CHICAGO
INTERNATIONAL
SOFTWARE DEVELOPMENT
CONFERENCE 2015

goto; conference

Types vs Tests

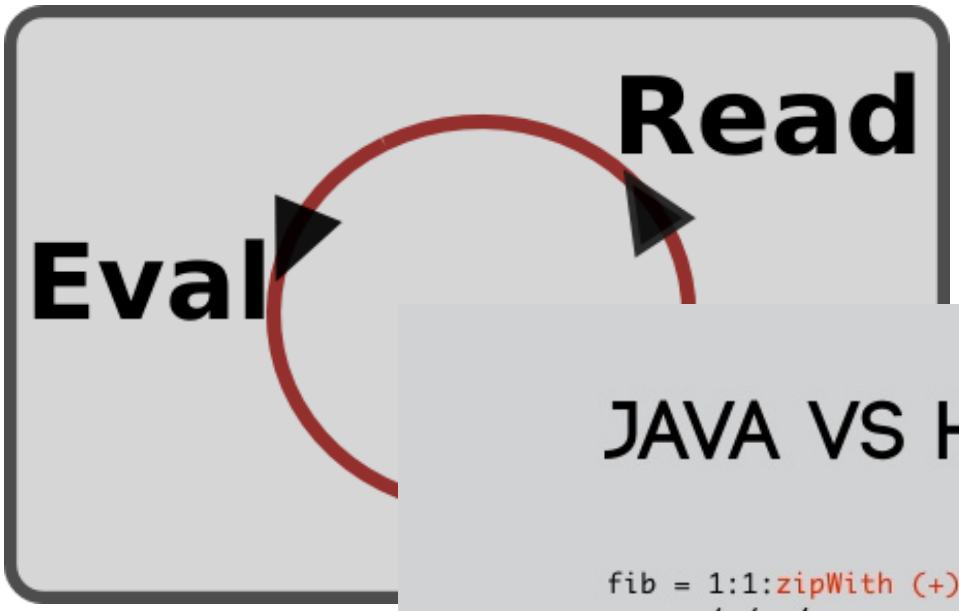
Amanda Laucher


@pandamonial


ILLINOIS
1818
Land
of
Lincoln
21st
State
Century
DW
2003
E PLURIBUS UNUM

Intersubjectivity

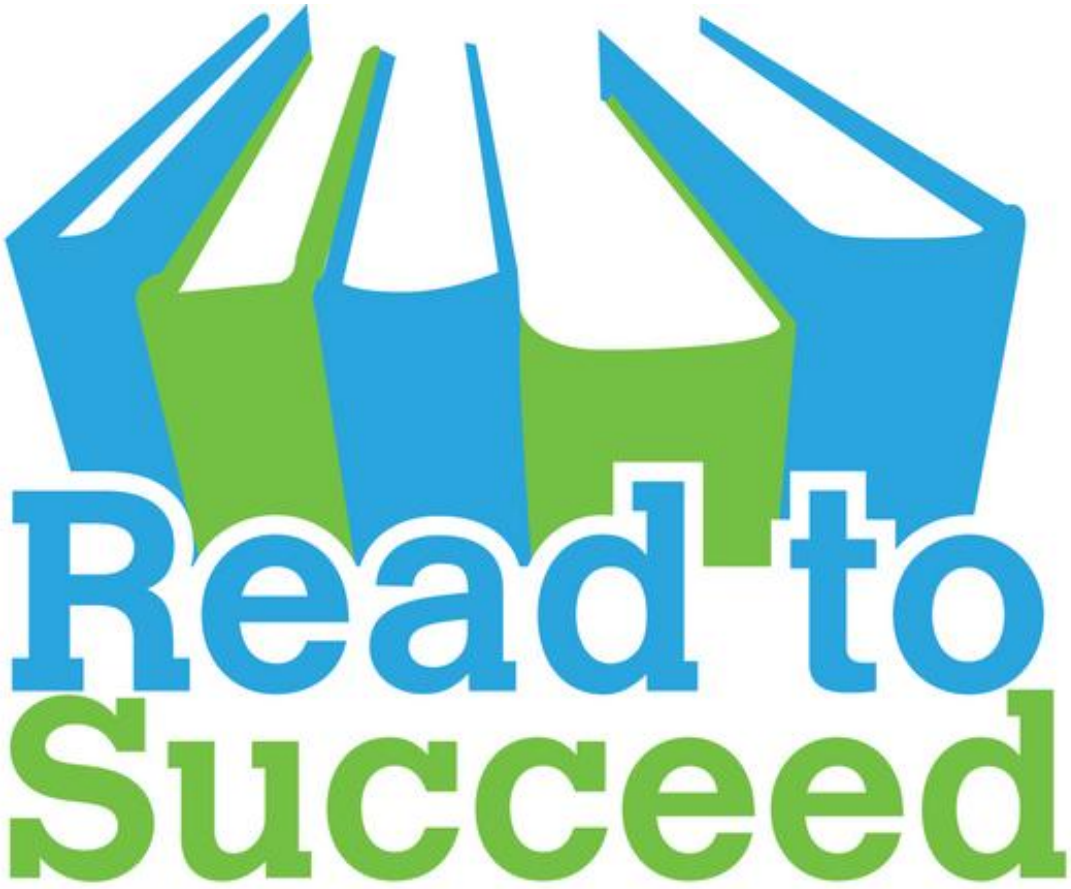
Assumptions



JAVA VS HASKELL

```
fib = 1:1:zipWith (+) fib (tail fib)
```

```
List <int> seq = new ArrayList(n);
seq[0] = 1;
seq[1] = 1;
for(int i = 2; i < n; i++) {
  seq[i] = seq[i-2] + seq[i-1];
}
```


Read to
Succeed

Craftsmanship

Quotes

“When in doubt create a type.” Martin Fowler

“Make illegal states unrepresentable.” Yaron Minsky

Michael Feathers describes legacy code as code without an automated test suite and now designs his code type signature first.

“In 5 years we will view compilation as the weakest form of unit testing” Stuart Halloway

“Given a good test suite the return on investment simply does not justify the use of static typing” Jay Fields

TDD is dead

David Heinemeier Hansson-

http://david.heinemeierhansson.com/2014/tdd-is-dead-long-live-testing.html
http://david.heinemeierhansson.com/2014/test-induced-design-damage.html

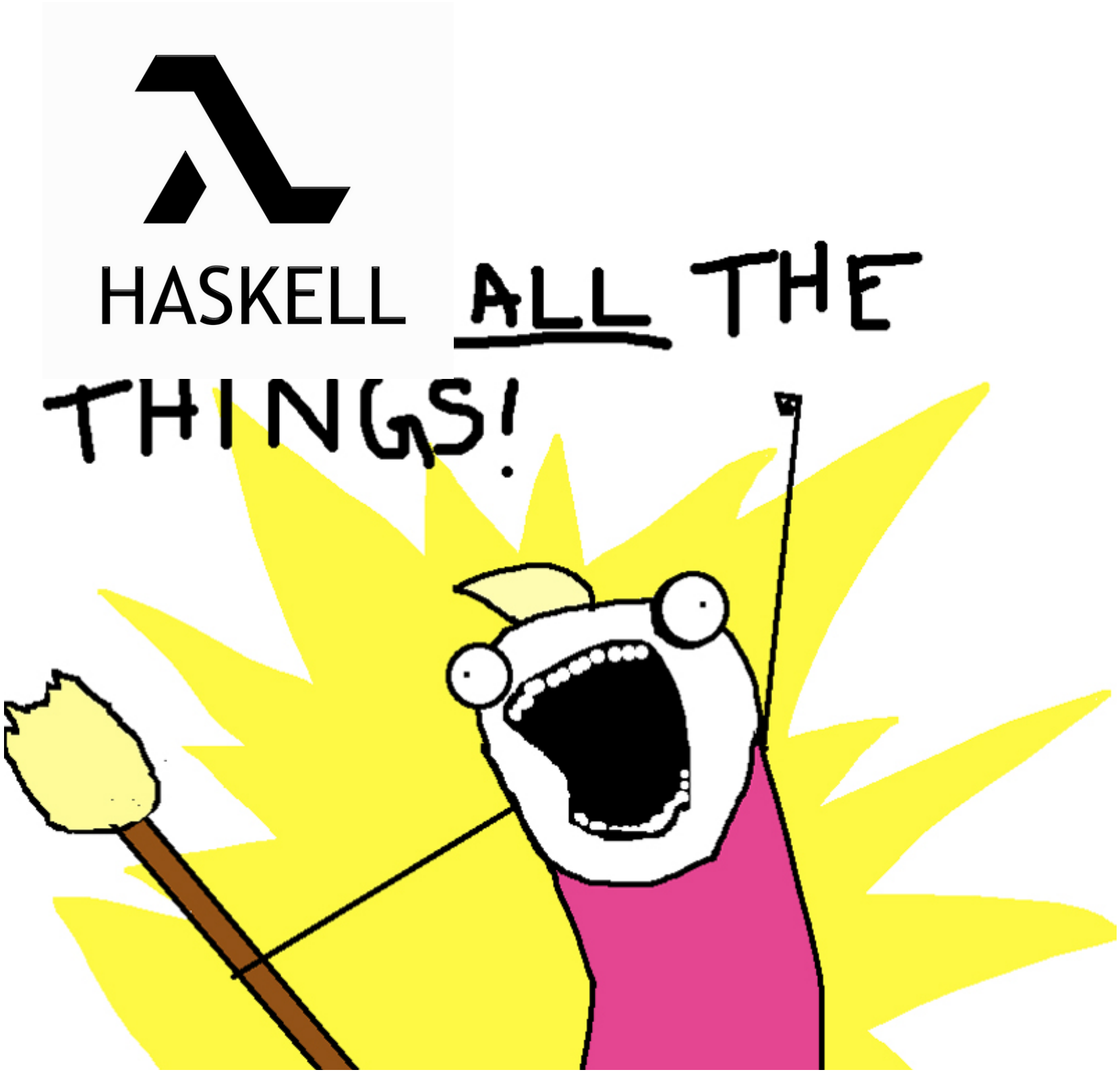
TDD isn't useful anymore

- We've learnt what we needed
- Unit tests aren't useful
- Testability hurts the design

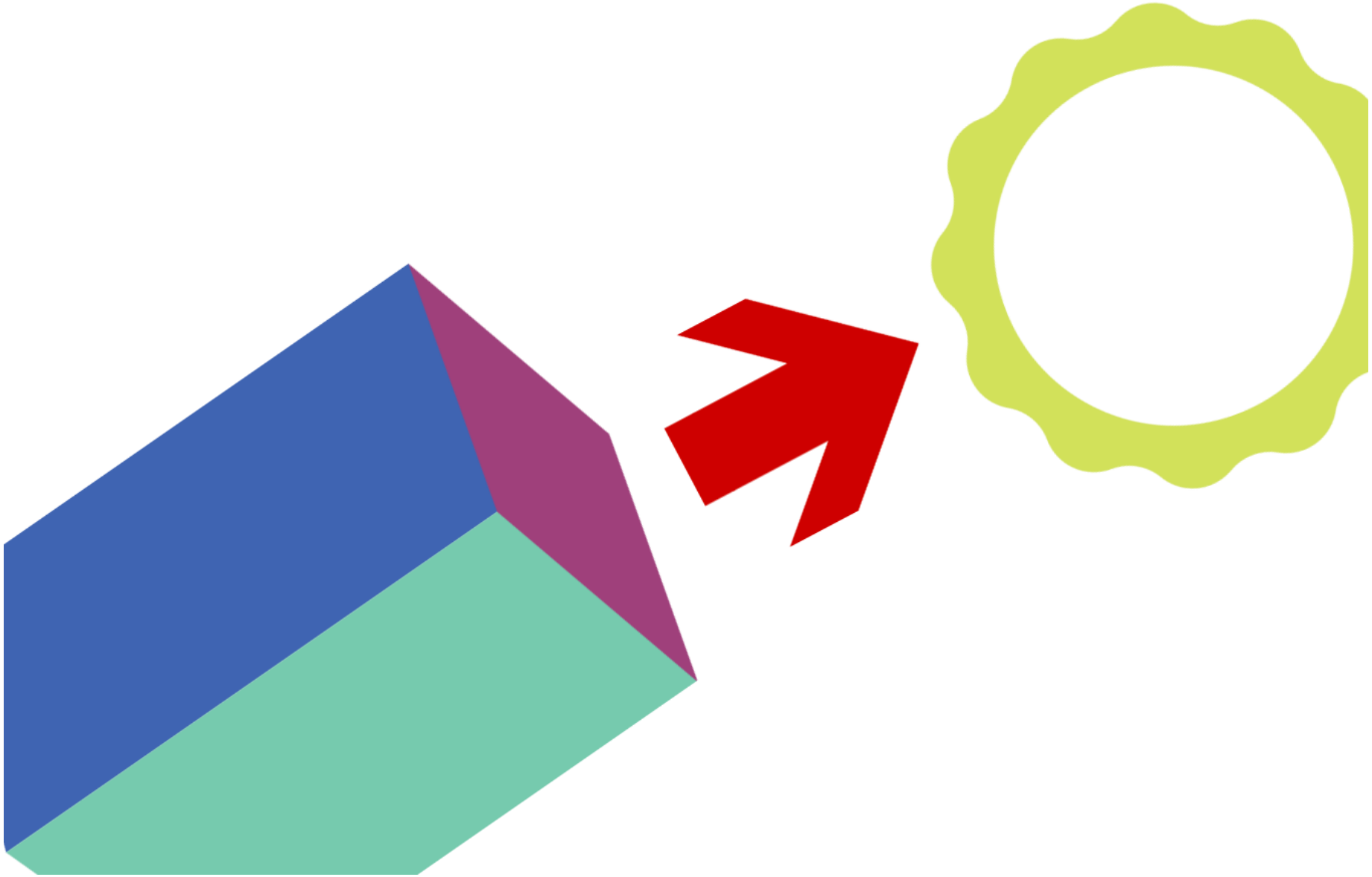
Seb Rose

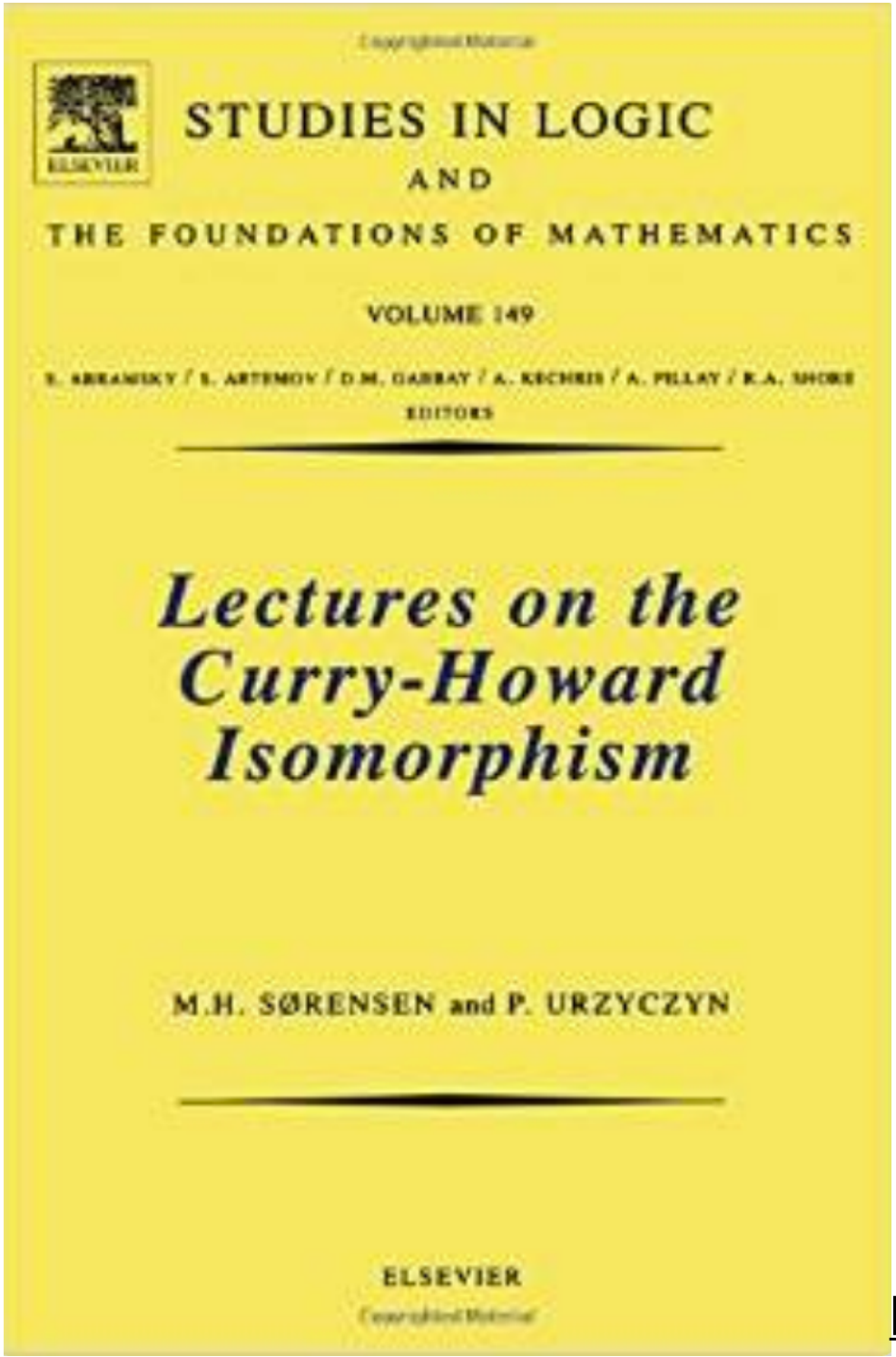
Claysnow Limited

@sebrose

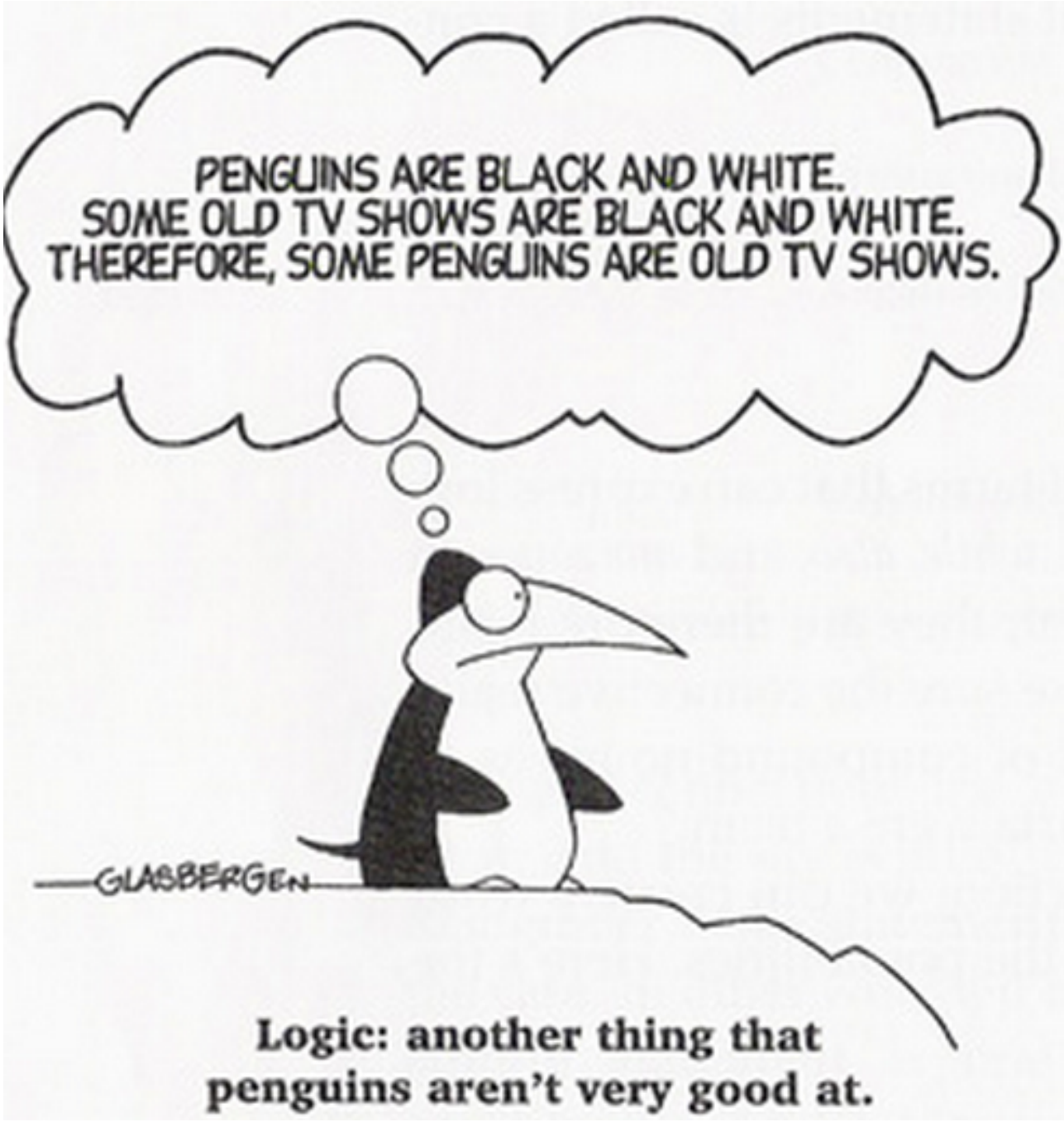

HASKELL ALL THE THINGS!

```
type Shape = Circle of int | Cuboid of int * int
```





http://bit.ly/1vvsXWC


PENGUINS ARE BLACK AND WHITE.
SOME OLD TV SHOWS ARE BLACK AND WHITE.
THEREFORE, SOME PENGUINS ARE OLD TV SHOWS.
GLASBERGEN
Logic: another thing that
penguins aren't very good at.

Type signature is a **Theorem**
Function definition is the **Proof**

Types:

Reduce bugs

Make code run faster

Define interfaces

Check compliance

Document model

Types:

Reduce bugs

Make code run faster

Define interfaces

Check compliance

Document **model**

Tests:

Reduce bugs

~~Make code run faster~~

~~Define interfaces~~

Check compliance

Document model

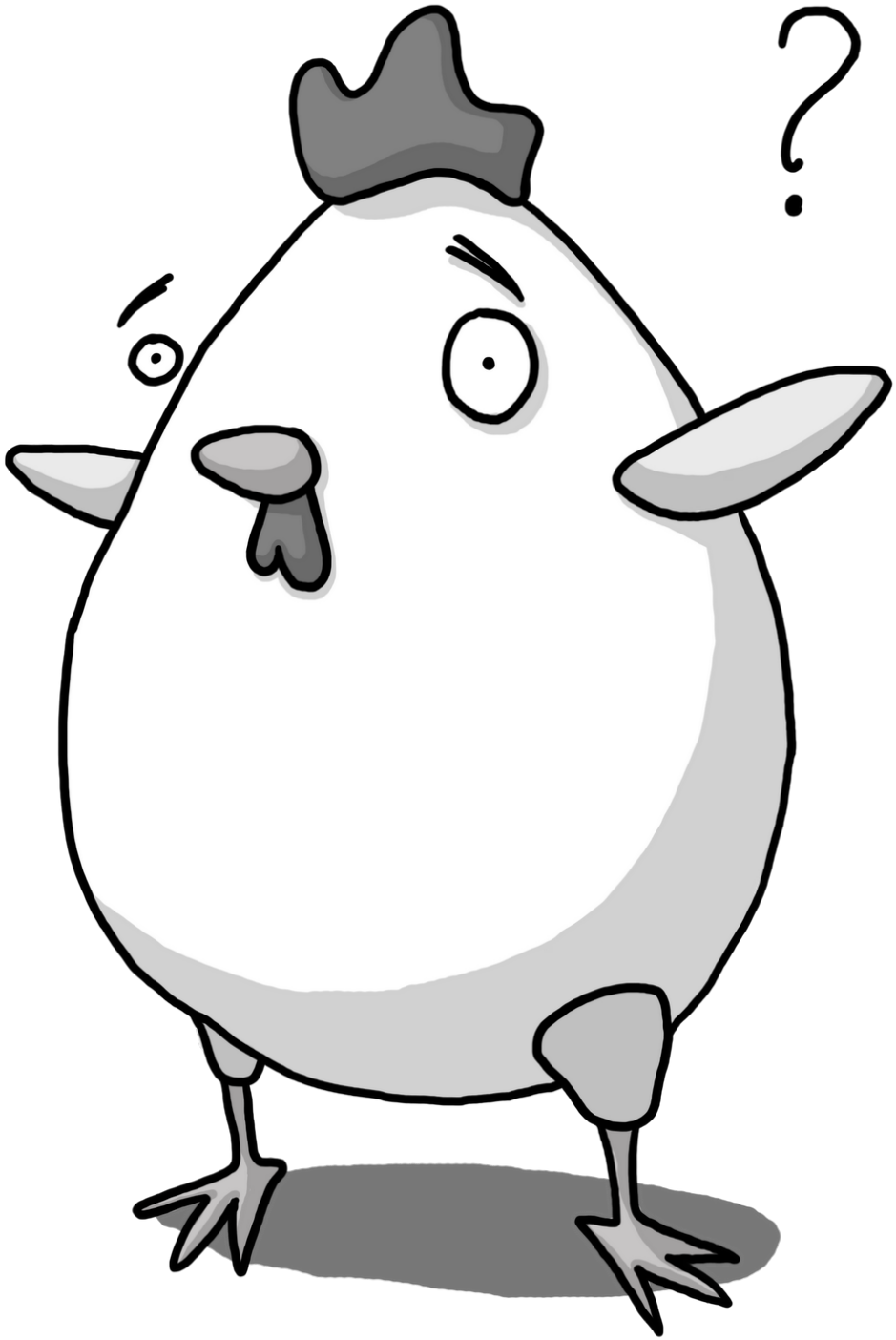
Test “logic”

Functional Tests

Property Based Testing

Unit Tests

REPL Tests

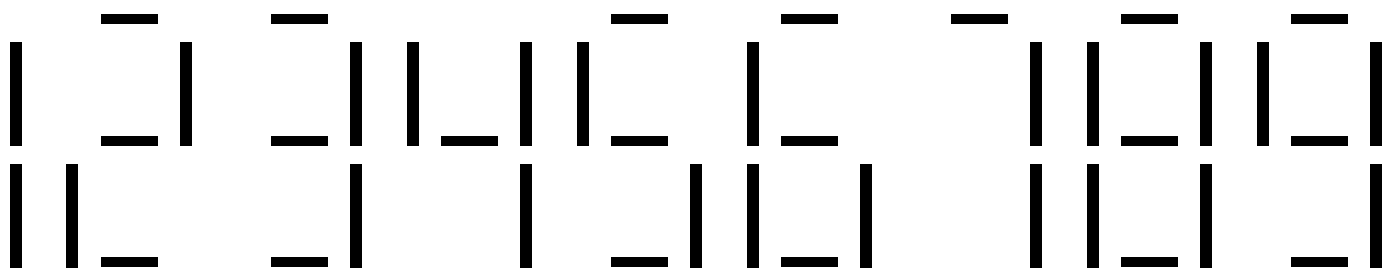



Braintree

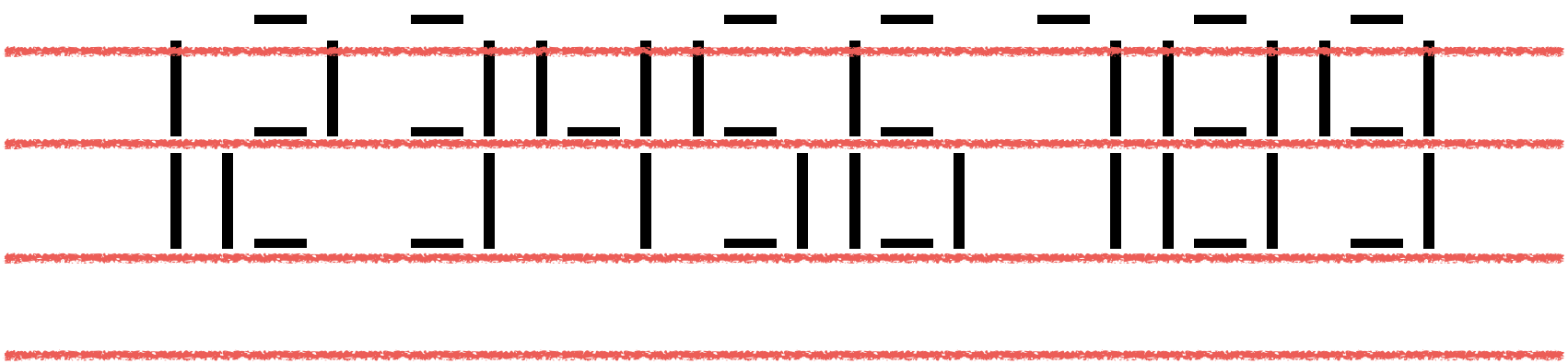

Easy St
Lazy Way


Science
With Me!

BANK OCR CODE KATA



=> 123456789



=> 123456789

Story 2

Account number:

3 4 5 8 8 2 8 6 5

Position names:

d9 d8 d7 d6 d5 d4 d3 d2 d1

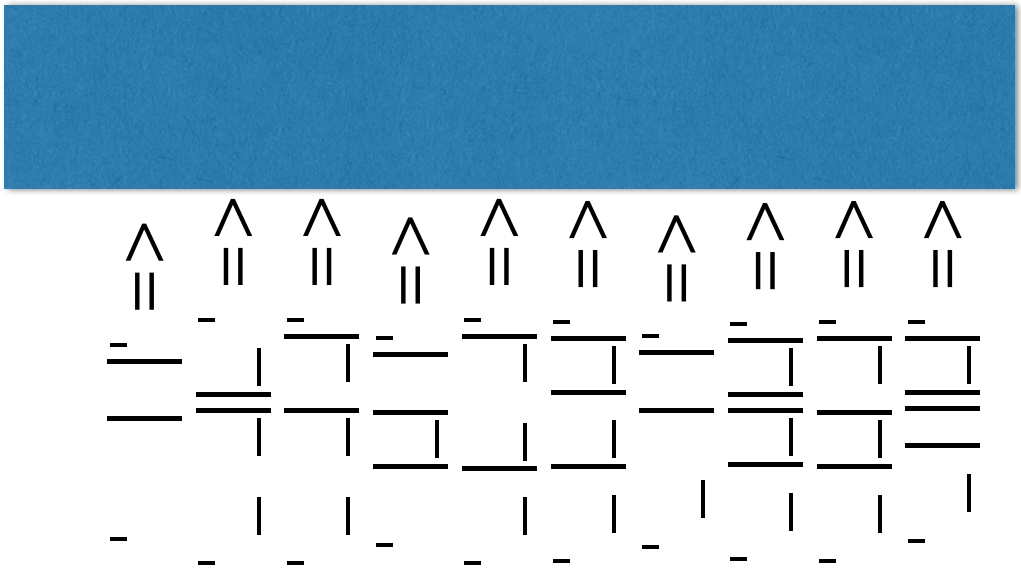
Checksum calculation:

(d1 + 2 + 3*d3 + …+9* d9) mod 11 = 0

- TDD
- Unit testing throughout or after
- Functional Tests
- Type signatures first
- REPL driven
- Property based testing first TDD
- Property based testing throughout or after

Analysis

- 100’s of code samples
- Every language we could think of
- Github/web examples



```
type Digit = Zero | One | Two | Three
with member x.toInt = match x with
                      |Zero -> 0
                      |One -> 1
                      |Two -> 2
                      |Three -> 3
let stringToDigit = function
        |"
          _
         | |
         |_|
           " -> Some Zero
        |
           |
           |
            " -> Some One
        |"
          _
           _|
         |_  " -> Some Two
        |"
          _
           _|
           _|" -> Some Three
       |_-> None
```

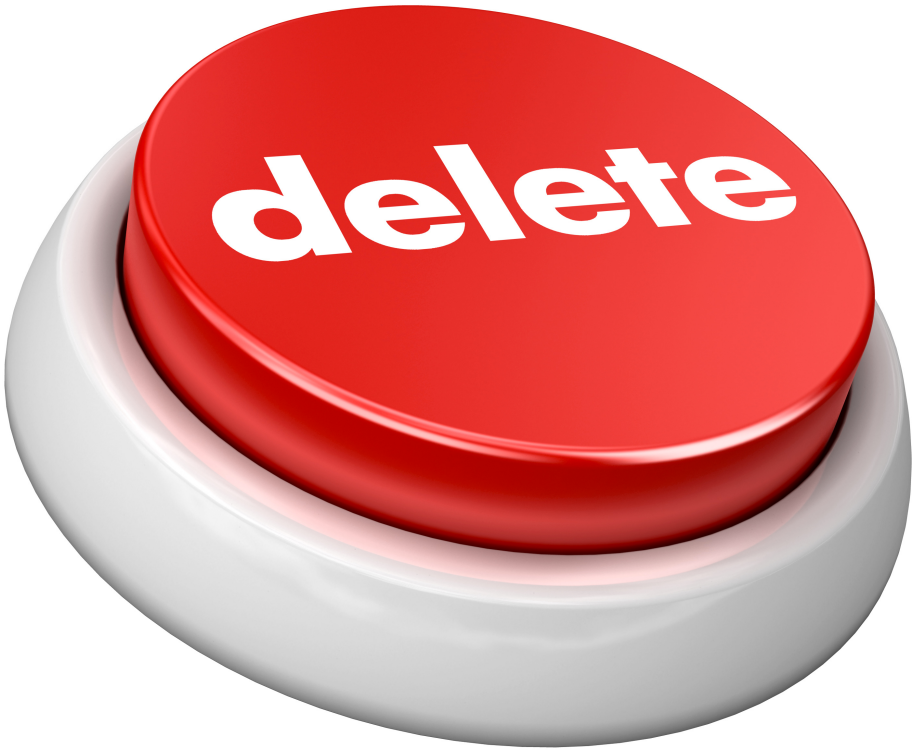
```
type AccountType =
   |Valid of Account
   |Invalid
and Account = {d9 :int; d8 : int; d7 : int; d6 : int}
   with member x.validate =
      if int x.d9 + 2 * int x.d8 + 3 *
         int x.d7 + 4 * int x.d6 % 11 = 0
      then Valid x
      else Invalid
```

Removed Types

```
type LegalChar =
        |Underscore
        |Pipe
        |Space
```

- Tests validate what types are not able to prove
- Property based testing : when there is a forAll, you should consider a type


HELP


delete

```clojure
(deftest valid-checksums
  (are [result] (= 0 (mod result 11))
       (checksum [0 0 0 0 0 0 0 5 1])
       (checksum [3 4 5 8 8 2 8 6 5])
       (checksum [4 5 7 5 0 8 0 0 0])))

(deftest invalid-checksums
  (are [result] (not (= 0 (mod result 11)))
       (checksum [1 2 3 4 5 6 7 8 0])
       (checksum [6 6 4 3 7 1 4 9 5])
       (checksum [9 8 7 6 5 4 3 2 1])))

(deftest valid-account-numbers
  (are [-vector] (valid? -vector)
       [0 0 0 0 0 0 0 5 1]
       [3 4 5 8 8 2 8 6 5]
       [4 5 7 5 0 8 0 0 0]))

(deftest invalid-account-numbers
  (are [-vector] (not (valid? -vector))
       [1 2 3 4 5 6 7 8 0]
       [6 6 4 3 7 1 4 9 5]
       [9 8 7 6 5 4 3 2 1]
       [0 0 0 0 0 0 \? 5 1]))

(deftest legibility
  (is (legible? [0 0 0 0 0 0 0 5 1]))
  (is (not (legible? [0 0 0 0 0 0 \? 5 1]))))

(deftest describe-validity
  (are [result -vector]
       (= result (error-description -vector))
       nil   [0 0 0 0 0 0 0 5 1]
       "ERR" [6 6 4 3 7 1 4 9 5]
       "ILL" [0 0 0 0 0 0 \? 5 1]))
```

- Types save me from having to even think about certain categories of tests.
- It’s easy to get lost when you never have to deliver.
- Syntax matters!

```scala
trait HasChecksum[L <: HList, S <: Nat]

implicit object hnilHasChecksum extends HasChecksum[HNil, _0]

implicit def hlistHasChecksum[
  H <: Nat,  T <: HList, S <: Nat,
  TL <: Nat, TS <: Nat,
  HL <: Nat, HS <: Nat
](implicit
  tl: LengthAux[T, TL],
  ts: HasChecksum[T, TS],
  hl: ProdAux[H, Succ[TL], HL],
  hs: SumAux[HL, TS, HS],
  sm: ModAux[HS, _11, S]
) = new HasChecksum[H :: T, S] {}

// Check that the list has nine elements and a checksum of zero.
def isValid[L <: HList](l: L)(implicit
  len: LengthAux[L, _9],
  hcs: HasChecksum[L, _0]
) {}
```

```scala
// Now the following valid sequence (an example from the kata) compiles:
isValid(_3 :: _4 :: _5 :: _8 :: _8 :: _2 :: _8 :: _6 :: _5 :: HNil)

// But these invalid sequences don't:
// isValid(_3 :: _1 :: _5 :: _8 :: _8 :: _2 :: _8 :: _6 :: _5 :: HNil)
// isValid(_3 :: _4 :: _5 :: _8 :: _8 :: _2 :: _8 :: _6 :: HNil)
```

```
describe "#check?" do
  context "when the account number is good" do
    # good account numbers were taken from the user story specs
    Then { checker.check?("000000000").should be_true }
    Then { checker.check?("000000051").should be_true }
    Then { checker.check?("123456789").should be_true }
    Then { checker.check?("200800000").should be_true }
    Then { checker.check?("333393333").should be_true }
    Then { checker.check?("490867715").should be_true }
    Then { checker.check?("664371485").should be_true }
    Then { checker.check?("711111111").should be_true }
    Then { checker.check?("777777177").should be_true }
  end
```

I haven’t found a language that does a great job of making illegal states completely unrepresentable.

- Types scale better than tests
- Tests can be valuable for open source or distributed teams as a form of safety and documentation (especially functional tests)
- Small/short lived codebase means little value for types and great value for tests
- Types make it easy to refactor
- Types help to modularize code
- Tests take a long time to run and types to compile
- Refactor to types

All Type Systems Are Not Created Equal

Sum Types

Inclusive OR

Either [Failure, Success]

+ Pattern Matching

Product Types

(AND

records, objects, tuples

x*y*z)

+ Currying

Safety of nominal vs structural typing

```
type X = Bool
type Y = Bool
```

Type inference

Incomplete but expressive

OR

Complete but weak

In a perfect world…

Dependent Types

```
datatype 'a list with nat =
    nil(0)
  | {n:nat} cons(n+1) 'a * 'a list(n)
fun zip ([], []) = []
  | zip (x :: xs, y :: ys) = (x, y) :: zip (xs, ys)
withtype {n:nat} <n> =>
      'a list(n) * 'b list(n) -> ('a * 'b) list(n)
```

In a perfect world…

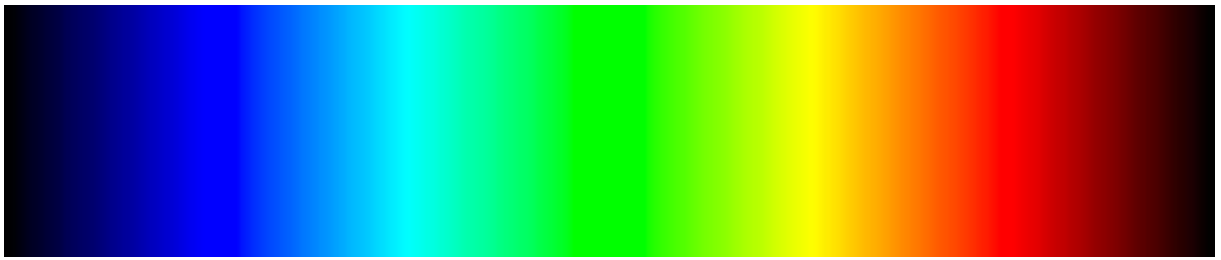
Dependent Types

```
append : Vect n a -> Vect m a -> Vect (n + m) a

append Nil       ys = ys
append (x :: xs) ys = x :: app xs ys
```

Final Thoughts

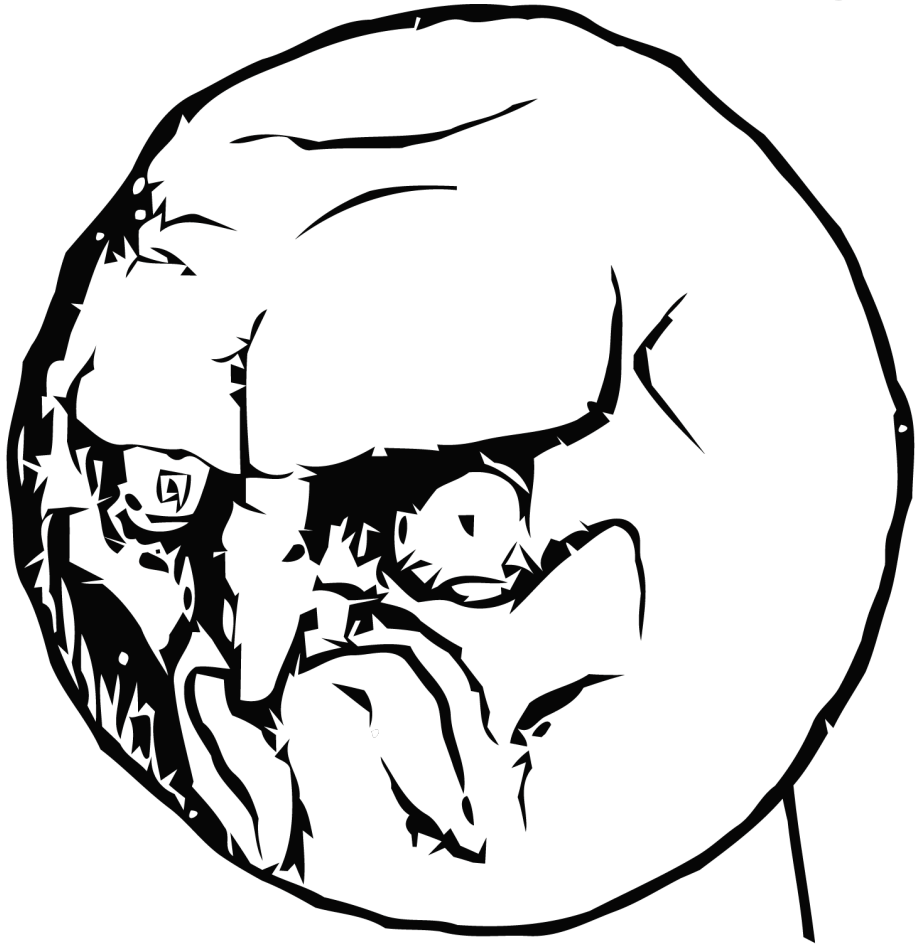
Tests



Types

Types = For All
Tests = There Exists

Stringly Typed Programming in a statically typed language?



NO.


Future

Future languages will make type level programming indistinguishable from the rest of the code

Where does simulation testing fit in? Mutation testing?

Type signature is a Theorem
Function definition is the Proof

Types = For All
Tests = There Exists

Use the facilities available

CHICAGO
INTERNATIONAL
SOFTWARE DEVELOPMENT
CONFERENCE 2015

goto; conference

Questions?

Please remember to evaluate via the GOTO Guide App


@pandamonial

follow us @gotochgo

Conference: May 11-12 / Workshops: 13-14