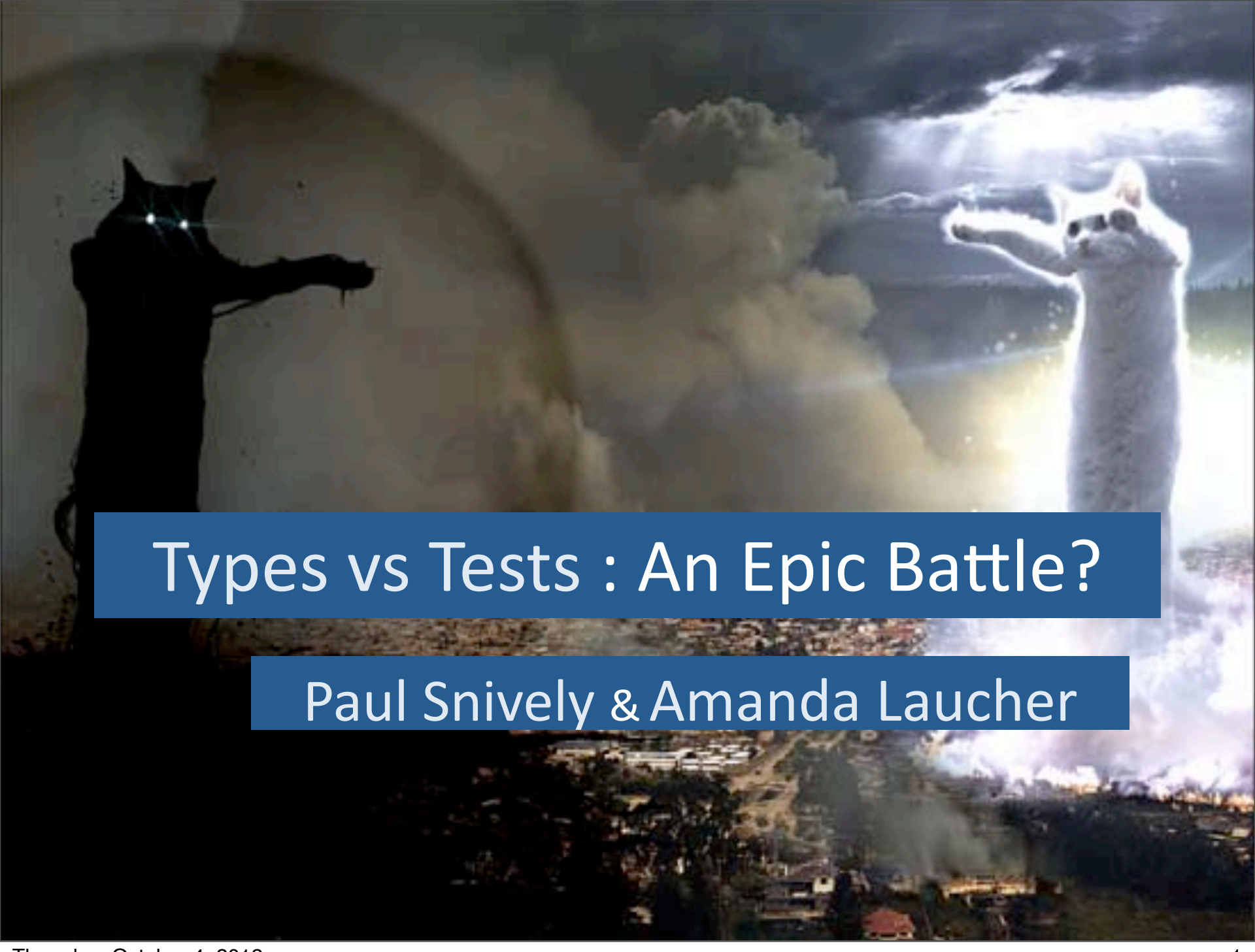

Types vs Tests : An Epic Battle?
Paul Snively & Amanda Laucher

Logic is in the air!

"The resurgence of Datalog" — ELC

MiniKanren—Scheme

core.logic—Clojure

Michael Focus
David Nolan
Dan Friedman

Assumptions

REPL

Logic

Some languages are better suited for correct code

Read vs write

We are lazy bastards!

Assumptions

REPL

Logic

Some languages are better suited for correct code

Read vs write

We are lazy bastards!

Assumptions

REPL

Logic

Some languages are better suited for correct code

Read vs write

We are lazy bastards!

Assumptions

REPL

Logic

Some languages are better suited for correct code

Read vs write

We are lazy bastards!

Assumptions

REPL

Logic

Some languages are better suited for correct code

Read vs write

We are lazy bastards!

Quotes

When in doubt, create a type : *Martin Fowler*

Quotes

When in doubt, create a type : *Martin Fowler*

Make illegal states unrepresentable : *Yaron Minsky*

Quotes

Michael Feathers defines legacy code as code without automated tests

Quotes

Michael Feathers defines legacy code as code without automated tests

In 5 years, we'll view compilation as the weakest form of unit testing. : *Stuart Halloway*

Quotes

Michael Feathers defines legacy code as code without automated tests

In 5 years, we'll 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*

Stereotypes

It's easier to refactor with tests than types

No tests = no trust

Tests take a long time to run & types to compile

Property based testing can replace unit testing

Modular design only occurs only with TDD

I don't get errors that can be prevented by types

Stereotypes

It's easier to refactor with tests than types

No tests = no trust

Tests take a long time to run & types to compile

Property based testing can replace unit testing

Modular design only occurs only with TDD

I don't get errors that can be prevented by types

Stereotypes

It's easier to refactor with tests than types

No tests = no trust

Tests take a long time to run & types to compile

Property based testing can replace unit testing

Modular design only occurs only with TDD

I don't get errors that can be prevented by types

Stereotypes

It's easier to refactor with tests than types

No tests = no trust

Tests take a long time to run & types to compile

Property based testing can replace unit testing

Modular design only occurs only with TDD

I don't get errors that can be prevented by types

Stereotypes

It's easier to refactor with tests than types

No tests = no trust

Tests take a long time to run & types to compile

Property based testing can replace unit testing

Modular design only occurs only with TDD

I don't get errors that can be prevented by types

Stereotypes

It's easier to refactor with tests than types

No tests = no trust

Tests take a long time to run & types to compile

Property based testing can replace unit testing

Modular design only occurs only with TDD

I don't get errors that can be prevented by types

Stereotypes

Ivory Tower vs hippies

100% coverage

Typed code is verbose

Types take too long

Testing is for QA

Stereotypes

Ivory Tower vs hippies

100% coverage

Typed code is verbose

Types take too long

Testing is for QA

Stereotypes

Ivory Tower vs hippies

100% coverage

Typed code is verbose

Types take too long

Testing is for QA

Stereotypes

Ivory Tower vs hippies

100% coverage

Typed code is verbose

Types take too long

Testing is for QA

Stereotypes

Ivory Tower vs hippies

100% coverage

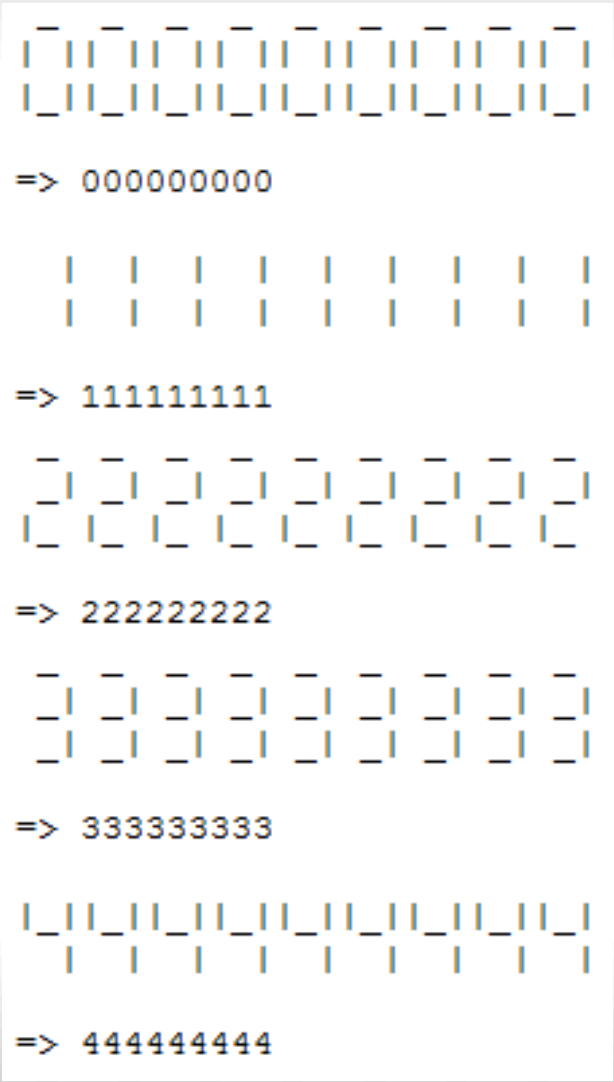
Typed code is verbose

Types take too long

Testing is for QA

A walk on the wildside?

Our Approach – The Kata



We used different languages to do the same Kata
We stepped outside of our comfort zone to try a different approach (or multiple rewrites)
We didn't pair
We chatted regularly about our outcome

Story 1

- Each entry is 4 lines long
- Each line has 27 characters
- The first 3 lines of each entry contain an account number written using pipes and underscores, and the fourth line is blank.
- Each account number should have 9 digits, all of which should be in the range 0-9.
- A normal file contains around 500 entries.

OCRChars no value

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*d2+3*d3 +..+9*d9) mod 11 = 0

Story 3

- The file has one account number per row
- If some characters are illegible, they are replaced by a ?.
- In the case of a wrong checksum, or illegible number, this is noted in a second column indicating status.

457508000
664371495 ERR
86110??36 ILL

Amanda takes a stab

The good!

Amanda's Approach

- F#
- Signatures first
- Types
- REPL play for algorithms
- Tests for validation

Interesting Code

```
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
```

Interesting Code

```
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
```


Deleted types as well. Sometimes the types get removed with new use cases(story 4)

Lessons learned

Types save me from having to even think about certain categories of tests

Tests help me out when I get stuck but I mostly run them in the REPL and **delete**

Code is structured differently with REPL

Most modern languages don't have a strong enough type system to make illegal states un-representable

It's easy to get lost in a space where you never deliver

Test verify when types can't prove

Lessons learned

Types save me from having to even think about certain categories of tests

Tests help me out when I get stuck but I mostly run them in the REPL and delete

Code is structured differently with REPL

Most modern languages don't have a strong enough type system to make illegal states un-representable

It's easy to get lost in a space where you never deliver

Test verify when types can't prove

Lessons learned

Types save me from having to even think about certain categories of tests

Tests help me out when I get stuck but I mostly run them in the REPL and **delete**

Code is structured differently with REPL

Most modern languages don't have a strong enough type system to make illegal states un-representable

It's easy to get lost in a space where you never deliver

Test verify when types can't prove

Lessons learned

Types save me from having to even think about certain categories of tests

Tests help me out when I get stuck but I mostly run them in the REPL and **delete**

Code is structured differently with REPL

Most modern languages don't have a strong enough type system to make illegal states un-representable

It's easy to get lost in a space where you never deliver

Test verify when types can't prove

Lessons learned

Types save me from having to even think about certain categories of tests

Tests help me out when I get stuck but I mostly run them in the REPL and **delete**

Code is structured differently with REPL

Most modern languages don't have a strong enough type system to make illegal states un-representable

It's easy to get lost in a space where you never deliver

Test verify when types can't prove

Lessons learned

Types save me from having to even think about certain categories of tests

Tests help me out when I get stuck but I mostly run them in the REPL and **delete**

Code is structured differently with REPL

Most modern languages don't have a strong enough type system to make illegal states un-representable

It's easy to get lost in a space where you never deliver

Test verify when types can't prove

Paul's turn

The evil!

Paul's approach

- Scala
- Tests without types
- Property-based testing
- Types
- Delete some tests

Interesting code

```
def makeLegit: String = {
  val result = nextInt(9000000) + 1000000

  val guess = checksum(result.toString + "00")
  val goal = (guess / 11 + 1) * 11
  val diff = goal - guess
  val quotient = diff / 2
  val remainder = diff % 2

  val answer = (result * 100) + (quotient * 10) + remainder
  answer.toString
}

def legit: Gen[String] = Gen(_ => Some(makeLegit))

"All legitimate OCR scans" should {
  "evaluate to their unique value" in {
    forAll(legit) { (v: String) => evaluate(digitsToScan(v)) must_== v }
  }
}
```

Interesting code—h/t Travis Brown

```
import shapeless._
import Nat._
import HList._

// Evidence that the sum of (each item in L multiplied by (its distance from
// the end of the list plus one) modulo eleven) is S.
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]
) {}

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

isValid(_1 :: _2 :: _3 :: _4 :: _5 :: _6 :: _7 :: _8 :: _9 :: 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)
```

Interesting code—h/t Travis Brown

```
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] {}

 def isValid[L <: HList](l: L)(implicit
   len: LengthAux[L, _9],
   hcs: HasChecksum[L, _0]
 ) {}

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

 isValid(_1 :: _2 :: _3 :: _4 :: _5 :: _6 :: _7 :: _8 :: _9 :: 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)
```

Interesting code—h/t Travis Brown

```
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] {}

 def isValid[L <: HList](l: L)(implicit
   len: LengthAux[L, _9],
   hcs: HasChecksum[L, _0]
 ) {}

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

  isValid(_1 :: _2 :: _3 :: _4 :: _5 :: _6 :: _7 :: _8 :: _9 ::
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)
```

Tim Sweeney, CEO, Epic Games

```
Program Fixpoint quicksort (l : list t) {measure length l} :
    { l' : list t | sort le l' /\ permutation l l'}
```

"True dependent types would be preferable to the solution that has evolved in both Haskell and C++, where the type level is Turing-complete yet remains a separate and bizarre computational realm where you can't directly reason about numbers and strings but need to reconstruct them from inductive data types at terrible cost in complexity."

Lessons learned

Even for such a small problem, spelling out unit tests made me want to gouge out my eyeballs with a rusty spoon.

When developing property-based tests, every forAll made me think “could/should that be a type?”

For some use-cases, having examples of correct input/output gave no real guidance whatsoever.

Test code is still code, with its own correctness, maintenance, etc. burden.

Lessons learned

Even for such a small problem, spelling out unit tests made me want to gouge out my eyeballs with a rusty spoon.

When developing property-based tests, every forAll made me think "could/should that be a type?"

For some use-cases, having examples of correct input/output gave no real guidance whatsoever.

Test code is still code, with its own correctness, maintenance, etc. burden.

Lessons learned

Even for such a small problem, spelling out unit tests made me want to gouge out my eyeballs with a rusty spoon.

When developing property-based tests, every forAll made me think “could/should that be a type?”

For some use-cases, having examples of correct input/output gave no real guidance whatsoever.

Test code is still code, with its own correctness, maintenance, etc. burden.

Lessons learned

Even for such a small problem, spelling out unit tests made me want to gouge out my eyeballs with a rusty spoon.

When developing property-based tests, every forAll made me think “could/should that be a type?”

For some use-cases, having examples of correct input/output gave no real guidance whatsoever.

Test code is still code, with its own correctness, maintenance, etc. burden.

Our discussion throughout

Small codebase = little value for type system

Types scale better than tests

Types have little value when talking with non technical end users

The hardest part is understanding the requirements

We rarely have the luxury of sample input/output

Tests can be good for forming ideas but then can be deleted

Our discussion throughout

Small codebase = little value for type system

Types scale better than tests

Types have little value when talking with non technical end users

The hardest part is understanding the requirements

We rarely have the luxury of sample input/output

Tests can be good for forming ideas but then can be deleted

Our discussion throughout

Small codebase = little value for type system

Types scale better than tests

Types have little value when talking with non technical end users

The hardest part is understanding the requirements

We rarely have the luxury of sample input/output

Tests can be good for forming ideas but then can be deleted

Our discussion throughout

Small codebase = little value for type system

Types scale better than tests

Types have little value when talking with non technical end users

The hardest part is understanding the requirements

We rarely have the luxury of sample input/output

Tests can be good for forming ideas but then can be deleted

Our discussion throughout

Small codebase = little value for type system

Types scale better than tests

Types have little value when talking with non technical end users

The hardest part is understanding the requirements

We rarely have the luxury of sample input/output

Tests can be good for forming ideas but then can be deleted

Our discussion throughout

Small codebase = little value for type system

Types scale better than tests

Types have little value when talking with non technical end users

The hardest part is understanding the requirements

We rarely have the luxury of sample input/output

Tests can be good for forming ideas but then can be deleted

I considered making it a type of it's own, but it makes the rest of the code ugly…

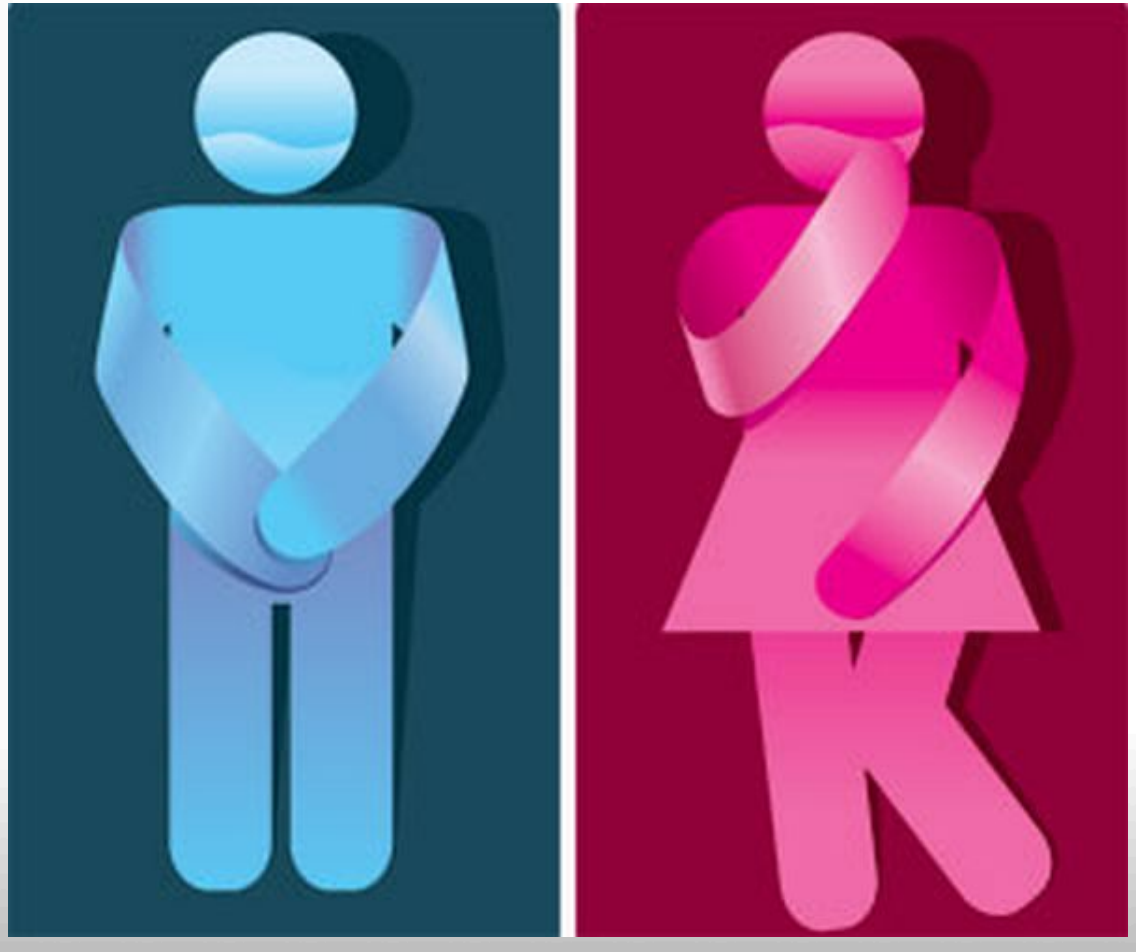
I considered making it a type of it's own, but it makes the rest of the code ugly…

Context matters. One reason I like the notion of "code smell" is indeed that code is a language, and things either read well or they don't, and if they don't, it's telling us something.

If you're going to use types to make distinctions between states you end up with types with only one inhabitant, that's OK

English is a piss-poor specification language, but there's got to be a better way than slavishly enumerating input/output examples, which isn't realistic in any scenario of any combinatorial complexity, either. Where that puts me is very firmly in a "property-based test, then translate to types as reasonable"

I want a type for gender



I want a test for pregnancy

Final Thoughts

Tests = There exists

Types = For all


PROFILEBRAND.COM

Spectrum

Considerations

- Do we need to think about codebase scale?
- How long will this code be in production?
- Business value of the code
- Documentation
- You don't have to be a type system genius to get some major value
- There are an abundance of languages to suit both needs
- LOGIC!

Find us for follow-up debate

Paul Snively

@psnively

psnively@me.com

Amanda Laucher

@pandamonial

pcprogrammer@gmail.com

Trolling

prime -> prime -> int

Aaron and Daniel

String -> [String] -> [[String]] -> [String] -> String

Michael