Mathematics

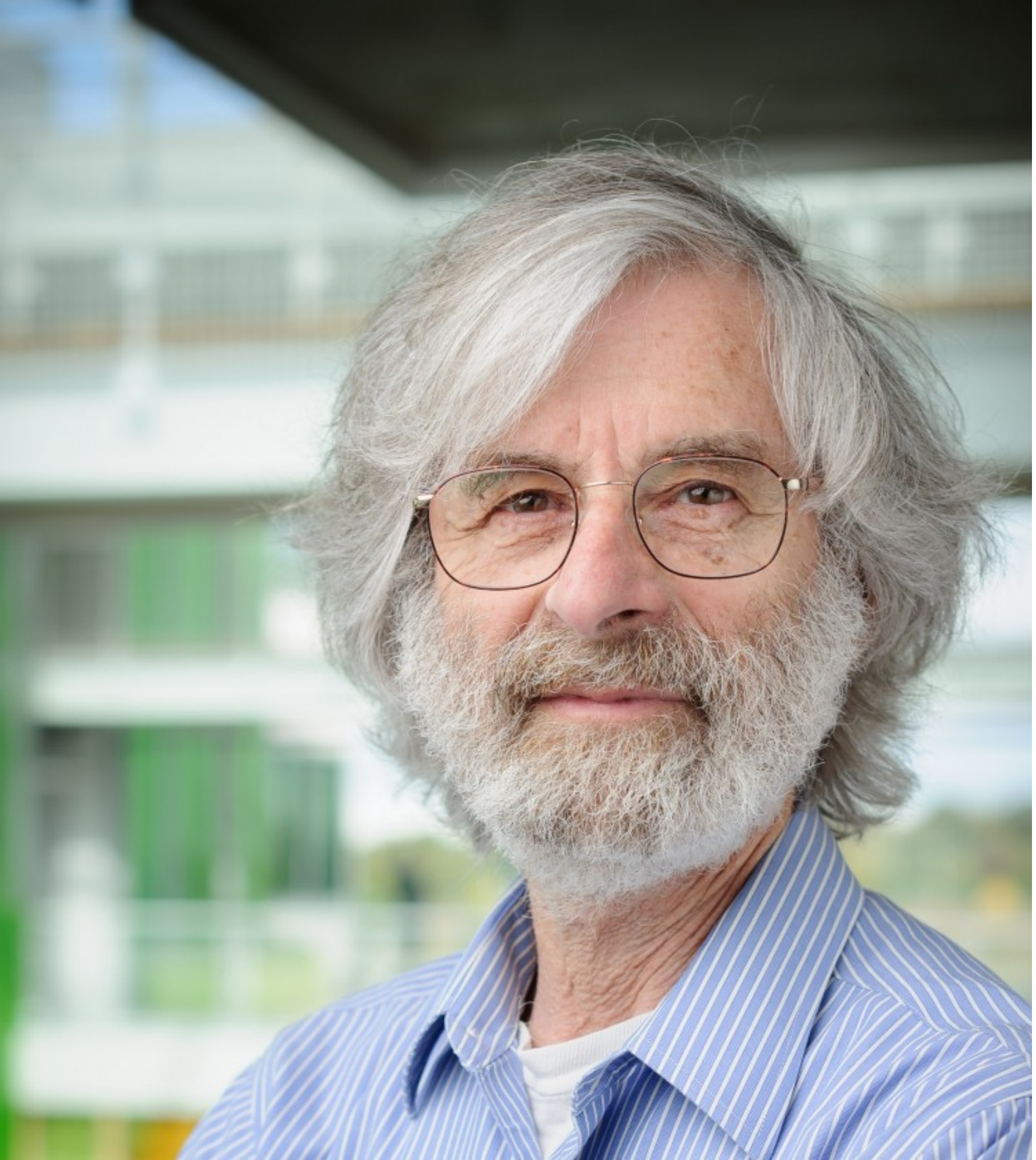
The missing ingredient in modern software development

Rahul Goma Phulore (@missingfaktor)
2014.11.22
XConf Bengaluru, 2014

Software Development Today

- "Agile"
- "TDD"
- "Object oriented"
- "Composition over inheritance"
- "Too many private methods is a smell"

Throw away your buzzwords and blanket statements, and use some math!

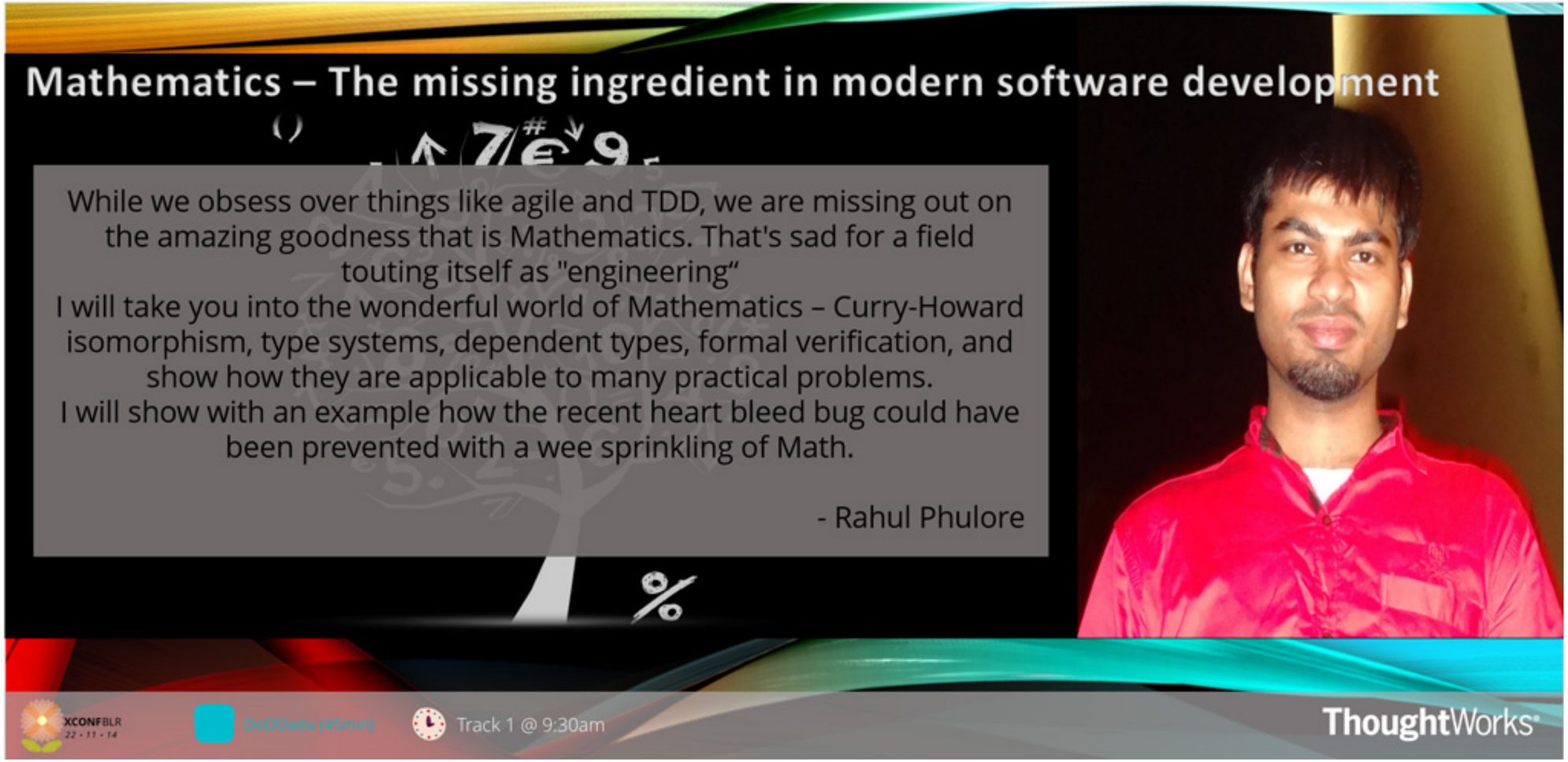


There's something about the culture of software that has impeded the use of specification. We have a wonderful way of describing things precisely that's been developed over the last couple of millennia, called mathematics. I think that's what we should be using as a way of thinking about what we build.

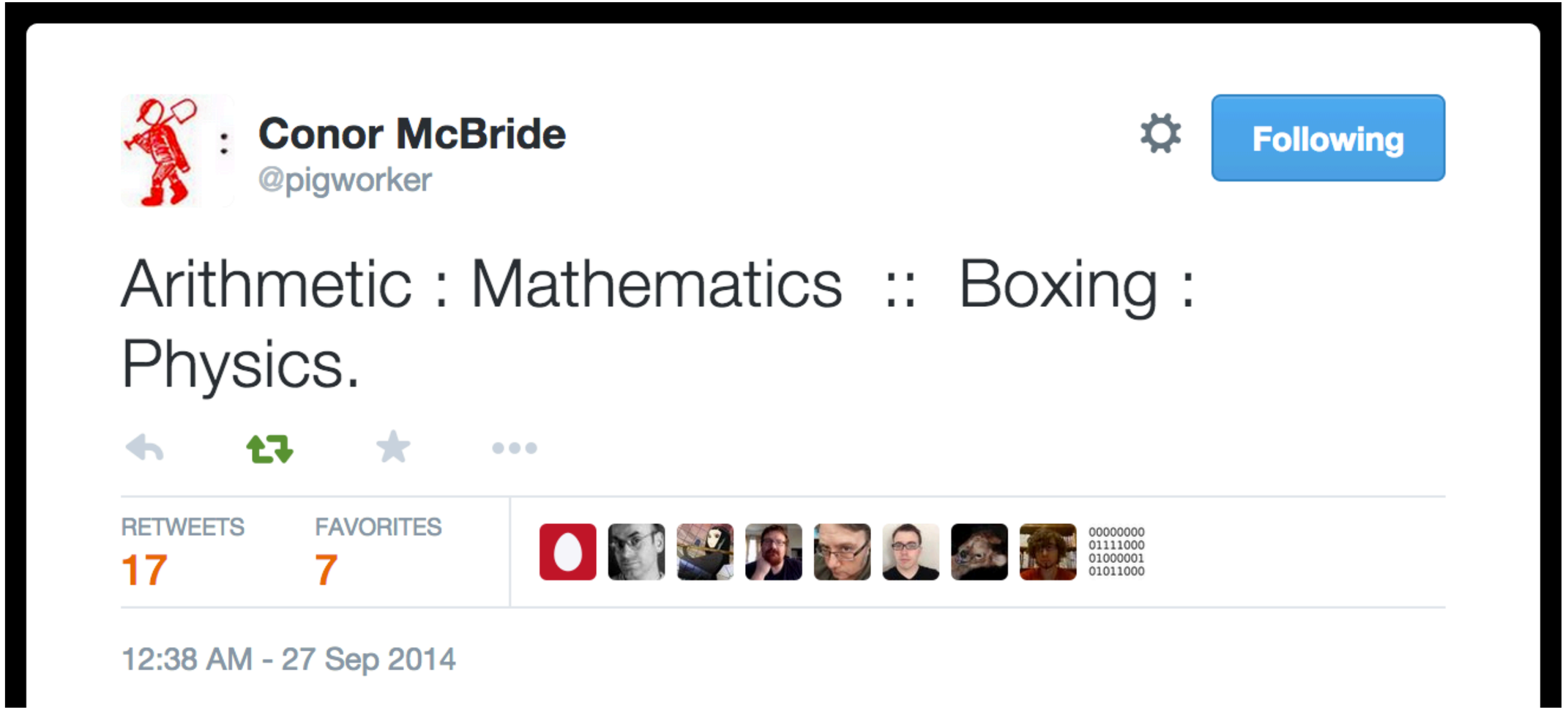
-- Leslie Lamport

What comes to your mind when I say the word "mathematics"?

XConf Poster (See those numbers on the tree?)



But...


Conor McBride
@pigworker
Following
Arithmetic : Mathematics :: Boxing : Physics.
RETWEETS
17
FAVORITES
7
00000000
01111000
01000001
01011000
12:38 AM - 27 Sep 2014

math·e·mat·ics (ˌmæθəˈmætɪks)

— **Mathematicians seek out patterns and use them to formulate new conjectures.**

math·e·mat·ics (ˌmæθəˈmætɪks)

— **Mathematicians seek out patterns and use them to formulate new conjectures.**
— **Mathematicians resolve the truth or falsity of conjectures by mathematical proof.**

math·e·mat·ics (ˌmæθəˈmætɪks)

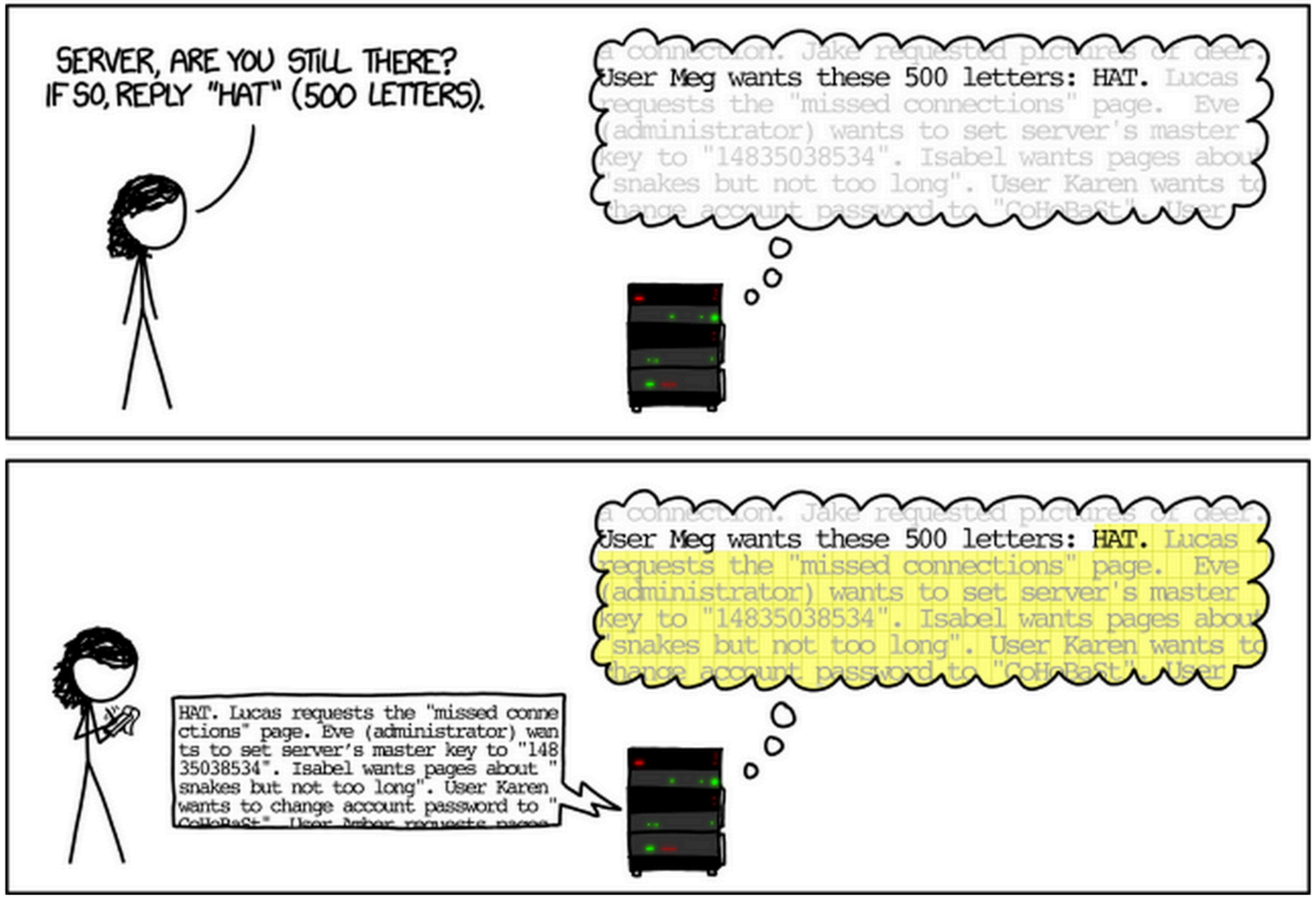
— **Mathematicians seek out patterns and use them to formulate new conjectures.**
— **Mathematicians resolve the truth or falsity of conjectures by mathematical proof.**
— **When mathematical structures are good models of real phenomena, then mathematical reasoning can provide insight or predictions about nature.**

A Motivating Example

OpenSSL Heartbleed

Even these rookie crackers exploited it!

OpenSSL Heartbleed



Language used for the example: ATS

Why do we test our software?

We test because:

— **We want our software to be reliable and correct.**

We test because:

— We want our software to be reliable and correct.
— Correctness matters.

We test because:

— We want our software to be reliable and correct.
— Correctness matters.
— It's about basic professionalism.

What if I told you that there are better ways to verify your software?

Formal Methods

In computer science, formal methods are a particular kind of mathematically based techniques for the specification, development and verification of software and hardware systems.

Type Systems

A type system is a syntactic method for automatically checking the absence of certain erroneous behaviors by classifying program phrases according to the kinds of values they compute.

What types are not

- Classes
- Runtime tags

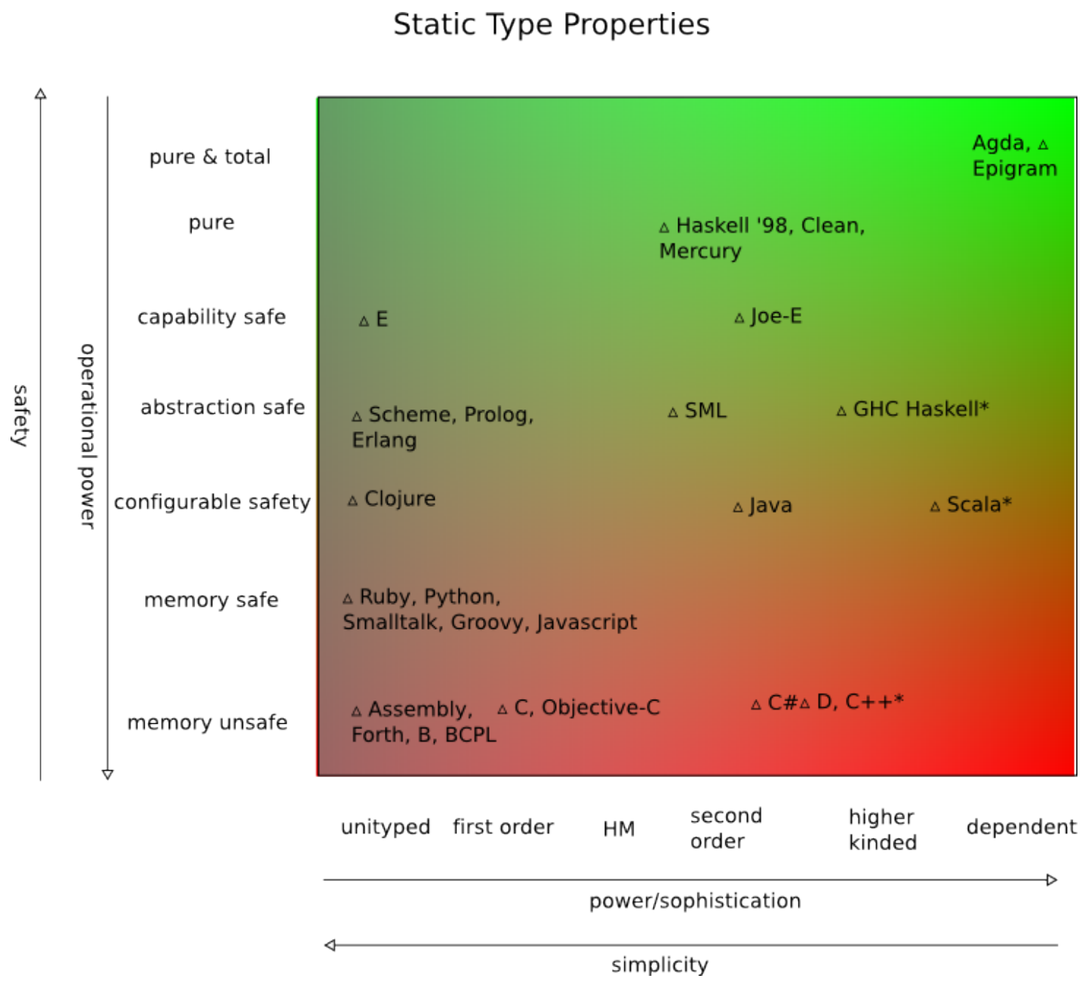
What types are

- Types categorize terms, based on their properties and the kind of operations they can support
- Types encode invariants/knowledge about your program
- Intuition: sets

Curry-Howard Correspondence

- **Types ≅ Propositions**
- **Programs ≅ Proofs**

Not all type systems are created equal



Java ≈ Bat

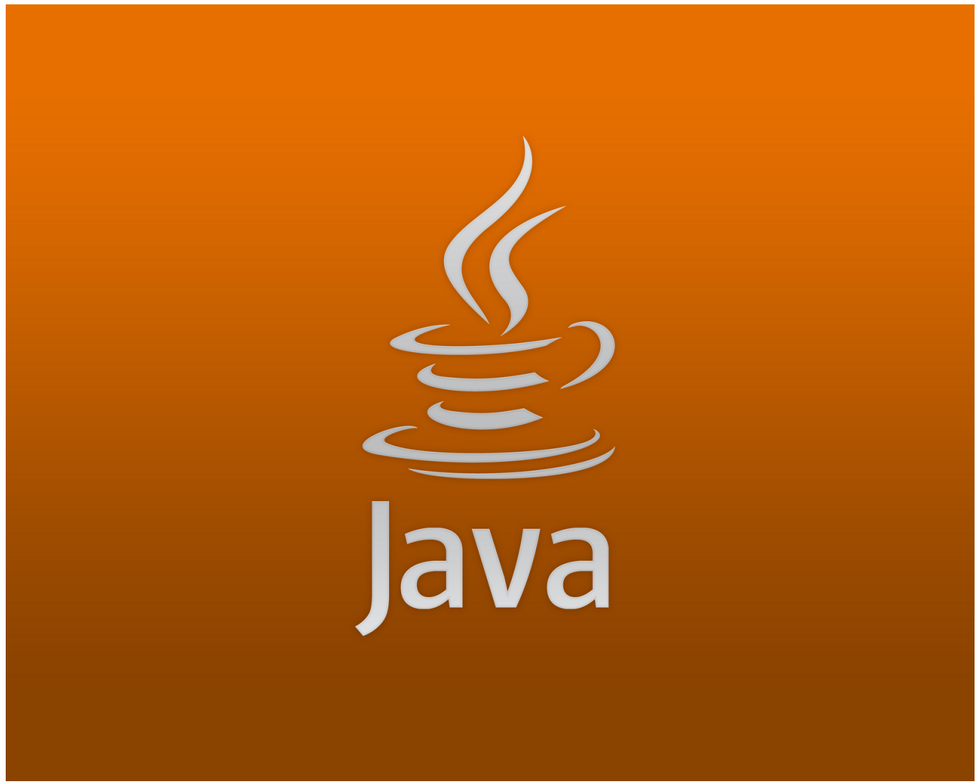


Exhibit A

NullPointerException!

NullPointerException - Diagnosis

$$null \in A$$

NullPointerException - Diagnosis

```
f :: A -> B
```

- **f can accept a non-null A and return a non-null B.**
- **f can accept a null A and return a non-null B.**
- **f can accept a non-null A and return null B.**
- **f can accept a null A and return a null B.**

NullPointerException - Diagnosis

- **Check every single reference for `null`? Madness!**
- **Have to rely on documentation or tribal knowledge.**

NullPointerException - Broken solutions

- Null object pattern
- Elvis operator (`?:`)
- Safe navigation operator (`?.`)

NullPointerException - Broken solutions

Michael Feathers

« Guiding Software Development with Design Challenges | Main | Flipping Assumptions with 'Programmer Anarc

June 17, 2013

Avoid Null Checks by Replacing Finders with Tellers

One of my pet peeves in programming is null checks. Many codebases are littered with them and, as a result they often very hard to understand.

It's easy enough to complain about null checks, but it's harder to root out all of the places they occur and find alternatives. The typical advice is to move your code toward using the null object pattern or to just check for nu immediately and make sure that you don't pass nulls along in your program. After all, we do often need to retriev objects and sometimes they aren't there

NullPointerException - Let's apply some math!

— **Ditch** `null`.

NullPointerException - Let's apply some math!

— **Capture the algebra of "nullity".**
— **Algebraic data types.**

```
data Option a = Some a
              | None

-- inhabitants(Option a) = inhabitants(a) + 1
```

NullPointerException - Let's apply some math!

```
f :: A -> B
```

— `f` **can accept an** `A` **and return a** `B`**.**

NullPointerException - Let's apply some math!

```
f :: A -> Option B
```

— `f` can accept an `A` and *optionally* return a `B`.

NullPointerException - Let's apply some math!

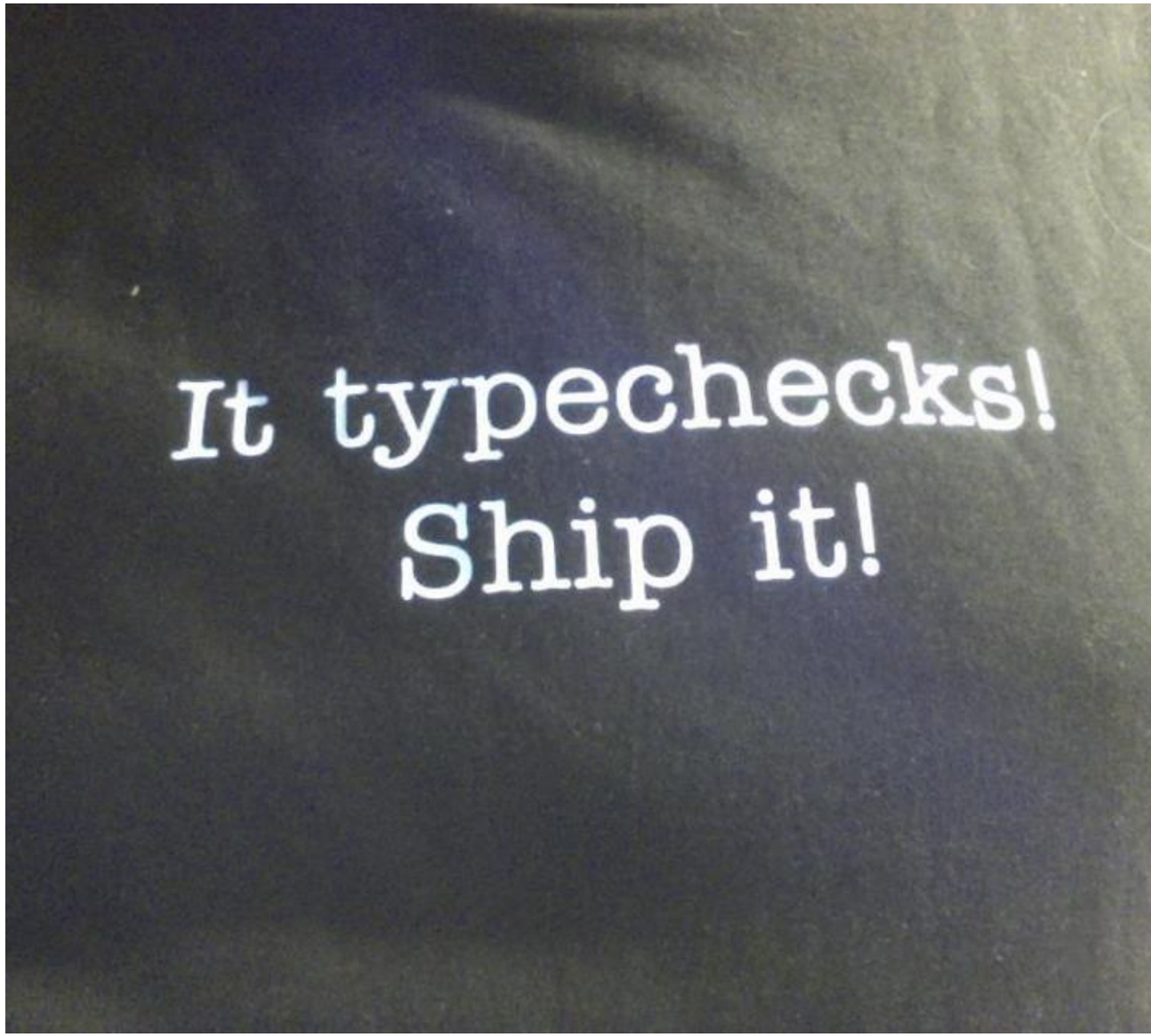
— **No way to mistake an `A` for an `Option A`!**

Algebraic data types + First class functions

- Help in capturing constraints precisely.
- Constraints propagate through program.
- Patterns emerge easily.
- Lead to algebraic patterns such as functors, applicatives, monads.
- Better composability. Extremely high degree of code reuse.

We have barely scratched the surface!

- Dependent types
- Theorem provers
- Substructural types
- Effects
- Coeffects
- Region systems
- Sequent calculus

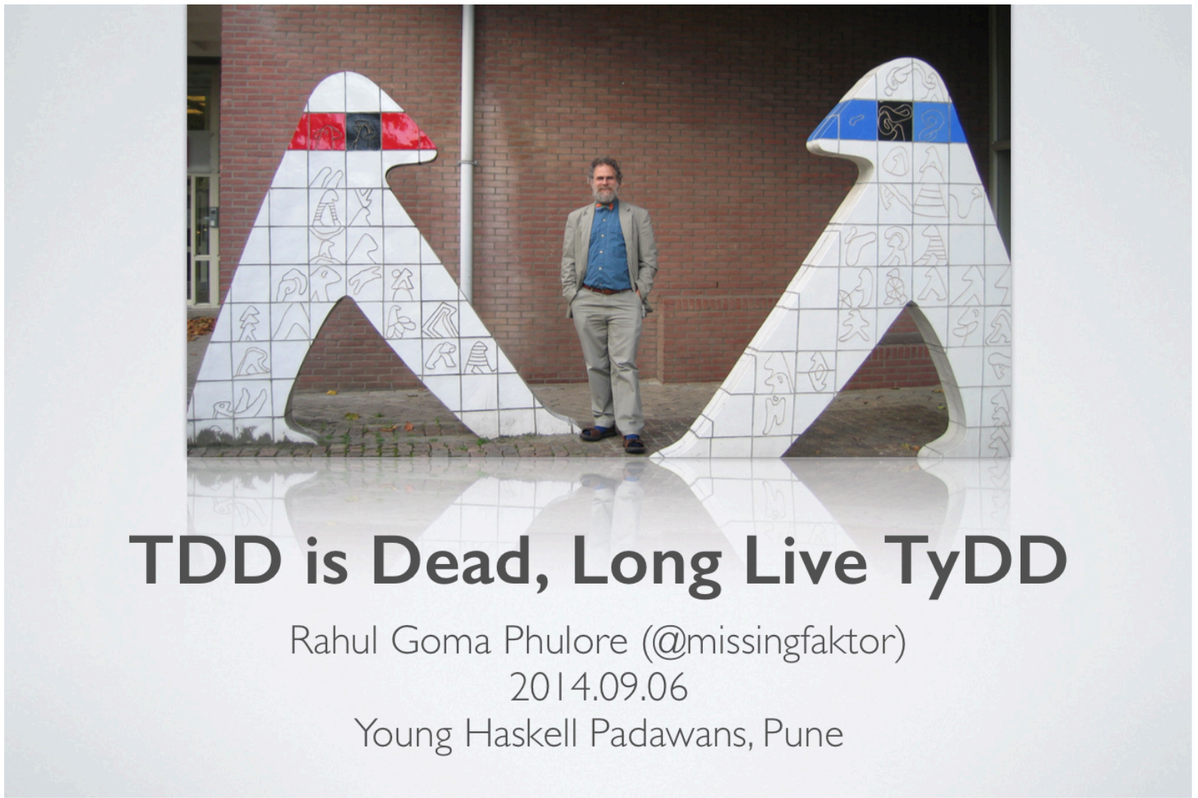


When you don't have Math vs When you have Math

- Callbacks - Monadic futures
- Go error handling - Monadic error handling (Either)
- AspectJ - Higher order functions and combinators
- Spring DI - ML modules, Reader
- Polymer - Elm

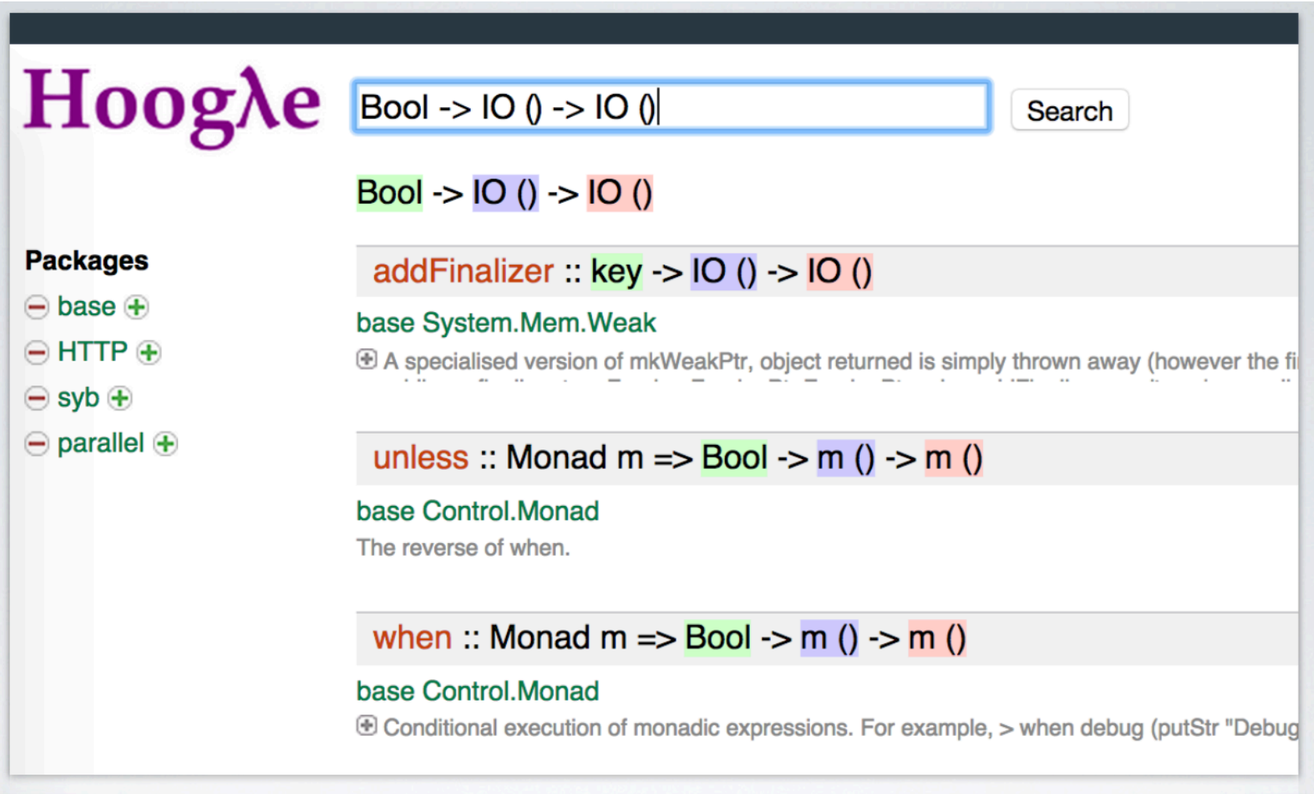
Other Benefits

Excellent design tool (See my TyDD presentation)



Other Benefits

Discoverability - Hoogle



Other Benefits

Equational reasoning

```
replicate (m * n) x

-- Assume: n = 1 + n'
= replicate (m * (1 + n')) x

-- m * (1 + n') = m + m * n'
= replicate (m + m * n') x

-- replicate distributes over addition
= replicate m x ++ replicate (m * n') x

-- Induction: reuse the premise
= replicate m x ++ concatMap (replicate m) (replicate n' x)

-- Definition of `concatMap`
= replicate m x ++ foldr ((++) . replicate m) [] (replicate n' x)

-- Definition of `foldr`, in reverse
= foldr ((++) . replicate m)) [] (x:replicate n' x)

-- Definition of `concatMap`, in reverse
= concatMap (replicate m) (x:replicate n' x)

-- Definition of `replicate`
```

Other Benefits

Amazingly capable tools

Code editing, IDEs, and type systems

Code editing will be done in structural editors, which will look nothing like the existing batch of IDEs that are little more than glorified text editors (and they are actually rather poor text editors). In a structural editor, the programmer will construct expressions which may have holes in them not yet filled with terms. Importantly, these structural editors will be type-directed, so for any given hole the programmer can be presented with set of values matching the expected type, ordered in some sensible way. The editor will perform local program search to enable autocompleting of multiple expressions. If you've ever seen someone live-code Agda, you'll know how powerful and productive this idea could be. Yeah, the actual interface for programming Agda is still kind of 1970s (a custom Emacs mode), but the idea of type-directed editors is powerful. It makes it clear that types are effective not just at preventing many software errors, but also in *guiding development*. They are a powerful tool to augment our puny programming brains.

Applications

Reactive Extensions (Rx)

Applications

Games

Applications

Systems programming

Applications

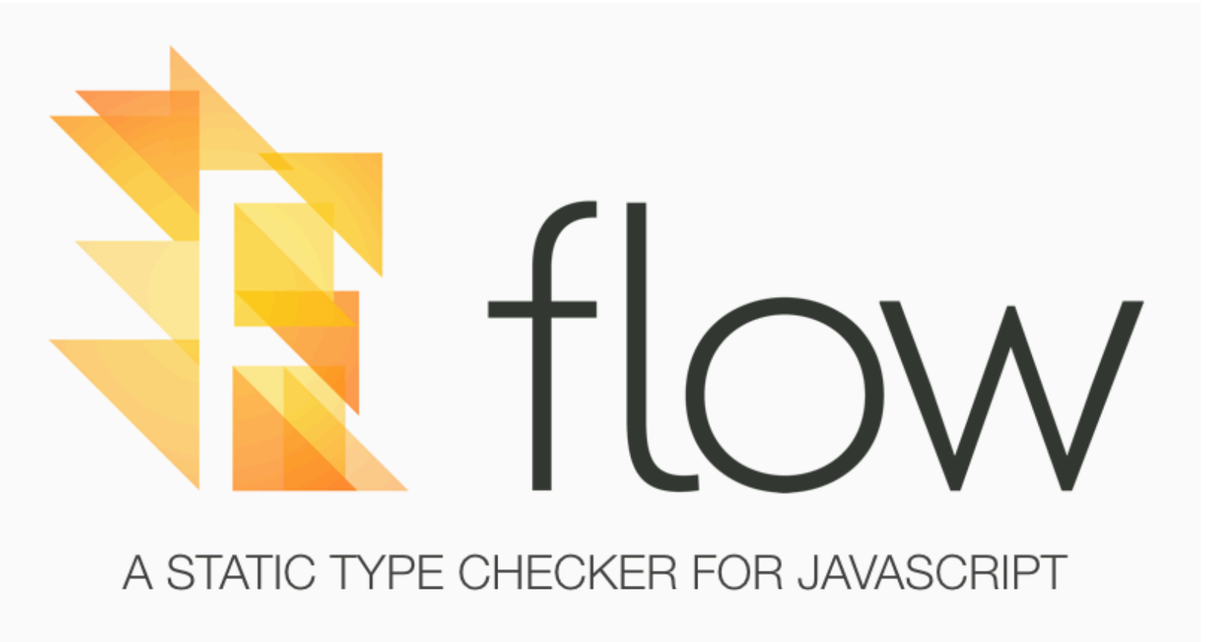
Functional reactive programming (FRP)

Applications

Formally verified OS kernel - seL4

Applications

Facebook's Hack and FlowType

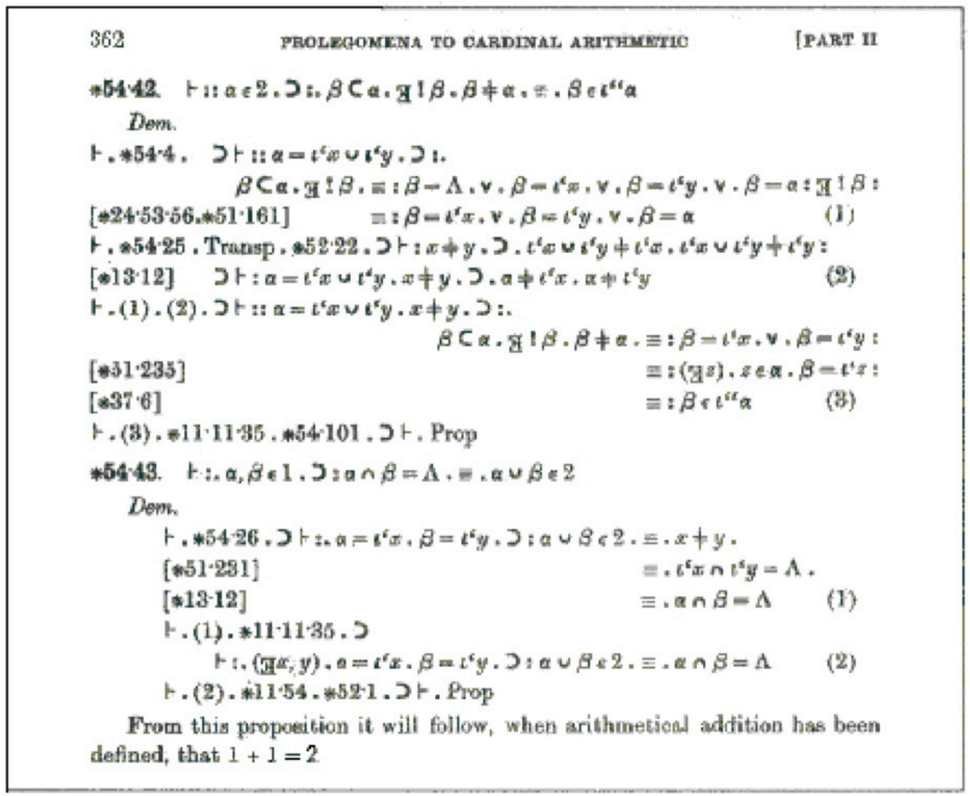


Does this replace testing?

No.

Does this replace testing?

1 + 1 = 2 proof



362 PROLEGOMENA TO CARDINAL ARITHMETIC [PART II

*54·42. $\vdash :: \alpha \in 2 . \supset :. \beta \subset \alpha . \exists ! \beta . \beta \neq \alpha . \equiv . \beta \in \iota''\alpha$

Dem.

$\vdash .$ *54·4. $\supset \vdash :: \alpha = \iota'x \cup \iota'y . \supset :.$

$\beta \subset \alpha . \exists ! \beta . \equiv : \beta = \Lambda . \vee . \beta = \iota'x . \vee . \beta = \iota'y . \vee . \beta = \alpha : \exists ! \beta :$

[*24·53·56.*51·161] $\equiv : \beta = \iota'x . \vee . \beta = \iota'y . \vee . \beta = \alpha$ (1)

$\vdash .$ *54·25. Transp. *52·22. $\supset \vdash : x \neq y . \supset . \iota'x \cup \iota'y \neq \iota'x . \iota'x \cup \iota'y \neq \iota'y :$

[*13·12] $\supset \vdash : \alpha = \iota'x \cup \iota'y . x \neq y . \supset . \alpha \neq \iota'x . \alpha \neq \iota'y$ (2)

$\vdash .(1).(2). \supset \vdash :: \alpha = \iota'x \cup \iota'y . x \neq y . \supset :.$

$\beta \subset \alpha . \exists ! \beta . \beta \neq \alpha . \equiv : \beta = \iota'x . \vee . \beta = \iota'y :$

[*31·235] $\equiv : (\exists z) . z \in \alpha . \beta = \iota'x :$

[*37·6] $\equiv : \beta \in \iota''\alpha$ (3)

$\vdash .(3). *11·11·35 . *54·101 . \supset \vdash .$ Prop

*54·43. $\vdash :. \alpha, \beta \in 1 . \supset : \alpha \cap \beta = \Lambda . \equiv . \alpha \cup \beta \in 2$

Dem.

$\vdash .$ *54·26. $\supset \vdash :. \alpha = \iota'x . \beta = \iota'y . \supset : \alpha \cup \beta \in 2 . \equiv . x \neq y .$

[*51·231] $\equiv . \iota'x \cap \iota'y = \Lambda .$

[*13·12] $\equiv . \alpha \cap \beta = \Lambda$ (1)

$\vdash .(1). *11·11·35 . \supset$

$\vdash :. (\exists x, y) . \alpha = \iota'x . \beta = \iota'y . \supset : \alpha \cup \beta \in 2 . \equiv . \alpha \cap \beta = \Lambda$ (2)

$\vdash .(2). *11·54 . *52·1 . \supset \vdash .$ Prop

From this proposition it will follow, when arithmetical addition has been defined, that $1 + 1 = 2$.

Does this replace testing?

- Formalization has a cost; not always justified. You need to draw a line.
- You can get a lot of mileage from the type systems of Scala, Haskell, F#. And even C# and Java.
- Tests to cover what hasn't been formally verified.

Thank You!


Are you too busy to improve?
No thanks!
We are too busy
Håkan Forss @hakanforss http://hakanforss.wordpress.com
This illustration is inspired by and in part derived from the work by Scott Simmerman, "The Square Wheels Guy" http://www.performancemanagementcompany.com/

References

— See the source markdown.

Credits

— Many giants on whose shoulders us mortals stand.
— Rahul Kavale, Shripad Agashe, and Priti Biyani for ideas and early feedback.