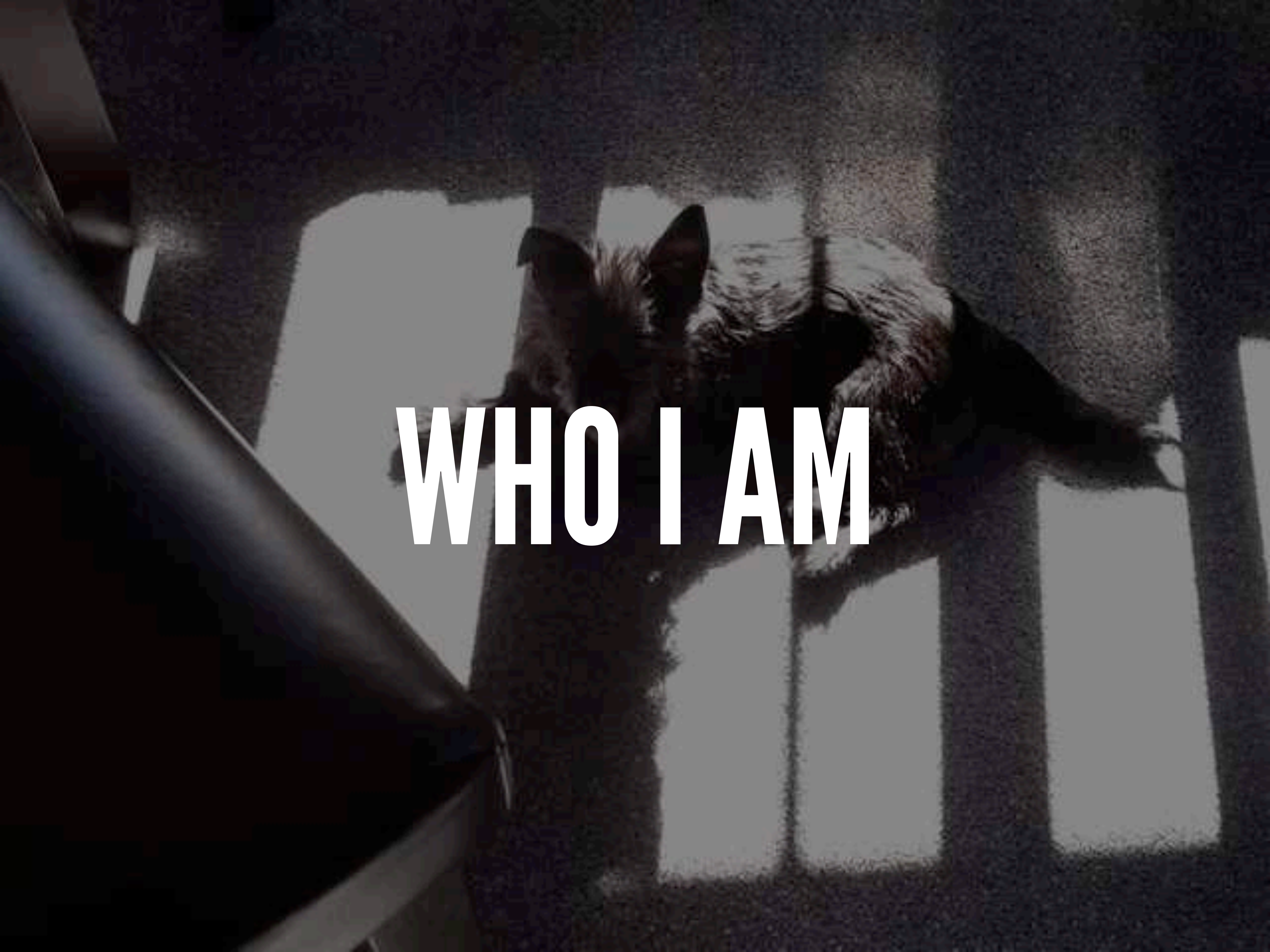
LEARN DEPENDENTLY-TYPED PROGRAMMING

WITH IDRIS

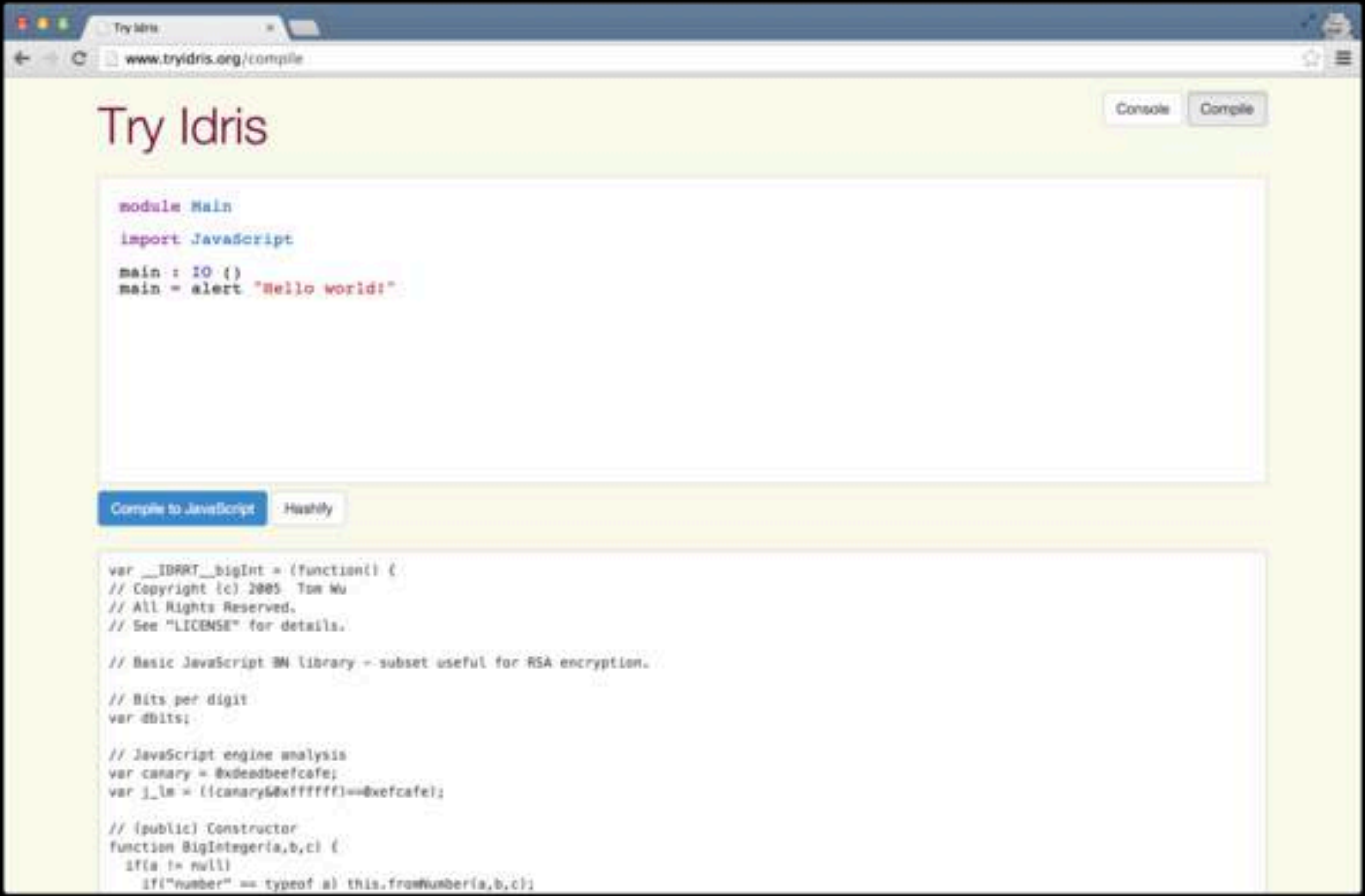

WHO I AM

- @puffnfresh
- Tiny contributor to Idris (18 commits)
- Played with dependent types for 2 years
- Been doing Idris for 6 months


ASSUMPTIONS

- Small experience with Haskell
- Have an install of Idris (can be tricky)

```
$ brew install ghc cabal-install
$ cabal update
$ cabal install alex
$ cabal install idris
```


Try Idris
www.tryidris.org/compile
Try Idris
Console
Compile
module Main
import JavaScript
main : IO ()
main = alert "Hello world!"
Compile to JavaScript
Hashify
var __IDRRT__bigInt = (function() {
// Copyright (c) 2005 Tom Wu
// All Rights Reserved.
// See "LICENSE" for details.
// Basic JavaScript BN library - subset useful for RSA encryption.
// Bits per digit
var dbits;
// JavaScript engine analysis
var canary = 0xdeadbeefcafe;
var j_lm = ((canary&0xffffff)==0xefcafe);
// (public) Constructor
function BigInteger(a,b,c) {
if(a != null)
if("number" == typeof a) this.fromNumber(a,b,c);

OUTLINE

1. Overview of dependent types and Idris
2. Work through exercises, I lead
3. Work through exercises, I help


MOTIVATION

Bad news: most software cannot be reasoned about

– Paul Phillips

- Curry-Howard; programs are proofs
- Let's make our proofs interesting
- Therefore let's use a powerful type system


MISCONCEPTIONS

- Idris is harder than Haskell
- Dependent types are hard


DEPENDENT TYPES
EVERYTHING IS A TERM

```
isIdris : Bool
isIdris = True

one : Nat
one = if isIdris then S Z else Z

StringList : Type
StringList = if isIdris then List Char else Int
```

- Types and kinds are values in universes
- Types can depend on values
- *Free* polymorphism, type constructors

```
the : (t : Type) -> (x : t) -> t
the _ a = a

one : Nat
one = the Nat Z
```

```
id1 : {t : Type} -> (x : t) -> t
id1 {t} a = a

id2 : (x : t) -> t
id2 a = a

id3 : t -> t
id3 a = a
```

```
Option : Type -> Type
Option = Maybe
```


TOTALITY

```
$ idris --total
$ idris --warnpartial
```

```
%default total

total plusOne : Nat -> Nat
plusOne Z = S Z
plusOne (S n) = S (S n)
```

I am often asked ‘how do I implement a server as a program in your terminating language?’

— Conor McBride

I reply that I do not: a server is a coprogram in a language guaranteeing liveness

— Conor McBride

- We always make progress
- Watch out for the totality checker!
- Church-Rosser theorem
- Evaluation is really normalisation!
- Can still do it all!


EQUALITY

```
data (=) : a -> b -> Type where
  refl : x = x

x : 1 = 1
x = refl

y : 1 + 1 = 2
y = refl
```

```
x : {a : Nat} -> a - a = Z
x {a=Z} = refl
x {a=S k} = x {a=k}

y : {a : Nat} -> a - a = Z
y {a} = replace {P = \x => (a - x = Z)}
          (plusZeroRightNeutral a)
          (minusPlusZero a Z)
```

```
x : {a : Nat} -> a - a = Z
x = ?xproof

xproof = proof
  intros
  rewrite (minusPlusZero a Z)
  rewrite (plusZeroRightNeutral a)
  trivial
```

- *The* problem of dependent types
  - Values are unified
  - Checked for syntactic/term equality


WHY IDRIS?

- LLVM, C, Java, JS backends
- FFI
- Lots of syntactic sugar
- Tactic rewriting
- Allows more lying/cheating
- REPL, editor modes, doc tools

```
    ____    __     _
   /  _/___/ /____(_)____
   / // __  / ___/ / ___/     Version 0.9.9.2
 _/ // /_/ / /  / (__  )      http://www.idris-lang.org/
/___/\__,_/_/  /_/____/       Type :? for help

Idris> :i Monad
Methods:

Prelude.Monad.>>= : (m a) -> (a -> m b) -> m b

Instances:

Monad PrimIO
Monad IO
Monad Maybe
(e : Type) -> Monad (Either e)
Monad List
(n : Nat) -> Monad (Vect n)
Idris> the Nat 1
1 : Nat
Idris> :t filter
Prelude.List.filter : (a -> Bool) -> (List a) -> List a
Prelude.Vect.filter : (a -> Bool) -> (Vect n a) -> (p ** Vect p a)
Idris> 
```

INSERT ./[vimshell] - default   unix vimshell   27:21

```
total minus : Nat -> Nat -> Nat
minus Z        right      = Z
minus left     Z          = left
minus (S left) (S right) = minus left right

total power : Nat -> Nat -> Nat
power base Z        = S Z
power base (S exp) = mult base $ power base exp

hyper : Nat -> Nat -> Nat -> Nat
hyper Z         a b       = S b
hyper (S Z)     a Z       = a
hyper (S(S Z)) a Z       = Z
hyper n         a Z       = S Z
hyper (S pn)    a (S pb) = hyper pn a (hyper (S pn) a pb)


--------------------------------------------------------------------------------
-- Comparisons
--------------------------------------------------------------------------------

data LTE  : Nat -> Nat -> Type where
  lteZero : LTE Z    right
  lteSucc : LTE left right -> LTE (S left) (S right)

```

Prelude/Nat.idr   6%   53:1

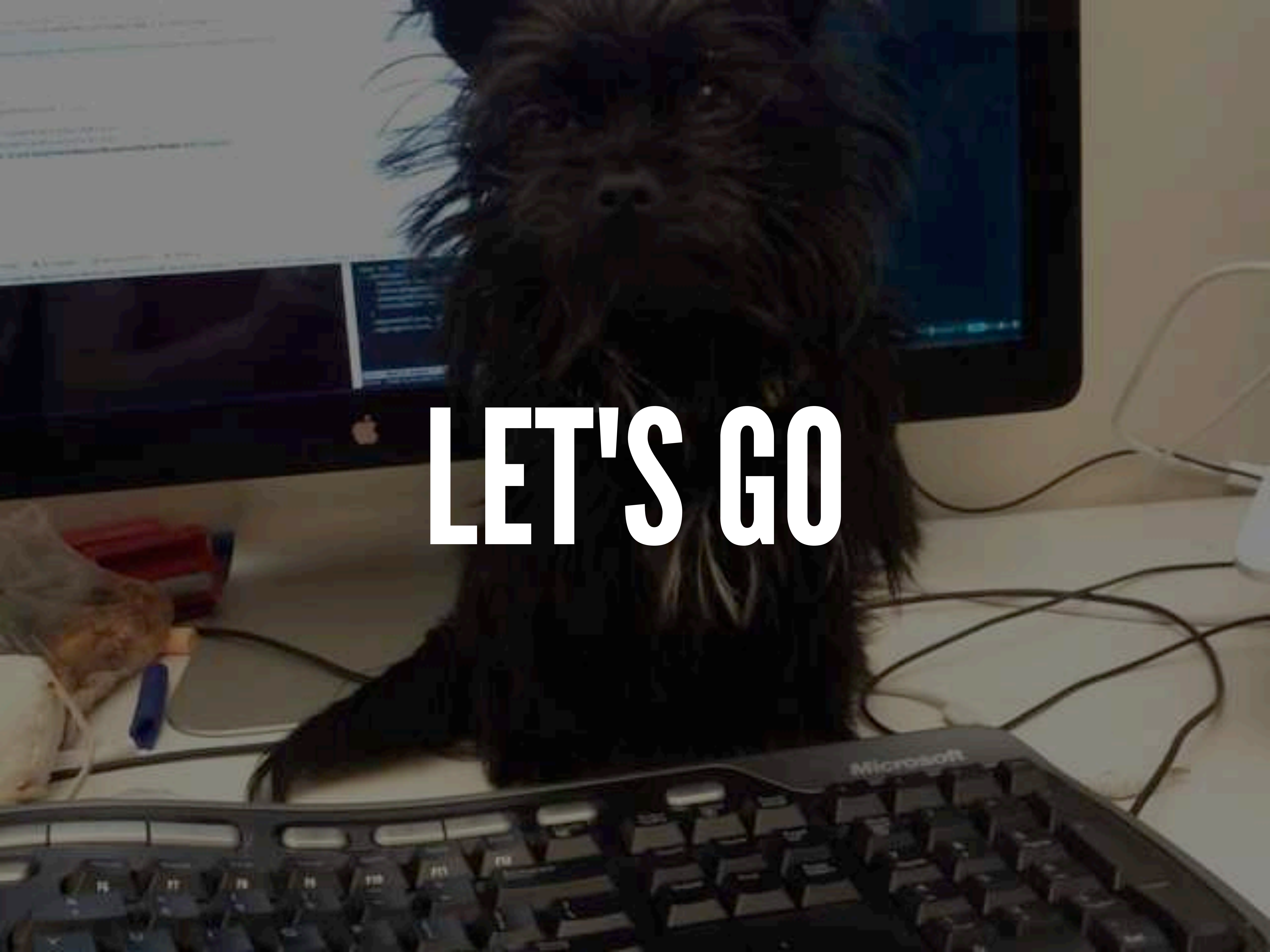
```
                   , prim__indexB64x2 x (prim__truncBigInt_B32 1))
                   )

instance Show Bits64x2 where
  show x =
    case viewB64x2 x of
      (a, b) =>
        "<" ++ prim__toStrB64 a
        ++ ", " ++ prim__toStrB64 b
        ++ ">"

instance (Show a, Show b) => Show (a, b) where
    show (x, y) = "(" ++ show x ++ ", " ++ show y ++ ")"

instance Show a => Show (List a) where
    show xs = "[" ++ show' "" xs ++ "]" where
        show' acc []          = acc
        show' acc [x]         = acc ++ show x
        show' acc (x :: xs) = show' (acc ++ show x ++ ", ") xs

instance Show a => Show (Vect n a) where
    show xs = "[" ++ show' xs ++ "]" where
        show' : Vect n a -> String
        show' []         = ""
        show' [x]        = show x
        show' (x :: xs) = show x ++ ", " ++ show' xs

instance Show a => Show (Maybe a) where
    show Nothing = "Nothing"
    show (Just x) = "Just " ++ show x

---- Functor instances

instance Functor PrimIO where
    map f io = prim_io_bind io (prim_io_return . f)

instance Functor IO where
    map f io = io_bind io (\b => io_return (f b))

instance Functor Maybe where
    map f (Just x) = Just (f x)
    map f Nothing  = Nothing

instance Functor (Either e) where
    map f (Left l) = Left l
    map f (Right r) = Right (f r)

---- Applicative instances

instance Applicative PrimIO where
    pure = prim_io_return
```

./Prelude.idr   29%   151:15

```
data Parity : Nat -> Type where
  Even : (n : Nat) -> Parity (n + n)
  Odd  : (n : Nat) -> Parity (S (n + n))
```


HOW TO IDRIS

- Idris Tutorial
- Idris library docs
- Idris library source
- Beginning Haskell: a Project Based Approach


LET'S GO
Microsoft

- printf
- Equality proofs
- Verified algebra
- Vector filtering

http://goo.gl/gfCJne