Introduction to monads: the programming semicolon that lies at the heart of category theory

Mark Wildon

Heilbronn Institute for Mathematical Research, Bristol University

June 16 2025

§1 Currying and the product-hom adjunction

The Haskell category **Hask** has as objects all types, such as `Integer`, `String`, `Integer -> Bool`, and morphisms all . . .

§1 Currying and the product-hom adjunction

The Haskell category **Hask** has as objects all types, such as `Integer`, `String`, `Integer -> Bool`, and morphisms all Haskell values of the appropriate type.

§1 Currying and the product-hom adjunction

The Haskell category **Hask** has as objects all types, such as `Integer`, `String`, `Integer -> Bool`, and morphisms all Haskell values of the appropriate type.

- `6 :: Integer`
- `"Is this too easy?" :: String`
- `adams x = if x == 42 then True else False`

Thus `adams :: Integer -> Bool` and `adams 42` $\rightsquigarrow$ `True`.

Question. What about multiple arguments? Unidiomatic:

§1 Currying and the product-hom adjunction

The Haskell category **Hask** has as objects all types, such as `Integer`, `String`, `Integer -> Bool`, and morphisms all Haskell values of the appropriate type.

- `6 :: Integer`
- `"Is this too easy?" :: String`
- `adams x = if x == 42 then True else False`

Thus `adams :: Integer -> Bool` and `adams 42` $\rightsquigarrow$ `True`.

Question. What about multiple arguments? Unidiomatic:

- `fPair (b, x) = if b then floor x else floor (x+1)`

of type `(Bool, Double) -> Integer`. Idiomatic:

§1 Currying and the product-hom adjunction

The Haskell category **Hask** has as objects all types, such as `Integer`, `String`, `Integer -> Bool`, and morphisms all Haskell values of the appropriate type.

- `6 :: Integer`
- `"Is this too easy?" :: String`
- `adams x = if x == 42 then True else False`

Thus `adams :: Integer -> Bool` and `adams 42` $\rightsquigarrow$ `True`.

Question. What about multiple arguments? Unidiomatic:

- `fPair (b, x) = if b then floor x else floor (x+1)`

of type `(Bool, Double) -> Integer`. Idiomatic:

- `f b x = if b then floor x else floor (x+1)`

of type

§1 Currying and the product-hom adjunction

The Haskell category **Hask** has as objects all types, such as `Integer`, `String`, `Integer -> Bool`, and morphisms all Haskell values of the appropriate type.

- `6 :: Integer`
- `"Is this too easy?" :: String`
- `adams x = if x == 42 then True else False`

Thus `adams :: Integer -> Bool` and `adams 42` ⇝ `True`.

Question. What about multiple arguments? Unidiomatic:

- `fPair (b, x) = if b then floor x else floor (x+1)`

of type `(Bool, Double) -> Integer`. Idiomatic:

- `f b x = if b then floor x else floor (x+1)`

of type `Bool -> Double -> Integer`. Thus

- `f True :: Double -> Integer`

is a single variable function and `f` is a function taking as input a boolean and returning a function of type `Double -> Integer`.

§1 Currying and the product-hom adjunction

The Haskell category **Hask** has as objects all types, such as `Integer`, `String`, `Integer -> Bool`, and morphisms all Haskell values of the appropriate type.

- `6 :: Integer`
- `"Is this too easy?" :: String`
- `adams x = if x == 42 then True else False`

Thus `adams :: Integer -> Bool` and `adams 42` $\rightsquigarrow$ `True`.

Question. What about multiple arguments? Unidiomatic:

- `fPair (b, x) = if b then floor x else floor (x+1)`

of type `(Bool, Double) -> Integer`. Idiomatic:

- `f b x = if b then floor x else floor (x+1)`

of type `Bool -> Double -> Integer`. Thus

- `f True :: Double -> Integer`

is a single variable function and `f` is a function taking as input a boolean and returning a function of type `Double -> Integer`.

Conclusion. You can't program in Haskell without meeting the product-hom adjunction in almost every line.

Some examples of the product-hom adjunction

In **Hask**:

$$\mathrm{Hom}_{\mathbf{Hask}}\big((a, b), c\big) \cong \mathrm{Hom}_{\mathbf{Hask}}\big(a, b \to c\big).$$

Exercise (with a one word answer from the standard prelude)

Define the forwards isomorphism in Haskell by writing

- `idiomatize :: ((a, b) -> c) -> (a -> (b -> c))`

Some examples of the product-hom adjunction

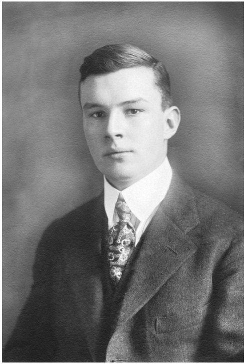
In **Hask**:

$$\mathrm{Hom}_{\mathbf{Hask}}\big((a,b),c\big) \cong \mathrm{Hom}_{\mathbf{Hask}}\big(a, b \to c\big).$$

Exercise (with a one word answer from the standard prelude)

Define the forwards isomorphism in Haskell by writing

- `idiomatize :: ((a, b) -> c) -> (a -> (b -> c))`



Haskell Brooks Curry 1900–1982

Some examples of the product-hom adjunction

In **Hask**:

$$\mathrm{Hom}_{\mathbf{Hask}}\big((a, b), c\big) \cong \mathrm{Hom}_{\mathbf{Hask}}\big(a, b \to c\big).$$

Exercise (with a one word answer from the standard prelude)

Define the forwards isomorphism in Haskell by writing

- `idiomatize :: ((a, b) -> c) -> (a -> (b -> c))`

In **Set**:

$$\mathrm{Hom}_{\mathbf{Set}}\big(X \times Y, Z\big) \cong \mathrm{Hom}_{\mathbf{Set}}\big(X, \mathrm{Hom}_{\mathbf{Set}}(Y, Z)\big)$$

Some examples of the product-hom adjunction

In **Hask**:

$$\mathrm{Hom}_{\mathbf{Hask}}\big((a, b), c\big) \cong \mathrm{Hom}_{\mathbf{Hask}}\big(a, b \to c\big).$$

Exercise (with a one word answer from the standard prelude)

Define the forwards isomorphism in Haskell by writing

- `idiomatize :: ((a, b) -> c) -> (a -> (b -> c))`

In **Set**:

$$\mathrm{Hom}_{\mathbf{Set}}\big(- \times Y, -\big) \cong \mathrm{Hom}_{\mathbf{Set}}\big(-, \mathrm{Hom}_{\mathbf{Set}}(Y, -)\big)$$

Some examples of the product-hom adjunction

In **Hask**:

$$\mathrm{Hom}_{\mathbf{Hask}}\big((a,b),c\big) \cong \mathrm{Hom}_{\mathbf{Hask}}\big(a, b \to c\big).$$

Exercise (with a one word answer from the standard prelude)

Define the forwards isomorphism in Haskell by writing

- `idiomatize :: ((a, b) -> c) -> (a -> (b -> c))`

In **Set**: there are isomorphisms natural in X and Z

$$\mathrm{Hom}_{\mathbf{Set}}\big(X \times Y, Z\big) \cong \mathrm{Hom}_{\mathbf{Set}}\big(X, \mathrm{Hom}_{\mathbf{Set}}(Y, Z)\big)$$

Exercise (from Linderholm, *Mathematics Made Difficult*)

Prove that $z^{xy} = (z^y)^x$ *for* $x, y, z \in \mathbb{N}_0$.

Some examples of the product-hom adjunction

In **Hask**:

$$\mathrm{Hom}_{\mathbf{Hask}}\big((a,b),c\big) \cong \mathrm{Hom}_{\mathbf{Hask}}\big(a, b \to c\big).$$

Exercise (with a one word answer from the standard prelude)

Define the forwards isomorphism in Haskell by writing

- `idiomatize :: ((a, b) -> c) -> (a -> (b -> c))`

In **Set**: there are isomorphisms natural in X and Z

$$\mathrm{Hom}_{\mathbf{Set}}\big(X \times Y, Z\big) \cong \mathrm{Hom}_{\mathbf{Set}}\big(X, \mathrm{Hom}_{\mathbf{Set}}(Y,Z)\big)$$

Exercise (from Linderholm, *Mathematics Made Difficult*)

Prove that $z^{xy} = (z^y)^x$ *for* $x, y, z \in \mathbb{N}_0$.

In **mod**-$\mathbb{C}G$ and **mod**-$\mathbb{C}G$ for H a subgroup of G:

$$\mathrm{Hom}_{\mathbb{C}G}(X \otimes_{\mathbb{C}H} \mathbb{C}G, Z) \cong \mathrm{Hom}_{\mathbb{C}H}\big(X, \mathrm{Hom}_{\mathbb{C}G}(\mathbb{C}G, Z)\big)$$

or using standard notation for induction and restriction,

$$\mathrm{Hom}_{\mathbb{C}G}(X\!\uparrow_H^G, Z) \cong \mathrm{Hom}_{\mathbb{C}H}(X, Z\!\downarrow_H^G).$$

This is Frobenius reciprocity. A deep result true for trivial reasons.

§2: A free-forgetful monad

Let X be a set. The *free monoid* on X is the set of all words in X with concatenation as the product.

- Let $F(X)$ be the free monoid on the set X.
- Let $U(N)$ be the underlying set of the monoid N.

Thus we have functors $F : \mathbf{Set} \to \mathbf{Mon}$, $U : \mathbf{Mon} \to \mathbf{Set}$. For example: $F\{\blacklozenge\} = \{\varnothing, \blacklozenge, \blacklozenge\blacklozenge, \blacklozenge\blacklozenge\blacklozenge, \ldots\}$ and $U(\mathbb{N}_0, +) = \{0, 1, 2, \ldots\}$.

Claim

Let $X \in \mathbf{Set}$ and let $N \in \mathbf{Mon}$. Then

$$\mathrm{Hom}_{\mathrm{Mon}}\big(F(X), N)\big) \cong \mathrm{Hom}_{\mathrm{Set}}\big(X, U(N)\big).$$

§2: A free-forgetful monad

Let X be a set. The *free monoid* on X is the set of all words in X with concatenation as the product.

- Let $F(X)$ be the free monoid on the set X.
- Let $U(N)$ be the underlying set of the monoid N.

Thus we have functors $F : \mathbf{Set} \to \mathbf{Mon}$, $U : \mathbf{Mon} \to \mathbf{Set}$. For example: $F\{\blacklozenge\} = \{\varnothing, \blacklozenge, \blacklozenge\blacklozenge, \blacklozenge\blacklozenge\blacklozenge, \ldots\}$ and $U(\mathbb{N}_0, +) = \{0, 1, 2, \ldots\}$.

Claim

Let $X \in \mathbf{Set}$ and let $N \in \mathbf{Mon}$. Then

$$\mathrm{Hom}_{\mathrm{Mon}}\big(F(X), N)\big) \cong \mathrm{Hom}_{\mathrm{Set}}\big(X, U(N)\big).$$

Let $M = UF : \mathbf{Set} \to \mathbf{Set}$. For instance

$$M\big\{w, o, r, d, s\big\} = \big\{word, sword, door, roodwords, \ldots\big\}$$

Question. Let X be a set. What is $M(M(X))$?

§2: A free-forgetful monad

Let X be a set. The *free monoid* on X is the set of all words in X with concatenation as the product.

- Let $F(X)$ be the free monoid on the set X.
- Let $U(N)$ be the underlying set of the monoid N.

Thus we have functors $F : \mathbf{Set} \to \mathbf{Mon}$, $U : \mathbf{Mon} \to \mathbf{Set}$. For example: $F\{\blacklozenge\} = \{\varnothing, \blacklozenge, \blacklozenge\blacklozenge, \blacklozenge\blacklozenge\blacklozenge, \ldots\}$ and $U(\mathbb{N}_0, +) = \{0, 1, 2, \ldots\}$.

Claim

Let $X \in \mathbf{Set}$ and let $N \in \mathbf{Mon}$. Then

$$\mathrm{Hom}_{\mathsf{Mon}}\big(F(X), N)\big) \cong \mathrm{Hom}_{\mathsf{Set}}\big(X, U(N)\big).$$

Let $M = UF : \mathbf{Set} \to \mathbf{Set}$. For instance

$$M\big\{w, o, r, d, s\big\} = \big\{word, sword, door, roodwords, \ldots\big\}$$

Question. Let X be a set. What is $M(M(X))$?

Exercise

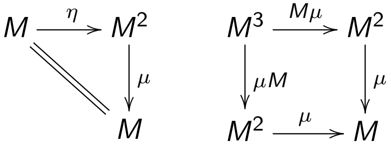
Formally words are tuples and $F(X) = \bigsqcup_{n \geq 0} X^n$, where the monoid product is concatenation of tuples. Does $M^2 = M$ hold? Is there a natural isomorphism $\mu : M^2 \cong M$?

§3: The mathematical definition of monads

Let $\mathcal{D}$ be a category. A monad is a functor $M : \mathcal{D} \to \mathcal{D}$ together with natural transformations

- $\mu : M^2 \to M$ (join)
- $\eta : \mathrm{id}_{\mathcal{D}} \to M$ (unit)

such that the diagrams below commute.



Example ($M(X) = \bigsqcup_{n \geq 0} X^n$, the free monoid monad)

We saw that $\mu : M^2 \to M$ *is defined by 'remove inner parentheses':*

$$\mu_{\{w,o,r,d,s\}}\big((d,o,o,r),(w,o,r,d)\big) = (d,o,o,r,w,o,r,d,s)$$

The unit $\eta : \mathrm{id}_{\mathsf{Set}} \to \mathsf{Set}$ *is defined on each set* X *so that* $\eta_X : X \to \bigsqcup_{n \geq 0} X^n$ *is the canonical inclusion. For instance* $\eta_{\{w,o,r,d,s\}} x = (x)$ *for each* $x \in \{w,o,r,d,s\}$ *and*

$$\eta_{M\{w,o,r,d,s\}}(d,o,o,r) = ((d,o,o,r)).$$

The infamous one-line definition

Saunders MacLane, *Categories for the working mathematician*

All told, a monad in X is just a monoid in the category of endofunctors of X, with product $\times$ replaced by composition of endofunctors and unit set by the identity endofunctor

James Ivry, *A brief incomplete, and mostly wrong history of programming languages*

Wadler tries to appease critics [of Haskell 1997] by explaining that: "A monad is a monoid in the category of endofunctors: what's the problem?"

§4: Monads in the Haskell category **Hask**

Type `:info Functor` and `:info Monad` at the Haskell prompt:

```
class Functor f where
   fmap :: (a -> b) -> f a -> f b
class Functor m => Monad m where
   unit :: a -> m a
   join :: m m a -> m a
   (>>=) :: m a -> (a -> m b) -> m b
   mx >>= f = join (fmap f mx)
```

And here is a complete definition of the list monad

```
instance Functor [] where fmap f xs = map f xs
instance Monad [] where unit x = [x]; join = concat
```

§4: Monads in the Haskell category **Hask**

Type `:info Functor` and `:info Monad` at the Haskell prompt:

```
class Functor f where
   fmap :: (a -> b) -> f a -> f b
class Functor m => Monad m where
   unit :: a -> m a
   join :: m m a -> m a
   (>>=) :: m a -> (a -> m b) -> m b
   mx >>= f = join (fmap f mx)
```

And here is a complete definition of the list monad

```
instance Functor [] where fmap f xs = map f xs
instance Monad [] where unit x = [x]; join = concat
```

I have to confess that in fact Haskell expects you to define >>= and unit derives all the rest, including `fmap`. (Except that recent versions make you define an Applicative instance.) And, for reasons that made sense when monads were often used for IO, `unit` is called `return`. But it *could* work exactly as this slide claims.

§4: Monads in the Haskell category **Hask**

Type `:info Functor` and `:info Monad` at the Haskell prompt:

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
class Functor m => Monad m where
    unit :: a -> m a
    join :: m m a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
    mx >>= f = join (fmap f mx)
```

And here is a complete definition of the list monad

```
instance Functor [] where fmap f xs = map f xs
instance Monad [] where unit x = [x]; join = concat
```

I have to confess that in fact Haskell expects you to define >>= and unit derives all the rest, including `fmap`. (Except that recent versions make you define an Applicative instance.) And, for reasons that made sense when monads were often used for IO, `unit` is called `return`. But it *could* work exactly as this slide claims.

And on the theme of excusable oversimplifications, I should also admit that, without an explicit type declaration, the function `f b x = if b then floor x else floor (x+1)` gets the polymorphic type `f :: (RealFrac a, Integral p) => Bool -> a -> p`. But I didn't want to drag in typeclasses on the first slide.

Haskell has exactly two special pieces of syntax: list comprehension and ‘do’ notation:

- `[(x,y) | x <- [1,2,3], y <- [4,5]]`
- `do x <- [1,2,3]; y <- [4,5]; unit (x,y)`
- `[1,2,3] >>= \x -> [4,5] >>= \y -> unit (x,y)`

Haskell has exactly two special pieces of syntax: list comprehension and 'do' notation:

- `[(x,y) | x <- [1,2,3], y <- [4,5]]`
- `do x <- [1,2,3]; y <- [4,5]; unit (x,y)`
- `[1,2,3] >>= \x -> [4,5] >>= \y -> unit (x,y)`
- ```
  join (fmap f [1,2,3]) where
       f x = [4,5] >>= \y -> unit (x,y)
  ```
  ```

Haskell has exactly two special pieces of syntax: list comprehension and 'do' notation:

- `[(x,y) | x <- [1,2,3], y <- [4,5]]`
- `do x <- [1,2,3]; y <- [4,5]; unit (x,y)`
- `[1,2,3] >>= \x -> [4,5] >>= \y -> unit (x,y)`
- ```
  join (fmap f [1,2,3]) where
        f x = [4,5] >>= \y -> unit (x,y)
  ```
- ```
  join (fmap f [1,2,3]) where
        f x = join (fmap g [4,5]) where g y = [(x,y)]
  ```

Haskell has exactly two special pieces of syntax: list comprehension and ‘do’ notation:

- `[(x,y) | x <- [1,2,3], y <- [4,5]]`
- `do x <- [1,2,3]; y <- [4,5]; unit (x,y)`
- `[1,2,3] >>= \x -> [4,5] >>= \y -> unit (x,y)`
-
```
join (fmap f [1,2,3]) where
     f x = [4,5] >>= \y -> unit (x,y)
```
-
```
join (fmap f [1,2,3]) where
     f x = join (fmap g [4,5]) where g y = [(x,y)]
```
-
```
join (fmap f [1,2,3]) where
     f x = join [[(x,4)], [(x,5)]]
```

Haskell has exactly two special pieces of syntax: list comprehension and 'do' notation:

- `[(x,y) | x <- [1,2,3], y <- [4,5]]`
- `do x <- [1,2,3]; y <- [4,5]; unit (x,y)`
- `[1,2,3] >>= \x -> [4,5] >>= \y -> unit (x,y)`
- ```
  join (fmap f [1,2,3]) where
       f x = [4,5] >>= \y -> unit (x,y)
  ```
- ```
  join (fmap f [1,2,3]) where
       f x = join (fmap g [4,5]) where g y = [(x,y)]
  ```
- ```
  join (fmap f [1,2,3]) where
       f x = join [[(x,4)], [(x,5)]]
  ```
- ```
  join (fmap f [1,2,3]) where
       f x = [(x,4), (x,5)]
  ```

Haskell has exactly two special pieces of syntax: list comprehension and ‘do’ notation:

- `[(x,y) | x <- [1,2,3], y <- [4,5]]`
- `do x <- [1,2,3]; y <- [4,5]; unit (x,y)`
- `[1,2,3] >>= \x -> [4,5] >>= \y -> unit (x,y)`
- ```
  join (fmap f [1,2,3]) where
        f x = [4,5] >>= \y -> unit (x,y)
  ```
- ```
  join (fmap f [1,2,3]) where
        f x = join (fmap g [4,5]) where g y = [(x,y)]
  ```
- ```
  join (fmap f [1,2,3]) where
        f x = join [[(x,4)], [(x,5)]]
  ```
- ```
  join (fmap f [1,2,3]) where
        f x = [(x,4), (x,5)]
  ```
- `join [(1,4),(1,5)],[(2,4),(2,5)],[(3,4),(3,5)]]`

all evaluate to `[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]`

The programmable semicolon

Using default lists,

- `do x <- [1,2,3]; y <- [4,5..]; unit (x,y)`
  ⇝ `[(1,4),(1,5),(1,6), ...`

Because of lazy evaluation there's no problem with the infinite stream, except that it means we never get beyond the head of `[1,2,3]`.

```
newtype DiagonalList a = DL {unDL :: [a]}
    deriving (Functor, Show)
instance Monad DiagonalList where
    unit x = DL [x]
    join = concat . stripe
```

where `stripe :: [[a]] -> [[a]]` returns the diagonal stripes of a list of a list. We now run the same computation in the diagonal list monad:

- `do x <- DL [1,2,3]; y <- DL [4,5..]; unit (x,y)`
  ⇝ `DL [(1,4),(1,5),(2,4),(1,6),(2,5),(3,4)..`

The programmable semicolon

Using default lists,

- `do x <- [1,2,3]; y <- [4,5..]; unit (x,y)`
  ⇝ `[(1,4),(1,5),(1,6), ...`

Because of lazy evaluation there's no problem with the infinite stream, except that it means we never get beyond the head of `[1,2,3]`.

```
newtype DiagonalList a = DL {unDL :: [a]}
    deriving (Functor, Show)
instance Monad DiagonalList where
    unit x = DL [x]
    join = concat . stripe
```

where `stripe :: [[a]] -> [[a]]` returns the diagonal stripes of a list of a list. We now run the same computation in the diagonal list monad:

- `do x <- DL [1,2,3]; y <- DL [4,5..]; unit (x,y)`
  ⇝ `DL [(1,4),(1,5),(2,4),(1,6),(2,5),(3,4)..`

As defined `join` has type `[[a]]->[[a]]` not `DL DL a -> DL a` and in truth `join = DL . concat . stripe . map unDL . unDL`

§5: In praise of types

Question. Write $\neg a$ for $a \implies \bot$, i.e. 'a implies false'. Which of the following are tautologies?

- $a \implies a$
- $a \implies (b \implies a)$
- $(a \implies b) \implies a$
- $\left(a \implies (b \implies c)\right) \implies (a \implies b) \implies (a \implies c)$
- $a \implies \neg\neg a$
- $\neg\neg a \implies a$
- $((a \implies b) \implies a) \implies a$

§5: In praise of types

Question. Write $\neg a$ for $a \implies \bot$, i.e. 'a implies false'. Which of the following are tautologies?

- $a \implies a$
- $a \implies (b \implies a)$
- $(a \implies b) \implies a$
- $\left(a \implies (b \implies c)\right) \implies (a \implies b) \implies (a \implies c)$
- $a \implies \neg\neg a$
- $\neg\neg a \implies a$
- $((a \implies b) \implies a) \implies a$

Answer, all except $(a \implies b) \implies a$.

§5: In praise of types

Question. Write $\neg a$ for $a \implies \bot$, i.e. 'a implies false'. Which of the following are tautologies?

- $a \implies a$
- $a \implies (b \implies a)$
- $(a \implies b) \implies a$
- $\big(a \implies (b \implies c)\big) \implies (a \implies b) \implies (a \implies c)$
- $a \implies \neg\neg a$
- $\neg\neg a \implies a$
- $((a \implies b) \implies a) \implies a$

Answer, all except $(a \implies b) \implies a$. Moreover all the tautologies except the last two can be proved in intuitionistic logic. Why?

§5: In praise of types

Question. Write $\neg a$ for $a \implies \bot$, i.e. 'a implies false'. Which of the following are tautologies?

- $a \implies a$
- $a \implies (b \implies a)$
- $(a \implies b) \implies a$
- $\big(a \implies (b \implies c)\big) \implies (a \implies b) \implies (a \implies c)$
- $a \implies \neg\neg a$
- $\neg\neg a \implies a$
- $((a \implies b) \implies a) \implies a$

Answer, all except $(a \implies b) \implies a$. Moreover all the tautologies except the last two can be proved in intuitionistic logic. Why? Because, replacing $\implies$ with $\rightarrow$ they are the types of Haskell programs.

This is the Curry–Howard correspondence between mathematical proofs and computer programs.

The type checker is your friend (in Haskell)

Recall the function `f : Bool -> Double -> Integer` defined by

```
f b x = if b then floor x else floor (x+1)
```

- In Haskell, if you forget which order the arguments come in, the interpreter/compiler will give you a helpful error message

```
> f 3.5 False
<interactive>:88:7: error:
  Couldn't match expected type ‘Double’ with actual
  type ‘Bool’.
     • In the second argument of ‘f’, namely ‘False’
```

- In Magma, the interpreter will wait until you've done a long calculation, and then pounce:

```
function f(b, x); if b then return Floor(x); else ..
                     ^
     Runtime error in if: Logical expected
```

- In C, the compiler need neither notice nor care and, under the rules of undefined behaviour, your program might erase your user directory. The Haskell type checker promises this can't happen, since `deleteFile :: IO ()` is in the IO monad.

§6: Every adjunction defines a monad

Let $L : \mathcal{D} \to \mathcal{C}$ and $R : \mathcal{C} \to \mathcal{D}$ be adjoint functors, so

$$\mathrm{Hom}_{\mathcal{C}}(Lx, z) \cong \mathrm{Hom}_{\mathcal{D}}(x, Rz)$$

naturally in $x \in \mathcal{D}$ and $z \in \mathcal{C}$. For instance

- $L = - \times Y : \mathbf{Set} \to \mathbf{Set}$; $R = \mathrm{Hom}_{\mathrm{Set}}(Y, -) : \mathbf{Set} \to \mathbf{Set}$,
- $L = F : \mathbf{Set} \to \mathbf{Mon}$; $R = U : \mathbf{Mon} \to \mathbf{Set}$.

Theorem

The composition $RL : \mathcal{D} \to \mathcal{D}$ is a monad in a canonical way.

Why you might believe this.

Pretend that L is 'free' and R is 'forget'. Forget R. Then 'free on free' is no more complicated than 'free', and there is a canonical unit map from X into the 'free thing' on X. □

§6: Every adjunction defines a monad

Let $L : \mathcal{D} \to \mathcal{C}$ and $R : \mathcal{C} \to \mathcal{D}$ be adjoint functors, so

$$\mathrm{Hom}_{\mathcal{C}}(Lx, z) \cong \mathrm{Hom}_{\mathcal{D}}(x, Rz)$$

naturally in $x \in \mathcal{D}$ and $z \in \mathcal{C}$. For instance

- $L = - \times Y : \mathbf{Set} \to \mathbf{Set}$; $R = \mathrm{Hom}_{\mathrm{Set}}(Y, -) : \mathbf{Set} \to \mathbf{Set}$,
- $L = F : \mathbf{Set} \to \mathbf{Mon}$; $R = U : \mathbf{Mon} \to \mathbf{Set}$.

Theorem

The composition $RL : \mathcal{D} \to \mathcal{D}$ is a monad in a canonical way.

The canonical way to define unit and join is by chasing through the adjunction to get the only maps that can possibly be defined:

- $\eta : \mathrm{id}_{\mathcal{D}} \to M$ is defined so that η_x is the image of id_{Lx} under the isomorphism $\mathrm{Hom}_{\mathcal{C}}(Lx, Lx) \cong \mathrm{Hom}_{\mathcal{D}}\big(x, RLx\big)$
- μ is the natural transformation

$$M^2 = (RL)(RL) = R(LR)L \xrightarrow{R\epsilon L} R\mathrm{id}_{\mathcal{C}}L = RL = M$$

where $\epsilon_z : LRz \to z$ is the image of id_{Rz} under the isomorphism $\mathrm{Hom}_{\mathcal{C}}(Rz, Rz) \cong \mathrm{Hom}_{\mathcal{D}}\big(LRz, z\big)$.

§7: Every monad comes from an adjunction

Let $M : \mathcal{D} \to \mathcal{D}$ be a monad. An M-algebra is an object $z \in \mathcal{D}$ together with a map $Mz \overset{\vartheta}{\to} z$ such that the diagrams below commute.

$$
\begin{array}{ccc}
z & \xrightarrow{\eta} & Mz \\
& \searrow{=} & \downarrow{\scriptstyle \vartheta} \\
& & z
\end{array}
\qquad
\begin{array}{ccc}
M^2 z & \xrightarrow{\mu} & Mz \\
\downarrow{\scriptstyle M\vartheta} & & \downarrow{\scriptstyle \vartheta} \\
Mz & \xrightarrow{\vartheta} & z
\end{array}
$$

The Eilenberg–Moore category $\mathcal{D}^M$ is the category of M algebras. The object Mx is a T-algebra with maps $\mu : M^2x \to x$. It satisfies

$$
\mathrm{Hom}_{\mathcal{D}^M}\left(\begin{array}{c} M^2x \\ \downarrow{\scriptstyle \mu} \\ Mx \end{array}, \begin{array}{c} Mz \\ \downarrow{\scriptstyle \vartheta} \\ z \end{array} \right) \cong \mathrm{Hom}_{\mathcal{D}}(x, z)
$$

Hence

- $F : \mathcal{D} \to \mathcal{D}^M$ defined by $Fx = M^2x \overset{\mu}{M}\; x$
- $U : \mathcal{D}^M \to \mathcal{D}$ defined by $U(Mz \xrightarrow{\vartheta} z) = z$

are adjoint functors. Since $U(F(x)) = Mx$, the monad M comes from an adjunction.

Algebras for monads can be remarkably deep. Algebras for the

- free monoid monad are monoids;
- power set monad are associative, symmetric, idempotent binary operations;
- ultfrafilter monad are compact Hausdorff spaces;
- distribution monad on a set of size n are divisions of the n-vertex simplex into n convex sets, one containing each vertex;
- state monad on **Hask** are mnemoids, a functional version of global state.

We saw that 'induce then restrict' is an instance of the state monad from the product-hom adjunction, interpreted in module categories.

Proposition

Let H be a subgroup of G and let $\Omega = G/H$ be the G-set of H-cosets. An $\mathbb{F}_2$-algebra for the 'induce to G then restrict' monad on $\mathbb{F}_2H$-mod is a union Δ of H-orbits on Ω such that

$$\omega, \omega g, \omega g' \in \Delta \implies \omega g g' \in \Delta$$

for all $\omega \in \Omega$ and $g, g' \in G$.

Thank you!