The Power of Pi

Wouter Swierstra
joint work with Nicolas Oury

Galois

Cryptol

Cryptol: example

```
x : [32]; -- a 32-bit word
x = 1337;
```

The **type** of a word records its **size**.

Cryptol: example

```
swab : [32] -> [32]

swab [a b c d] = [b a c d]
```

You can eliminate a word of size *n*k* by pattern matching on it as *n* words of size *k*.

Agda 101

```
data Nat : Set where
  Zero : Nat
  Succ : Nat -> Nat

data List (A : Set) : Set where
  Nil : List A
  Cons : A -> List A -> List
```

Words

```
data Vec (A : Set) : Nat -> Set where
  Nil : Vec A Zero
  _::_ : A -> Vec A n -> Vec A (Succ n)

Word : Nat -> Set
Word n = Vec Bit n
```

Views

- Introducing Cryptol-style pattern matching on words entails:
  - Defining a data type `WordView` indexed by a `Word (n * k)`;
  - Defining a function `view` that produces a suitable `WordView xs`, for every `xs : Word (n * k)`.

WordView

```
data WordView : Vec A (n*k) -> Set where
  Split : (xss : Vec (Vec A k) n)
        -> WordView (concat xss)
```

View

```
chop : (k : Nat) -> Vec A (n * k)
     -> Vec (Vec A k) n

view : (n : Nat) -> (k : Nat)
  -> (xs : Vec A (n * k)) ->  WordView xs
view n k xs = ... Split (chop k xs) ...
```

Example

```
swab : Word 32 -> Word 32

swab xs with view 4 8 xs

swab xs | Split (a :: b :: c :: d :: Nil)

   = concat (b :: a :: c :: d :: Nil)
```

Why index?

- Any view with type `(x : A) -> View x` has a left-inverse.
- Pattern matching maintains relation between original data and view.

Haskell

```
data Zero

data Succ n = Succ n

data Vec a n where

  Nil :: Vec a Zero

  Cons :: a -> Vec a n -> Vec a (Succ n)
```

Bitmaps

The PBM monochrome bitmap format is one way to generate black-and-white images:

```
P1 50 100\n OIIOOOIOOIO...
```

Real world Haskell

```
data PBM = PBM
  { width :: Integer
  , height :: Integer
  , data :: [[Bit]]
  }
```

Dependent types to the rescue!

- In dependently typed languages:
  - you precisely define your file format;
  - and get parsers and printers for free.

A small universe

```
data U : Set where
  CHAR : U
  VEC : Nat -> U -> U
  BIT : U
  ...
el : U -> Set
el CHAR = Char
...
```

Formats

```
data Format : Set where
  EOF : Format
  Bad : Format
  Read : (u : U)
       -> (el u -> Format)
       -> Format
_>>=_ = Read
```

PBM Format

```
PBM : Format
PBM = char ‘P’ >>
      char ‘1’ >>
      Read NAT >>= \n ->
      Read NAT >>= \m ->
      Read (VEC n (VEC m) BIT)
char c f = Read CHAR (\c’ -> ...)
```

Format Universe

```
type : Format -> Set

type EOF          = Unit

type Bad          = Empty

type (Read u f) = Sigma (el u) (type . f)
```

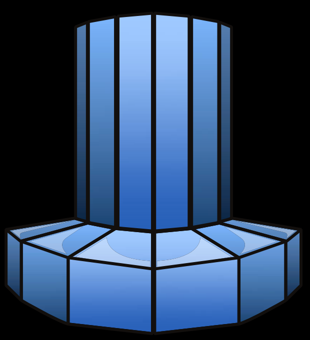
Read and Show

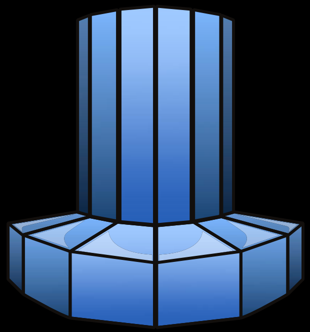
```
read : (f : Format) -> List Bit
     -> Maybe (type f )

show : (f : Format) -> (type f)
     -> List Bit
```

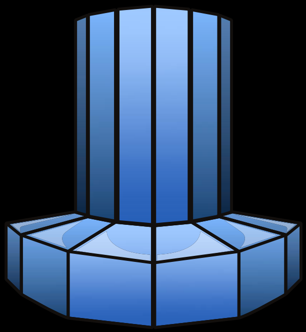
Haskell & Databases

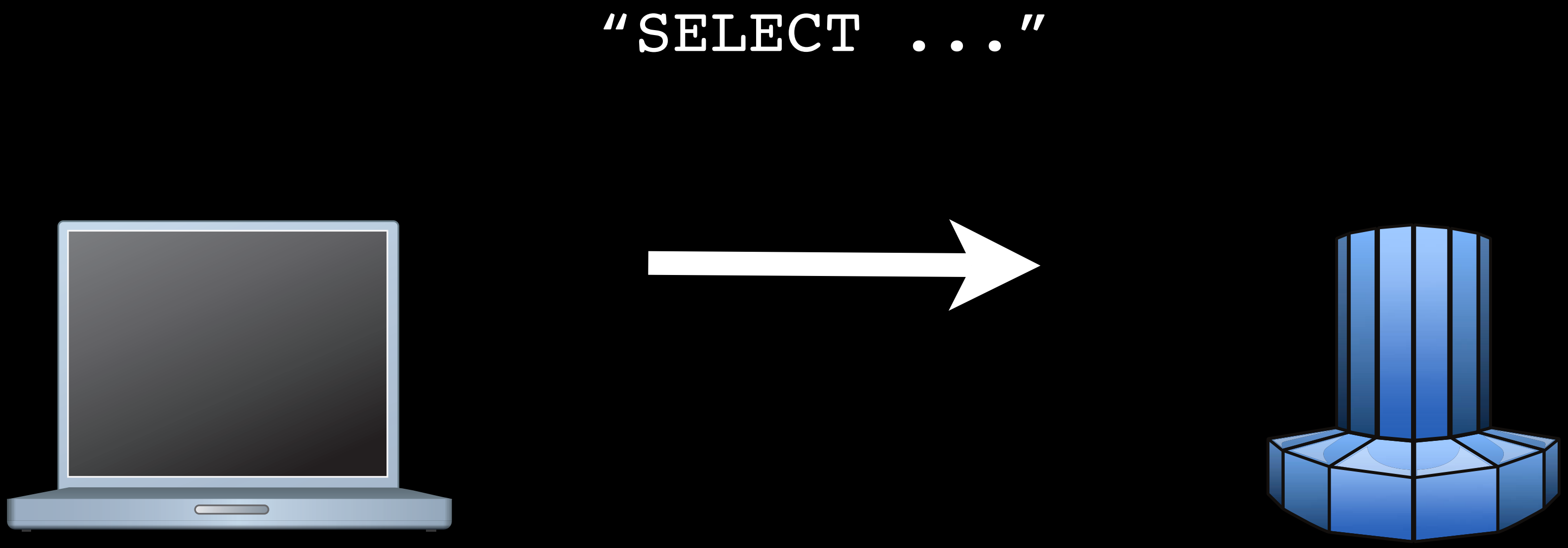
- Haskell interfaces need to:
  - use extensible records;
  - rely on type class tomfoolery;
  - represent everything by a String.
- ... accompanied by a preprocessor.

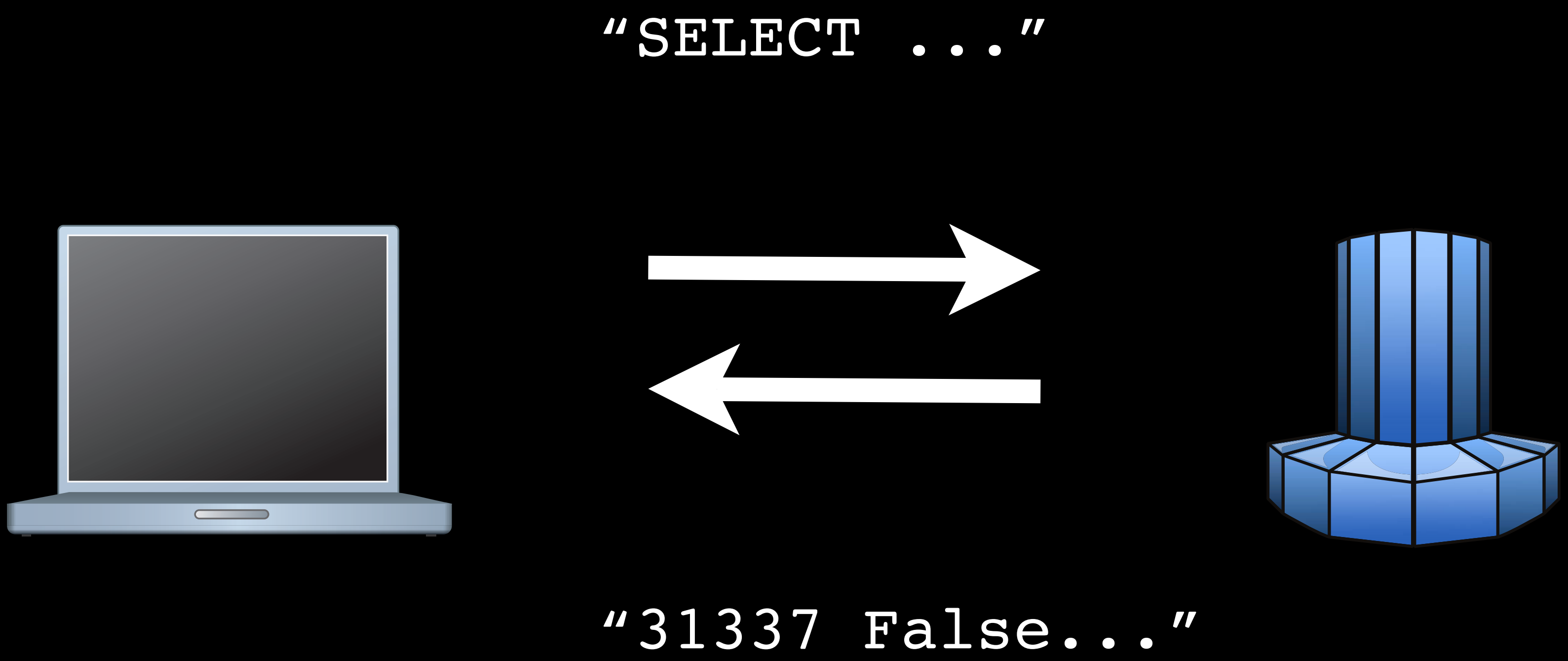


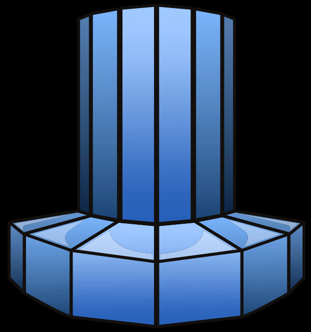


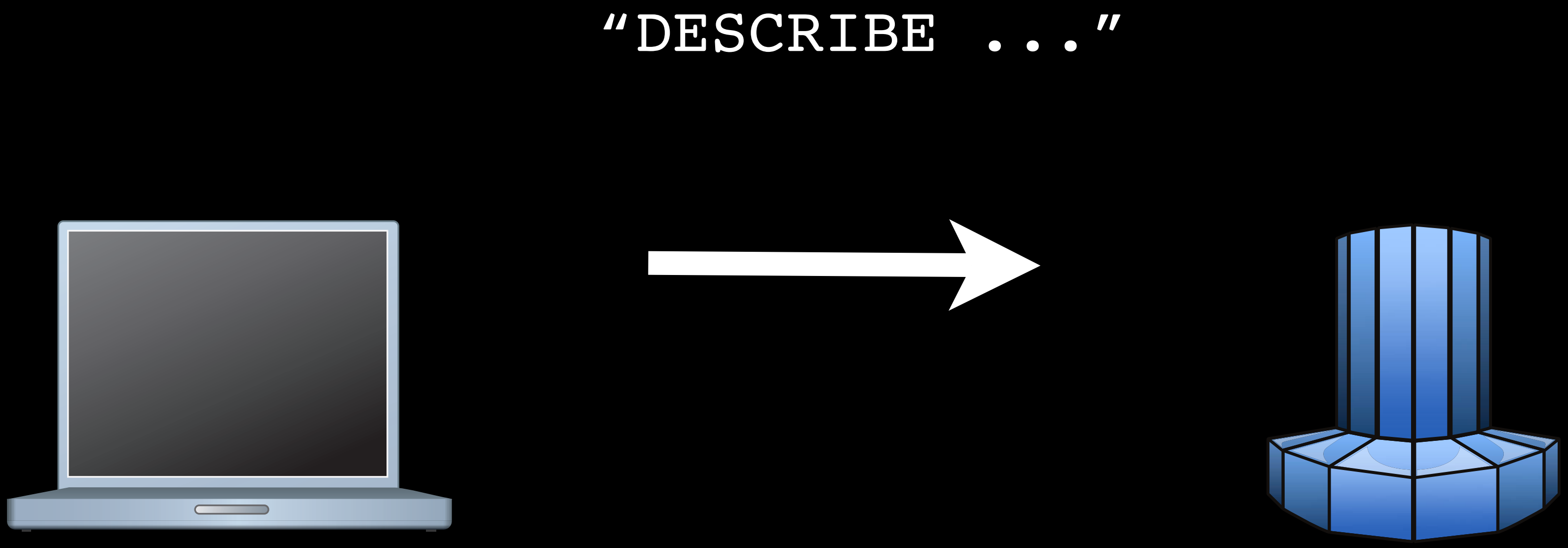
```
main = do
   db <- getLine
   connectServer db
   ...
```

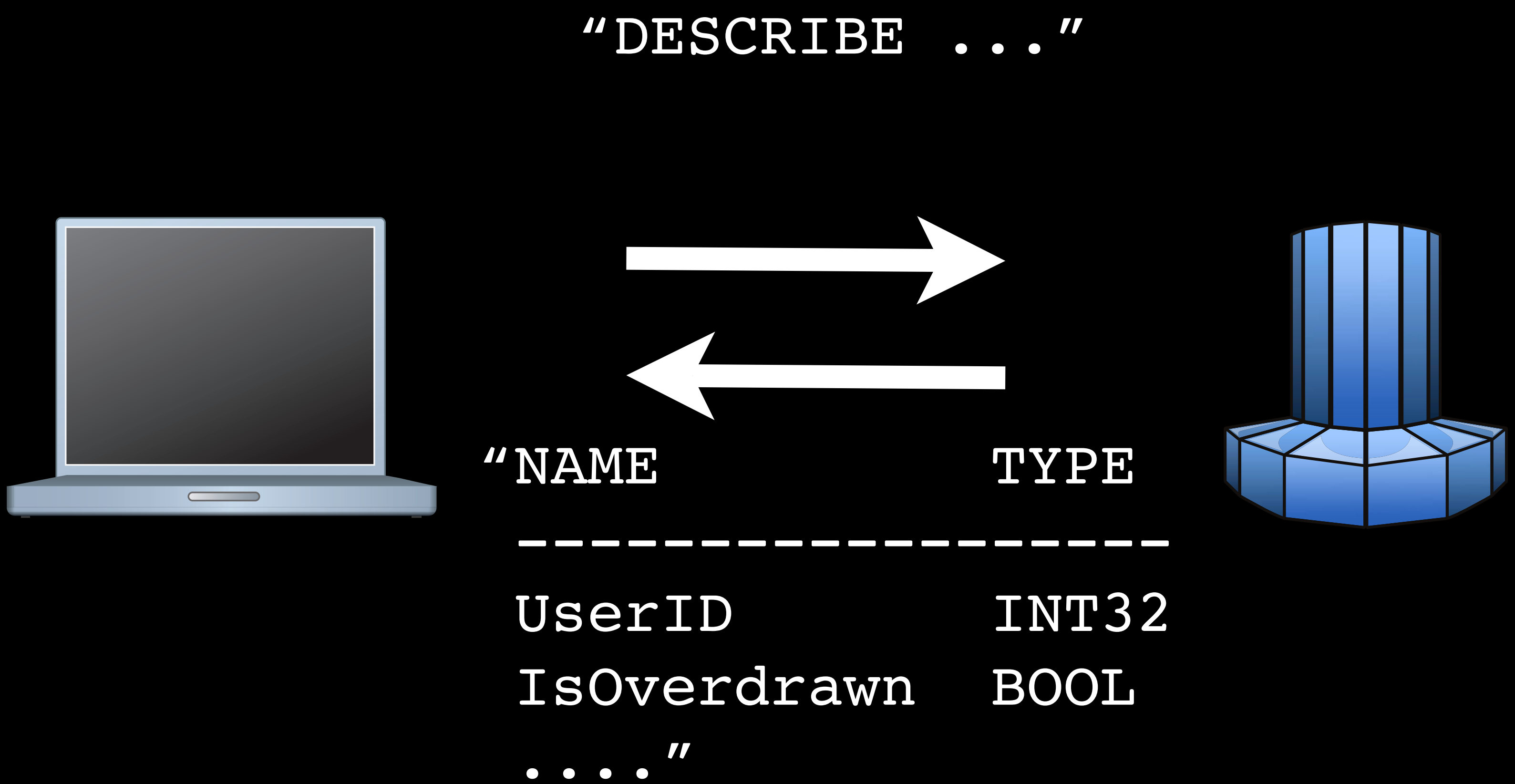



“SELECT ...”


"SELECT ..."
"31337 False..."




“DESCRIBE ...”


"DESCRIBE ..."
"NAME TYPE

UserID INT32
IsOverdrawn BOOL
...."

- Precise data types

- Precise data types
- Views

- Precise data types
- Views
- Universes

Conclusion