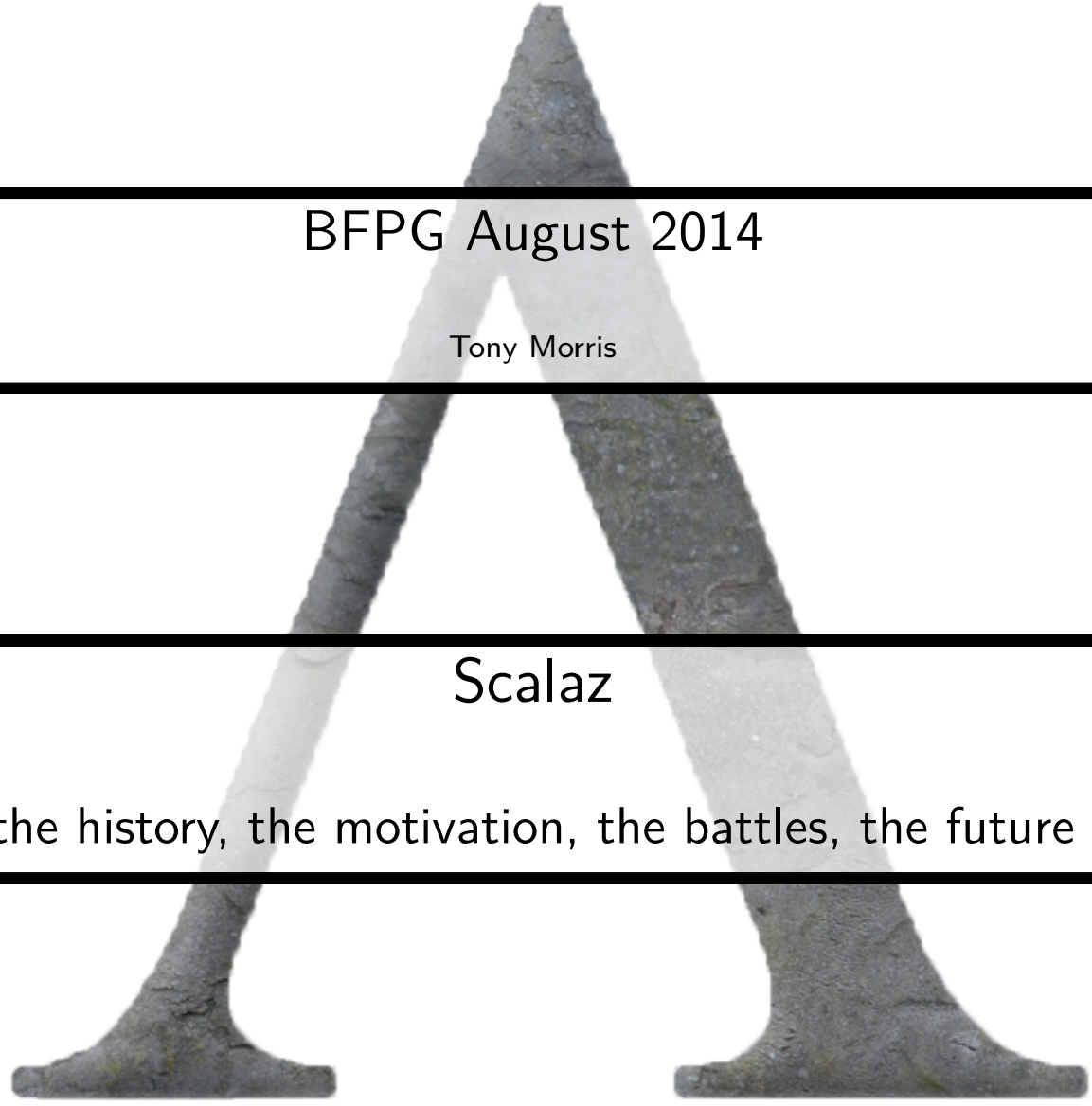


BFPG August 2014

Tony Morris

Scalaz

the history, the motivation, the battles, the future

From which ivory tower did this nonsense come from?

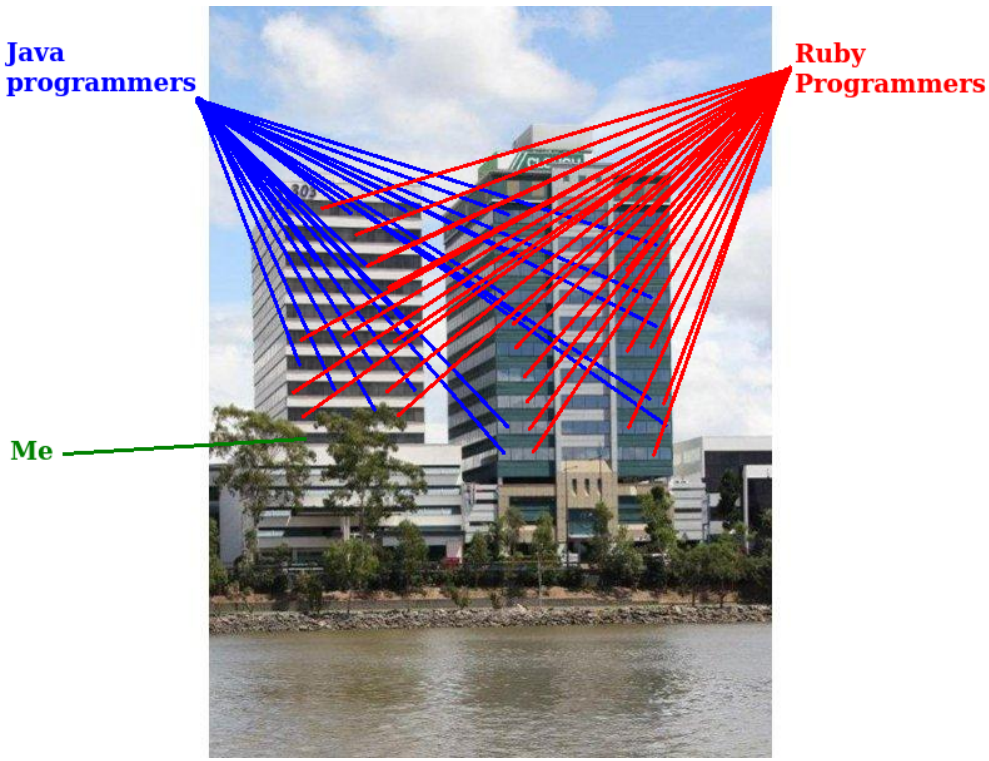
Scalaz started in the corner of a room in Milton in 2007.

303 Coronation Drive, Milton, Queensland, Australia

NICTA

From which ivory tower did this nonsense come from?

Scalaz started in the corner of a room in Milton in 2007.



303 Coronation Drive, Milton, Queensland, Australia

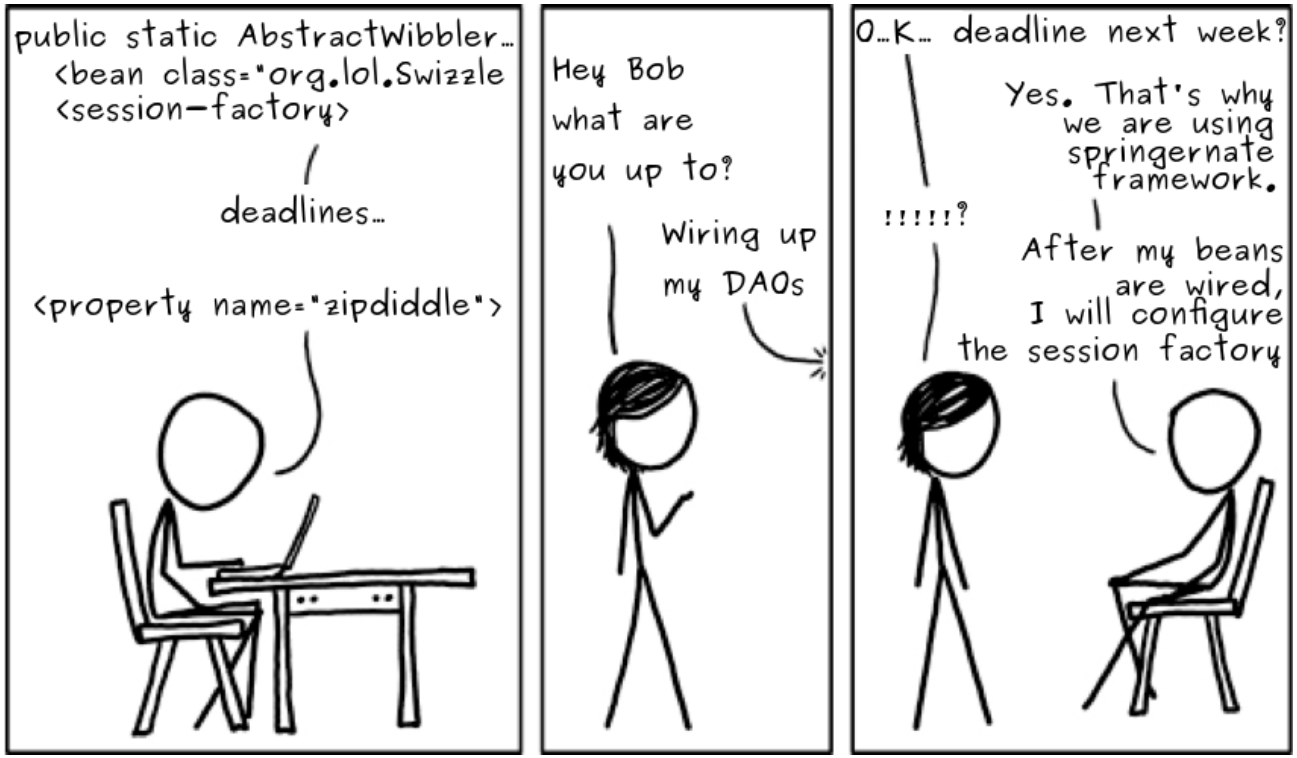
NICTA

From which ivory tower did this nonsense come from?

- Working for a small Java-based consultancy
- Short deadlines, demanding quality
- Excellent people to work with . . .

From which ivory tower did this nonsense come from?

. . . but they started saying strange things at my head



NICTA

From which ivory tower did this nonsense come from?

. . . all while holding a serious face

and so I decided . . .

NICTA

From which ivory tower did this nonsense come from?

. . . I didn't want to end up like this

∴ Scalaz

and so Scalaz came to exist

The ultimate goal

- reason about code to know what it does
- incrementally improve code
- everything it takes to hit the goal **and nothing more**

The ultimate goal

This means we must

- exploit existing ideas that are known to work
- efficiently reject ideas that are known to not work
- explore new ideas for which there is no known answer

Readabillaty Factory

To assist in code readability

- we count the number of characters in our source code
- we look up the meaning of our identifier names in a dictionary
- we use the latest test tools to report the number of pixels in our critical source code

Careful analysis

Diagram of typical whiteboard session

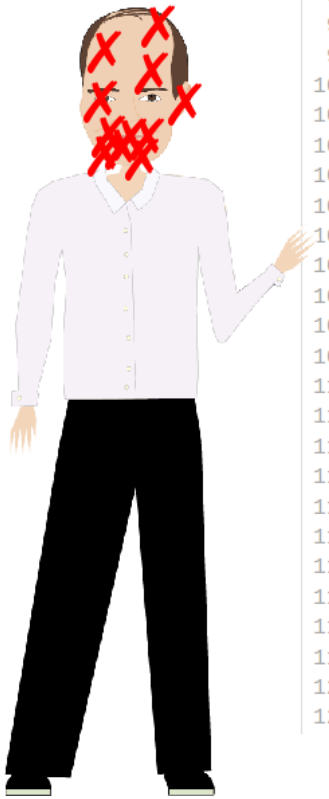
```
97
98  public abstract class AbstractTypeHierarchyTraversingFilter
99        implements TypeFilter {
100   private final boolean considerInherited;
101   private final boolean considerInterfaces;
102
103   @Override
104   public boolean match(MetadataReader metadataReader,
105                        MetadataReaderFactory metadataReaderFactory)
106
107 public class AspectJTypeFilter implements TypeFilter {
108   private final World world;
109   private final TypePattern typePattern;
110
111 public interface MetadataReaderFactory
112
113 public class SimpleMetadataReaderFactory implements MetadataReaderFactory {
114   private final ResourceLoader resourceLoader;
115
116   @Override
117   public MetadataReader getMetadataReader(String className) throws IOException {
118     String resourcePath = ResourceLoader.CLASSPATH_URL_PREFIX +
119         ClassUtils.convertClassNameToResourcePath(className) + ClassUtils.CLASS_F
120     return getMetadataReader(this.resourceLoader.getResource(resourcePath));
121   }
```

Careful selection of identifier names, ensuring a correspondence to the real world.

NICTA

Reject the dunce cap

Just kidding!



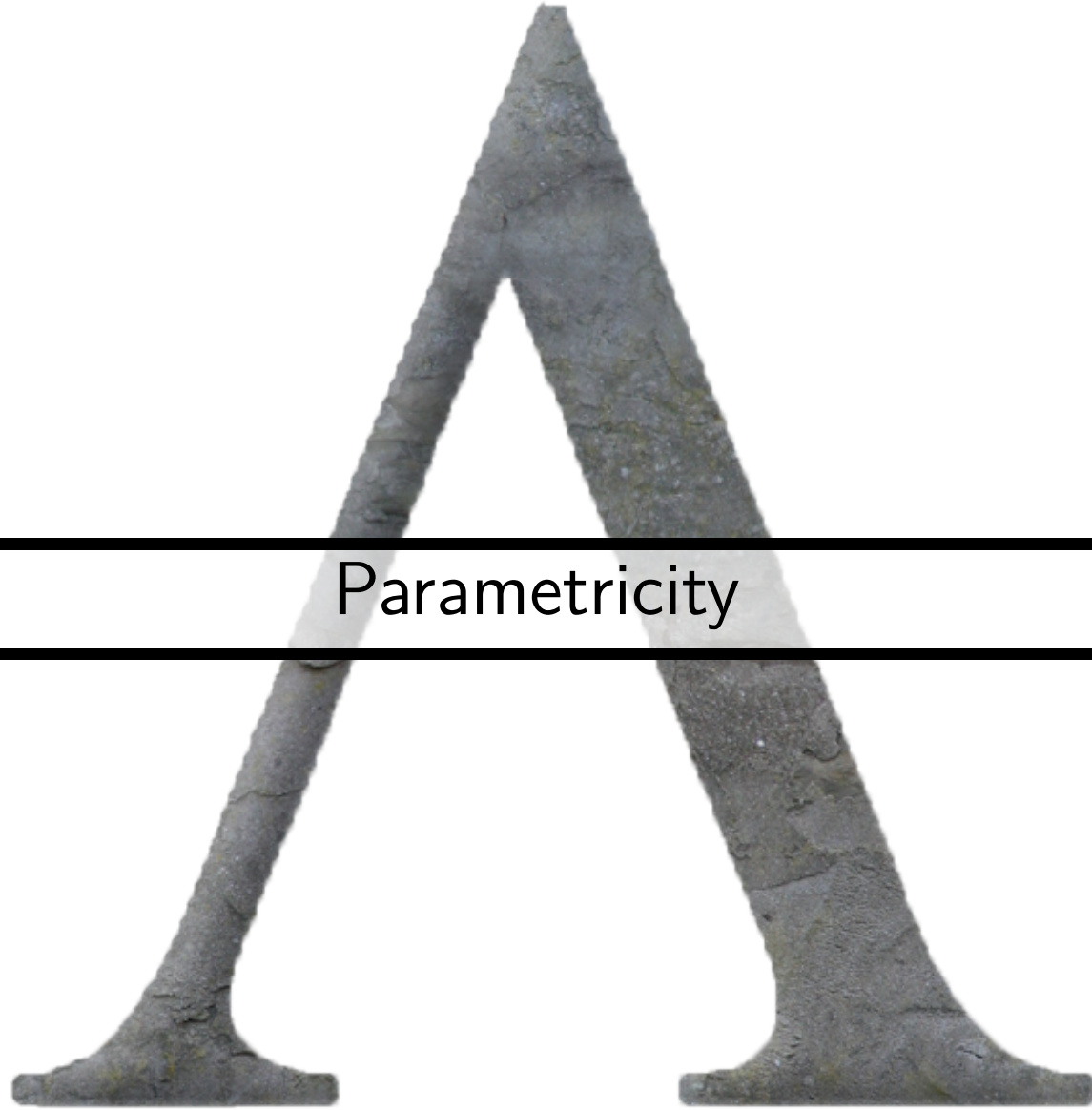
```
97
98  public abstract class AbstractTypeHierarchyTraversingFilter
99        implements TypeFilter {
100   private final boolean considerInherited
101   private final boolo    siderInterfa
102
103   @Override
104   public boolean match     aRe     dataReader,
105                                    tory metadataReaderFactory)
106
107 public class AspectJTypeF          ents TypeFilter {
108   private final World wor
109   private final TypePatte            ern;
110
111 public interface Meta
112
113 public class Simpl          ead          implements MetadataReaderFactory {
114   private final R          ader re          ader;
115
116   @Override
117   public Met          r getMetadataRe          r(String className) throws IOException {
118     String          ath = ResourceLoader.CLASSPATH_URL_PREFIX +
119         Cla          convertClassNameToResourcePath(className) + ClassUtils.CLASS_F
120     return          dataReader(this.resourceLoader.getResource(resourcePath));
121   }
```

Crass bullshizzles is rejected from the comfort of an ivory tower.

Principles

Scalaz explores three principles

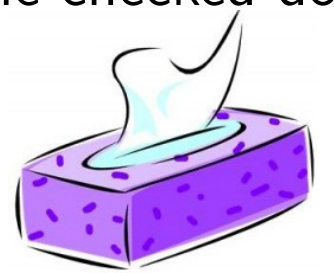
1. **parametricity** or **free theorems**
   parametric generalisation to eliminate candidate implementations of a type and construct theorems (documentation) based on type
2. **equational reasoning**
   construct large programs from smaller programs and vice versa
3. **abstraction**
   discard expenditure of repetitious effort



Parametricity

Parametricity
Types are Documentation

Using a technique described by Philip Wadler[Wad89], we can obtain machine-checked documentation.



Let's use an example to progressively exploit this technique.

Parametricity
Types are Documentation

```
def value(
  f: List[Int => String]
, x: Int
): List[String] =
  ...
```

- what does this function `value` do?
- how many of the resulting strings come from application of the list of functions?
- are some of those strings created without using the list of functions?
- if so, what are their values?
- do we apply those functions to a different integer to the one given?
- are the arguments used at all, or maybe just one of them?

At present, there is no way for us to answer these questions

The **only** thing we know is that it returns one of many possible list of strings

Parametricity
Types are Documentation

```
def value[A, B](
  f: List[A => B]
, x: A
): List[B] =
  ...
```

By using type parameters, we know some things

- every `B` in the result came from application of one and only one of the given functions
- (it follows that) if the resulting list is empty, the input list is empty
- for a given function application (to arrive at `B`), we applied the given `A`
- either this function uses its list argument or it returns `Nil`

But we still have questions

- what arrangement of application of the list of functions occurs?
- do we use each function a specific number of times?
- do we use the arguments at all (we know that if not, we always get `Nil`)?

But we still have questions

- what arrangement of application of the list of functions occurs?
- do we use each function a specific number of times?
- do we use the arguments at all (we know that if not, we always get `Nil`)?

But we still have questions

- what arrangement of application of the list of functions occurs?
- do we use each function a specific number of times?
- do we use the arguments at all (we know that if not, we always get `Nil`)?

Parametricity
Types are Documentation

```
def value[F[_]: Functor, A, B](
  f: F[A => B]
, x: A
): F[B] =
  ...
```

where Functor is given by

```
trait Functor[F[_]] {
  def map[A, B](f: A => B): F[A] => F[B]
}
```

NICTA

Parametricity
Types are Documentation

```
def value[F[_]: Functor, A, B](
  f: F[A => B]
, x: A
): F[B] =
  ...
```

Everything we need and nothing more

This function maps application to the given `A` across every function in the context of `F`

Parametricity
Types are Documentation

Importantly

- I can *prove this by parametricity*.
- $\therefore$ if two voices disagree what this function does, at least one of them is wrong.

Parametricity
Types are Documentation

Importantly

- I can *prove this by parametricity*.
- $\therefore$ if two voices disagree what this function does, at least one of them is wrong.

Parametricity
Types are Documentation

I don't need to . . .

- ask Fred what his function does
- inspect the source code under some pretention of "readability"
- perform magic woo-woo hocus-pocus pompous drivel with identifier names

Parametricity
Types are Documentation

I don't need to . . .

- ask Fred what his function does
- inspect the source code under some pretention of "readability"
- perform magic woo-woo hocus-pocus pompous drivel with identifier names

I don't need to . . .

- ask Fred what his function does
- inspect the source code under some pretention of "readability"
- perform magic woo-woo hocus-pocus pompous drivel with identifier names

Danielsson, Hughes, Jansson & Gibbons [DHJG06] tell us:

> *Functional programmers often reason about programs as if they were written in a total language, expecting the results to carry over to non-total (partial) languages. We justify such reasoning.*

Scala has a ~~few~~ lot of undermining escape hatches

- `null` (yes that `null`)
- exceptions
- Type-casing (`isInstanceOf`)
- Type-casting (`asInstanceOf`)
- Side-effects
- `equals`/`toString`/`hashCode`
- `notify`/`wait`
- `classOf`/`.getClass`
- General recursion

Scala has a ~~few~~ lot of undermining escape hatches

- `null` (yes that `null`)
- exceptions
- Type-casing (`isInstanceOf`)
- Type-casting (`asInstanceOf`)
- Side-effects
- `equals`/`toString`/`hashCode`
- `notify`/`wait`
- `classOf`/`.getClass`
- General recursion

Scala has a ~~few~~ lot of undermining escape hatches

- `null` (yes that `null`)
- exceptions
- Type-casing (`isInstanceOf`)
- Type-casting (`asInstanceOf`)
- Side-effects
- `equals`/`toString`/`hashCode`
- `notify`/`wait`
- `classOf`/`.getClass`
- General recursion

Scala has a ~~few~~ lot of undermining escape hatches

- `null` (yes that `null`)
- exceptions
- Type-casing (`isInstanceOf`)
- Type-casting (`asInstanceOf`)
- Side-effects
- `equals`/`toString`/`hashCode`
- `notify`/`wait`
- `classOf`/`.getClass`
- General recursion

Scala has a ~~few~~ lot of undermining escape hatches

- `null` (yes that `null`)
- exceptions
- Type-casing (`isInstanceOf`)
- Type-casting (`asInstanceOf`)
- Side-effects
- `equals`/`toString`/`hashCode`
- `notify`/`wait`
- `classOf`/`.getClass`
- General recursion

Parametricity
Fast and Loose Reasoning

Scala has a ~~few~~ lot of undermining escape hatches

- `null` (yes that `null`)
- exceptions
- Type-casing (`isInstanceOf`)
- Type-casting (`asInstanceOf`)
- Side-effects
- `equals`/`toString`/`hashCode`
- `notify`/`wait`
- `classOf`/`.getClass`
- General recursion

NICTA

Scala has a ~~few~~ lot of undermining escape hatches

- `null` (yes that `null`)
- exceptions
- Type-casing (`isInstanceOf`)
- Type-casting (`asInstanceOf`)
- Side-effects
- `equals`/`toString`/`hashCode`
- `notify`/`wait`
- `classOf`/`.getClass`
- General recursion

Scala has a ~~few~~ lot of undermining escape hatches

- `null` (yes that `null`)
- exceptions
- Type-casing (`isInstanceOf`)
- Type-casting (`asInstanceOf`)
- Side-effects
- `equals`/`toString`/`hashCode`
- `notify`/`wait`
- `classOf`/`.getClass`
- General recursion

Scala has a ~~few~~ lot of undermining escape hatches

- `null` (yes that `null`)
- exceptions
- Type-casing (`isInstanceOf`)
- Type-casting (`asInstanceOf`)
- Side-effects
- `equals`/`toString`/`hashCode`
- `notify`/`wait`
- `classOf`/`.getClass`
- General recursion

Scala has a ~~few~~ lot of undermining escape hatches

- `null` (yes that `null`)
- exceptions
- Type-casing (`isInstanceOf`)
- Type-casting (`asInstanceOf`)
- Side-effects
- `equals`/`toString`/`hashCode`
- `notify`/`wait`
- `classOf`/`.getClass`
- General recursion

To trade it off

- What would be the costs and benefits by eliminating these escape hatches?
- The Scalaz project explores this question with much rigour.

To trade it off

- What would be the costs and benefits by eliminating these escape hatches?
- The Scalaz project explores this question with much rigour.

Fast and Loose Reasoning

The Scalazzi Safe Scala Subset

- ~~`null`~~
- ~~exceptions~~
- ~~Type-casing (`isInstanceOf`)~~
- ~~Type-casting (`asInstanceOf`)~~
- ~~Side-effects~~
- ~~`equals`/`toString`/`hashCode`~~
- ~~`notify`/`wait`~~
- ~~`classOf`/`.getClass`~~
- General recursion

Fast and Loose Reasoning

By the way

There exist reasonable objections to the morality of fast and loose reasoning.

Parametricity
Types are Documentation

Types do not always disambiguate

```
def reverse[A](x: List[A]): List[A] =
  ...
```

but we can still tell a lot

```
def reverse[A](x: List[A]): List[A] =
  ...
```

Theorem: Every element in the result appears in the input

Parametricity
Types are Documentation

```
def reverse[A](x: List[A]): List[A] =
  ...
```

to disambiguate, compromise by using the next best thing

$\forall x.$ `reverse(reverse(x)) == x`
$\forall x.\forall y.$ `reverse(x ++ y) == reverse(y) ++ reverse(x)`

Parametricity
Types are Documentation

In practice, the examples will be less trivial

- ```
  def filterM[F[_]: Monad]
    (p: A => F[Boolean]):
    List[A] => F[List[A]]
  ```

- ```
  trait Profunctor[F[_]] {
    def dimap[A, B, C, D]
      (p: B => A, q: C => D):
      F[A, C] => F[B, D]
  }
  ```

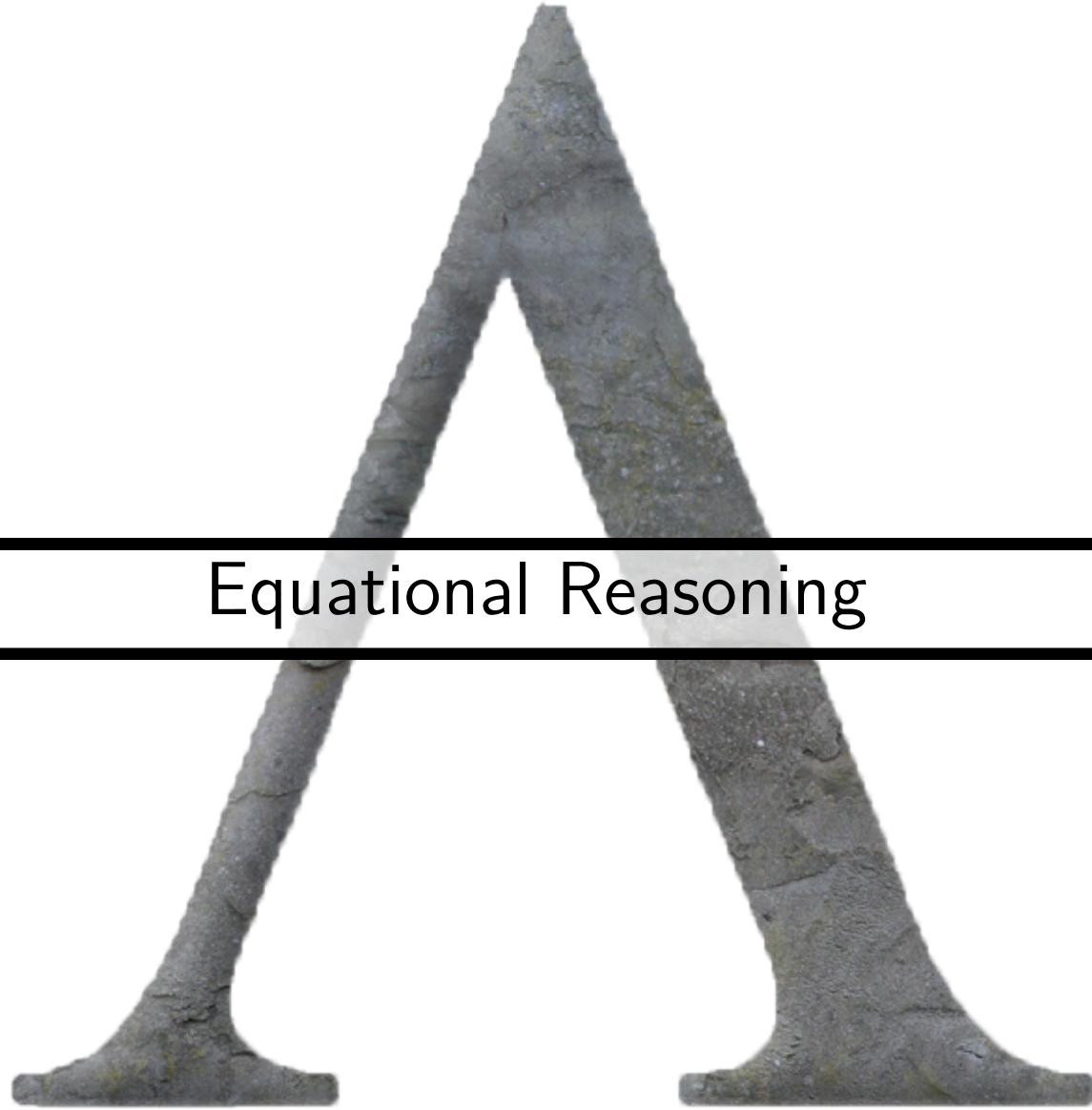
In practice, the examples will be less trivial

- ```
  def filterM[F[_]: Monad]
    (p: A => F[Boolean]):
    List[A] => F[List[A]]
  ```

- ```
  trait Profunctor[F[_]] {
    def dimap[A, B, C, D]
      (p: B => A, q: C => D):
      F[A, C] => F[B, D]
  }
  ```

Parametricity

"parametricity constantly tests more conditions than your unit test suite ever will" —*Someone*, 16 July 2014



Equational Reasoning

Equational Reasoning

Equational reasoning is a program property

that allows replacing arbitrary program expressions with values, ultimately providing a tool to construct non-trivial programs.

Equational Reasoning

We want **the potential** to replace expressions with values:

- without consequence on program behaviour
- for the general case i.e. not having to reason about concretes for abstract concepts *cough* `scala.Seq` *cough*

Equational Reasoning

We want **the potential** to replace expressions with values:

- without consequence on program behaviour
- for the general case i.e. not having to reason about concretes for abstract concepts *cough* `scala.Seq` *cough*

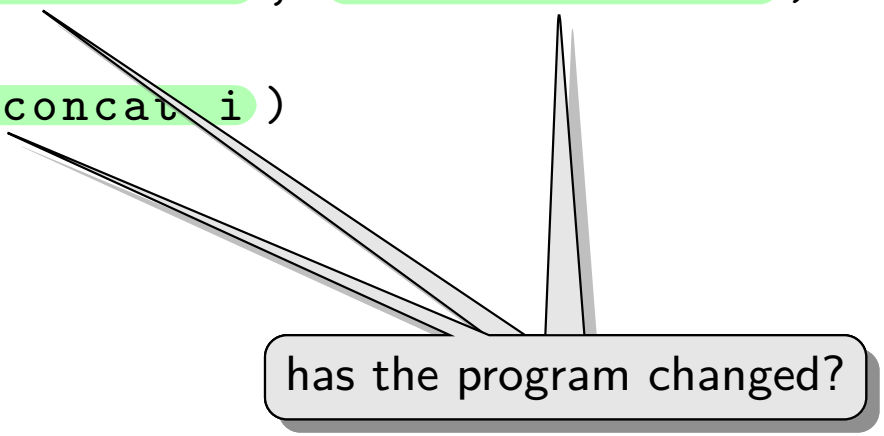
Equational Reasoning

```
val i: String = ...
val result: String = "abc" concat i
method1(result, result)
...
method2(result)
...
```

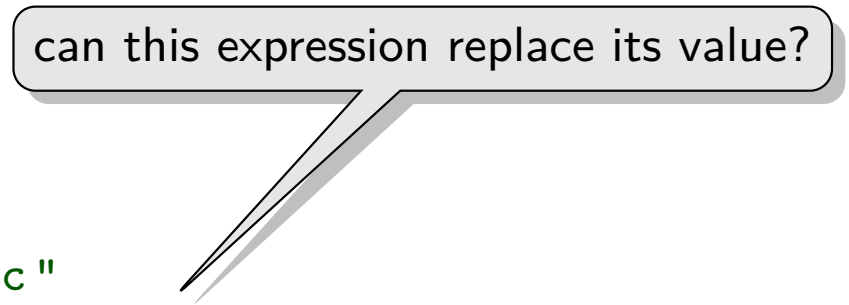
can this expression replace its value?

Equational Reasoning

```
val i: String = ...
// val result: String = "abc" concat i
method1("abc" concat i, "abc" concat i)
...
method2("abc" concat i)
...
```



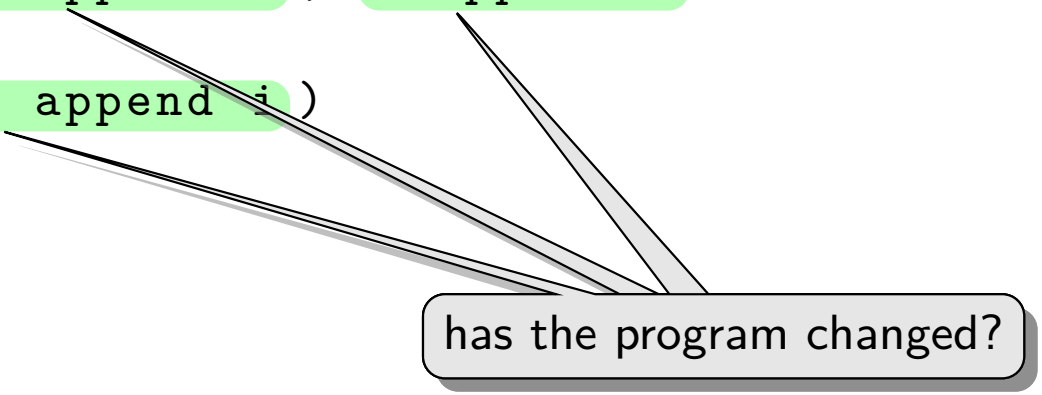
Unequational Unreasoning




```
val i: StringBuilder = ...
val b: StringBuilder = "abc"
val result: StringBuilder = b append i
method1(result, result)
...
method2(result)
...
```

Unequational Unreasoning

```
val i: StringBuilder = ...
val b: StringBuilder = "abc"
// val result: StringBuilder = b append i
method1(b append i, b append i)
...
method2(b append i)
...
```



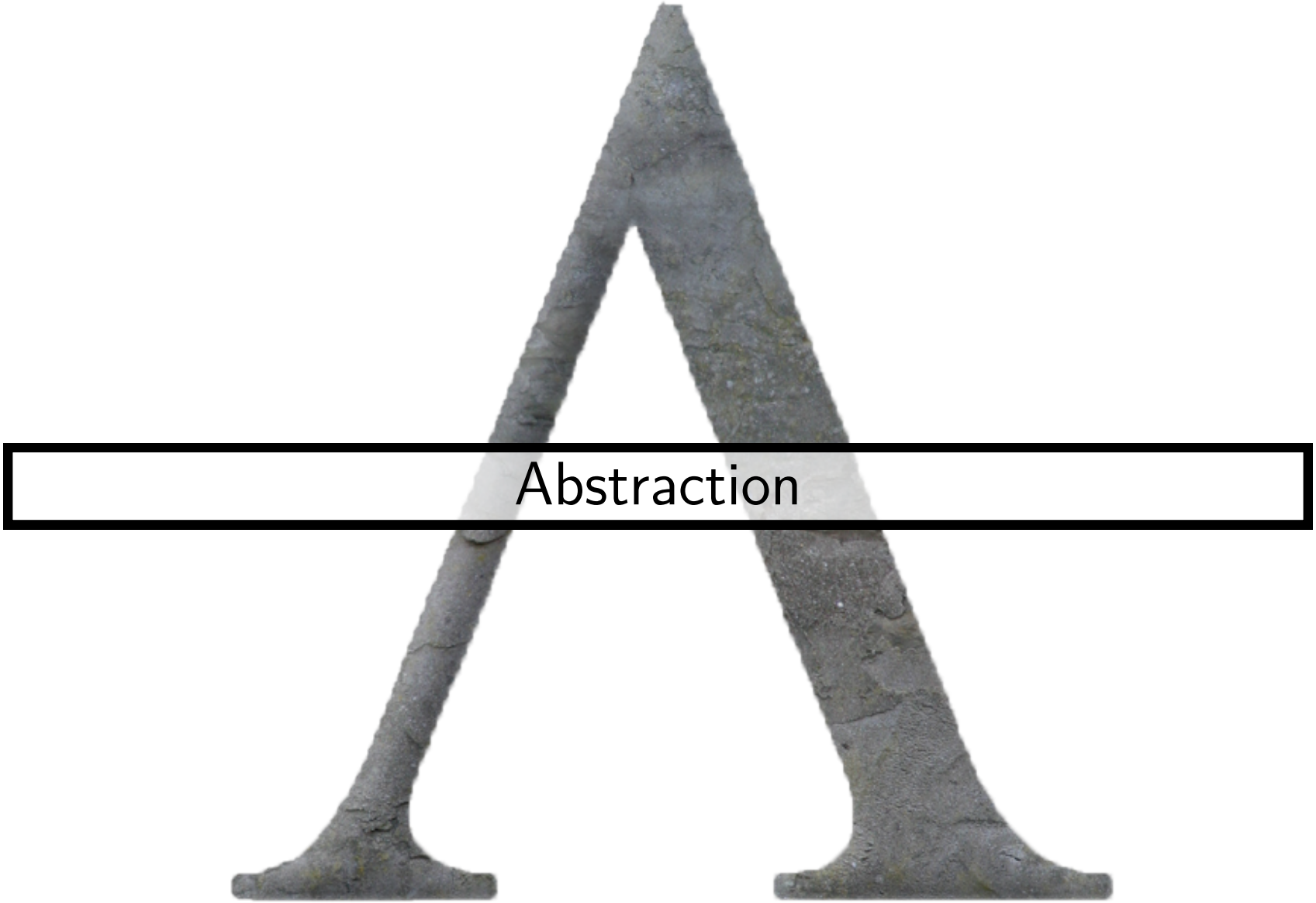
Equational Reasoning

The program has changed. Equational reasoning has been lost.

Equational Reasoning

Equational reasoning . . .

- gives rise to a catalogue of practical benefits
- is *essential* to a high-performing software development team


Abstraction

Principles and Goals of Abstraction

Construct a constraint

- minimise the requirements to satisfy the constraint to increase instances
- maximise potential for deriving operations as a consequence of satisfying the constraint

Principles and Goals of Abstraction

Trade-off between the two

- stronger constraint
  - fewer instances
  - more derived operations
- weaker constraint
  - more instances
  - fewer derived operations

Principles and Goals of Abstraction

The ultimate goal

Avoid repetition of the same work

Principles and Goals of Abstraction

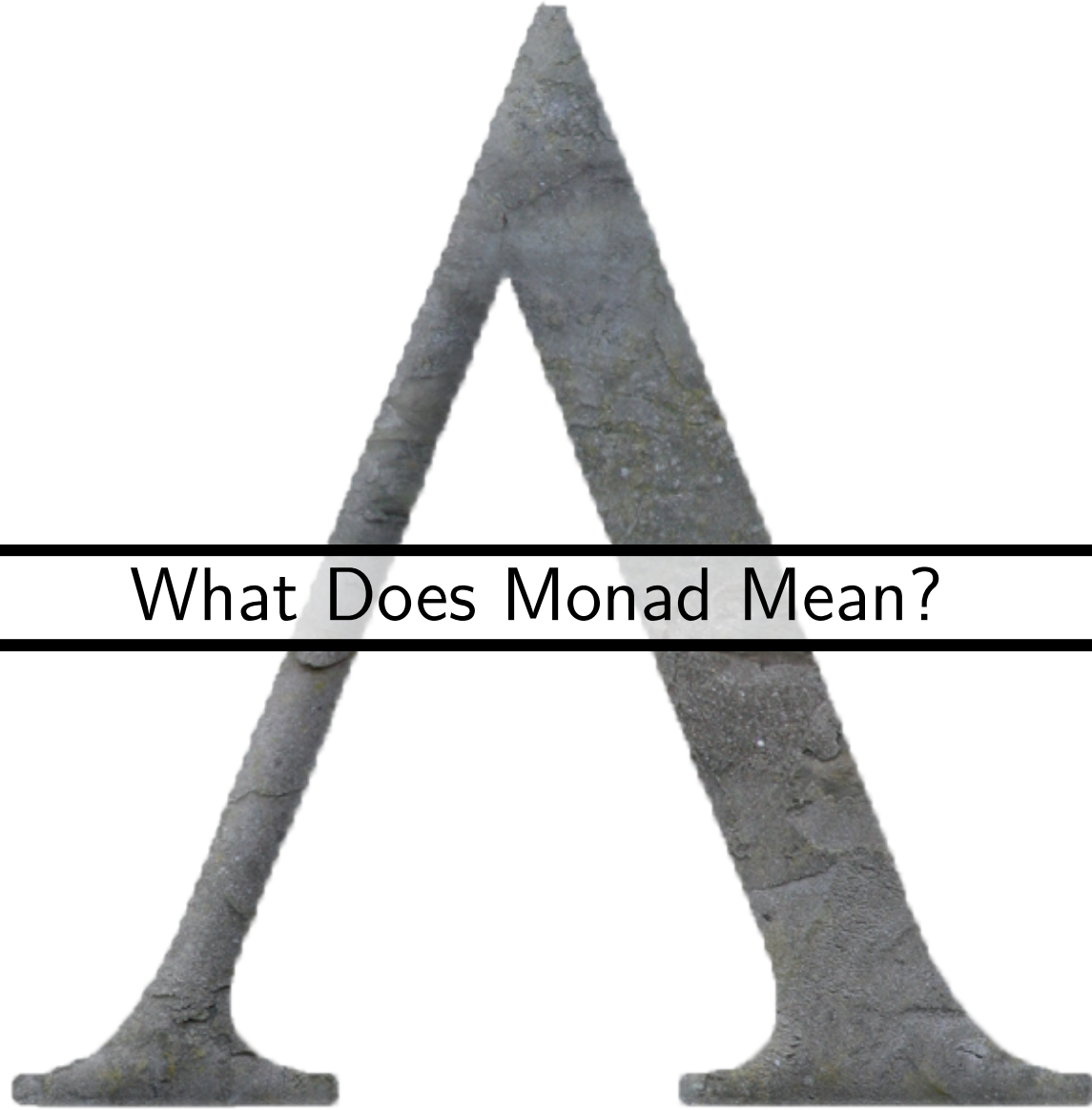
Consequently

A proposed abstraction that loses in both directions is a *false economy* and must be efficiently discarded.

cough `scala.collection.generic.CanBuildFrom` *cough*

Abstraction

For example



What Does Monad Mean?

What Does Monad Mean?

Well if you google it

burritos, spacesuits or some silly shenanigans like that

Monad

But if we decline the distraction

```
trait Monad[F[_]] {
  def bind[A, B]
    (f: A => F[B]):
    F[A] => F[B]
  def unit[A]:
    A => F[A]
}
```

Monad

This abstraction has many instances (values for `F`)

- list
- continuations
- nullable values
- exception chaining
- state
- I/O actions
- argument threading
- logging
- *hundreds more*

Monad

This abstraction gives us many useful operations

- sequencing a list of effect values
  `List[F[A]] -> F[List[A]]`
- replicating an effect a given number of times
  `Int -> F[A] -> F[List[A]]`
- *bazillions more*

Monad

We just saw

- the monad abstraction expressed as a constraint
- instances that satisfy the constraint
- operations that are derived from the constraint

Monad

We just saw

- the monad abstraction expressed as a constraint
- instances that satisfy the constraint
- operations that are derived from the constraint

Monad

We just saw

- the monad abstraction expressed as a constraint
- instances that satisfy the constraint
- operations that are derived from the constraint

Abstraction nomenclature

Other abstractions trade off along the two competing principles

- covariant functor
- applicative functor
- semigroupoid
- comonad
- profunctor
- monoid
- *hundreds more*

Now that we know this

We can do some mythbusting

- Monads are for side-effects
- Monads are for functional programming only
- Monads are for doing I/O
- Monads don't apply to my programming tasks
- Monads are for those who want Scala to be like Haskell

Now that we know this

We can do some mythbusting

- Monads are for side-effects
- Monads are for functional programming only
- Monads are for doing I/O
- Monads don't apply to my programming tasks
- Monads are for those who want Scala to be like Haskell

Now that we know this

We can do some mythbusting

- Monads are for side-effects
- Monads are for functional programming only
- Monads are for doing I/O
- Monads don't apply to my programming tasks
- Monads are for those who want Scala to be like Haskell

Now that we know this

We can do some mythbusting

- Monads are for side-effects
- Monads are for functional programming only
- Monads are for doing I/O
- Monads don't apply to my programming tasks
- Monads are for those who want Scala to be like Haskell

Now that we know this

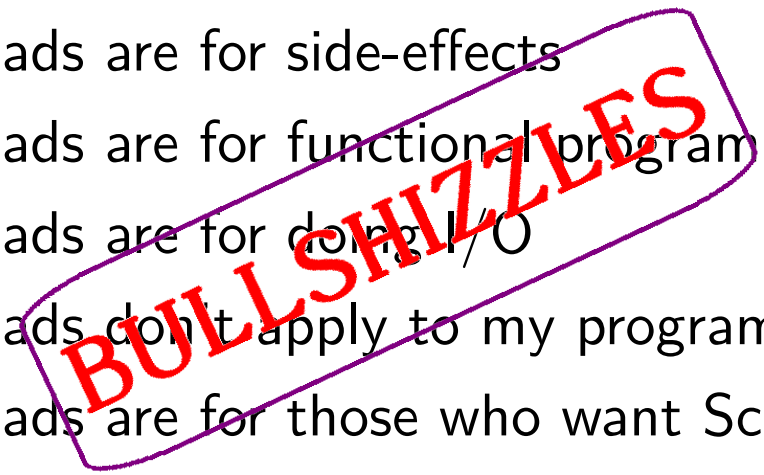
We can do some mythbusting

- Monads are for side-effects
- Monads are for functional programming only
- Monads are for doing I/O
- Monads don't apply to my programming tasks
- Monads are for those who want Scala to be like Haskell

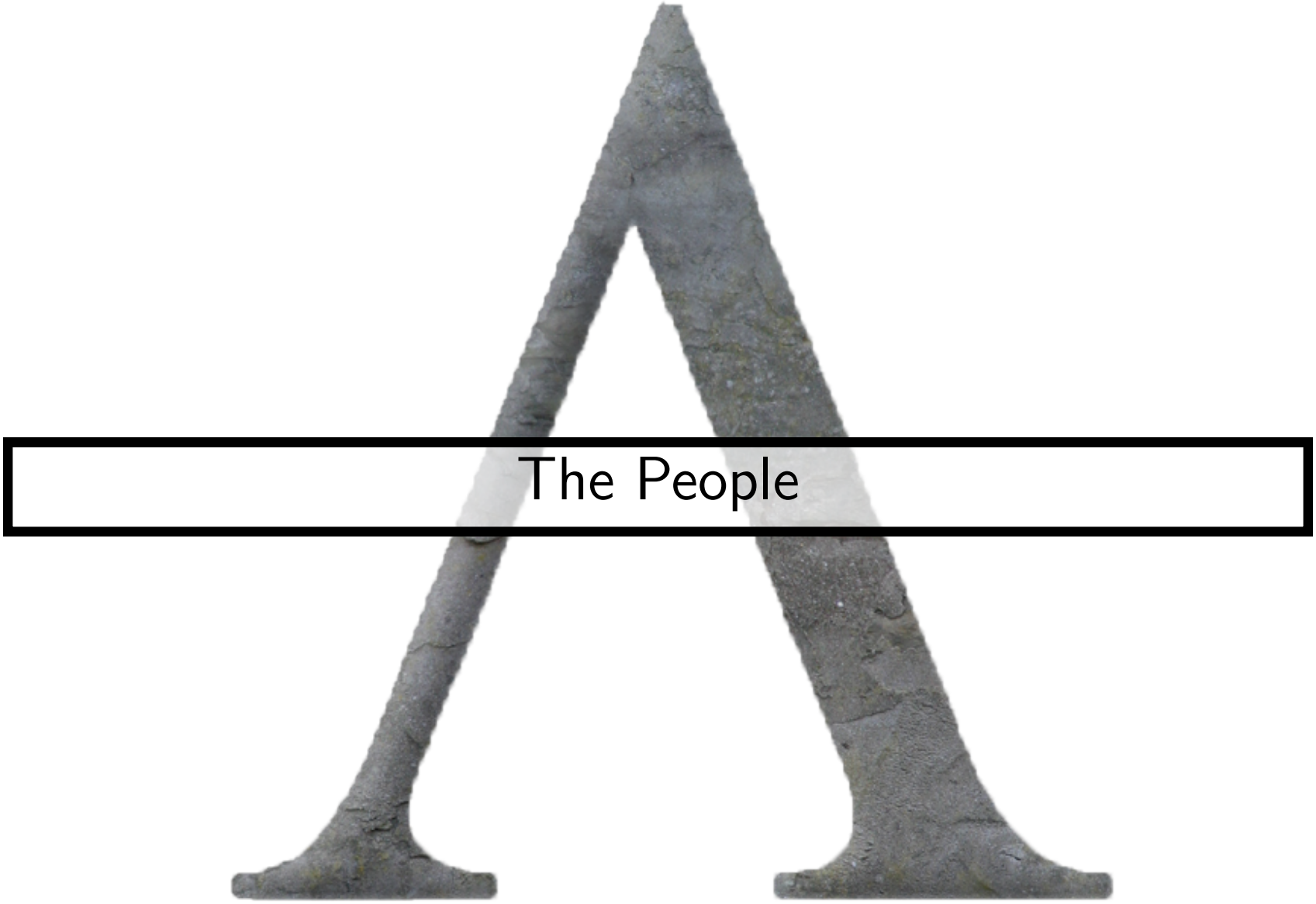
Now that we know this

We can do some mythbusting

- Monads are for side-effects
- Monads are for functional programming only
- Monads are for doing I/O
- Monads don't apply to my programming tasks
- Monads are for those who want Scala to be like Haskell



NICTA


The People

The idea of the "Scalaz community"

Originally, all contributors to Scalaz

- reject the idea of "community" —a euphemism most often deployed for the purpose of exclusionary tribalism
- cannot speak for all, but this theme carries on today

Ideas, not people

Consequently

- people are not scrutinised, but ideas are, heavily
- the rejection of "community" is an essential mode to technical success

The subsequent battles

This idea is commonly misunderstood

You are not one of us

"Scalaz developers come from Haskell"

Ideas are personal

"That person was mean to me"

Conflating the values of a person with values of an idea is a distraction from the goal

True things must never exist

"That's just your opinion mate"

Knowledge cannot possibly lead to conclusions

```
enum Fact { True, False, JustYourOpinion }
```

More tribalist horse doo-doo

"Scalaz is for extremists (or purists)"

Knowledge is inaccessible to those who are not in the tribe

I am not smart, and neither should you be

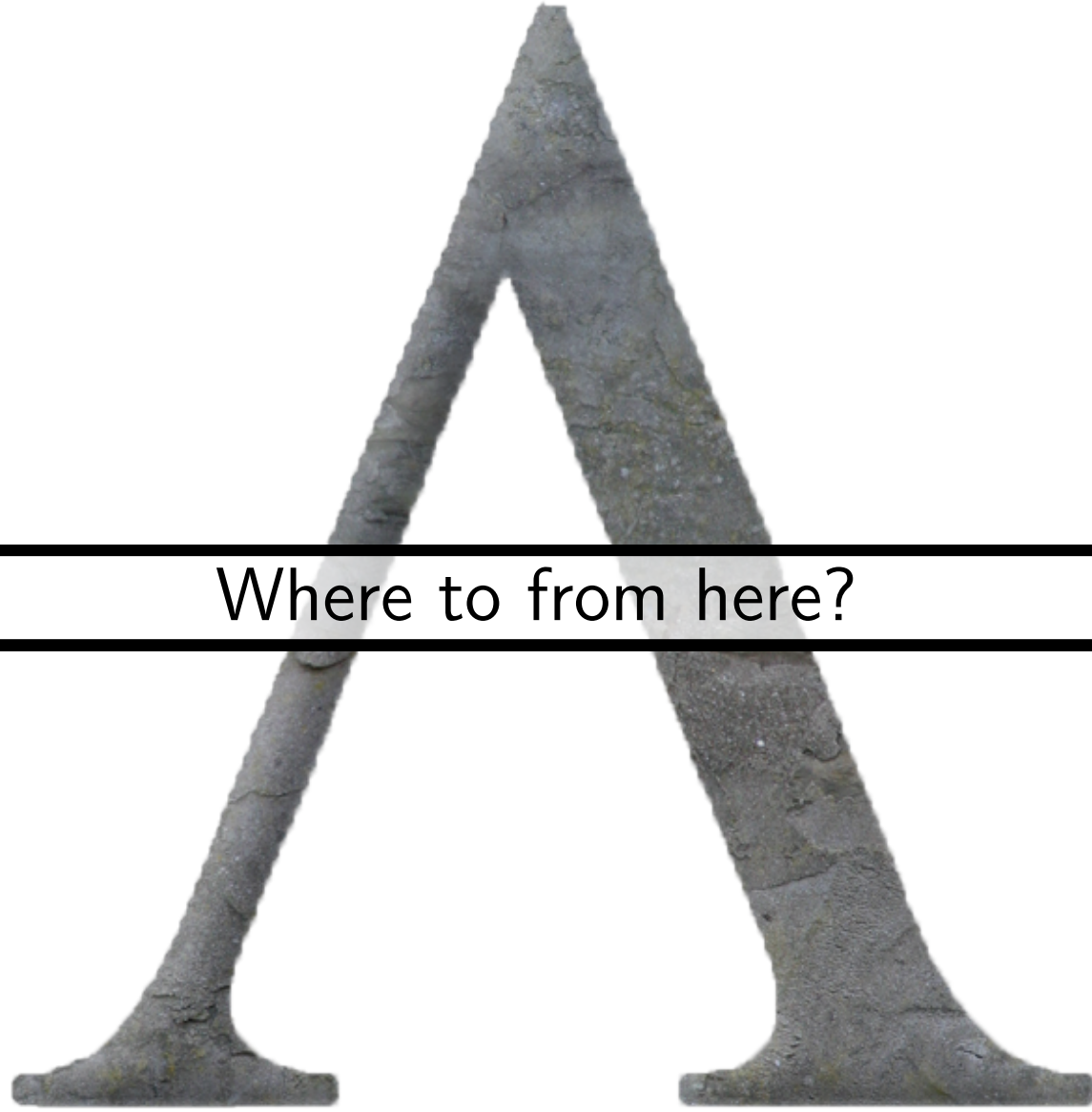
"Scalaz tries to be too clever, with a disregard for the real world"

Knowledge is inaccessible to me and I require that it is for you too

So do you just ignore it?

All this would be typically ignored, but that

it invites candidates to believe that they cannot possibly contribute to the pool of potential ideas



Where to from here?

Where to from here?

Scalaz has principled, well-defined goals, but . . .

Where to from here?

NICTA

Where to from here?

So what for Scalaz?

Can we achieve better with less effort?

Where to from here?

Options

- abandon the JVM and therefore, Scala
- continue to attempt to improve Scala, accepting relentless conflict
- accept the JVM, abandon Scala

Where to from here?

Options

- abandon the JVM and therefore, Scala
- continue to attempt to improve Scala, accepting relentless conflict
- accept the JVM, abandon Scala

Where to from here?

Options

- abandon the JVM and therefore, Scala
- continue to attempt to improve Scala, accepting relentless conflict
- accept the JVM, abandon Scala

Where to from here?

With a view to abandon Scala

- first, implement a working Scala abstract syntax tree
- program against that AST (not text)
- next, implement tools to transform the AST
- finally, begin at a Scala AST only for exceptional legacy purposes

Where to from here?

With a view to abandon Scala

- first, implement a working Scala abstract syntax tree
- program against that AST (not text)
- next, implement tools to transform the AST
- finally, begin at a Scala AST only for exceptional legacy purposes

Where to from here?

With a view to abandon Scala

- first, implement a working Scala abstract syntax tree
- program against that AST (not text)
- next, implement tools to transform the AST
- finally, begin at a Scala AST only for exceptional legacy purposes

Thanks

Principled software development demands principled compromise.

References

Nils Anders Danielsson, John Hughes, Patrik Jansson, and Jeremy Gibbons, *Fast and loose reasoning is morally correct*, ACM SIGPLAN Notices, vol. 41, ACM, 2006, pp. 206–217.

Philip Wadler, *Theorems for free!*, Proceedings of the fourth international conference on Functional programming languages and computer architecture, ACM, 1989, pp. 347–359.

Licence Attributions

- Some images used in this presentation are attributed to Joshua Morris
- Some images used in this presentation are attributed to XKCD released under the CC-BY-NC licence
- Some images used in this presentation are released under the CC0 1.0 licence