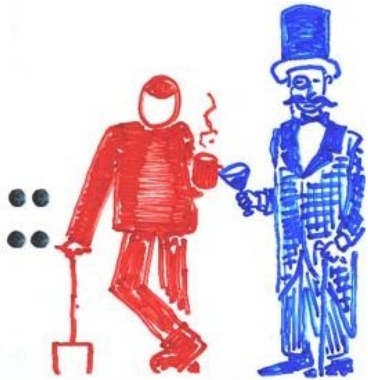
The Strathclyde Haskell Enhancement

Conor McBride

February 18, 2010



A Break with Tradition?

A Break with Tradition?

- I'm going to use a computer.

A Break with Tradition?

- I'm going to use a computer.
- I'm going to use Haskell.

A Break with Tradition?

- I'm going to use a computer.
- I'm going to use Haskell.
- I'm going to run a program.

A Break with Tradition?

- I'm going to use a computer.
- I'm going to use Haskell.
- I'm going to run a program.

- What have I done with the real Conor?

It starts like this...

It starts like this...

```
{-# OPTIONS_GHC -F -pgmF she #-}
```

It starts like this...

```
{-# OPTIONS_GHC -F -pgmF she #-}
```

...and this ain't looking much better...

It starts like this...

```
{-# OPTIONS_GHC -F -pgmF she #-}
```

...and this ain't looking much better...

```
{-# LANGUAGE KindSignatures, RankNTypes #-}
{-# LANGUAGE TypeOperators, GADTs #-}
{-# LANGUAGE TypeFamilies, MultiParamTypeClasses #-}
```

It starts like this...

```
{-# OPTIONS_GHC -F -pgmF she #-}
```

...and this ain't looking much better...

```
{-# LANGUAGE KindSignatures, RankNTypes #-}
{-# LANGUAGE TypeOperators, GADTs #-}
{-# LANGUAGE TypeFamilies, MultiParamTypeClasses #-}
```

...but this...

It starts like this...

```
{-# OPTIONS_GHC -F -pgmF she #-}
```

...and this ain't looking much better...

```
{-# LANGUAGE KindSignatures, RankNTypes #-}
{-# LANGUAGE TypeOperators, GADTs #-}
{-# LANGUAGE TypeFamilies, MultiParamTypeClasses #-}
```

...but this...

```
module FileDemo where


import System.FilePath
import System.IO
import System.IO.Error
```

It starts like this...

```
{-# OPTIONS_GHC -F -pgmF she #-}
```

...and this ain't looking much better...

```
{-# LANGUAGE KindSignatures, RankNTypes #-}
{-# LANGUAGE TypeOperators, GADTs #-}
{-# LANGUAGE TypeFamilies, MultiParamTypeClasses #-}
```

...but this...

```
module FileDemo where


import System.FilePath
import System.IO
import System.IO.Error
```

...suggests that we might even do something.

(Monkey) Business as usual

(Monkey) Business as usual

```
import ShePrelude   -- voodoo
import IFunctor     -- second-order jiggery-pokery
import IMonad       -- third-order jiggery-pokery-transformers
```

File Handles with State

```
data State :: * where
  Open   :: State
  Closed :: State
  deriving SheSingleton    -- what's that?
```

File Handles with State

```
data State :: * where
  Open   :: State
  Closed :: State
  deriving SheSingleton   -- what's that?
```

and a choice of operations.

```
type FH   -- ::({State} → *) → {State} → *
  = FilePath :−{Closed} :>>: (::State)                  -- fOpen
  :+: ()       :−{Open}   :>>: Maybe Char :−{Open}       -- fGetC
  :+: ()       :−{Open}   :>>: () :−{Closed}             -- fClose
```

File Handles with State

```
data State :: * where
  Open   :: State
  Closed :: State
  deriving SheSingleton    -- what's that?
```

and a choice of operations.

```
type FH    -- ::({State} → *) → {State} → *
   = FilePath :−{Closed} :>>: (::State)                    -- fOpen
  :+: ()      :−{Open}   :>>: Maybe Char :−{Open}   -- fGetC
  :+: ()      :−{Open}   :>>: () :−{Closed}            -- fClose
```

hint: *‘precondition’* :>>: *‘postcondition’*

File Handles with State

```
data State :: * where
  Open   :: State
  Closed :: State
  deriving SheSingleton    -- what's that?
```

and a choice of operations.

```
type FH    -- ::({State} → *) → {State} → *
   = FilePath :-{Closed} :>> (::State)                  -- fOpen
  :+: ()      :-{Open}   :>> Maybe Char :-{Open}        -- fGetC
  :+: ()      :-{Open}   :>> () :-{Closed}              -- fClose
```

hint: *'precondition'* :>> *'postcondition'*

hint: *thingIHave* :-{ *stateI'mIn*}   (some data, some logic)

File Handles with State

```
data State :: * where
  Open   :: State
  Closed :: State
  deriving SheSingleton   -- what's that?
```

and a choice of operations.

```
type FH   -- ::({State} → *) → {State} → *
  = FilePath :-{Closed} :>> (::State)                    -- fOpen
  :+: ()     :-{Open}   :>> Maybe Char :-{Open}          -- fGetC
  :+: ()     :-{Open}   :>> () :-{Closed}                -- fClose
```

hint: *'precondition'* :>> *'postcondition'*
hint: *thingIHave* :-{ *stateI'mIn*}   (some data, some logic)
hint: (::State)   means 'is a State known at run time'

I fiddle about in the back of the room,...

```
pattern FOpen p k = Do (InL (V p :& k))
pattern FGetC   k = Do (InR (InL (V () :& k)))
pattern FClose  k = Do (InR (InR (V () :& k)))
```

(**pattern** synonyms are linear constructor-form definitions you can use on either side of your program)

```
fOpen   :: FilePath → (FH :* (::State)) {Closed}
fOpen p = FOpen p Ret
fGetC   :: (FH :* (Maybe Char :−{Open})) {Open}
fGetC   = FGetC Ret
fClose  :: (FH :* (() :−{Closed})) {Open}
fClose  = FClose Ret
```

Ret and Do are the constructors of :*, as we'll see in a bit.

...I write an interpreter,...

```
runFH :: (FH :* (a :-{Closed})) {Closed} → IO a
runFH (Ret (V a)) = return a
runFH (FOpen s k) = catch
  (openFile s ReadMode ≫= openFH (k {Open}))
  (λ_ → runFH (k {Closed}))
   where
    openFH :: (FH :* (a :-{Closed})) {Open} → Handle → IO a
    openFH (FClose k) h = hClose h ≫ runFH (k (V ()))
    openFH (FGetC k) h = catch
      (hGetChar h ≫= λc → openFH (k (V (Just c))) h)
      (λ_ → openFH (k (V Nothing)) h)
```

...and then I write a little program,...

```
fileContents :: FilePath →
                (FH :* (Maybe String :−{Closed})) {Closed}
fileContents p = fOpen p ?>= λs → case s of
  {Closed} → (| Nothing |)
  {Open}   → (| Just readOpenFile (−fClose−) |)

readOpenFile :: (FH :* (String :−{Open})) {Open}
readOpenFile = fGetC =>= λx → case x of
  Nothing → (| "" |)
  Just c  → (| ~c : readOpenFile |)
```

...but is it Haskell?

How about I run this program?

I suppose that means I should suspend Preview and run ghci, in some sort of emacs buffer.

What's Going On?

What's Going On?

- Dependent types?

What's Going On?

- Dependent types?
- Macros?

What's Going On?

- Dependent types?
- Macros?
- A new kind of monad?

What's Going On?

- Dependent types?
- Macros?
- A new kind of monad?

Yeah

What's Going On?

- Dependent types?
- Macros?
- A new kind of monad?

Yeah, but no, but it is...

What's Going On?

- Dependent types?
- Macros?
- A new kind of monad?

Yeah, but no, but it is...
...the *Strathclyde Haskell Enhancement*!

Bizarre Brackets in Peculiar Places

Let's see that again...

$$\begin{array}{l}
\mathsf{fileContents} :: \mathsf{FilePath} \rightarrow \\
\qquad\qquad (\mathsf{FH} \mathrel{:*} (\mathsf{Maybe\ String} \mathrel{:-}\{\mathsf{Closed}\}))\ \{\mathsf{Closed}\} \\
\mathsf{fileContents}\ p = \mathsf{fOpen}\ p \mathrel{?\!\!>\!\!=} \lambda s \rightarrow \mathbf{case}\ s\ \mathbf{of} \\
\quad \{\mathsf{Closed}\} \rightarrow (\!|\ \mathsf{Nothing}\ |\!) \\
\quad \{\mathsf{Open}\} \rightarrow (\!|\ \mathsf{Just\ readOpenFile}\ (-\mathsf{fClose}-)\ |\!)
\end{array}$$

$$\begin{array}{l}
\mathsf{readOpenFile} :: (\mathsf{FH} \mathrel{:*} (\mathsf{String} \mathrel{:-}\{\mathsf{Open}\}))\ \{\mathsf{Open}\} \\
\mathsf{readOpenFile} = \mathsf{fGetC} \mathrel{=\!\!\!>\!\!=} \lambda x \rightarrow \mathbf{case}\ x\ \mathbf{of} \\
\quad \mathsf{Nothing} \rightarrow (\!|\ \text{``''}\ |\!) \\
\quad \mathsf{Just}\ c \rightarrow (\!|\ {\sim}c : \mathsf{readOpenFile}\ |\!)
\end{array}$$

Bizarre Brackets in Peculiar Places

Let's see that again... braces in types.

$$
\begin{array}{l}
\mathsf{fileContents} :: \mathsf{FilePath} \rightarrow \\
\qquad\qquad (\mathsf{FH} : * (\mathsf{Maybe\ String} :\!- \{\mathsf{Closed}\})) \ \{\mathsf{Closed}\} \\
\mathsf{fileContents}\ p = \mathsf{fOpen}\ p\ ?\!\!\gg\!= \lambda s \rightarrow \textbf{case}\ s\ \textbf{of} \\
\quad \{\mathsf{Closed}\} \rightarrow (\!|\ \mathsf{Nothing}\ |\!) \\
\quad \{\mathsf{Open}\} \ \ \rightarrow (\!|\ \mathsf{Just\ readOpenFile}\ (-\mathsf{fClose}-)\ |\!) \\
\\
\mathsf{readOpenFile} :: (\mathsf{FH} : * (\mathsf{String} :\!- \{\mathsf{Open}\})) \ \{\mathsf{Open}\} \\
\mathsf{readOpenFile} = \mathsf{fGetC} \ggg\!= \lambda x \rightarrow \textbf{case}\ x\ \textbf{of} \\
\quad \mathsf{Nothing} \rightarrow (\!|\ \text{``''}\ |\!) \\
\quad \mathsf{Just}\ c \ \ \ \rightarrow (\!|\ {\sim}c : \mathsf{readOpenFile}\ |\!)
\end{array}
$$

Bizarre Brackets in Peculiar Places

Let's see that again... braces around patterns (and expressions)

```
fileContents :: FilePath →
                (FH : ∗ (Maybe String :−{ Closed })) { Closed }
fileContents p = fOpen p ?>= λs → case s of
  {Closed} → (| Nothing |)
  {Open}   → (| Just readOpenFile (−fClose−) |)

readOpenFile :: (FH : ∗ (String :−{ Open })) { Open }
readOpenFile = fGetC >>= λx → case x of
  Nothing → (| "" |)
  Just c  → (| ∼c : readOpenFile |)
```

Bizarre Brackets in Peculiar Places

Let's see that again... banana brackets

$$
\begin{array}{l}
\mathsf{fileContents} :: \mathsf{FilePath} \rightarrow \\
\qquad\qquad (\mathsf{FH} : * (\mathsf{Maybe\ String} :\!- \{\mathsf{Closed}\}))\ \{\mathsf{Closed}\} \\
\mathsf{fileContents}\ p = \mathsf{fOpen}\ p\ ?\!\!\gg\!= \lambda s \rightarrow \mathbf{case}\ s\ \mathbf{of} \\
\quad \{\mathsf{Closed}\} \rightarrow (|\mathsf{Nothing}|) \\
\quad \{\mathsf{Open}\} \quad \rightarrow (|\mathsf{Just\ readOpenFile}\ (-\mathsf{fClose}-)|)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{readOpenFile} :: (\mathsf{FH} : * (\mathsf{String} :\!- \{\mathsf{Open}\}))\ \{\mathsf{Open}\} \\
\mathsf{readOpenFile} = \mathsf{fGetC} \gg\!\!= \lambda x \rightarrow \mathbf{case}\ x\ \mathbf{of} \\
\quad \mathsf{Nothing} \rightarrow (|\text{``''}|) \\
\quad \mathsf{Just}\ c \quad \rightarrow (|{\sim}c : \mathsf{readOpenFile}|)
\end{array}
$$

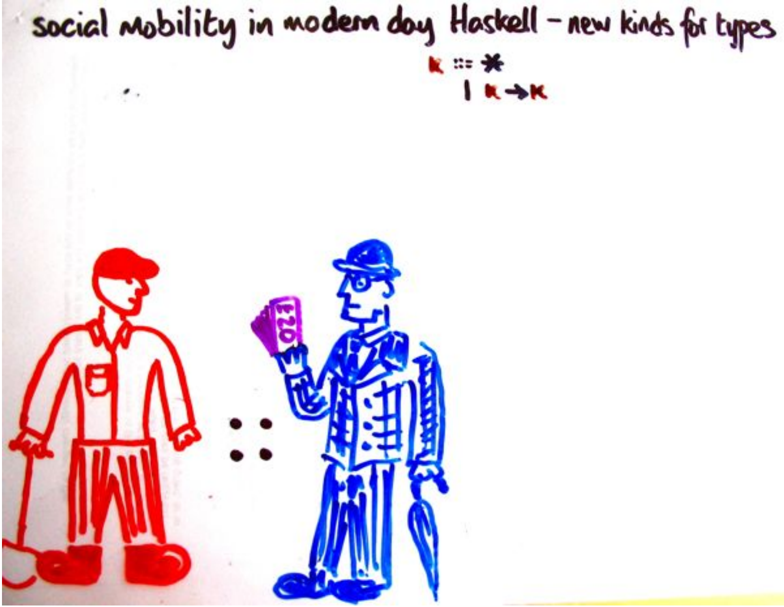
Bizarre Brackets in Peculiar Places

Let's see that again... banana brackets with tack brackets inside

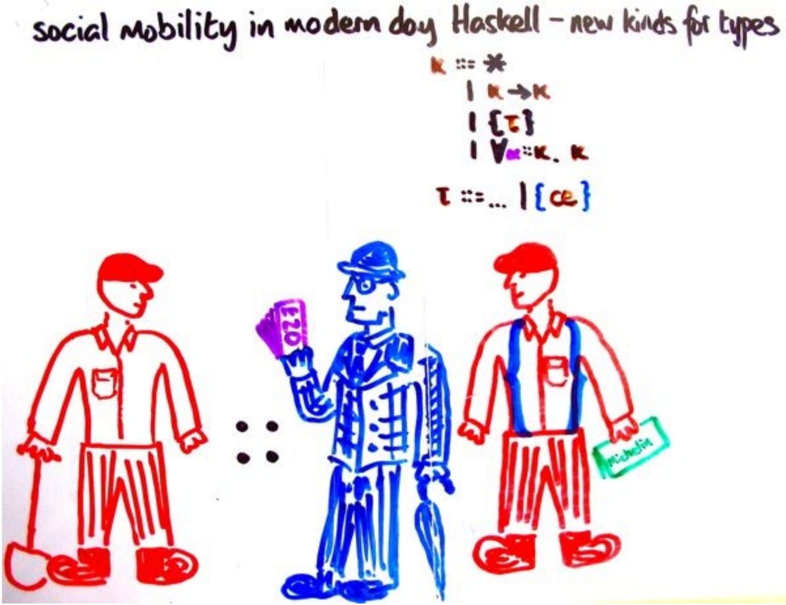
```
fileContents :: FilePath →
                (FH : ∗ (Maybe String :−{ Closed })) { Closed }
fileContents p = fOpen p ?>= λs → case s of
   { Closed } → (| Nothing |)
   { Open }   → (| Just readOpenFile (−fClose−) |)

readOpenFile :: (FH : ∗ (String :−{ Open })) { Open }
readOpenFile = fGetC =>= λx → case x of
   Nothing → (| "" |)
   Just c  → (| ∼c : readOpenFile |)
```

The Braces of Upward Mobility



The Braces of Upward Mobility



Synchronized Swimming from Above and Below

Synchronized Swimming from Above and Below

- types like State become kinds like $\{$State$\}$

Synchronized Swimming from Above and Below

- types like State become kinds like $\{\text{State}\}$
- constructor forms (made from constructors and variables), like Open, move up to the type-level, with $\{\text{Open}\} :: \{\text{State}\}$

Synchronized Swimming from Above and Below

- types like State become kinds like $\{\text{State}\}$
- constructor forms (made from constructors and variables), like Open, move up to the type-level, with $\{\text{Open}\} :: \{\text{State}\}$
- you can now make State-indexed GADTs of kind $\{\text{State}\} \to *$

Synchronized Swimming from Above and Below

- types like State become kinds like $\{\mathsf{State}\}$
- constructor forms (made from constructors and variables), like Open, move up to the type-level, with $\{\mathsf{Open}\} :: \{\mathsf{State}\}$
- you can now make State-indexed GADTs of kind $\{\mathsf{State}\} \to *$
- you can even make GADTs with *polymorphic* kinds

$$\textbf{data}\ (:\!\!-)\ :: \forall\,(x :: *)\,.\ * \to \{x\} \to \{x\} \to *\ \textbf{where}$$
$$\mathsf{V} :: a \to (a :\!\!-\{k\})\,\{k\}$$

Synchronized Swimming from Above and Below

- types like State become kinds like $\{\mathsf{State}\}$
- but SHE turns $\{\tau\}$ into $*$
- constructor forms (made from constructors and variables), like Open, move up to the type-level, with $\{\mathsf{Open}\} :: \{\mathsf{State}\}$
- you can now make State-indexed GADTs of kind $\{\mathsf{State}\} \rightarrow *$
- you can even make GADTs with *polymorphic* kinds

$$\textbf{data}\ (:\!\dashv) :: \forall\, (x :: *)\, .\, * \rightarrow \{x\} \rightarrow \{x\} \rightarrow *\ \textbf{where}$$
$$\mathsf{V} :: a \rightarrow (a :\!\dashv \{k\})\ \{k\}$$

Synchronized Swimming from Above and Below

- types like State become kinds like $\{\text{State}\}$
- but SHE turns $\{\tau\}$ into $*$
- constructor forms (made from constructors and variables), like Open, move up to the type-level, with $\{\text{Open}\} :: \{\text{State}\}$
- but SHE declares SheTyOpen and maps $\{\text{Open}\}$ to it
- you can now make State-indexed GADTs of kind $\{\text{State}\} \to *$

- you can even make GADTs with *polymorphic* kinds

$$\textbf{data}\ (:\!\!-\!\!) :: \forall\ (x :: *)\ .\ * \to \{x\} \to \{x\} \to *\ \textbf{where}$$
$$\quad \mathsf{V} :: a \to (a :\!\!-\!\!\{k\})\ \{k\}$$

Synchronized Swimming from Above and Below

- types like State become kinds like $\{$State$\}$
-     but SHE turns $\{\tau\}$ into $*$
- constructor forms (made from constructors and variables), like Open, move up to the type-level, with $\{$Open$\}$ :: $\{$State$\}$
-     but SHE declares SheTyOpen and maps $\{$Open$\}$ to it
- you can now make State-indexed GADTs of kind $\{$State$\} \rightarrow *$
-     but SHE knows they're really of kind $* \rightarrow *$
- you can even make GADTs with *polymorphic* kinds

$$\textbf{data}\ (:\!\!-)\ ::\ \forall\ (x :: *)\ .\ * \rightarrow \{x\} \rightarrow \{x\} \rightarrow *\ \textbf{where}$$
$$\mathsf{V} :: a \rightarrow (a :\!\!- \{k\})\ \{k\}$$

Synchronized Swimming from Above and Below

- types like State become kinds like $\{\text{State}\}$
- but SHE turns $\{\tau\}$ into $*$
- constructor forms (made from constructors and variables), like Open, move up to the type-level, with $\{\text{Open}\} :: \{\text{State}\}$
- but SHE declares SheTyOpen and maps $\{\text{Open}\}$ to it
- you can now make State-indexed GADTs of kind $\{\text{State}\} \rightarrow *$
- but SHE knows they're really of kind $* \rightarrow *$
- you can even make GADTs with *polymorphic* kinds

$$\textbf{data}\ (:\!\!-)\ ::\ \forall\ (x :: *)\ .\ * \rightarrow \{x\} \rightarrow \{x\} \rightarrow *\ \textbf{where}$$
$$\mathsf{V} :: a \rightarrow (a :\!\!- \{k\})\ \{k\}$$

- but SHE erases $\forall\ (x :: \kappa)\ .$ *n.b.*, x occurs only in $\{..\}$

Synchronized Swimming from Above and Below

- types like State become kinds like $\{\mathsf{State}\}$
- but SHE turns $\{\tau\}$ into $*$
- constructor forms (made from constructors and variables), like Open, move up to the type-level, with $\{\mathsf{Open}\} :: \{\mathsf{State}\}$
- but SHE declares SheTyOpen and maps $\{\mathsf{Open}\}$ to it
- you can now make State-indexed GADTs of kind $\{\mathsf{State}\} \to *$
- but SHE knows they're really of kind $* \to *$
- you can even make GADTs with *polymorphic* kinds

$$\textbf{data}\ (:\!\!-) :: \forall\,(x :: *)\,.\,* \to \{x\} \to \{x\} \to *\ \textbf{where}$$
$$\mathsf{V} :: a \to (a :\!\!- \{k\})\ \{k\}$$

- but SHE erases $\forall\,(x :: \kappa)\,.$ (*n.b.*, x occurs only in $\{..\}$)

Indexed Sets, Data as Witnesses

What does a kind like $\{\text{State}\} \rightarrow *$ contain?

Indexed Sets, Data as Witnesses

What does a kind like $\{\text{State}\} \rightarrow *$ contain?

Sets *indexed by* States,

Indexed Sets, Data as Witnesses

What does a kind like $\{\text{State}\} \to *$ contain?

Sets *indexed by* States,

capturing *properties* of States,

Indexed Sets, Data as Witnesses

What does a kind like $\{\text{State}\} \rightarrow *$ contain?

Sets *indexed by* States,

capturing *properties* of States,

containing data *relevant* to a given State.

Indexed Sets, Data as Witnesses

What does a kind like $\{\mathsf{State}\} \to *$ contain?

Sets *indexed by* States,
capturing *properties* of States,
containing data *relevant* to a given State.

Data carry significant dynamic information *and* witness properties of their static index.

Indexed Sets, Data as Witnesses

What does a kind like $\{\text{State}\} \rightarrow *$ contain?

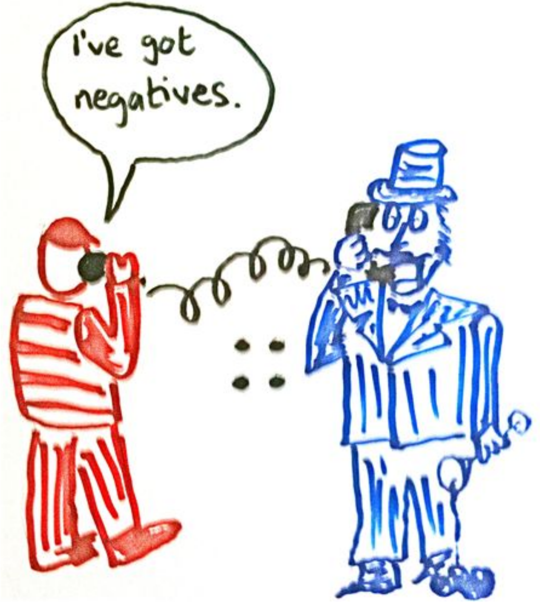
Sets *indexed by* States,
capturing *properties* of States,
containing data *relevant* to a given State.

Data carry significant dynamic information *and* witness properties of their static index.

$$\textbf{data}\ (:\!-)\ ::\ \forall\ (x :: *)\ .\ * \rightarrow \{x\} \rightarrow \{x\} \rightarrow *\ \textbf{where}$$
$$\mathsf{V} :: a \rightarrow (a :\!- \{k\})\ \{k\}$$

$(a :\!- \{k\}) :: \{x\} \rightarrow *$ (pronounced "a atkey k") carries values in a at the *key* index k, and is *empty* at other indices.

Or, to put it another way,



An Old Favourite

$$
\begin{aligned}
&\textbf{data}\ \mathsf{Nat} :: * \ \textbf{where} \\
&\quad \mathsf{Z} :: \mathsf{Nat} \\
&\quad \mathsf{S} :: \mathsf{Nat} \rightarrow \mathsf{Nat} \\
&\textbf{data}\ \mathsf{Vec} :: * \rightarrow \{\mathsf{Nat}\} \rightarrow * \ \textbf{where} \\
&\quad \mathsf{Nil} \quad :: \mathsf{Vec}\ a\ \{\mathsf{Z}\} \\
&\quad \mathsf{Cons} :: a \rightarrow \mathsf{Vec}\ a\ \{n\} \rightarrow \mathsf{Vec}\ a\ \{\mathsf{S}\ n\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{vmap} :: (a \rightarrow b) \rightarrow \mathsf{Vec}\ a\ \{n\} \rightarrow \mathsf{Vec}\ b\ \{n\} \\
&\mathsf{vmap}\ f\ \mathsf{Nil} \qquad\qquad\quad = \mathsf{Nil} \\
&\mathsf{vmap}\ f\ (\mathsf{Cons}\ a\ as) = \mathsf{Cons}\ (f\ a)\ \mathsf{vmap}\ f\ as
\end{aligned}
$$

An Old Favourite

$$\begin{array}{l}
\textbf{data}\ \mathsf{Nat} :: * \ \textbf{where} \\
\quad \mathsf{Z} :: \mathsf{Nat} \\
\quad \mathsf{S} :: \mathsf{Nat} \rightarrow \mathsf{Nat}
\end{array}$$

$$\begin{array}{l}
\textbf{data}\ \mathsf{Vec} :: * \rightarrow \{\mathsf{Nat}\} \rightarrow * \ \textbf{where} \\
\quad \mathsf{Nil} \quad :: \mathsf{Vec}\ a\ \{\mathsf{Z}\} \\
\quad \mathsf{Cons} :: a \rightarrow \mathsf{Vec}\ a\ \{n\} \rightarrow \mathsf{Vec}\ a\ \{\mathsf{S}\ n\}
\end{array}$$

$$\textbf{type}\ s :\rightarrow t = \forall\ i\ .\ s\ \{i\} \rightarrow t\ \{i\}$$

$$\begin{array}{l}
\mathsf{vmap} :: (a \rightarrow b) \rightarrow \mathsf{Vec}\ a\ \{n\} \rightarrow \mathsf{Vec}\ b\ \{n\} \\
\mathsf{vmap}\ f\ \mathsf{Nil} \qquad\qquad\quad = \mathsf{Nil} \\
\mathsf{vmap}\ f\ (\mathsf{Cons}\ a\ as) = \mathsf{Cons}\ (f\ a)\ \mathsf{vmap}\ f\ as
\end{array}$$

An Old Favourite

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Nat} :: * \ \textbf{where} \\
\quad \mathsf{Z} :: \mathsf{Nat} \\
\quad \mathsf{S} :: \mathsf{Nat} \to \mathsf{Nat}
\end{array}
$$

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Vec} :: * \to \{\mathsf{Nat}\} \to * \ \textbf{where} \\
\quad \mathsf{Nil} \quad :: \mathsf{Vec}\ a\ \{\mathsf{Z}\} \\
\quad \mathsf{Cons} :: a \to \mathsf{Vec}\ a\ \{n\} \to \mathsf{Vec}\ a\ \{\mathsf{S}\ n\}
\end{array}
$$

$$
\textbf{type}\ s \mathrel{\dot{\to}} t = \forall\, i\,.\, s\ \{i\} \to t\ \{i\}
$$

$$
\begin{array}{l}
\mathsf{vmap} :: (a \to b) \to \mathsf{Vec}\ a \mathrel{\dot{\to}} \mathsf{Vec}\ b \\
\mathsf{vmap}\ f\ \mathsf{Nil} \qquad\qquad\quad = \mathsf{Nil} \\
\mathsf{vmap}\ f\ (\mathsf{Cons}\ a\ as) = \mathsf{Cons}\ (f\ \ a)\ \mathsf{vmap}\ f\ \ as
\end{array}
$$

A New Favourite *(reflexive-transitive closure)*

$$\textbf{data}\ \mathsf{Path} :: (\{i,i\} \to *) \to \{i,i\} \to *\ \textbf{where}$$
$$\quad \mathsf{Stop} :: \mathsf{Path}\ \sigma\ \{i,i\}$$
$$\quad (:\!-\!:) :: \sigma\ \{i,j\} \to \mathsf{Path}\ \sigma\ \{j,k\} \to \mathsf{Path}\ \sigma\ \{i,k\}$$

You can write $\{i,j\}$ for $\{(i,j)\}$, and $\{\}$ for $\{()\}$.

A New Favourite *(reflexive-transitive closure)*

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Path} :: (\{i,i\} \rightarrow *) \rightarrow \{i,i\} \rightarrow *\ \textbf{where} \\
\quad \mathsf{Stop} :: \mathsf{Path}\ \sigma\ \{i,i\} \\
\quad (:\!-\!:) :: \sigma\ \{i,j\} \rightarrow \mathsf{Path}\ \sigma\ \{j,k\} \rightarrow \mathsf{Path}\ \sigma\ \{i,k\}
\end{array}
$$

You can write $\{i,j\}$ for $\{(i,j)\}$, and $\{\}$ for $\{()\}$.
An index- (*i.e.*, endpoint-) respecting function on steps induces an index-respecting map on paths.

$$
\begin{array}{l}
\mathsf{imap} :: (\sigma :\rightarrow \tau) \rightarrow \mathsf{Path}\ \sigma :\rightarrow \mathsf{Path}\ \tau \\
\mathsf{imap}\ f\ \mathsf{Stop} \qquad\quad = \mathsf{Stop} \\
\mathsf{imap}\ f\ (s :\!-\!: ss) = f\ s :\!-\!: \mathsf{imap}\ f\ ss
\end{array}
$$

Nostrils twitching?

Nostrils twitching? They should be...

Nostrils twitching? They should be...

$$\textbf{class}\ \mathsf{IFunctor}\ (\phi :: (\{\,i\,\} \rightarrow *) \rightarrow \{\,o\,\} \rightarrow *)\ \ \textbf{where}$$
$$\quad \mathsf{imap} :: (\sigma \mathrel{:\!\longrightarrow} \tau) \rightarrow \phi\ \sigma \mathrel{:\!\longrightarrow} \phi\ \tau$$

$$\textbf{instance}\ \mathsf{IFunctor}\ \mathsf{Path}\ \ \textbf{where}$$
$$\quad \mathsf{imap}\ f\ \mathsf{Stop} \qquad\quad = \mathsf{Stop}$$
$$\quad \mathsf{imap}\ f\ (r \mathrel{:\!-\!:} rs) = f\ r \mathrel{:\!-\!:} \mathsf{imap}\ f\ rs$$

Nostrils twitching? They should be...

$$
\begin{array}{l}
\textbf{class}\ \mathsf{IFunctor}\ (\phi :: (\{\,i\,\} \rightarrow *) \rightarrow \{\,o\,\} \rightarrow *)\ \ \textbf{where} \\
\quad \mathsf{imap} :: (\sigma :\rightarrow \tau) \rightarrow \phi\ \sigma :\rightarrow \phi\ \tau
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ \mathsf{IFunctor}\ \mathsf{Path}\ \ \textbf{where} \\
\quad \mathsf{imap}\ f\ \mathsf{Stop} \qquad\quad = \mathsf{Stop} \\
\quad \mathsf{imap}\ f\ (r :\!-\!: rs) = f\ r :\!-\!: \mathsf{imap}\ f\ rs
\end{array}
$$

Make Vec an IFunctor by the power of *one*...

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Vec}' :: (\{\,\} \rightarrow *) \rightarrow \{\,\mathsf{Nat}\,\} \rightarrow *\ \textbf{where} \\
\quad \mathsf{Nil} \quad\ :: \mathsf{Vec}'\ \alpha\ \{\,\mathsf{Z}\,\} \\
\quad \mathsf{Cons}' :: \alpha\ \{\,\} \rightarrow \mathsf{Vec}\ \alpha\ \{\,n\,\} \rightarrow \mathsf{Vec}\ \alpha\ \{\,\mathsf{S}\ n\,\}
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ \mathsf{IFunctor}\ \mathsf{Vec}'\ \ \textbf{where} \\
\quad \mathsf{imap}\ f\ \mathsf{Nil} \qquad\qquad\quad = \mathsf{Nil} \\
\quad \mathsf{imap}\ f\ (\mathsf{Cons}'\ a\ as) = \mathsf{Cons}'\ (f\ a)\ \mathsf{vmap}\ f\ as
\end{array}
$$

Nostrils twitching? They should be...

```
class IFunctor (φ :: ({i} → ∗) → {o} → ∗)  where
  imap :: (σ :→ τ) → φ σ :→ φ τ

instance IFunctor Path  where
  imap f Stop        = Stop
  imap f (r :−: rs) = f r :−: imap f rs
```

Make Vec an IFunctor by the power of *one*...

```
data Vec′ :: ({ } → ∗) → {Nat} → ∗ where
  Nil    :: Vec′ α {Z}
  Cons′ :: α { } → Vec α {n} → Vec α {S n}

instance IFunctor Vec′  where
  imap f Nil              = Nil
  imap f (Cons′ a as) = Cons′ (f a) vmap f as
```

... and atkey back to where you were.

```
type      Vec a {n} = Vec′ (a :−{ }) {n}
pattern Cons a as  = Cons′ (V a) as
```

No Invention Needed

I didn't *invent* IFunctors. I remembered that *each* kind of indexed set $\{i\} \rightarrow *$ has morphisms, $\sigma :\rightarrow \tau$ obeying categorical laws, and I *instantiated* the categorical notion of functor accordingly.

No Invention Needed

I didn't *invent* IFunctors. I remembered that *each* kind of indexed set $\{i\} \to *$ has morphisms, $\sigma :\to \tau$ obeying categorical laws, and I *instantiated* the categorical notion of functor accordingly.

Haskell's Functor is *just* the special case for *functors from* $*$ *to* $*$.

No Invention Needed

I didn't *invent* IFunctors. I remembered that *each* kind of indexed set $\{i\} \to *$ has morphisms, $\sigma \mathrel{:\to} \tau$ obeying categorical laws, and I *instantiated* the categorical notion of functor accordingly.

Haskell's Functor is *just* the special case for *functors from* $*$ *to* $*$.

However, IFunctor is a richer notion, as I may have mentioned before. It doesn't just allow fixpoints; it's *closed* under fixpoints. But that's another story...

No Invention Needed

I didn't *invent* IFunctors. I remembered that *each* kind of indexed set $\{i\} \to *$ has morphisms, $\sigma :\to \tau$ obeying categorical laws, and I *instantiated* the categorical notion of functor accordingly.

Haskell's Functor is *just* the special case for *functors from* $*$ *to* $*$.

However, IFunctor is a richer notion, as I may have mentioned before. It doesn't just allow fixpoints; it's *closed* under fixpoints. But that's another story...

Guess what I'm not going to invent next..?

Indexed Monads

$$\begin{array}{l}\textbf{class}\ \mathsf{IFunctor}\ \phi \Rightarrow \mathsf{IMonad}\ (\phi :: (\{\,i\,\} \to *) \to \{\,i\,\} \to *)\ \ \textbf{where} \\ \quad \mathsf{iskip} \quad\ :: \sigma \mathrel{\dot{\to}} \phi\ \sigma \\ \quad \mathsf{iextend} :: (\sigma \mathrel{\dot{\to}} \phi\ \tau) \to (\phi\ \sigma \mathrel{\dot{\to}} \phi\ \tau)\end{array}$$

It's quite like what you're used to, but with funny names (explanation shortly), and I've flipped 'bind' (back to the way it was when monads were 'tribbles' rather than 'warm fuzzy things').

Indexed Monads

$$\textbf{class}\ \mathsf{IFunctor}\ \phi \Rightarrow \mathsf{IMonad}\ (\phi :: (\{\,i\,\} \to *) \to \{\,i\,\} \to *)\ \ \textbf{where}$$
$$\mathsf{iskip} :: \sigma \mathrel{\dot\to} \phi\ \sigma$$
$$\mathsf{iextend} :: (\sigma \mathrel{\dot\to} \phi\ \tau) \to (\phi\ \sigma \mathrel{\dot\to} \phi\ \tau)$$

It's quite like what you're used to, but with funny names (explanation shortly), and I've flipped 'bind' (back to the way it was when monads were 'tribbles' rather than 'warm fuzzy things').

Interpret $\phi\ \tau\ \{\,i\,\}$ as **'τ is reachable from state $\{\,i\,\}$'**.

Indexed Monads

$$
\begin{array}{l}
\textbf{class}\ \mathsf{IFunctor}\ \phi \Rightarrow \mathsf{IMonad}\ (\phi :: (\{\,i\,\} \to *) \to \{\,i\,\} \to *)\ \ \textbf{where} \\
\quad \mathsf{iskip} \quad\ \ :: \sigma \mathrel{:\!\to} \phi\ \sigma \\
\quad \mathsf{iextend} :: (\sigma \mathrel{:\!\to} \phi\ \tau) \to (\phi\ \sigma \mathrel{:\!\to} \phi\ \tau)
\end{array}
$$

It's quite like what you're used to, but with funny names (explanation shortly), and I've flipped 'bind' (back to the way it was when monads were 'tribbles' rather than 'warm fuzzy things').

Interpret $\phi\ \tau\ \{\,i\,\}$ as **'τ is reachable from state** $\{\,i\,\}$'. So, $\sigma \mathrel{:\!\to} \phi\ \tau$ means **'whenever precondition σ holds, postcondition τ is reachable'**.

Indexed Monads

$$\textbf{class}\ \text{IFunctor}\ \phi \Rightarrow \text{IMonad}\ (\phi :: (\{i\} \to *) \to \{i\} \to *)\ \ \textbf{where}$$
$$\text{iskip} \quad :: \sigma \mathrel{\dot{\to}} \phi\ \sigma$$
$$\text{iextend} :: (\sigma \mathrel{\dot{\to}} \phi\ \tau) \to (\phi\ \sigma \mathrel{\dot{\to}} \phi\ \tau)$$

It's quite like what you're used to, but with funny names (explanation shortly), and I've flipped 'bind' (back to the way it was when monads were 'tribbles' rather than 'warm fuzzy things').

Interpret $\phi\ \tau\ \{i\}$ as **'τ is reachable from state $\{i\}$'**. So, $\sigma \mathrel{\dot{\to}} \phi\ \tau$ means **'whenever precondition σ holds, postcondition τ is reachable'**. Or, as Peter Hancock put it, *'But, Conor, that's just Hoare Logic!'*.

Indexed Monads

$$\textbf{class}\ \mathsf{IFunctor}\ \phi \Rightarrow \mathsf{IMonad}\ (\phi :: (\{\,i\,\} \to *) \to \{\,i\,\} \to *)\ \ \textbf{where}$$
$$\mathsf{iskip}\quad :: \sigma :\to \phi\ \sigma$$
$$\mathsf{iextend} :: (\sigma :\to \phi\ \tau) \to (\phi\ \sigma :\to \phi\ \tau)$$

It's quite like what you're used to, but with funny names (explanation shortly), and I've flipped 'bind' (back to the way it was when monads were 'tribbles' rather than 'warm fuzzy things').

Interpret $\phi\ \tau\ \{\,i\,\}$ as **'τ is reachable from state $\{\,i\,\}$'**. So, $\sigma :\to \phi\ \tau$ means **'whenever precondition σ holds, postcondition τ is reachable'**. Or, as Peter Hancock put it, *'But, Conor, that's just Hoare Logic!'*.

$$\mathsf{iseq} :: \mathsf{IMonad}\ \phi \Rightarrow (\rho :\to \phi\ \sigma) \to (\sigma :\to \phi\ \tau) \to \rho :\to \phi\ \tau$$
$$\mathsf{iseq}\ f\ g = \mathsf{iextend}\ g\ .\ f$$

Key Example: Typed Terms

$$\textbf{data}\ \mathsf{Ty} = \mathsf{BB} \mid \mathsf{NN}$$

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Tm} :: (\{\mathsf{Ty}\} \to *) \to \{\mathsf{Ty}\} \to *\ \textbf{where} \\
\quad \mathsf{Var} :: \alpha\ \{t\} \to \mathsf{Tm}\ \alpha\ \{t\} \\
\quad \mathsf{Le} :: \mathsf{Tm}\ \alpha\ \{\mathsf{NN}\} \to \mathsf{Tm}\ \alpha\ \{\mathsf{NN}\} \to \mathsf{Tm}\ \alpha\ \{\mathsf{BB}\} \\
\quad \mathsf{Add} :: \mathsf{Tm}\ \alpha\ \{\mathsf{NN}\} \to \mathsf{Tm}\ \alpha\ \{\mathsf{NN}\} \to \mathsf{Tm}\ \alpha\ \{\mathsf{NN}\} \\
\quad \mathsf{If} :: \mathsf{Tm}\ \alpha\ \{\mathsf{BB}\} \to \mathsf{Tm}\ \alpha\ \{t\} \to \mathsf{Tm}\ \alpha\ \{t\} \to \mathsf{Tm}\ \alpha\ \{t\}
\end{array}
$$

Key Example: Typed Terms

$$\textbf{data}\ \mathsf{Ty} = \mathsf{BB} \mid \mathsf{NN}$$

$$\begin{array}{l}
\textbf{data}\ \mathsf{Tm} :: (\{\mathsf{Ty}\} \to *) \to \{\mathsf{Ty}\} \to *\ \textbf{where} \\
\quad \mathsf{Var} \ :: \alpha\ \{t\} \to \mathsf{Tm}\ \alpha\ \{t\} \\
\quad \mathsf{Le} \quad :: \mathsf{Tm}\ \alpha\ \{\mathsf{NN}\} \to \mathsf{Tm}\ \alpha\ \{\mathsf{NN}\} \to \mathsf{Tm}\ \alpha\ \{\mathsf{BB}\} \\
\quad \mathsf{Add} :: \mathsf{Tm}\ \alpha\ \{\mathsf{NN}\} \to \mathsf{Tm}\ \alpha\ \{\mathsf{NN}\} \to \mathsf{Tm}\ \alpha\ \{\mathsf{NN}\} \\
\quad \mathsf{If} \quad\ :: \mathsf{Tm}\ \alpha\ \{\mathsf{BB}\} \to \mathsf{Tm}\ \alpha\ \{t\} \to \mathsf{Tm}\ \alpha\ \{t\} \to \mathsf{Tm}\ \alpha\ \{t\}
\end{array}$$

The IMonad behaviour is *type-respecting substitution*.

$$\begin{array}{l}
\textbf{instance}\ \mathsf{IMonad}\ \mathsf{Tm}\ \ \textbf{where} \\
\quad \mathsf{iskip} = \mathsf{Var} \\
\quad \mathsf{iextend}\ f\ (\mathsf{Var}\ x) \quad\ = f\ x \\
\quad \mathsf{iextend}\ f\ (\mathsf{Le}\ s\ t) \quad = \mathsf{Le}\ (\mathsf{iextend}\ f\ s)\ (\mathsf{iextend}\ f\ t) \\
\quad \mathsf{iextend}\ f\ (\mathsf{Add}\ s\ t) = \mathsf{Add}\ (\mathsf{iextend}\ f\ s)\ (\mathsf{iextend}\ f\ t) \\
\quad \mathsf{iextend}\ f\ (\mathsf{If}\ b\ s\ t) \ = \mathsf{If}\ (\mathsf{iextend}\ f\ b)\ (\mathsf{iextend}\ f\ s)\ (\mathsf{iextend}\ f\ t)
\end{array}$$

The IFunctor behaviour is *type-respecting renaming*.

$$\begin{array}{l}
\textbf{instance}\ \mathsf{IFunctor}\ \mathsf{Tm}\ \ \textbf{where} \\
\quad \mathsf{imap}\ f = \mathsf{iextend}\ (\mathsf{Var}\ .\ f)
\end{array}$$

Free Monads (I)

Seen this?

$$\textbf{data}\ f \mathbin{:\!*} t = \mathsf{Ret}\ t \mid \mathsf{Do}\ (f\ (f \mathbin{:\!*} t))$$

You can see this as a kind of 'generalized syntax', where f describes the *constructors* but $(f\,{:}{*})$ chucks in *variables*, too.

Free Monads (I)

Seen this?

$$\mathbf{data}\ f \mathbin{:\!*} t = \mathrm{Ret}\ t \mid \mathrm{Do}\ (f\ (f \mathbin{:\!*} t))$$

You can see this as a kind of 'generalized syntax', where f describes the *constructors* but $(f{:}{*})$ chucks in *variables*, too. The Monad behaviour is exactly substitution.

$$
\begin{array}{l}
\mathbf{instance}\ \mathrm{Functor}\ f \Rightarrow \mathrm{Monad}\ (f{:}{*})\ \ \mathbf{where} \\
\quad \mathrm{return} = \mathrm{Ret} \\
\quad \mathrm{Ret}\ t \ \ggg\!= g = g\ t \\
\quad \mathrm{Do}\ \mathit{fft} \ggg\!= g = \mathrm{Do}\ (\mathit{fmap}\ ({\ggg\!=}g)\ \mathit{fft})
\end{array}
$$

Or you can think of it as the Monad with *commands* given by f, and we throw in return. Elements of (f :* t) are *strategies* for doing f commands in a quest to deliver an t, and $\ggg\!=$ pastes stratgies together.

Free Monads (II)

Let me just rejig that **data** declaration, GADT style.

$$\begin{array}{l} \textbf{data}\ ({:}{*}) :: (* \to *) \quad \to \\ \qquad\qquad\qquad * \to * \quad \textbf{where} \\ \quad \mathsf{Ret} :: t \qquad\qquad\quad \to f\ {:}{*}\ t \\ \quad \mathsf{Do}\ :: f\ (f\ {:}{*}\ t) \to f\ {:}{*}\ t \end{array}$$

Free Monads (III)

Let me just index that.

$$\textbf{data}\ ({:}{*}) :: ((\{i\} \to *) \to \{i\} \to *) \quad \to$$
$$(\{i\} \to *) \to \{i\} \to * \quad \textbf{where}$$
$$\text{Ret} :: t \qquad :\to f\ {:}{*}\ t$$
$$\text{Do} :: f\ (f\ {:}{*}\ t)\ :\to f\ {:}{*}\ t$$

$$\textbf{instance}\ \text{IFunctor}\ f \Rightarrow \text{IMonad}\ (f\,{:}{*})\ \ \textbf{where}$$
$$\text{iskip} = \text{Ret}$$
$$\text{iextend}\ g\ (\text{Ret}\ t)\ \ = g\ t$$
$$\text{iextend}\ g\ (\text{Do}\ \mathit{fft}) = \text{Do}\ (\text{imap}\ (\text{iextend}\ g)\ \mathit{fft})$$

Free Monads (III)

Let me just index that.

$$\textbf{data}\ (^{:*}) :: ((\{i\} \to *) \to \{i\} \to *) \quad \to$$
$$(\{i\} \to *) \to \{i\} \to * \quad \textbf{where}$$
$$\text{Ret} :: t \quad \dot{\to} f\ ^{:*}\ t$$
$$\text{Do} :: f\ (f\ ^{:*}\ t) \dot{\to} f\ ^{:*}\ t$$

- Ret says 't is reachable if it's already witnessed'.

$$\textbf{instance}\ \text{IFunctor}\ f \Rightarrow \text{IMonad}\ (f^{:*})\ \ \textbf{where}$$
$$\text{iskip} = \text{Ret}$$
$$\text{iextend}\ g\ (\text{Ret}\ t) \ = g\ t$$
$$\text{iextend}\ g\ (\text{Do}\ \mathit{fft}) = \text{Do}\ (\text{imap}\ (\text{iextend}\ g)\ \mathit{fft})$$

Free Monads (III)

Let me just index that.

$$
\begin{aligned}
&\textbf{data}\ (^{:*}) :: ((\{i\} \to *) \to \{i\} \to *) \quad \to \\
&\qquad\qquad ((\{i\} \to *) \to \{i\} \to * \quad \textbf{where} \\
&\quad \mathrm{Ret} :: t \qquad\qquad\ \ \dot{\to}\ f\ ^{:*}\ t \\
&\quad \mathrm{Do}\ \ :: f\ (f\ ^{:*}\ t)\ \dot{\to}\ f\ ^{:*}\ t
\end{aligned}
$$

- Ret says 't is reachable if it's already witnessed'.
- Do says 'if doing *one* f-command makes t reachable, then it's reachable already'

$$
\begin{aligned}
&\textbf{instance}\ \mathrm{IFunctor}\ f \Rightarrow \mathrm{IMonad}\ (f^{:*})\ \ \textbf{where} \\
&\quad \mathrm{iskip} = \mathrm{Ret} \\
&\quad \mathrm{iextend}\ g\ (\mathrm{Ret}\ t)\ \ = g\ t \\
&\quad \mathrm{iextend}\ g\ (\mathrm{Do}\ \mathit{fft}) = \mathrm{Do}\ (\mathrm{imap}\ (\mathrm{iextend}\ g)\ \mathit{fft})
\end{aligned}
$$

Free Monads (IV)

Let me expand $:\!\longrightarrow$ to fix the syntax errors.

$$
\begin{aligned}
&\textbf{data}\ (\,{:}\!*\,) :: ((\{\,i\,\} \to *) \to \{\,i\,\} \to *) \quad \to \\
&\qquad\qquad\quad (\{\,i\,\} \to *) \to \{\,i\,\} \to * \quad \textbf{where} \\
&\quad \mathsf{Ret} :: t\ \{\,i\,\} \qquad\qquad\quad \to (f\ {:}\!*\ t)\ \{\,i\,\} \\
&\quad \mathsf{Do}\ \ :: f\ (f\ {:}\!*\ t)\ \{\,i\,\} \to (f\ {:}\!*\ t)\ \{\,i\,\}
\end{aligned}
$$

What would go wrong if we expanded **type** synonyms before checking GADT constructors?

Indexed Monads, Demonic Bind

$$\textbf{class}\ \text{IFunctor}\ \phi \Rightarrow \text{IMonad}\ (\phi :: (\{\,i\,\} \to *) \to \{\,i\,\} \to *)\ \ \textbf{where}$$

$$\text{iskip} :: \sigma \mathrel{\dot{\to}} \phi\ \sigma$$

$$\text{iextend} :: (\sigma \mathrel{\dot{\to}} \phi\ \tau) \to (\phi\ \sigma \mathrel{\dot{\to}} \phi\ \tau)$$

We can also define handy two infix binds.

Indexed Monads, Demonic Bind

$$\textbf{class}\ \mathsf{IFunctor}\ \phi \Rightarrow \mathsf{IMonad}\ (\phi :: (\{\,i\,\} \rightarrow *) \rightarrow \{\,i\,\} \rightarrow *)\ \ \textbf{where}$$
$$\quad \mathsf{iskip} \quad :: \sigma :\rightarrow \phi\ \sigma$$
$$\quad \mathsf{iextend} :: (\sigma :\rightarrow \phi\ \tau) \rightarrow (\phi\ \sigma :\rightarrow \phi\ \tau)$$

We can also define handy two infix binds. *Demonic* bind

$$(?\!\!\succcurlyeq) :: \mathsf{IMonad}\ \phi \Rightarrow$$
$$\qquad \phi\ \sigma\ \{\,i\,\} \rightarrow (\sigma :\rightarrow \phi\ \tau) \rightarrow \phi\ \tau\ \{\,i\,\}$$
$$c\ ?\!\!\succcurlyeq f = \mathsf{iextend}\ f\ c$$

models the general situation: you must be ready for *any* state satisfying σ.

Indexed Monads, Demonic Bind

$$\mathbf{class}\ \mathrm{IFunctor}\ \phi \Rightarrow \mathrm{IMonad}\ (\phi :: (\{\,i\,\} \to *) \to \{\,i\,\} \to *)\ \ \mathbf{where}$$
$$\quad \mathrm{iskip} \quad :: \sigma \mathrel{:\!\to} \phi\ \sigma$$
$$\quad \mathrm{iextend} :: (\sigma \mathrel{:\!\to} \phi\ \tau) \to (\phi\ \sigma \mathrel{:\!\to} \phi\ \tau)$$

We can also define handy two infix binds. *Demonic* bind

$$(?\!\gg\!=) :: \mathrm{IMonad}\ \phi \Rightarrow$$
$$\qquad \forall\, i\,.\,\phi\ \sigma\ \{i\} \to (\forall\, j\,.\,\sigma\ \{j\} \to \phi\ \tau\ \{j\}) \to \phi\ \tau\ \{i\}$$
$$c\ ?\!\gg\!=\ f = \mathrm{iextend}\ f\ c$$

models the general situation: you must be ready for *any* state satisfying σ. We choose i but the demon (*i.e.*, reality) chooses j.

Indexed Monads, Demonic Bind

$$\textbf{class}\ \text{IFunctor}\ \phi \Rightarrow \text{IMonad}\ (\phi :: (\{\,i\,\} \to *) \to \{\,i\,\} \to *)\ \ \textbf{where}$$
$$\quad \text{iskip} \quad :: \sigma \mathrel{:\to} \phi\ \sigma$$
$$\quad \text{iextend} :: (\sigma \mathrel{:\to} \phi\ \tau) \to (\phi\ \sigma \mathrel{:\to} \phi\ \tau)$$

We can also define handy two infix binds. *Demonic* bind

$$(?\!\succcurlyeq) :: \text{IMonad}\ \phi \Rightarrow$$
$$\qquad \forall\ i\ .\ \phi\ \sigma\ \{i\} \to (\forall\ j\ .\ \sigma\ \{j\} \to \phi\ \tau\ \{j\}) \to \phi\ \tau\ \{i\}$$
$$c\ ?\!\succcurlyeq f = \text{iextend}\ f\ c$$

models the general situation: you must be ready for *any* state satisfying σ. We choose i but the demon (*i.e.*, reality) chooses j.

IMonad**s model uncertainty about the state of the world in which computation happens, and what we can learn by interacting with it.**

Demonic Bind, Angelic Bind

$$
\begin{aligned}
&(?\!\gg\!=) :: \mathrm{IMonad}\ \phi \Rightarrow \\
&\qquad \forall\, i\,.\, \phi\ \sigma\ \{i\} \to (\forall\, j\,.\, \sigma\ \{j\} \to \phi\ \tau\ \{j\}) \to \phi\ \tau\ \{i\} \\
&c\ ?\!\gg\!=\ f = \mathrm{iextend}\ f\ c
\end{aligned}
$$

Demonic Bind, Angelic Bind

$$
\begin{aligned}
&(?\!\!\gg\!\!=) :: \text{IMonad}\ \phi \Rightarrow \\
&\qquad\qquad \forall\, i\,.\,\phi\ \sigma\ \{i\} \rightarrow (\forall\, j\,.\,\sigma\ \{j\} \rightarrow \phi\ \tau\ \{j\}) \rightarrow \phi\ \tau\ \{i\} \\
&c\ ?\!\!\gg\!\!=\ f = \text{iextend}\ f\ c
\end{aligned}
$$

Angelic bind constricts the demon with atkey.

$$
\begin{aligned}
&(=\!\!\gg\!\!=) :: \text{IMonad}\ \phi \Rightarrow \phi\ (a :\!- \{j\})\ \{i\} \rightarrow (a \rightarrow \phi\ \tau\ \{j\}) \rightarrow \phi\ \tau\ \{i\} \\
&c =\!\!\gg\!\!= f = c\ ?\!\!\gg\!\!=\ \lambda(\text{V}\ a) \rightarrow f\ a
\end{aligned}
$$

Demonic Bind, Angelic Bind

$$(?\!\!\succcurlyeq) :: \text{IMonad}\ \phi \Rightarrow \forall\, i\,.\, \phi\ \sigma\ \{i\} \to (\forall\, j\,.\, \sigma\ \{j\} \to \phi\ \tau\ \{j\}) \to \phi\ \tau\ \{i\}$$

$$c\ ?\!\!\succcurlyeq f = \text{iextend}\ f\ c$$

Angelic bind constricts the demon with atkey.

$$(\Rrightarrow) :: \text{IMonad}\ \phi \Rightarrow \phi\ (a :\!- \{j\})\ \{i\} \to (a \to \phi\ \tau\ \{j\}) \to \phi\ \tau\ \{i\}$$

$$c \Rrightarrow f = c\ ?\!\!\succcurlyeq \lambda(\text{V}\ a) \to f\ a$$

$$\text{ireturn} :: \text{IMonad}\ \phi \Rightarrow a \to \phi\ (a :\!- \{i\})\ \{i\}$$

$$\text{ireturn}\ a = \text{iskip}\ (\text{V}\ a)$$

Demonic Bind, Angelic Bind

$$(?\!\!\succcurlyeq) :: \mathrm{IMonad}\ \phi \Rightarrow$$
$$\qquad \forall\, i\,.\,\phi\ \sigma\ \{i\} \to (\forall\, j\,.\,\sigma\ \{j\} \to \phi\ \tau\ \{j\}) \to \phi\ \tau\ \{i\}$$
$$c\ ?\!\!\succcurlyeq f = \mathrm{iextend}\ f\ c$$

Angelic bind constricts the demon with atkey.

$$(\ggg\!=) :: \mathrm{IMonad}\ \phi \Rightarrow \phi\ (a :\!\!- \{j\})\ \{i\} \to (a \to \phi\ \tau\ \{j\}) \to \phi\ \tau\ \{i\}$$
$$c \ggg\!= f = c\ ?\!\!\succcurlyeq \lambda(\mathrm{V}\ a) \to f\ a$$

$$\mathrm{ireturn} :: \mathrm{IMonad}\ \phi \Rightarrow a \to \phi\ (a :\!\!- \{i\})\ \{i\}$$
$$\mathrm{ireturn}\ a = \mathrm{iskip}\ (\mathrm{V}\ a)$$

You can rebind return to ireturn and $\gg\!=$ to $\ggg\!=$.

Demonic Bind, Angelic Bind

$$(?\!\!\gg\!=) :: \mathrm{IMonad}\ \phi \Rightarrow$$
$$\forall\, i\,.\ \phi\ \sigma\ \{i\} \rightarrow (\forall\, j\,.\ \sigma\ \{j\} \rightarrow \phi\ \tau\ \{j\}) \rightarrow \phi\ \tau\ \{i\}$$
$$c\ ?\!\!\gg\!=\ f = \mathrm{iextend}\ f\ c$$

Angelic bind constricts the demon with atkey.

$$(\Rightarrow\!=) :: \mathrm{IMonad}\ \phi \Rightarrow \phi\ (a :\!-\{j\})\ \{i\} \rightarrow (a \rightarrow \phi\ \tau\ \{j\}) \rightarrow \phi\ \tau\ \{i\}$$
$$c \Rightarrow\!= f = c\ ?\!\!\gg\!=\ \lambda(\mathrm{V}\ a) \rightarrow f\ a$$

$$\mathrm{ireturn} :: \mathrm{IMonad}\ \phi \Rightarrow a \rightarrow \phi\ (a :\!-\{i\})\ \{i\}$$
$$\mathrm{ireturn}\ a = \mathrm{iskip}\ (\mathrm{V}\ a)$$

You can rebind return to ireturn and $\gg\!=$ to $\Rightarrow\!=$. Put $\tau = b :\!-\{k\}$

$$(\Rightarrow\!=) :: \mathrm{IMonad}\ \phi \Rightarrow \phi\ (a :\!-\{j\})\ \{i\}$$
$$\rightarrow \quad (a \rightarrow \phi\ (b :\!-\{k\})\ \{j\}) \rightarrow \phi\ (b :\!-\{k\})\ \{i\}$$

Demonic Bind, Angelic Bind

$$
\begin{aligned}
&(?\!\!\succcurlyeq) :: \mathsf{IMonad}\ \phi \Rightarrow \\
&\qquad \forall\, i\,.\ \phi\ \sigma\ \{i\} \to (\forall\, j\,.\ \sigma\ \{j\} \to \phi\ \tau\ \{j\}) \to \phi\ \tau\ \{i\} \\
&c\ ?\!\!\succcurlyeq\ f = \mathsf{iextend}\ f\ c
\end{aligned}
$$

Angelic bind constricts the demon with atkey.

$$
\begin{aligned}
&(\Rrightarrow) :: \mathsf{IMonad}\ \phi \Rightarrow \phi\ (a :\!-\{j\})\ \{i\} \to (a \to \phi\ \tau\ \{j\}) \to \phi\ \tau\ \{i\} \\
&c \Rrightarrow f = c\ ?\!\!\succcurlyeq\ \lambda(\mathsf{V}\ a) \to f\ a
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{ireturn} :: \mathsf{IMonad}\ \phi \Rightarrow a \to \phi\ (a :\!-\{i\})\ \{i\} \\
&\mathsf{ireturn}\ a = \mathsf{iskip}\ (\mathsf{V}\ a)
\end{aligned}
$$

You can rebind return to ireturn and $\gg\!=$ to $\Rrightarrow$. Put $\tau = b :\!-\{k\}$

$$
\begin{aligned}
(\Rrightarrow) :: \mathsf{IMonad}\ \phi \Rightarrow\ & \phi\ (a :\!-\{j\})\ \{i\} \\
\to\ & (a \to \phi\ (b :\!-\{k\})\ \{j\}) \to \phi\ (b :\!-\{k\})\ \{i\}
\end{aligned}
$$

cf Wadler, Uustalu, Kiselyov, Brady,...

Demonic Bind, Angelic Bind

$$
\begin{aligned}
&(?\!\!\succcurlyeq) :: \mathsf{IMonad}\ \phi \Rightarrow \\
&\qquad \forall\, i\,.\ \phi\ \sigma\ \{i\} \to (\forall\, j\,.\ \sigma\ \{j\} \to \phi\ \tau\ \{j\}) \to \phi\ \tau\ \{i\} \\
&c \mathrel{?\!\!\succcurlyeq} f = \mathsf{iextend}\ f\ c
\end{aligned}
$$

Angelic bind constricts the demon with atkey.

$$
\begin{aligned}
&(\Rrightarrow) :: \mathsf{IMonad}\ \phi \Rightarrow \phi\ (a \mathrel{:\!\!-} \{j\})\ \{i\} \to (a \to \phi\ \tau\ \{j\}) \to \phi\ \tau\ \{i\} \\
&c \Rrightarrow f = c \mathrel{?\!\!\succcurlyeq} \lambda(\mathsf{V}\ a) \to f\ a
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{ireturn} :: \mathsf{IMonad}\ \phi \Rightarrow a \to \phi\ (a \mathrel{:\!\!-} \{i\})\ \{i\} \\
&\mathsf{ireturn}\ a = \mathsf{iskip}\ (\mathsf{V}\ a)
\end{aligned}
$$

You can rebind return to ireturn and $\gg\!=$ to $\Rrightarrow$. Put $\tau = b \mathrel{:\!\!-} \{k\}$

$$
\begin{aligned}
(\Rrightarrow) :: \mathsf{IMonad}\ \phi \Rightarrow\ & \phi\ (a \mathrel{:\!\!-} \{j\})\ \{i\} \\
\to\ & (a \to \phi\ (b \mathrel{:\!\!-} \{k\})\ \{j\}) \to \phi\ (b \mathrel{:\!\!-} \{k\})\ \{i\}
\end{aligned}
$$

cf Wadler, Uustalu, Kiselyov, Brady,... and Bob of that ilk.

Angelic Applicatives

While I'm about it, let me define

$$
\begin{array}{l}
\textbf{class}\ \mathsf{IFunctor}\ \phi \Rightarrow \mathsf{IApplicative}\ (\phi :: (\{\,i\,\} \to *) \to \{\,i\,\} \to *)\ \ \textbf{where} \\
\quad \mathsf{pure} :: x \to \phi\ (x :\!\!-\{\,i\,\})\ \{\,i\,\} \\
\quad (\circledast)\ \ :: \phi\ ((s \to t) :\!\!-\{\,j\,\})\ \{\,i\,\} \to \\
\qquad\qquad \phi\ (s :\!\!-\{\,k\,\})\ \{\,j\,\} \to \phi\ (t :\!\!-\{\,k\,\})\ \{\,i\,\}
\end{array}
$$

Angelic Applicatives

While I'm about it, let me define

$$
\begin{aligned}
&\textbf{class}\ \mathsf{IFunctor}\ \phi \Rightarrow \mathsf{IApplicative}\ (\phi :: (\{\,i\,\} \rightarrow *) \rightarrow \{\,i\,\} \rightarrow *)\ \ \textbf{where}\\
&\quad \mathsf{pure} :: x \rightarrow \phi\ (x :\!\!-\{\,i\,\})\ \{\,i\,\}\\
&\quad (\circledast)\ :: \phi\ ((s \rightarrow t) :\!\!-\{\,j\,\})\ \{\,i\,\} \rightarrow\\
&\qquad\qquad \phi\ (s :\!\!-\{\,k\,\})\ \{\,j\,\} \rightarrow \phi\ (t :\!\!-\{\,k\,\})\ \{\,i\,\}
\end{aligned}
$$

This says ϕ allows us to build applications by (angelic) computation. pure computations preserve the state; $\circledast$ computes the function whilst evolving from $\{\,i\,\}$ to $\{\,j\,\}$ and its argument whilst evolving from $\{\,j\,\}$ to $\{\,k\,\}$.

Angelic Applicatives

While I'm about it, let me define

$$
\begin{array}{l}
\textbf{class}\ \mathsf{IFunctor}\ \phi \Rightarrow \mathsf{IApplicative}\ (\phi :: (\{\, i \,\} \rightarrow *) \rightarrow \{\, i \,\} \rightarrow *)\ \ \textbf{where} \\
\quad \mathsf{pure} :: x \rightarrow \phi\ (x :\!\!-\{\, i \,\})\ \{\, i \,\} \\
\quad (\circledast)\ :: \phi\ ((s \rightarrow t) :\!\!-\{\, j \,\})\ \{\, i \,\} \rightarrow \\
\qquad\qquad \phi\ (s :\!\!-\{\, k \,\})\ \{\, j \,\} \rightarrow \phi\ (t :\!\!-\{\, k \,\})\ \{\, i \,\}
\end{array}
$$

This says ϕ allows us to build applications by (angelic) computation. pure computations preserve the state; $\circledast$ computes the function whilst evolving from $\{\, i \,\}$ to $\{\, j \,\}$ and its argument whilst evolving from $\{\, j \,\}$ to $\{\, k \,\}$.

We're still lifting 'ordinary programming' to an effectful world,

Angelic Applicatives

While I'm about it, let me define

$$
\begin{array}{l}
\textbf{class}\ \mathsf{IFunctor}\ \phi \Rightarrow \mathsf{IApplicative}\ (\phi :: (\{\mathit{i}\} \to *) \to \{\mathit{i}\} \to *)\ \ \textbf{where} \\
\quad \mathsf{pure} :: x \to \phi\ (x :\!\!-\{\mathit{i}\})\ \{\mathit{i}\} \\
\quad (\circledast)\ :: \phi\ ((s \to t) :\!\!-\{\mathit{j}\})\ \{\mathit{i}\} \to \\
\qquad\qquad \phi\ (s :\!\!-\{\mathit{k}\})\ \{\mathit{j}\} \to \phi\ (t :\!\!-\{\mathit{k}\})\ \{\mathit{i}\}
\end{array}
$$

This says ϕ allows us to build applications by (angelic) computation. pure computations preserve the state; $\circledast$ computes the function whilst evolving from $\{i\}$ to $\{j\}$ and its argument whilst evolving from $\{j\}$ to $\{k\}$.

We're still lifting 'ordinary programming' to an effectful world, but now we're playing dominos, too.

Angelic Applicatives

While I'm about it, let me define

$$
\begin{array}{l}
\textbf{class}\ \mathsf{IFunctor}\ \phi \Rightarrow \mathsf{IApplicative}\ (\phi :: (\{i\} \to *) \to \{i\} \to *)\ \ \textbf{where} \\
\quad \mathsf{pure} :: x \to \phi\ (x \mathrel{:\!\!-} \{i\})\ \{i\} \\
\quad (\circledast)\ :: \phi\ ((s \to t) \mathrel{:\!\!-} \{j\})\ \{i\} \to \\
\qquad\qquad \phi\ (s \mathrel{:\!\!-} \{k\})\ \{j\} \to \phi\ (t \mathrel{:\!\!-} \{k\})\ \{i\}
\end{array}
$$

This says ϕ allows us to build applications by (angelic) computation. pure computations preserve the state; $\circledast$ computes the function whilst evolving from $\{i\}$ to $\{j\}$ and its argument whilst evolving from $\{j\}$ to $\{k\}$.

We're still lifting 'ordinary programming' to an effectful world, but now we're playing dominos, too.

Every IMonad is IApplicative, just as when we work over $*$.

Digressing further, let's peel those bananas...

$$\begin{array}{l}\mathsf{fileContents} :: \mathsf{FilePath} \rightarrow \\ \qquad\qquad (\mathsf{FH} \;{:}^{*}\; (\mathsf{Maybe\ String} :\!-\!\{\mathsf{Closed}\})) \;\{\mathsf{Closed}\} \\ \mathsf{fileContents}\; p = \mathsf{fOpen}\; p \;?\!\!\succcurlyeq \lambda s \rightarrow \textbf{case}\; s\; \textbf{of} \\ \quad \{\mathsf{Closed}\} \rightarrow (\!|\; \mathsf{Nothing} \;|\!) \\ \quad \{\mathsf{Open}\} \;\;\rightarrow (\!|\; \mathsf{Just\ readOpenFile}\; (-\mathsf{fClose}-) \;|\!)\end{array}$$

SHE turns applications

$$(\!|\; f\; a_1 \ldots a_n \;|\!)$$

in *idiom* brackets into

$$\mathsf{pure}\; f \circledast a_1 \circledast \ldots \circledast a_n$$

like in the paper by Ross and me, but round.

Digressing further, let's peel those bananas...

$$\begin{aligned}
&\mathsf{fileContents} :: \mathsf{FilePath} \rightarrow \\
&\qquad (\mathsf{FH} \mathrel{:\!*} (\mathsf{Maybe\ String} :\!\!-\{\mathsf{Closed}\})) \; \{\mathsf{Closed}\} \\
&\mathsf{fileContents}\ p = \mathsf{fOpen}\ p \mathrel{?\!\!\succcurlyeq} \lambda s \rightarrow \mathbf{case}\ s\ \mathbf{of} \\
&\quad \{\mathsf{Closed}\} \rightarrow (\!|\ \mathsf{Nothing}\ |\!) \\
&\quad \{\mathsf{Open}\} \;\; \rightarrow (\!|\ \mathsf{Just\ readOpenFile}\ (-\mathsf{fClose}-)\ |\!)
\end{aligned}$$

SHE turns applications

$$(\!|\ f\ a_1\ \ldots\ a_n\ |\!)$$

in *idiom* brackets into

$$\mathsf{pure}\ f \circledast a_1 \circledast \ldots \circledast a_n$$

like in the paper by Ross and me, but round. Meanwhile, *noise brackets*, $(-\ \ldots\ -)$, tack in effects but ignore their values, using $\lessdot\!\!*$:

$$thing \lessdot\!\!* noise = (\!|\ \mathsf{const}\ thing\ noise\ |\!)$$

Digressing further, let's peel those bananas...

```
fileContents :: FilePath →
                (FH :* (Maybe String :−{Closed})) {Closed}
fileContents p = fOpen p ?>= λs → case s of
   {Closed} → (| Nothing |)
   {Open}   → (| Just readOpenFile (−fClose−) |)
```

SHE turns applications

$$(\!|\ f\ a_1 \ldots a_n\ |\!)$$

in *idiom* brackets into

$$\mathsf{pure}\ f \circledast a_1 \circledast \ldots \circledast a_n$$

like in the paper by Ross and me, but round. Meanwhile, *noise brackets*, $(- \ldots -)$, tack in effects but ignore their values, using `<*`:

```
thing <* noise = (| const thing noise |)
```

Above, we get Just the String from the file, *and* we fClose the file.

Idiom Brackets de luxe

$$
\begin{array}{l}
\mathsf{readOpenFile} :: (\mathsf{FH} \mathbin{:\!*} (\mathsf{String} :\!- \{\mathsf{Open}\}))\ \{\mathsf{Open}\} \\
\mathsf{readOpenFile} = \mathsf{fGetC} \ggg \lambda x \rightarrow \mathbf{case}\ x\ \mathbf{of} \\
\quad \mathsf{Nothing} \rightarrow (\!|\ \text{""}\ |\!) \\
\quad \mathsf{Just}\ c \quad \rightarrow (\!|\ \sim c : \mathsf{readOpenFile}\ |\!)
\end{array}
$$

Idiom Brackets de luxe

$$\begin{array}{l}
\mathsf{readOpenFile} :: (\mathsf{FH} \mathrel{:\!*} (\mathsf{String} \mathrel{:\!-} \{\mathsf{Open}\}))\ \{\mathsf{Open}\} \\
\mathsf{readOpenFile} = \mathsf{fGetC} \ggg \lambda x \rightarrow \textbf{case}\ x\ \textbf{of} \\
\quad \mathsf{Nothing} \rightarrow (\!|\ \text{``''}\ |\!) \\
\quad \mathsf{Just}\ c \quad\ \rightarrow (\!|\ {\sim}c : \mathsf{readOpenFile}\ |\!)
\end{array}$$

SHE notices when the function in an application is infix.

Idiom Brackets de luxe

$$\begin{array}{l}
\mathsf{readOpenFile} :: (\mathsf{FH} \;{:}{*}\; (\mathsf{String} :\!-\{\mathsf{Open}\}))\;\{\mathsf{Open}\} \\
\mathsf{readOpenFile} = \mathsf{fGetC} \ggg \lambda x \rightarrow \textbf{case}\; x\; \textbf{of} \\
\quad \mathsf{Nothing} \rightarrow (\!|\; \text{""} \;|\!) \\
\quad \mathsf{Just}\; c \quad \rightarrow (\!|\; {\sim}c : \mathsf{readOpenFile} \;|\!)
\end{array}$$

SHE notices when the function in an application is infix.

SHE lets you mark pure *arguments* with $\sim$, so $\sim c$ means pure c.

Idiom Brackets de luxe

$$
\begin{array}{l}
\mathsf{readOpenFile} :: (\mathsf{FH} \mathbin{:\!*} (\mathsf{String} :\!\!-\{\mathsf{Open}\}))\ \{\mathsf{Open}\} \\
\mathsf{readOpenFile} = \mathsf{fGetC} \ggg\!= \lambda x \rightarrow \textbf{case}\ x\ \textbf{of} \\
\quad \mathsf{Nothing} \rightarrow (\!|\ \text{""}\ |\!) \\
\quad \mathsf{Just}\ c \quad\ \rightarrow (\!|\ {\sim}c : \mathsf{readOpenFile}\ |\!)
\end{array}
$$

SHE notices when the function in an application is infix.

SHE lets you mark pure *arguments* with $\sim$, so $\sim c$ means pure c.

I'm using idiom brackets with IApplicative here, but they also work for Applicative.

Idiom Brackets de luxe

$$\begin{array}{l}
\mathsf{readOpenFile} :: (\mathsf{FH} :\!\!* (\mathsf{String} :\!\!- \{\mathsf{Open}\}))\ \{\mathsf{Open}\} \\
\mathsf{readOpenFile} = \mathsf{fGetC} \ggg\!= \lambda x \rightarrow \textbf{case}\ x\ \textbf{of} \\
\quad \mathsf{Nothing} \rightarrow (\!|\ \text{``''}\ |\!) \\
\quad \mathsf{Just}\ c \quad \rightarrow (\!|\ {\sim}c : \mathsf{readOpenFile}\ |\!)
\end{array}$$

SHE notices when the function in an application is infix.

SHE lets you mark pure *arguments* with $\sim$, so $\sim c$ means pure c.

I'm using idiom brackets with IApplicative here, but they also work for Applicative.

Syntax remains negotiable: I'm open to suggestions.

Idiom Brackets de luxe

$$\begin{array}{l}
\mathsf{readOpenFile} :: (\mathsf{FH} \mathbin{:\!*} (\mathsf{String} :\!-\!\{\mathsf{Open}\}))\ \{\mathsf{Open}\} \\
\mathsf{readOpenFile} = \mathsf{fGetC} \ggg\!= \lambda x \rightarrow \textbf{case}\ x\ \textbf{of} \\
\quad \mathsf{Nothing} \rightarrow (\!|\ \text{``''}\ |\!) \\
\quad \mathsf{Just}\ c \quad \rightarrow (\!|\ {\sim}c : \mathsf{readOpenFile}\ |\!)
\end{array}$$

SHE notices when the function in an application is infix.

SHE lets you mark pure *arguments* with $\sim$, so $\sim c$ means pure c.

I'm using idiom brackets with IApplicative here, but they also work for Applicative.

Syntax remains negotiable: I'm open to suggestions.

Ha ha: $(-3-)$.

Where were we before we bananaed off?

We'd seen how to get a free monad from a functor describing commands. Here's a functor which describes commands via *Hoare Logic*.

$$\textbf{data}\ (\sigma \mathrel{:\!\gg\!\!:} \tau)\ \upsilon\ \{i\} = \sigma\ \{i\} \qquad \text{-- precondition holds now}$$
$$\mathrel{:\&}\ (\tau \mathrel{:\!\rightarrow} \upsilon) \qquad \text{-- postcondition delivers goal}$$

We can reach υ by doing a $(\sigma \mathrel{:\!\gg\!\!:} \tau)$ command if σ holds now, and we can get υ from τ.

Where were we before we bananaed off?

We'd seen how to get a free monad from a functor describing commands. Here's a functor which describes commands via *Hoare Logic*.

$$\mathbf{data}\ (\sigma \mathrel{:\!\gg\!\!:} \tau)\ \upsilon\ \{i\} = \sigma\ \{i\} \qquad \text{-- precondition holds now}$$
$$\mathrel{:\&}\ (\tau \mathrel{:\!\to} \upsilon) \qquad \text{-- postcondition delivers goal}$$

We can reach υ by doing a $(\sigma \mathrel{:\!\gg\!\!:} \tau)$ command if σ holds now, and we can get υ from τ.

$$\begin{aligned}&\mathbf{data}\ (\mathsf{IFunctor}\ \phi, \mathsf{IFunctor}\ \psi) \Rightarrow (\phi \mathrel{:\!+\!:} \psi)\ \tau\ \{i\}\\ &\quad = \mathsf{InL}\ (\phi\ \tau\ \{i\})\\ &\quad \mid\ \mathsf{InR}\ (\psi\ \tau\ \{i\})\end{aligned}$$

IFunctor is closed under *choice*, so you can offer a choice of commands.

That File System

$$\begin{array}{llll}
\textbf{type}\ \mathsf{FH} & \text{-- } ::(\{\mathsf{State}\} \to *) \to \{\mathsf{State}\} \to * & & \\
\quad = \ \mathsf{FilePath} & :\!-\{\mathsf{Closed}\} & :\gg (::\mathsf{State}) & \text{-- fOpen} \\
\quad :\!+\!: () & :\!-\{\mathsf{Open}\} & :\gg \mathsf{Maybe\ Char} :\!-\{\mathsf{Open}\} & \text{-- fGetC} \\
\quad :\!+\!: () & :\!-\{\mathsf{Open}\} & :\gg () :\!-\{\mathsf{Closed}\} & \text{-- fClose}
\end{array}$$

It's a choice of commands, specified in Hoare Logic. We get the corresponding IMonad, $(\mathsf{FH}^{:*})$ at no extra charge.

That File System

$$\begin{aligned}
&\textbf{type}\ \mathsf{FH} \quad \text{-- } ::(\{\mathsf{State}\} \to *) \to \{\mathsf{State}\} \to * \\
&\quad = \mathsf{FilePath} :\!\!-\!\{\mathsf{Closed}\} :\gg: (::\mathsf{State}) && \text{-- fOpen} \\
&\quad :+: () \quad :\!\!-\!\{\mathsf{Open}\} :\gg: \mathsf{Maybe\ Char} :\!\!-\!\{\mathsf{Open}\} && \text{-- fGetC} \\
&\quad :+: () \quad :\!\!-\!\{\mathsf{Open}\} :\gg: () :\!\!-\!\{\mathsf{Closed}\} && \text{-- fClose}
\end{aligned}$$

It's a choice of commands, specified in Hoare Logic. We get the corresponding IMonad, $(\mathsf{FH}^{:*})$ at no extra charge.

But what's that $(::\mathsf{State})$?

That File System

```
type FH   -- ::({State} → ∗) → {State} → ∗
   = FilePath :−{Closed} :≫ (::State)                    -- fOpen
   :+: ()     :−{Open}   :≫ Maybe Char :−{Open}          -- fGetC
   :+: ()     :−{Open}   :≫ () :−{Closed}                -- fClose
```

It's a choice of commands, specified in Hoare Logic. We get the corresponding IMonad, $(\mathsf{FH}^{:*})$ at no extra charge.

But what's that (::State)?

We can't predict the state after fOpen. We rather need to *check* it at run time.

Dependent Types to the Rescue

When you write...

```
data State :: * where
  Open   :: State
  Closed :: State
  deriving SheSingleton    -- ...this...
```

Dependent Types to the Rescue

When you write...

```
data State :: * where
  Open   :: State
  Closed :: State
  deriving SheSingleton   -- ...this...
```

... SHE constructs this (with uglier underwater names):

```
(::State) :: {State} → *
{Open}   :: (::State) {Open}
{Closed} :: (::State) {Closed}
```

Dependent Types to the Rescue

When you write...

```
data State :: * where
  Open   :: State
  Closed :: State
  deriving SheSingleton   -- ...this...
```

... SHE constructs this (with uglier underwater names):

```
(::State) :: {State} → *
{Open}   :: (::State) {Open}
{Closed} :: (::State) {Closed}
```

The point: if you do **case** analysis on a (::State) {*i*}, you find out what *i* is.

Dependent Types to the Rescue

When you write...

```
data State :: * where
  Open   :: State
  Closed :: State
  deriving SheSingleton   -- ...this...
```

... SHE constructs this (with uglier underwater names):

```
(::State) :: {State} → *
{Open}   :: (::State) {Open}
{Closed} :: (::State) {Closed}
```

The point: if you do **case** analysis on a (::State) { *i* }, you find out what *i* is.

SHE takes **pi** $(x :: s)\ .\ t$ to mean $\forall\ x\ .\ (::s)\ \{x\} \rightarrow t$

Putting it all together

```
fileContents :: FilePath →
                (FH :* (Maybe String :- {Closed})) {Closed}
fileContents p = fOpen p ?>= λs → case s of
  {Closed} → (| Nothing |)
  {Open}   → (| Just readOpenFile (-fClose-) |)
```

We *must* check if the file is open before reading it. We *must* close the file at the end.

```
readOpenFile :: (FH :* (String :- {Open})) {Open}
readOpenFile = fGetC >>= λx → case x of
  Nothing → (| "" |)
  Just c  → (| ~c : readOpenFile |)
```

Putting it all together

```
fileContents :: FilePath →
                  (FH :* (Maybe String :− {Closed})) {Closed}
fileContents p = fOpen p ?>= λs → case s of
   {Closed} → (| Nothing |)
   {Open}   → (| Just readOpenFile (−fClose−) |)
```

We *must* check if the file is open before reading it. We *must* close the file at the end.

```
readOpenFile :: (FH :* (String :− {Open})) {Open}
readOpenFile = fGetC =>= λx → case x of
   Nothing → (| "" |)
   Just c  → (| ~c : readOpenFile |)
```

We've captured a policy for safe interaction with a dangerous world.

Congratulations, Haskell!

Congratulations, Haskell!

You're the world's first mainstream dependently typed programming language!

The Scottish Society for the Prevention of Cruelty to Simons

confirms that no Simons were harmed in the making of this motion picture.