluate`: reducible to halting problem?

Wait just a minute!

"Hang on, Kenny! We don't know why `Tapes` are `ComonadApply`!"

Wait just a minute!

"Hang on, Kenny! We don't know why `Tapes` are `ComonadApply`!"

```
instance ComonadApply Tape where
   (Tape ls c rs) <@> (Tape ls' c' rs') =
      Tape (ls <@> ls') (c c') (rs <@> rs')
```

Wait just a minute!

"Hang on, Kenny! We don't know why Tapes are ComonadApply!"

```
instance ComonadApply Tape where
   (Tape ls c rs) <@> (Tape ls' c' rs') =
      Tape (ls <@> ls') (c c') (rs <@> rs')
```

"But that relies on the ComonadApply instance for Streams!"

Wait just a minute!

"Hang on, Kenny! We don't know why `Tape`s are `ComonadApply`!"

```
instance ComonadApply Tape where
   (Tape ls c rs) <@> (Tape ls' c' rs') =
      Tape (ls <@> ls') (c c') (rs <@> rs')
```

"But that relies on the `ComonadApply` instance for `Stream`s!"

```
instance ComonadApply Stream where (<@>) = (<*>)
```

Wait just a minute!

"Hang on, Kenny! We don't know why `Tape`s are `ComonadApply`!"

```
instance ComonadApply Tape where
   (Tape ls c rs) <@> (Tape ls' c' rs') =
      Tape (ls <@> ls') (c c') (rs <@> rs')
```

"But that relies on the `ComonadApply` instance for `Stream`s!"

```
instance ComonadApply Stream where (<@>) = (<*>)
```

```
instance Applicative Stream where
  pure  = repeat
  (<*>) = zipWith ($)
```

What *is* a `ComonadApply` anyhow?

What *is* a `ComonadApply` anyhow?

It is a strong lax symmetric semi-monoidal comonad on the category Hask of Haskell types. That it to say that w is a strong lax symmetric semi-monoidal functor on Hask, where both extract and duplicate are symmetric monoidal natural transformations.

—Edward Kmett

What *is* a `ComonadApply` anyhow?

It is a strong lax symmetric semi-monoidal comonad on the category Hask of Haskell types. That it to say that w is a strong lax symmetric semi-monoidal functor on Hask, where both extract and duplicate are symmetric monoidal natural transformations.

—Edward Kmett

ComonadApply is to Comonad like Applicative is to Monad.

—Edward Kmett

What *is* a `ComonadApply` anyhow?

The laws of `ComonadApply`:

```
(.) <$> u <@> v <@> w == u <@> (v <@> w)
extract   (p <@> q)   == extract p (extract q)
duplicate (p <@> q)   == (<@>) <$> duplicate p <@> duplicate q
```

What *is* a `ComonadApply` anyhow?

The laws of `ComonadApply`:

```
(.) <$> u <@> v <@> w == u <@> (v <@> w)
extract   (p <@> q)   == extract p (extract q)
duplicate (p <@> q)   == (<@>) <$> duplicate p <@> duplicate q
```

These laws mean `(<@>)` *must* be “zippy.”

What *is* a `ComonadApply` anyhow?

The laws of `ComonadApply`:

```
(.) <$> u <@> v <@> w == u <@> (v <@> w)
extract   (p <@> q)   == extract p (extract q)
duplicate (p <@> q)   == (<@>) <$> duplicate p <@> duplicate q
```

These laws mean `(<@>)` *must* be “zippy.”

Uustalu and Vene’s *The Essence of Dataflow Programming* calls it:

```
czip :: (ComonadZip d) => d a -> d b -> d (a,b)
```

What *is* a `ComonadApply` anyhow?

The laws of `ComonadApply`:

```
(.) <$> u <@> v <@> w == u <@> (v <@> w)
extract   (p <@> q)   == extract p (extract q)
duplicate (p <@> q)   == (<@>) <$> duplicate p <@> duplicate q
```

These laws mean `(<@>)` *must* be "zippy."

Uustalu and Vene's *The Essence of Dataflow Programming* calls it:

```
czip :: (ComonadZip d) => d a -> d b -> d (a,b)
```

Enlightening exercise: for an arbitrary `Functor f`, show how `czip` and `(<@>)` can be defined in terms of each other and `fmap`.

Zippy comonads $\rightarrow$ zippy computation

The "zippiness" required by the laws of `ComonadApply` is also the source of `evaluate`'s *computational* "zippiness."

Can going fast be total(ly safe)?

Can going fast be total(ly safe)?

Short answer: no.

Long answer: not in ways we would care about.

But I'm more than a one-dimensional character

But I'm more than a one-dimensional character

Nesting `Tape`s inside one another leads us into to higher-dimensional (discrete) spaces to explore.

$$
\begin{aligned}
\texttt{Tape a} &\cong \texttt{Integer} \to \texttt{a} \\
\texttt{Tape (Tape a)} &\cong \texttt{(Integer,Integer)} \to \texttt{a} \\
\texttt{Tape (Tape (Tape a))} &\cong \texttt{(Integer,Integer,Integer)} \to \texttt{a}
\end{aligned}
$$

Etcetera, ad infinitum!

But I'm more than a one-dimensional character

We could define a `newtype` for each added dimension, but this carries an overhead of between $O(n^2)$ and $O(n^3)$ boilerplate per dimension.

But I'm more than a one-dimensional character

We could define a `newtype` for each added dimension, but this carries an overhead of between $O(n^2)$ and $O(n^3)$ boilerplate per dimension.

```
newtype Tape2 a = Tape (Tape a)
newtype Tape3 a = Tape (Tape (Tape a))
...

instance Functor Tape2 where ...
instance Comonad Tape2 where ...
instance ComonadApply Tape2 where ...

instance Functor Tape3 where ...
instance Comonad Tape3 where ...
instance ComonadApply Tape3 where ...
...
```

That also really `succ`s.

But I'm more than a one-dimensional character

Composition of functors (from `Data.Functor.Compose`):

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

(Functor f, Functor g)         => Functor (Compose f g)
(Applicative f, Applicative g) => Applicative (Compose f g)
```

But I'm more than a one-dimensional character

Composition of functors (from `Data.Functor.Compose`):

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

(Functor f, Functor g)         => Functor (Compose f g)
(Applicative f, Applicative g) => Applicative (Compose f g)

instance (Comonad f, Comonad g) => Comonad (Compose f g) where
  extract   = extract . extract . getCompose
  duplicate = ...
```

Do you want to build a comonad?

(N.B. In this section, I've specialized many type signatures.)

What can you do with `(Compose f g a)`**?**

Do you want to build a comonad?

(N.B. In this section, I've specialized many type signatures.)

What can you do with `(Compose f g a)`**?**

Equivalent: what can you do with
`(Comonad f, Comonad g) => f (g a)`?

Do you want to build a comonad?

(N.B. In this section, I've specialized many type signatures.)

What can you do with `(Compose f g a)`**?**

Equivalent: what can you do with
`(Comonad f, Comonad g) => f (g a)`?

Duplicate outer layer:

```
duplicate :: f (g a) -> f (f (g a))
```

Do you want to build a comonad?

(N.B. In this section, I've specialized many type signatures.)

What can you do with `(Compose f g a)`**?**

Equivalent: what can you do with
`(Comonad f, Comonad g) => f (g a)`?

Duplicate outer layer:

```
duplicate :: f (g a) -> f (f (g a))
```

Duplicate inner layer:

```
fmap duplicate :: f (g a) -> f (g (g a))
```

Do you want to build a comonad?

(N.B. In this section, I've specialized many type signatures.)

What can you do with `(Compose f g a)`?

Equivalent: what can you do with
`(Comonad f, Comonad g) => f (g a)`?

Duplicate outer layer:

```
duplicate :: f (g a) -> f (f (g a))
```

Duplicate inner layer:

```
fmap duplicate :: f (g a) -> f (g (g a))
```

Duplicate both:

```
duplicate . fmap duplicate :: f (g a) -> f (f (g (g a)))
```

Do you want to build a comonad?

If only we had `f (f (g (g a))) -> f (g (f (g a)))`...

```
Compose . fmap Compose -- wrap again
. ???                  -- swap middle two layers
. duplicate            -- duplicate outside
. fmap duplicate       -- duplicate inside
. getCompose           -- unwrap
    :: Compose f g a -> Compose f g (Compose f g a)
```

Type sleuth vs. the mysterious functor-swapper

Whatever `???` is, it likely has a more generic type.

Type sleuth vs. the mysterious functor-swapper

Whatever ??? is, it likely has a more generic type.

```
???      ::    f (g x)       ->    g (f x)
fmap ??? :: f (f (g (g a))) -> f (g (f (g a)))
```

Type sleuth vs. the mysterious functor-swapper

Whatever ??? is, it likely has a more generic type.

```
???      ::    f (g x)       ->    g (f x)
fmap ??? :: f (f (g (g a))) -> f (g (f (g a)))
```

```
Compose . fmap Compose -- wrap again
. fmap ???             -- swap middle two layers
. duplicate            -- duplicate outside
. fmap duplicate       -- duplicate inside
. getCompose           -- unwrap
    :: Compose f g a -> Compose f g (Compose f g a)
```

Type sleuth vs. the mysterious functor-swapper

Two candidates (thanks Hoogle!):

```
sequenceA     -- from Data.Traversable
  :: (Traversable t, Applicative f) => t (f a) -> f (t a)

distribute    -- from Data.Distributive
  :: (Distributive g, Functor f)    => f (g a) -> g (f a)
```

Type sleuth vs. the mysterious functor-swapper

```
sequenceA     -- from Data.Traversable
  :: (Traversable t, Applicative f) => t (f a) -> f (t a)
```

Initially promising—I know and love `Traversable`.

Type sleuth vs. the mysterious functor-swapper

```haskell
sequenceA     -- from Data.Traversable
  :: (Traversable t, Applicative f) => t (f a) -> f (t a)
```

Initially promising—I know and love `Traversable`.

Requires two constraints:

- `Applicative f`: outer layer has to have `(<*>)` and `pure`—`pure` is a hard pill to swallow.
- `Traversable t`—that's a deal-breaker!

Type sleuth vs. the mysterious functor-swapper

```
sequenceA     -- from Data.Traversable
  :: (Traversable t, Applicative f) => t (f a) -> f (t a)
```

Initially promising—I know and love `Traversable`.

Requires two constraints:

- `Applicative f`: outer layer has to have `(<*>)` and `pure`—`pure` is a hard pill to swallow.
- `Traversable t`—that's a deal-breaker!

Jaskelioff & Rypacek, MSFP 2012, "An Investigation of the Laws of Traversals": "We are not aware of any functor that is traversable and is not a finitary container."

- Infinite streams are definitely not `Traversable`.

Type sleuth vs. the mysterious functor-swapper

```
distribute   -- from Data.Distributive
  :: (Distributive g, Functor f) => f (g a) -> g (f a)
```

But what does `Distributive` mean?

Type sleuth vs. the mysterious functor-swapper

```
distribute   -- from Data.Distributive
  :: (Distributive g, Functor f) => f (g a) -> g (f a)
```

But what does `Distributive` mean?

What can you do underneath a `Functor`?

Type sleuth vs. the mysterious functor-swapper

```
distribute   -- from Data.Distributive
  :: (Distributive g, Functor f) => f (g a) -> g (f a)
```

But what does `Distributive` mean?

What can you do underneath a `Functor`?

"Touch, don't look."

Type sleuth vs. the mysterious functor-swapper

```
distribute    -- from Data.Distributive
  :: (Distributive g, Functor f) => f (g a) -> g (f a)
```

Strategy/intuition for `distribute`:

- Start with `f (g a)`
- Create a `g` with `f (g a)` in each 'hole': `g (f (g a))`
- For each `f (g a)` on the inside of `g`:
  - navigate to a particular focus (using `fmap`)
  - `fmap extract` to eliminate the inner `g`
- Result: `g (f a)`

Mystery solved

```haskell
instance (Comonad f, Comonad g, Distributive g)
  => Comonad (Compose f g) where
extract   = extract . extract . getCompose
duplicate = Compose . fmap Compose -- wrap again
            . distribute           -- swap middle two layers
            . duplicate            -- duplicate outside
            . fmap duplicate       -- duplicate inside
            . getCompose           -- unwrap
```

Mystery solved

```
unfold prev center next x =
  Tape (S.unfold prev x) (center x) (S.unfold next x)

instance Distributive Tape where
  distribute =
    unfold (fmap (focus . moveL) &&& fmap moveL)
           (fmap focus)
           (fmap (focus . moveR) &&& fmap moveR)
```

The story so far

The story so far

Efficient evaluation:

```
evaluate :: ComonadApply w => w (w a -> a) -> w a
```

The story so far

Efficient evaluation:

```
evaluate :: ComonadApply w => w (w a -> a) -> w a
```

Elegant composition:

```
(Comonad f, Comonad g, Distributive g) => Comonad (Compose f g)
```

The story so far

Efficient evaluation:

```
evaluate :: ComonadApply w => w (w a -> a) -> w a
```

Elegant composition:

```
(Comonad f, Comonad g, Distributive g) => Comonad (Compose f g)
```

I could make a library out of this!

```
{-# LANGUAGE OverlappingInstances #-}
```

Baby, there's a shark in the water

```
type family ComposeCount f where
  ComposeCount (Compose f g a) = Succ (ComposeCount (f (g a)))
  ComposeCount x               = Zero

class CountCompose f where
  countCompose :: f -> ComposeCount f
```

Baby, there's a shark in the water

```haskell
{-# LANGUAGE FeelBadAboutYourself #-}

type family ComposeCount f where
  ComposeCount (Compose f g a) = Succ (ComposeCount (f (g a)))
  ComposeCount x               = Zero

class CountCompose f where
  countCompose :: f -> ComposeCount f
```

Baby, there's a shark in the water

```
{-# LANGUAGE OverlappingInstances #-}

type family ComposeCount f where
  ComposeCount (Compose f g a) = Succ (ComposeCount (f (g a)))
  ComposeCount x               = Zero

class CountCompose f where
  countCompose :: f -> ComposeCount f
```

Baby, there's a shark in the water

```
{-# LANGUAGE OverlappingInstances #-}

type family ComposeCount f where
  ComposeCount (Compose f g a) = Succ (ComposeCount (f (g a)))
  ComposeCount x               = Zero

class CountCompose f where
  countCompose :: f -> ComposeCount f

instance (CountCompose (f (g a)))
  => CountCompose (Compose f g a) where
  countCompose (Compose x) = Succ (countCompose x)
```

Baby, there's a shark in the water

```haskell
{-# LANGUAGE OverlappingInstances #-}

type family ComposeCount f where
  ComposeCount (Compose f g a) = Succ (ComposeCount (f (g a)))
  ComposeCount x               = Zero

class CountCompose f where
  countCompose :: f -> ComposeCount f

instance (CountCompose (f (g a)))
  => CountCompose (Compose f g a) where
  countCompose (Compose x) = Succ (countCompose x)

instance (ComposeCount f ~ Zero) => CountCompose f where
  countCompose _ = Zero
```

GADTs to the rescue!

Previously:

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

GADTs to the rescue!

Previously:

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

What if we put *depth of nesting* in the types?

```haskell
data Flat x
data Nest o i

data Nested fs a where
   Flat :: f a -> Nested (Flat f) a
   Nest :: Nested fs (f a) -> Nested (Nest fs f) a
```

GADTs to the rescue!

Previously:

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

What if we put *depth of nesting* in the types?

```
data Flat x
data Nest o i

data Nested fs a where
   Flat :: f a -> Nested (Flat f) a
   Nest :: Nested fs (f a) -> Nested (Nest fs f) a

            Just [1]    :: Maybe [Int]
      Flat (Just [1])   :: Nested (Flat Maybe) [Int]
Nest (Flat (Just [1])) :: Nested (Nest (Flat Maybe) []) Int
```

Nest it / `fmap` it / quick rewrap it

Two cases for each instance (base case/recursive case):

```
instance (Functor f) => Functor (Nested (Flat f)) where
   fmap f (Flat x) = Flat $ fmap f x

instance (Functor f, Functor (Nested fs))
   => Functor (Nested (Nest fs f)) where
   fmap f (Nest x) = Nest $ fmap (fmap f) x
```

The rest of the instances for look similar.

Nest it / `fmap` it / quick rewrap it

- Match on types, not constraints
- Base case is `Flat`, not every type, so no "universal instance"
- See ya later, `OverlappingInstances`!

Drag and drop it / zip - unzip it

What's in a reference?

Drag and drop it / zip - unzip it

What's in a reference?

```haskell
{-# LANGUAGE DataKinds #-}

data RefType = Relative | Absolute

data Ref (t :: RefType) where
   Rel :: Int -> Ref Relative
   Abs :: Int -> Ref Absolute
```

Drag and drop it / zip - unzip it

```
type family Combine a b where
   Combine Relative Absolute = Absolute
   Combine Absolute Relative = Absolute
   Combine Relative Relative = Relative
```

Drag and drop it / zip - unzip it

```haskell
type family Combine a b where
   Combine Relative Absolute = Absolute
   Combine Absolute Relative = Absolute
   Combine Relative Relative = Relative

class CombineRefs a b where ...
instance CombineRefs Absolute Relative where ...
instance CombineRefs Relative Absolute where ...
instance CombineRefs Relative Relative where ...
```

Drag and drop it / zip - unzip it

```
type family Combine a b where
   Combine Relative Absolute = Absolute
   Combine Absolute Relative = Absolute
   Combine Relative Relative = Relative

class CombineRefs a b where ...
instance CombineRefs Absolute Relative where ...
instance CombineRefs Relative Absolute where ...
instance CombineRefs Relative Relative where ...

... combine :: Ref a -> Ref b -> Ref (Combine a b)
... combine (Abs a) (Rel b) = Abs (a + b)
... combine (Rel a) (Abs b) = Abs (a + b)
... combine (Rel a) (Rel b) = Rel (a + b)
```

Drag and drop it / zip - unzip it

```
type family Combine a b where
   Combine Relative Absolute = Absolute
   Combine Absolute Relative = Absolute
   Combine Relative Relative = Relative

class CombineRefs a b where ...
instance CombineRefs Absolute Relative where ...
instance CombineRefs Relative Absolute where ...
instance CombineRefs Relative Relative where ...

... combine :: Ref a -> Ref b -> Ref (Combine a b)
... combine (Abs a) (Rel b) = Abs (a + b)
... combine (Rel a) (Abs b) = Abs (a + b)
... combine (Rel a) (Rel b) = Rel (a + b)
```

- Split presentation style due to Conor McBride, JFP 2001: *Faking It: Simulating Dependent Types in Haskell*

He's making a list and checking it statically

```
data x :-: y
data Nil

data ConicList f ts where
   (:-:) :: f x -> ConicList f xs -> ConicList f (x :-: xs)
   ConicNil  :: ConicList f Nil

type RefList = ConicList Ref
```

It's called a conic list because category theory: `(forall a. f a -> x)` is known as a *co-cone* from `f` to `x`, and this is sort of like that.

He's making a list and checking it statically

```
type family a & b where
   (a :-: as) & (b :-: bs) = Combine a b :-: (as & bs)
   Nil        & bs         = bs
   as         & Nil        = as
```

He's making a list and checking it statically

```
type family a & b where
   (a :-: as) & (b :-: bs) = Combine a b :-: (as & bs)
   Nil        & bs         = bs
   as         & Nil        = as

class CombineRefLists as bs where ...
instance (CombineRefs a b, CombineRefLists as bs)
      => CombineRefLists (a :-: as) (b :-: bs) where ...
instance CombineRefLists Nil        (b :-: bs) where ...
instance CombineRefLists (a :-: as) Nil        where ...
instance CombineRefLists Nil        Nil        where ...
```

He's making a list and checking it statically

```
type family a & b where
   (a :-: as) & (b :-: bs) = Combine a b :-: (as & bs)
   Nil        & bs         = bs
   as         & Nil        = as

class CombineRefLists as bs where ...
instance (CombineRefs a b, CombineRefLists as bs)
      => CombineRefLists (a :-: as) (b :-: bs) where ...
instance CombineRefLists Nil        (b :-: bs) where ...
instance CombineRefLists (a :-: as) Nil        where ...
instance CombineRefLists Nil        Nil        where ...

... (&) :: RefList as -> RefList bs -> RefList (as & bs)
... (a :-: as) & (b :-: bs) = combine a b :-: (as & bs)
... ConicNil   & bs         = bs
... as         & ConicNil   = as
... ConicNil   & ConicNil   = ConicNil
```

He's making a list and checking it statically

With suitable definition of names. . .

```
a :: RefList (Relative :-: Relative :-: Nil)
a = belowBy 3 & rightBy 14
```

He's making a list and checking it statically

With suitable definition of names…

```
a :: RefList (Relative :-: Relative :-: Nil)
a = belowBy 3 & rightBy 14

b :: RefList (Relative :-: Absolute :-: Nil)
b = columnAt 9000 & aboveBy 1
```

He's making a list and checking it statically

With suitable definition of names...

```
a :: RefList (Relative :-: Relative :-: Nil)
a = belowBy 3 & rightBy 14

b :: RefList (Relative :-: Absolute :-: Nil)
b = columnAt 9000 & aboveBy 1

c = columnAt 5 & columnAt 10
```

Take it / view it / go - insert it

```
class Take r t where
   type ListFrom t a
   take :: RefList r -> t a -> ListFrom t a

class View r t where
   type StreamFrom t a
   view :: RefList r -> t a -> StreamFrom t a

class Go r t where
   go :: RefList r -> t a -> t a
```

Take it / view it / go - insert it

```
class Take r t where
   type ListFrom t a
   take :: RefList r -> t a -> ListFrom t a

class View r t where
   type StreamFrom t a
   view :: RefList r -> t a -> StreamFrom t a

class Go r t where
   go :: RefList r -> t a -> t a
```

...and `insert` — I have discovered a truly marvelous type signature for this, which this margin is too narrow to contain.

What have we learned?

- Efficient comonadic fixed-point requires zipping
- Distributive comonads compose
- Dimension polymorphism needs type-indexed composition
- Heterogeneous lists unify absolute and relative references

What have we learned?

- Efficient comonadic fixed-point requires zipping
- Distributive comonads compose
- Dimension polymorphism needs type-indexed composition
- Heterogeneous lists unify absolute and relative references

- (Co)monads are (co)ol!

With great power comes code snippets for a tech talk

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

(I told you the syntax would get nicer!)

With great power comes code snippets for a tech talk

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

(I told you the syntax would get nicer!)

```
> slice (leftBy 2) (rightBy 17) fibonacci
[1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584]
```

With great power comes code snippets for a tech talk

```haskell
pascal :: Sheet2 Integer
pascal = evaluate . sheet 0 $
   repeat 1 <:> repeat (1 <:> pascalRow)
   where pascalRow = repeat $ cell above + cell left
```

With great power comes code snippets for a tech talk

```
pascal :: Sheet2 Integer
pascal = evaluate . sheet 0 $
   repeat 1 <:> repeat (1 <:> pascalRow)
   where pascalRow = repeat $ cell above + cell left

> take (belowBy 9 & rightBy 9) pascal
```

With great power comes code snippets for a tech talk

```
pascal :: Sheet2 Integer
pascal = evaluate . sheet 0 $
   repeat 1 <:> repeat (1 <:> pascalRow)
   where pascalRow = repeat $ cell above + cell left

> take (belowBy 9 & rightBy 9) pascal

[[1,  1,  1,   1,   1,    1,    1,     1,     1,     1],
 [1,  2,  3,   4,   5,    6,    7,     8,     9,    10],
 [1,  3,  6,  10,  15,   21,   28,    36,    45,    55],
 [1,  4, 10,  20,  35,   56,   84,   120,   165,   220],
 [1,  5, 15,  35,  70,  126,  210,   330,   495,   715],
 [1,  6, 21,  56, 126,  252,  462,   792,  1287,  2002],
 [1,  7, 28,  84, 210,  462,  924,  1716,  3003,  5005],
 [1,  8, 36, 120, 330,  792, 1716,  3432,  6435, 11440],
 [1,  9, 45, 165, 495, 1287, 3003,  6435, 12870, 24310],
 [1, 10, 55, 220, 715, 2002, 5005, 11440, 24310, 48620]]
```

With great power comes code snippets for a tech talk

```haskell
data Cell = X | O deriving ( Eq )

life :: ([Int],[Int]) -> [[Cell]] -> Sheet3 Cell
life ruleset seed =
   evaluate $ insert [map (map const) seed] blank where
     blank = sheet (const X) (repeat . tapeOf . tapeOf $ rule)
     rule place =
       case (neighbors place `elem`) `onBoth` ruleset of
            (True,_) -> O
            (_,True) -> cell inward place
            _        -> X
     neighbors = length . filter (O ==) . cells bordering
     bordering = map (inward &) (diag ++ vert ++ horz)
     diag = (&) <$> horizontals <*> verticals
     vert =        [above, below]
     horz = map d2 [right, left]

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

With great power comes code snippets for a tech talk

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

With great power comes code snippets for a tech talk

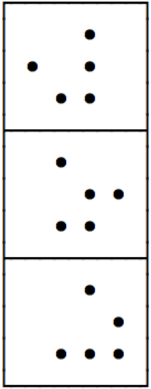
```
glider :: Sheet3 Cell
glider = conway [[X,X,O],
                 [O,X,O],
                 [X,O,O]]
```

```
> printLife glider
```

With great power comes code snippets for a tech talk

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

```
> printLife glider
```



`cabal install ComonadSheet`

`github.com/kwf/ComonadSheet`

Suggestions, bug reports, pull requests welcome!