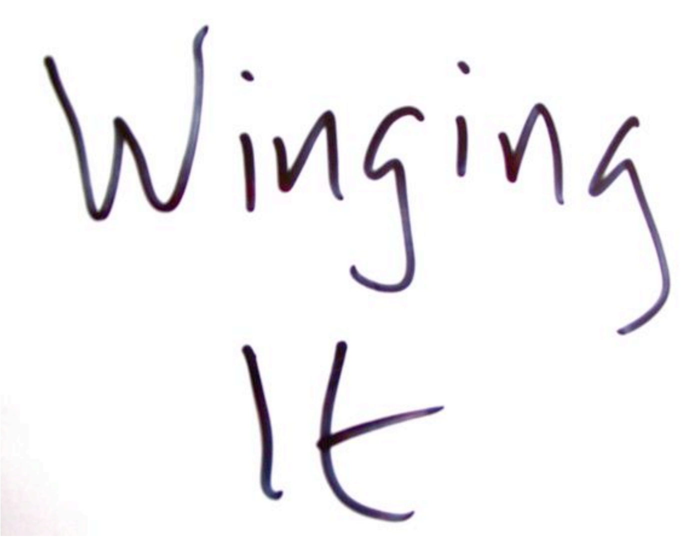


Winging It

alas, poor

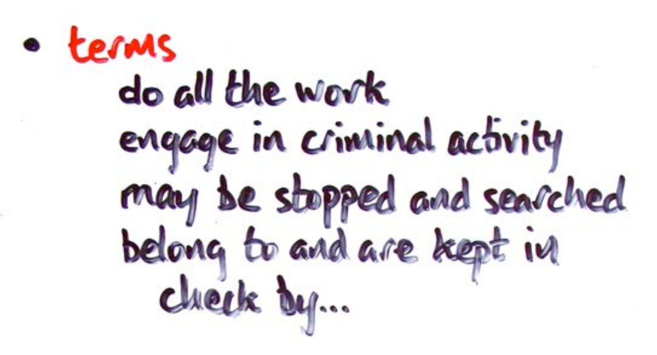
Conor McBride

(WG2.8 #26, Frauenchiemsee, 8-12 June 2009)

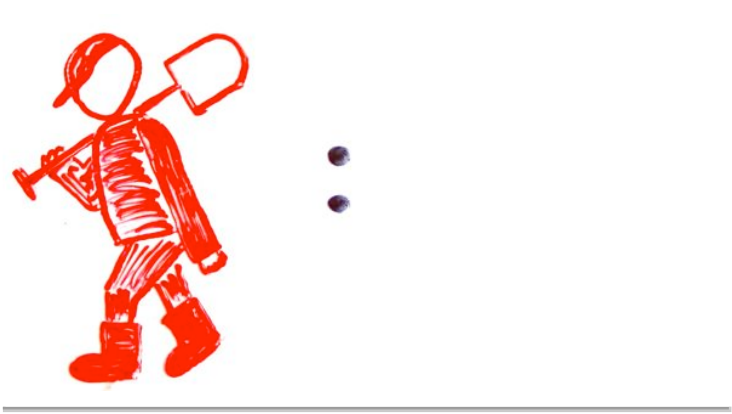
the established social order

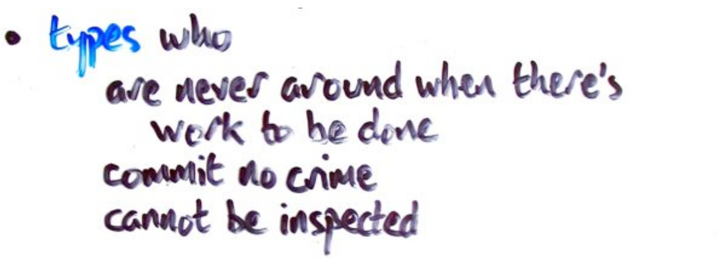
This slide is nearly ten years old. How times change!

:



- terms
  do all the work
  engage in criminal activity
  may be stopped and searched
  belong to and are kept in
  check by...





- types who
  - are never around when there's work to be done
  - commit no crime
  - cannot be inspected

breaking the old social order

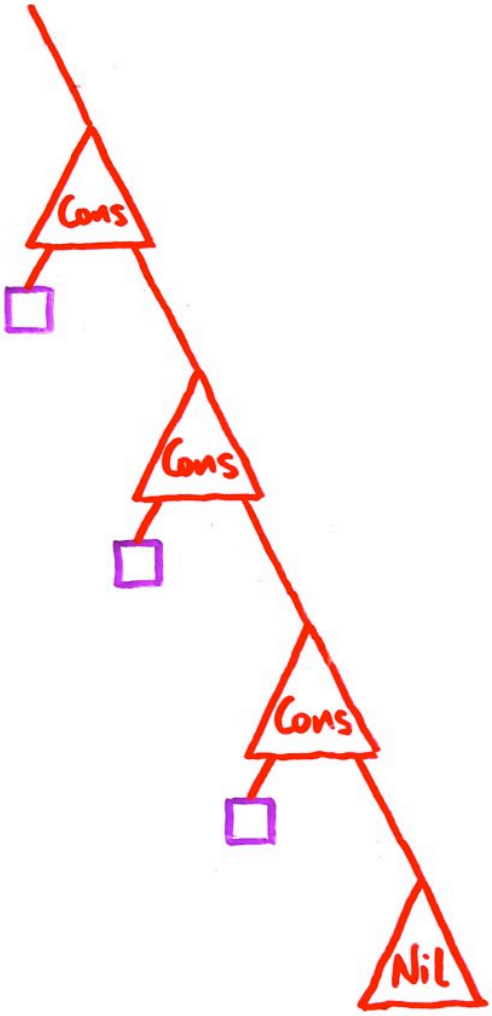
- data is validated with respect to other data
- if types are to capture valid data precisely, we must let them depend on terms

This remains the central point of the case for dependent types. In this talk, I consider validating containers with respect to shape, syntax with respect to type, but most especially, *interaction with respect to circumstances*.

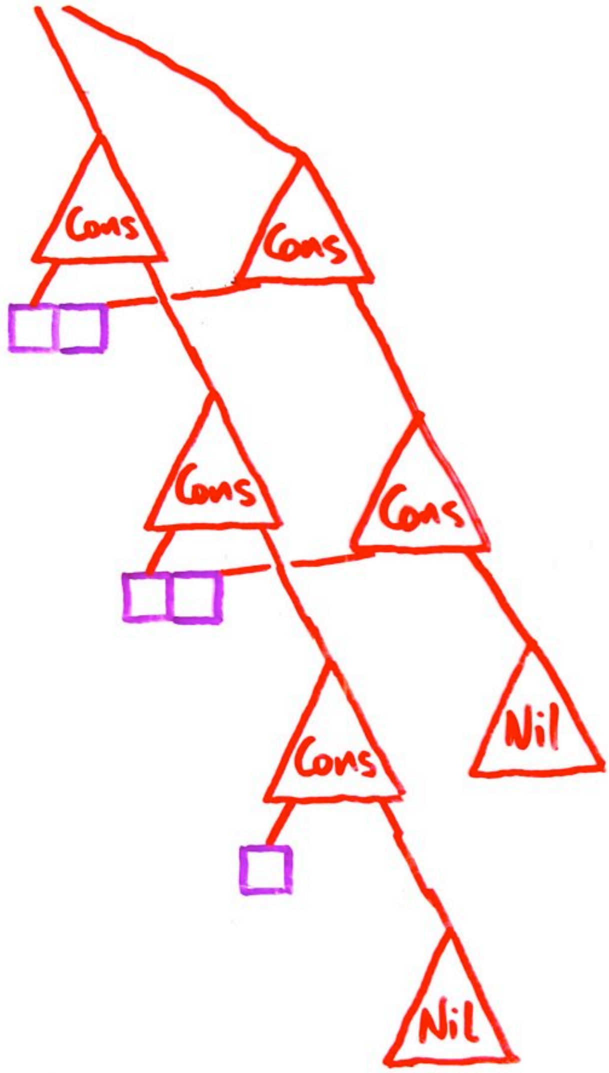
We still need the revolution, but it seems less controversial these days.

Do the types run off in terror?

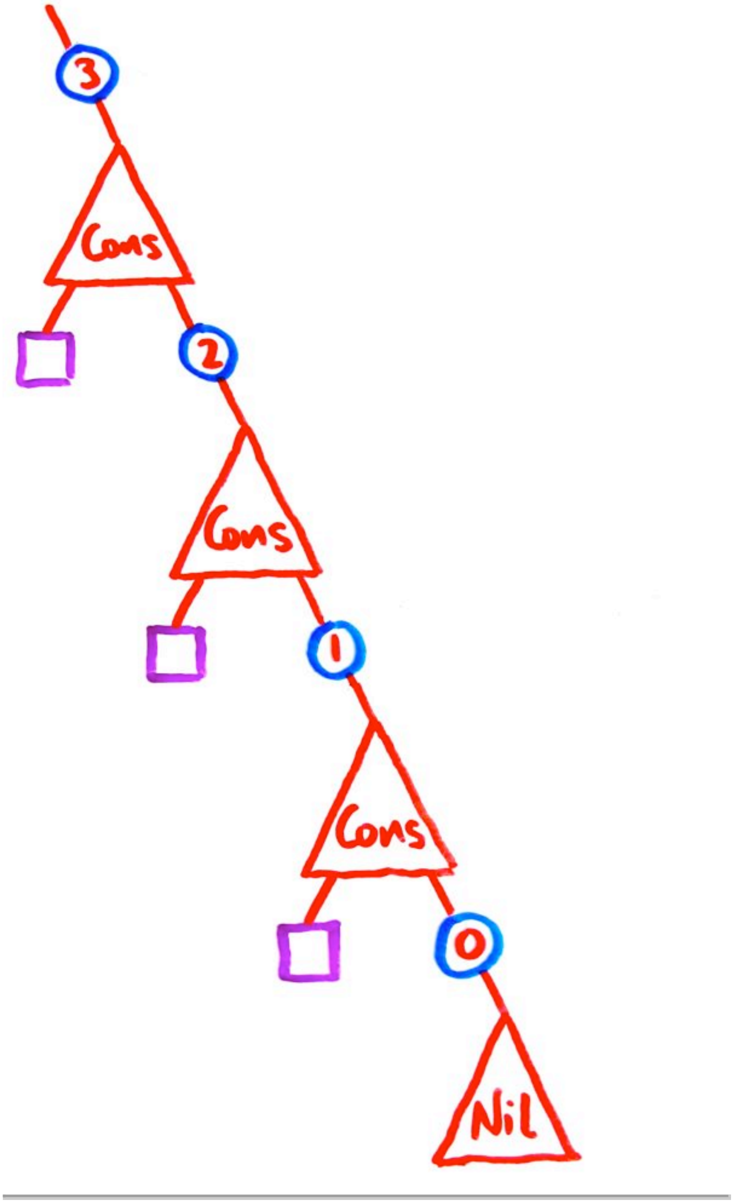
No! They're overjoyed at their new articulacy.



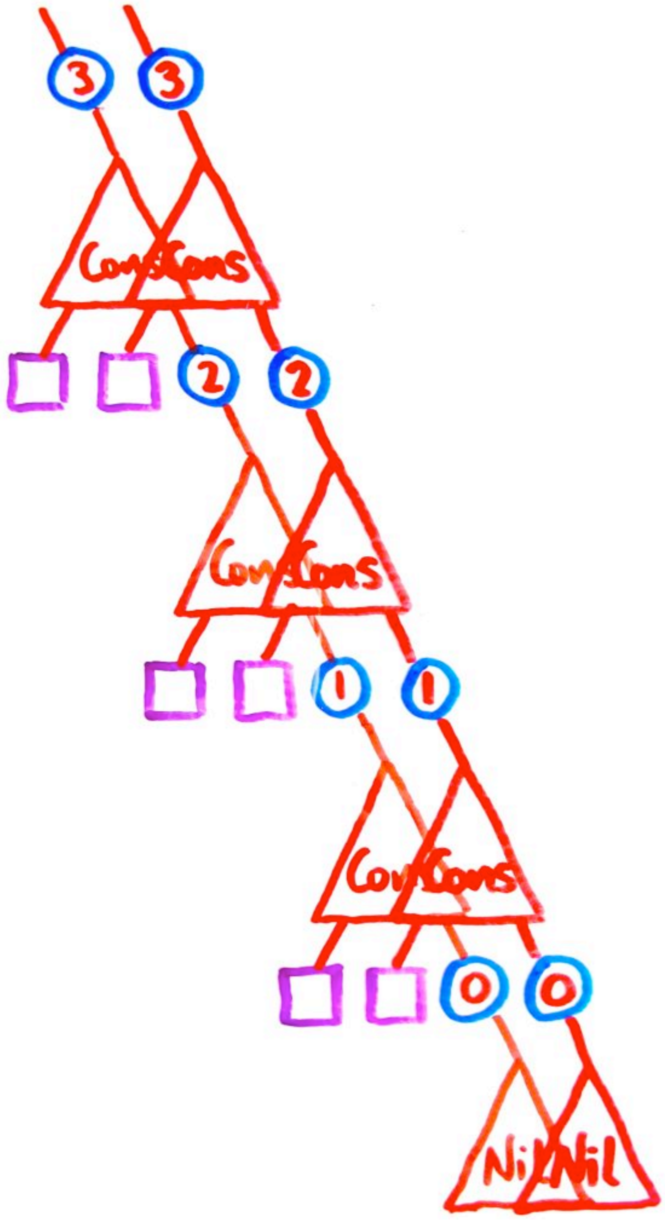
Here's a picture of a list, seen as a recursive data structure.



They don't zip so well, some times.



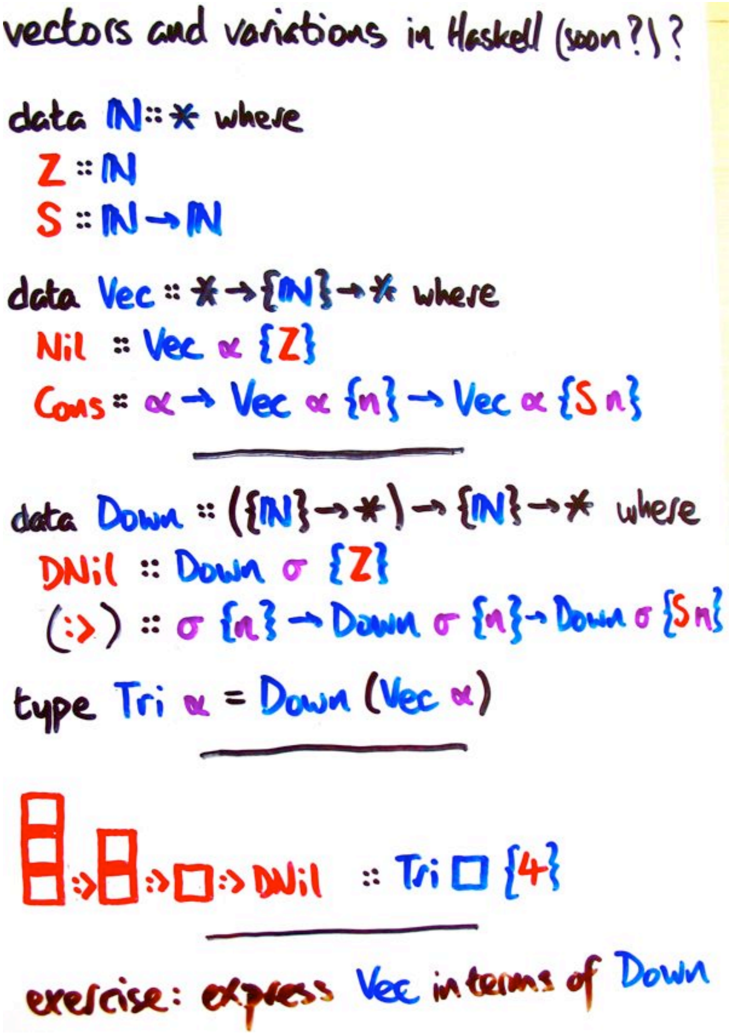
But if each node makes clear what length it delivers and what length subnode it is willing to accept,...



...zipping is always spot on.

social mobility in modern day Haskell – new kinds for types

```
κ ::= *
    | κ → κ
    | {τ}
    | ∀α::κ. κ

τ ::= … | {ce}
```



vectors and variations in Haskell (soon?)?

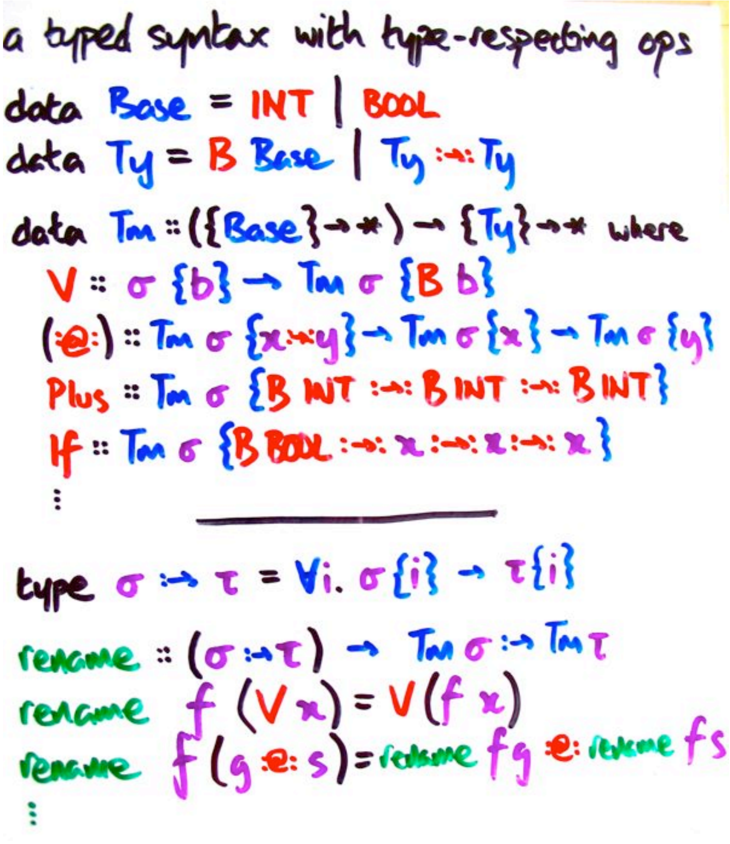
```
data ℕ :: * where
  Z :: ℕ
  S :: ℕ → ℕ

data Vec :: * → {ℕ} → * where
  Nil  :: Vec α {Z}
  Cons :: α → Vec α {n} → Vec α {S n}
```

```
data Down :: ({ℕ} → *) → {ℕ} → * where
  DNil :: Down σ {Z}
  (:>) :: σ {n} → Down σ {n} → Down σ {S n}

type Tri α = Down (Vec α)
```

```
:> :> □ :> DNil :: Tri □ {4}
```

---

exercise: express Vec in terms of Down



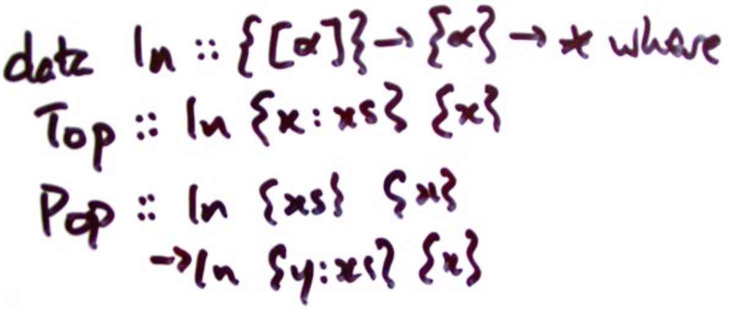
a typed syntax with type-respecting ops

```
data Base = INT | BOOL
data Ty = B Base | Ty :->: Ty

data Tm :: ({Base}->*) -> {Ty}->* where
  V :: σ {b} -> Tm σ {B b}
  (:@:) :: Tm σ {x:->:y} -> Tm σ {x} -> Tm σ {y}
  Plus :: Tm σ {B INT :->: B INT :->: B INT}
  If :: Tm σ {B BOOL :->: x :->: x :->: x}
  ⋮
```

```
type σ :-> τ = ∀i. σ{i} -> τ{i}
rename :: (σ :-> τ) -> Tm σ :-> Tm τ
rename f (V x) = V (f x)
rename f (g :@: s) = rename f g :@: rename f s
⋮
```

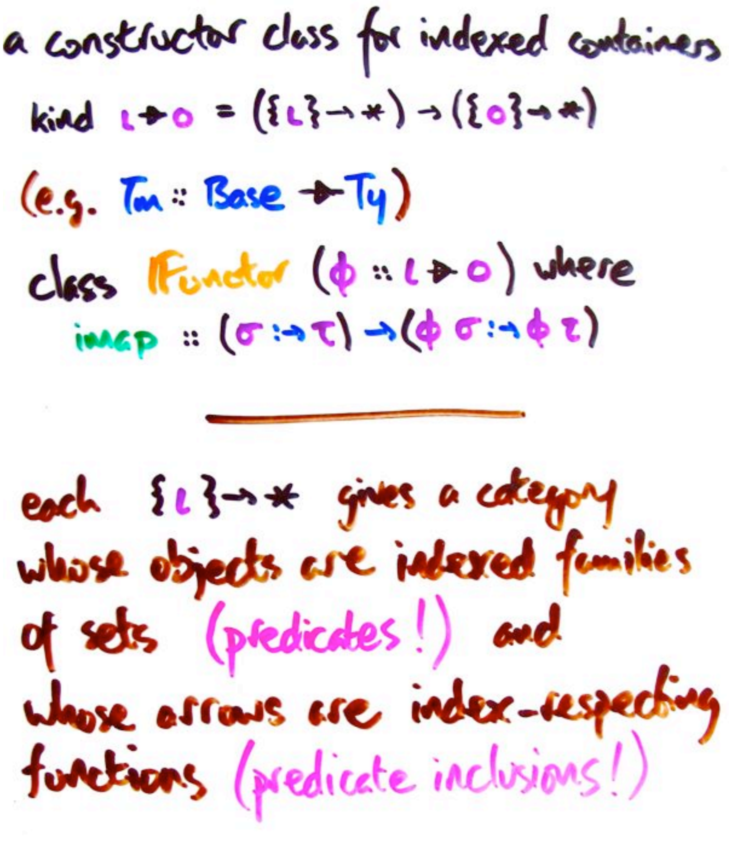
It's sensible, and even becoming traditional, to abstract your syntax over the type used to represent variables, pointing to the functorial structure of renaming and the monadic structure of substitution.

I'm just playing the same trick, with *typed* syntax. We're working in slightly fancier (i.e., slice) categories, but the structure is the same, so the code is the same. You'd expect that if your renaming maps each variable to another of the same (base) type, then deploying it will preserve the (arbitrary) type of any term. And that's what you get, at no extra charge.



```
data In :: {[α]} -> {α} -> * where
  Top :: In {x : xs} {x}
  Pop :: In {xs} {x}
      -> In {y : xs} {x}
```

It may help to give a candidate for σ :: {Base}→*, representing variables. A common choice is In {Γ}, where Γ :: [Base] represents a context. This definition is at least a decade old — Altenkirch and Reus used it exactly as this typed version of de Bruijn indices in their CSL 1999 paper.



a constructor class for indexed containers

kind $\iota \blacktriangleright o = (\{\iota\} \to *) \to (\{o\} \to *)$

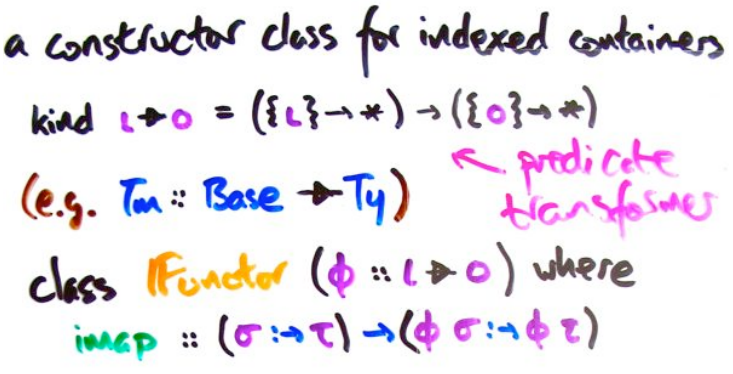
(e.g. $\mathsf{Tm} :: \mathsf{Base} \blacktriangleright \mathsf{Ty}$)

class IFunctor ($\phi :: \iota \blacktriangleright o$) where
  imap :: $(\sigma \mathrel{:\to} \tau) \to (\phi\ \sigma \mathrel{:\to} \phi\ \tau)$

each $\{\iota\} \to *$ gives a category whose objects are indexed families of sets (predicates!) and whose arrows are index-respecting functions (predicate inclusions!)

I should, of course, mention

```
instance IFunctor Tm where
  imap = rename
```



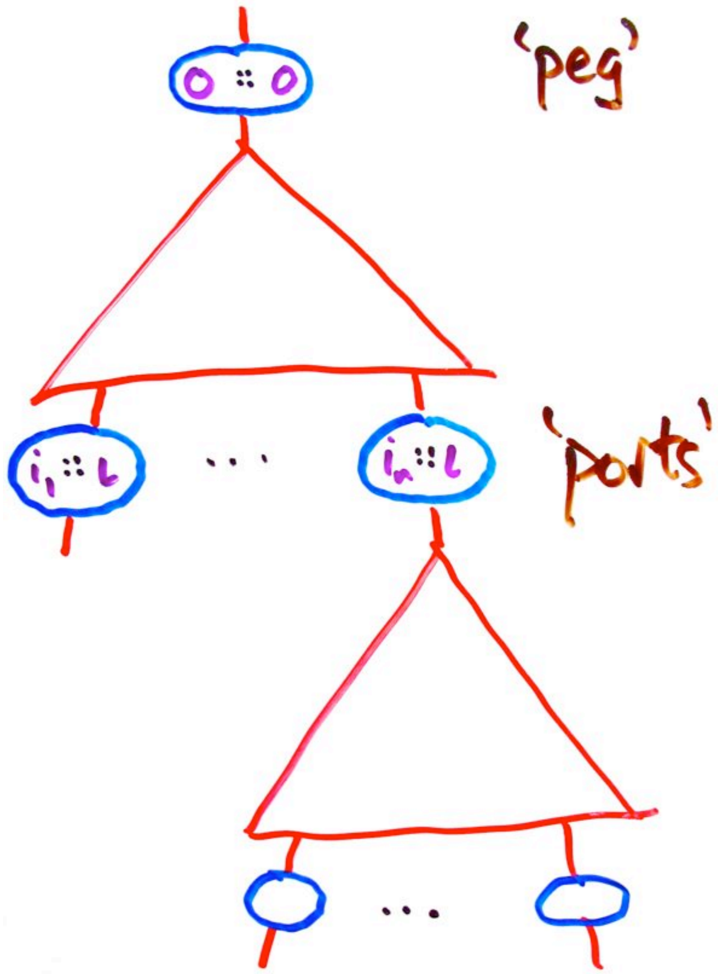
a constructor class for indexed containers

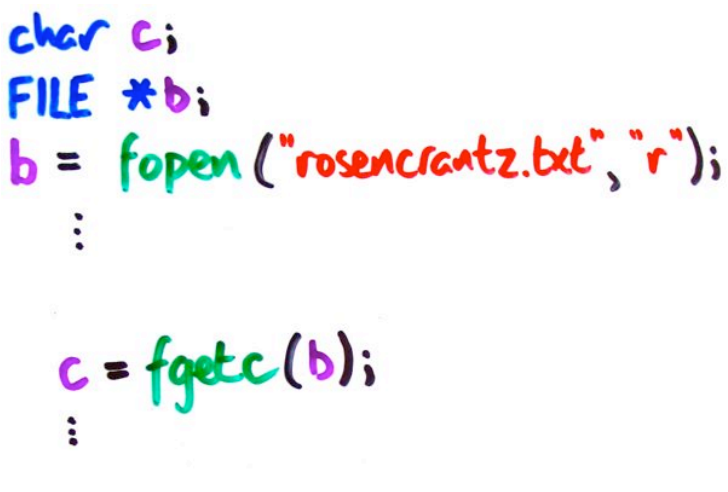
kind $\iota \blacktriangleright o = (\{\iota\} \to *) \to (\{o\} \to *)$ ← predicate transformer

(e.g. $\mathsf{Tm} :: \mathsf{Base} \blacktriangleright \mathsf{Ty}$)

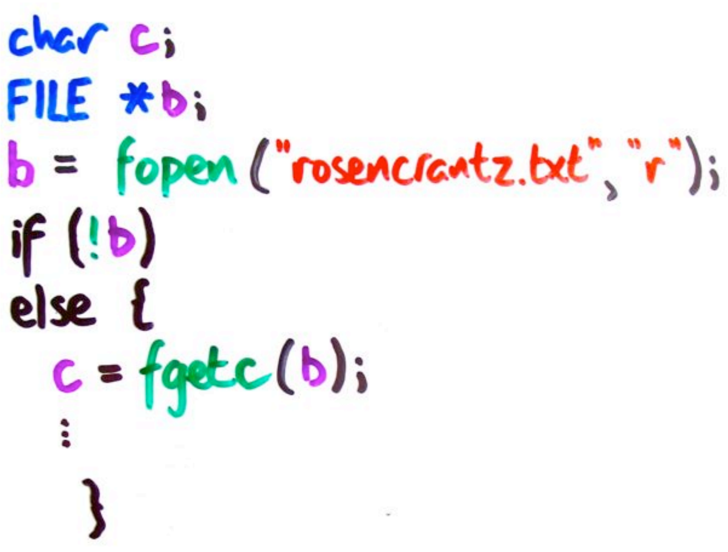
class IFunctor ($\phi :: \iota \blacktriangleright o$) where
  imap :: $(\sigma \mathrel{:\to} \tau) \to (\phi\ \sigma \mathrel{:\to} \phi\ \tau)$

An IFunctor is, specifically, a *monotone* predicate transformer.



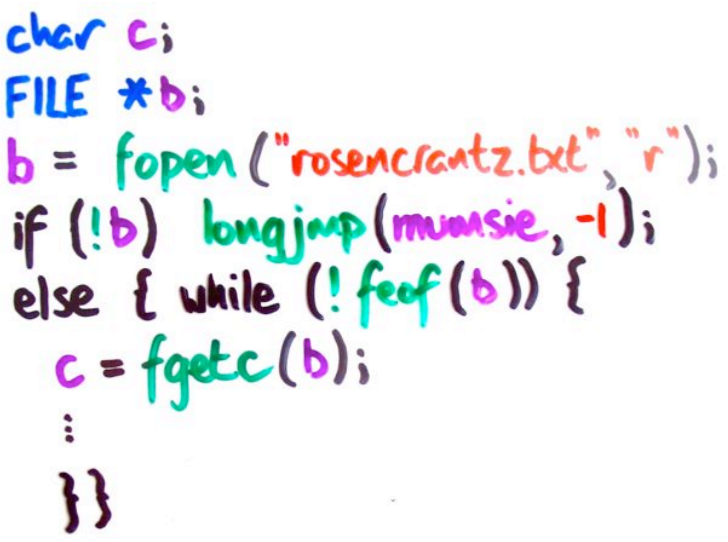

```
char c;
FILE *b;
b = fopen("rosencrantz.txt","r");
  ⋮

  c = fgetc(b);
  ⋮
```

b or not b, that is the question:



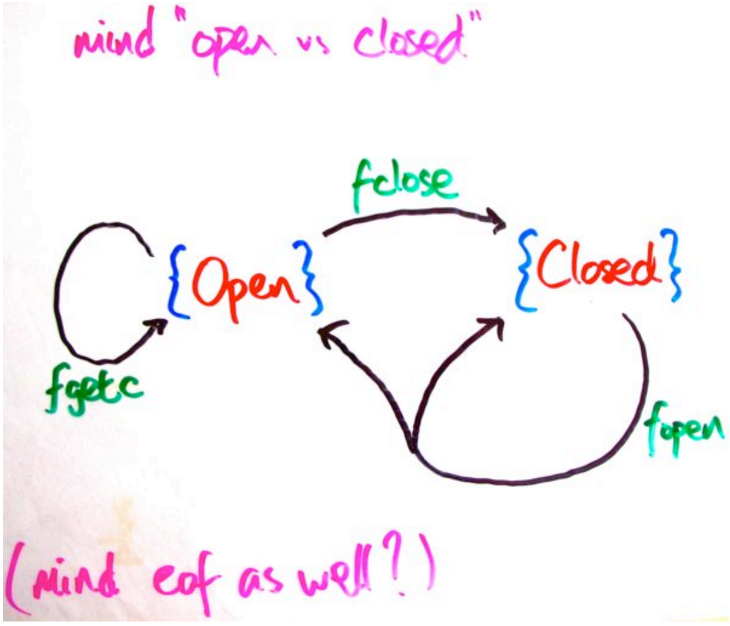
```
char c;
FILE *b;
b = fopen("rosencrantz.txt", "r");
if (!b)
else {
  c = fgetc(b);
  ⋮
   }
```

Whether 'tis nobler in the mind to suffer
The slings and arrows of outrageous fortune,



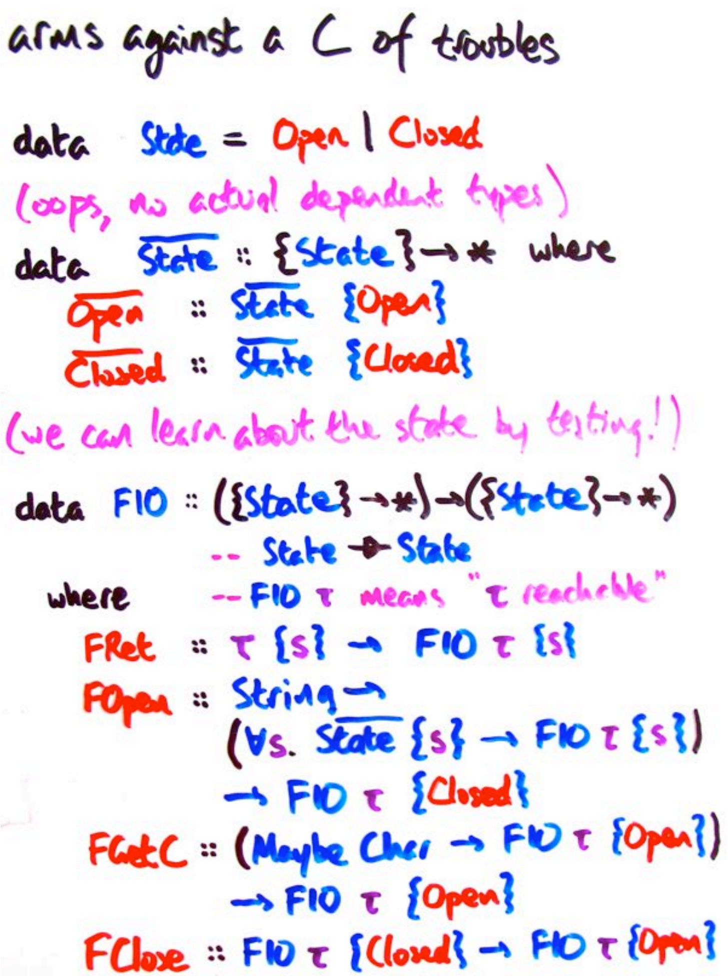
```
char c;
FILE *b;
b = fopen("rosencrantz.txt", "r");
if (!b)  longjmp(mumsie, -1);
else { while (!feof(b)) {
  c = fgetc(b);
  ⋮
  }}
```

Or to take arms against a sea of troubles



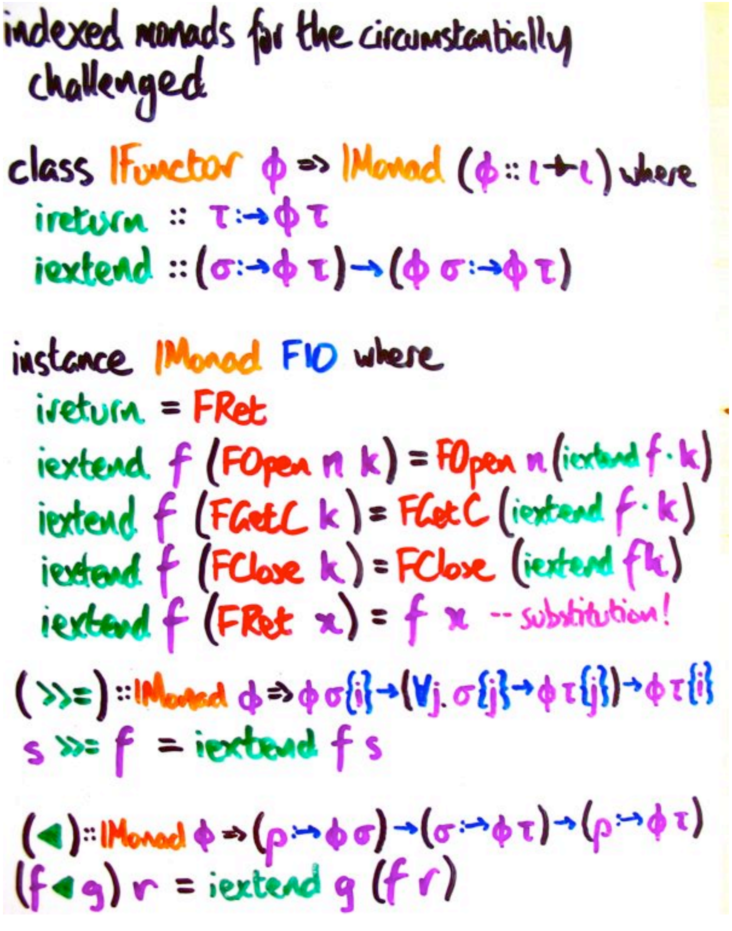
mind "open vs closed"

(mind eof as well?)



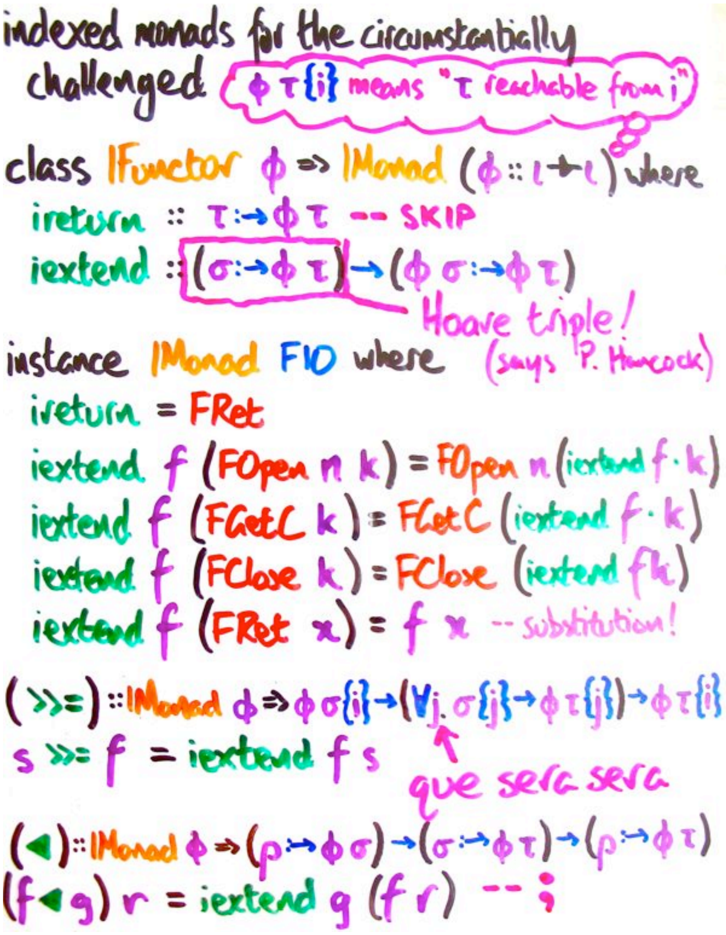
arms against a C of troubles

```
data  State = Open | Closed
(oops, no actual dependent types)
data  State :: {State} → *  where
  Open    :: State  {Open}
  Closed  :: State  {Closed}
(we can learn about the state by testing!)

data  FIO :: ({State} → *) → ({State} → *)
                -- State → State
  where         -- FIO τ  means  "τ reachable"
    FRet   :: τ {s} →  FIO τ {s}
    FOpen  :: String →
              (∀s. State {s} → FIO τ {s})
              → FIO τ  {Closed}
    FGetC  :: (Maybe Char → FIO τ {Open})
              → FIO  τ  {Open}
    FClose :: FIO τ {Closed} → FIO τ {Open}
```



indexed monads for the circumstantially challenged

```
class IFunctor φ => IMonad (φ :: ι → ι) where
  ireturn  ::  τ :→ φ τ
  iextend  :: (σ :→ φ τ) → (φ σ :→ φ τ)

instance IMonad FIO where
  ireturn = FRet
  iextend f (FOpen n k) = FOpen n (iextend f · k)
  iextend f (FGetC k) = FGetC (iextend f · k)
  iextend f (FClose k) = FClose (iextend f k)
  iextend f (FRet x) = f x  -- substitution!

(>>=) :: IMonad φ ⇒ φ σ {i} → (∀j. σ {j} → φ τ {j}) → φ τ {i}
s >>= f  = iextend f s

(◀) :: IMonad φ ⇒ (ρ :→ φ σ) → (σ :→ φ τ) → (ρ :→ φ τ)
(f ◀ g) r = iextend g (f r)
```



```
indexed monads for the circumstantially
  challenged   (φ τ{i} means "τ reachable from i")

class IFunctor φ => IMonad (φ :: ι → ι) where
  ireturn :: τ :→ φ τ  -- SKIP
  iextend :: (σ :→ φ τ) → (φ σ :→ φ τ)
                          Hoare triple!

instance IMonad FIO where   (says P. Hancock)
  ireturn = FRet
  iextend f (FOpen n k) = FOpen n (iextend f · k)
  iextend f (FGetC k) = FGetC (iextend f · k)
  iextend f (FClose k) = FClose (iextend f k)
  iextend f (FRet x) = f x  -- substitution!

(>>=) :: IMonad φ => φ σ{i} → (∀j. σ{j} → φ τ{j}) → φ τ{i}
s >>= f = iextend f s
                            que sera sera

(◄) :: IMonad φ => (ρ :→ φ σ) → (σ :→ φ τ) → (ρ :→ φ τ)
(f ◄ g) r = iextend g (f r)  -- ;
```

σ :→ φ τ = ∀ j. σ{j} → τ{j}
is the type of an arrow of outrageous fortune.

The *world* gets to choose what state we're in, but must provide evidence that it satisfies the precondition σ.

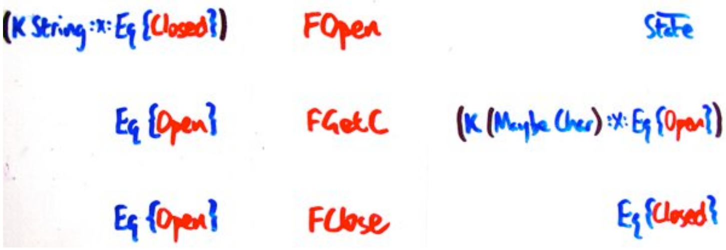
This is by contrast with the “parameterised monads” of Atkey and others.

```
class PMonad (ψ :: {ι} → * → {ι} → *) where
  preturn :: ∀ α, i.    α → ψ {i} α {i}
  pbind :: ∀ α, β, i, j, k.
           ψ {i} α {j} →
           (α → ψ {j} β {k}) →
           ψ {i} β {k}
```

A PMonad does not permit outrageous fortune: in any call to pbind, we fix the intermediate state {j} is fixed up front, and the world must deliver it. PMonads thus give access to the predictable fragment of effectful computation. Every IMonad can be specialized to the PMonad of its predictable fragment by using

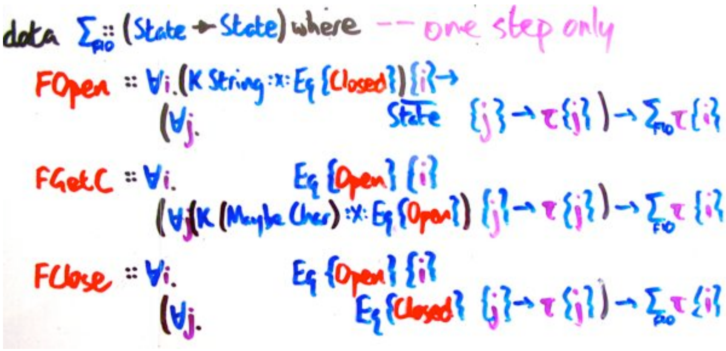
```
data Eq :: {ι} → {ι} → * where
  Refl :: Eq {i} {i}

data K :: * → {ι} → * where
  K :: α → K α {i}

data (:∧:) :: ({ι} → *) → ({ι} → *) → {ι} → *
where
  (:&:) :: σ{i} → τ{i} → (σ:∧:τ){i}

newtype Predict φ i α j =
  Ensure (φ (Eq {j} :∧: K α) {i})
```

That is, Predict φ i α j is the type of φ computations starting from state {i} which reach a state considered satisfactory if it happens to be the state {j} we predicted (and if we have a value in α, to boot).



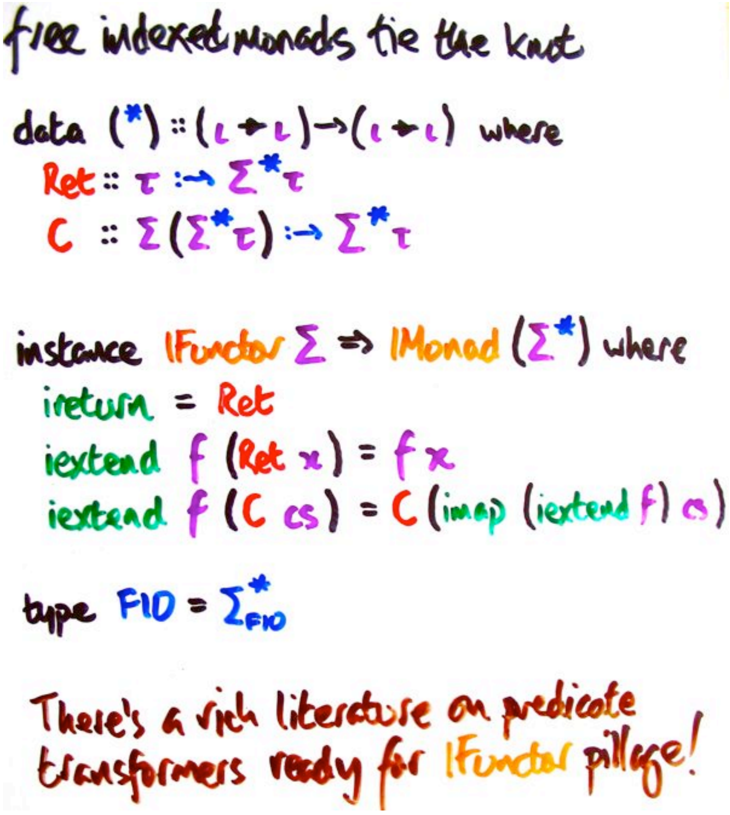
(K String :x: Eq {Closed})	FOpen	State
Eq {Open}	FGetC	(K (Maybe Char) :x: Eq {Open})
Eq {Open}	FClose	Eq {Closed}

Preconditions and postconditions for our file IO operations...



```
data Σ_FIO :: (State → State) where  -- one step only
  FOpen  :: ∀i. (K String :x: Eq {Closed}) {i} →
           (∀j.                    State {j} → τ {j}) → Σ_FIO τ {i}
  FGetC  :: ∀i.          Eq {Open} {i}
           (∀j. (K (Maybe Char) :x: Eq {Open}) {j} → τ {j}) → Σ_FIO τ {i}
  FClose :: ∀i.          Eq {Open} {i}
           (∀j.               Eq {Closed} {j} → τ {j}) → Σ_FIO τ {i}
```

...uniformly determine the predicate transformer characterizing one step of file IO.



free indexed monads tie the knot

```
data (*) :: (ι → ι) → (ι → ι) where
  Ret :: τ :→ Σ*τ
  C   :: Σ(Σ*τ) :→ Σ*τ

instance IFunctor Σ ⇒ IMonad (Σ*) where
  ireturn = Ret
  iextend f (Ret x) = f x
  iextend f (C cs) = C (imap (iextend f) cs)

type FIO = Σ*FIO
```

There's a rich literature on predicate transformers ready for IFunctor pillage!

Any monotonic one-step transformer yields an IMonad exactly by closure under skip and sequential composition.

Conor McBride 2009