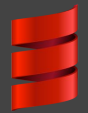
Ann Arbor FP

May 2017

Lazy Evaluation

Haskell vs. Scala



@filippovitale

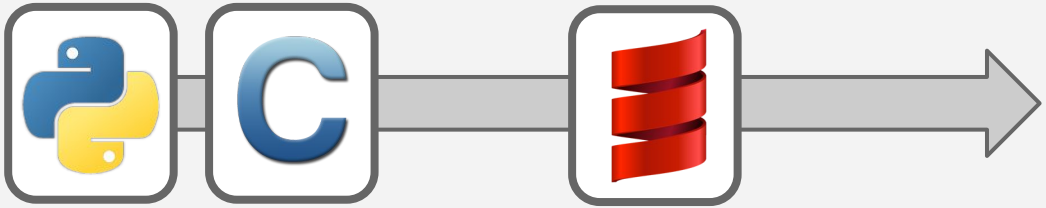
From Useless to Nirvana

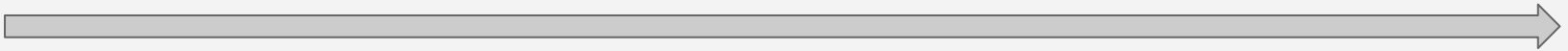
Effects Anywhere

Effects Anywhere

Limited Controlled Effects

https://www.youtube.com/watch?v=06x8Wf2r2Mc#t=21m52s

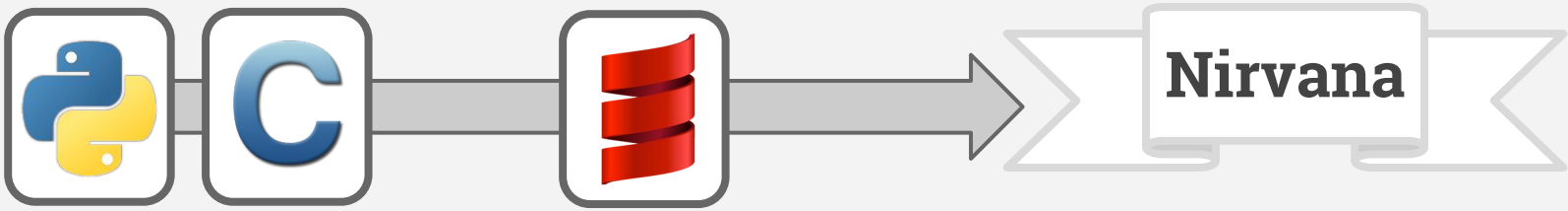


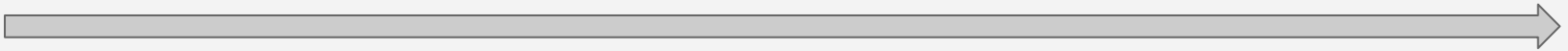


Effects Anywhere

Limited Controlled Effects

https://www.youtube.com/watch?v=06x8Wf2r2Mc#t=21m52s



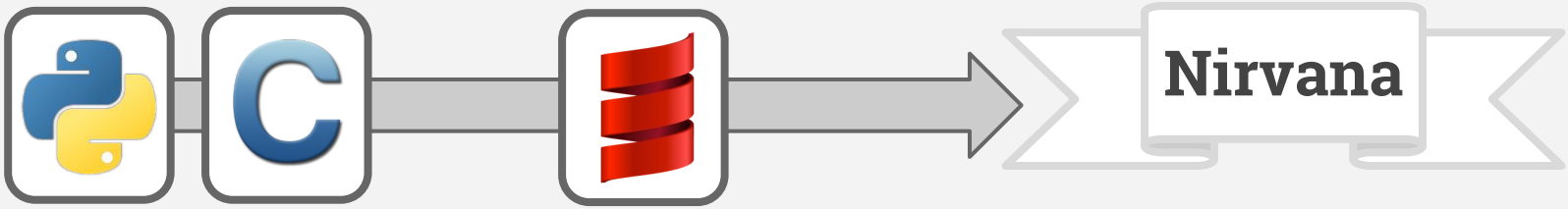


Effects Anywhere

Limited Controlled Effects

https://www.youtube.com/watch?v=06x8Wf2r2Mc#t=21m52s

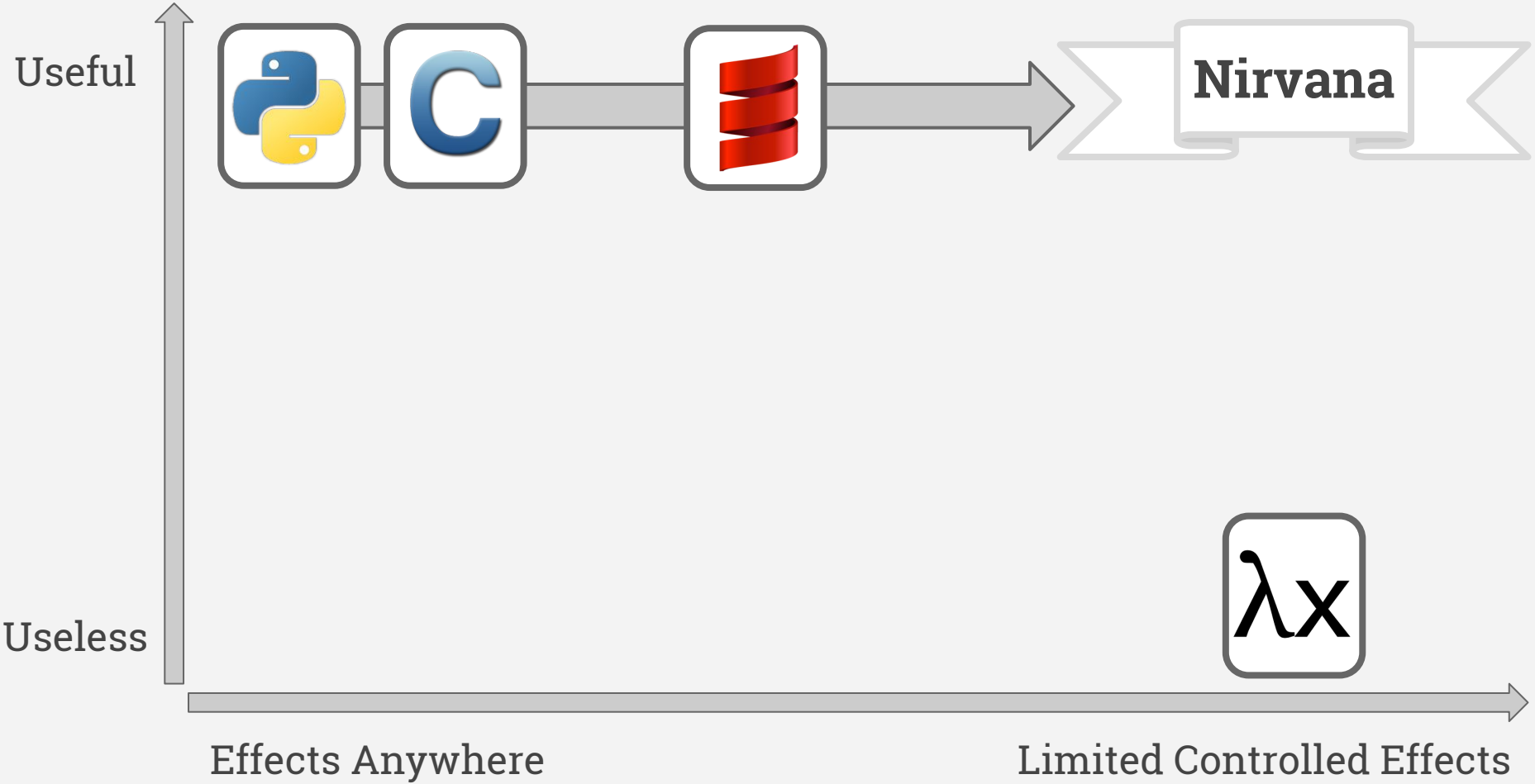
Useful



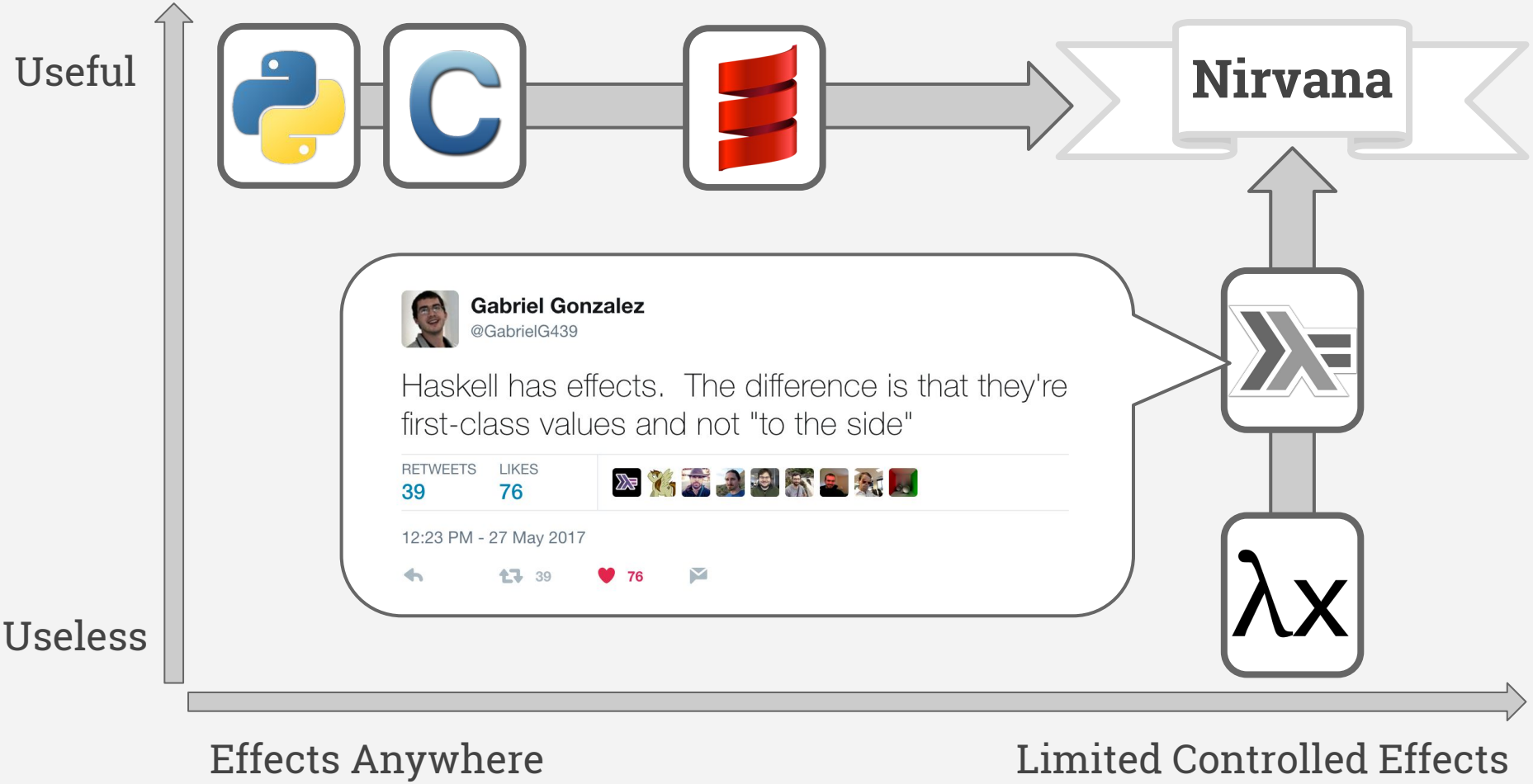
Effects Anywhere

Limited Controlled Effects

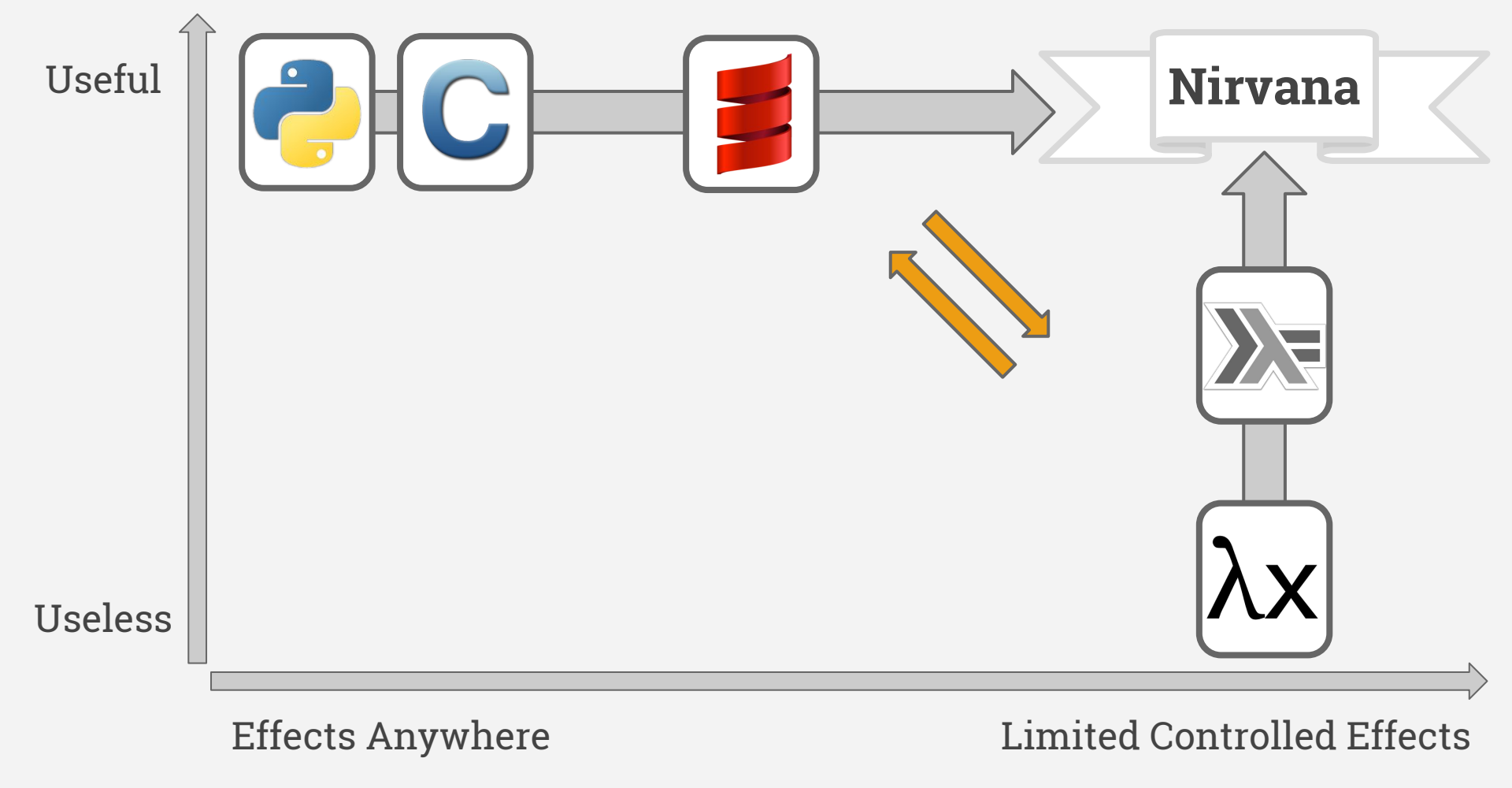
https://www.youtube.com/watch?v=06x8Wf2r2Mc#t=21m52s



https://www.youtube.com/watch?v=06x8Wf2r2Mc#t=21m52s



https://www.youtube.com/watch?v=06x8Wf2r2Mc#t=21m52s



https://www.youtube.com/watch?v=06x8Wf2r2Mc#t=21m52s

Effects & Evaluation Models

Evaluation Models

3 dominant models:

Call-by-value

- arguments are evaluated: before a function is entered

Evaluation Models

3 dominant models:

Call-by-value

- arguments are evaluated: before a function is entered

- **easy to predict** when and in what order things will happen

Evaluation Models

3 dominant models:

Call-by-value

- arguments are evaluated: before a function is entered

- **easy to predict** when and in what order things will happen
- `f(release_monkeys(), increment_counter())`
  - it doesn't matter if `f` uses those results or not

Evaluation Models

3 dominant models:

Call-by-value

- arguments are evaluated: before a function is entered

When “side effects” are allowed, strict evaluation is really what you want.

Evaluation Models

3 dominant models:

Call-by-value

- arguments are evaluated: before a function is entered

Call-by-name

- arguments are passed unevaluated

Evaluation Models

3 dominant models:

Call-by-value

- arguments are evaluated: before a function is entered

Call-by-name

- arguments are passed unevaluated

If an argument is not used in the function body
⇒ the argument is never evaluated
If it is used several times
⇒ it is re-evaluated each time it appears

Evaluation Models

3 dominant models:

Call-by-value

- arguments are evaluated: before a function is entered

Call-by-name

- arguments are passed unevaluated

Call-by-need

- arguments are passed unevaluated but an expression is only evaluated once and shared upon subsequent references

http://dev.stephendiehl.com/fun/005_evaluation.html

Evaluation Models

3 dominant models:

Lazy evaluation makes it hard to reason about when things will be evaluated
Side effects in a lazy language would be extremely unintuitive
Lazy evaluation strategy essentially forces you to also choose purity

- argu[illegible]re passed unevaluated

Call-by-need

- arguments are passed unevaluated but an expression is only evaluated once and shared upon subsequent references

https://en.wikipedia.org/wiki/Evaluation_strategy

Evaluation Models

3 dominant r

Call-by-v

- argume

Call-by-name

- arguments a passed unevaluated

Call-by-need

- arguments are passed unevaluated but an expression is only evaluated once and shared upon subsequent references

In a “pure“ (effect-free) setting
⇒ this produces the same results as call-by-name

https://en.wikipedia.org/wiki/Evaluation_strategy

Evaluation Models

3 dominant r

Call-by-v

- argume

Call-by-name

- arguments a passed unevaluated

Call-by-need

- arguments are passed unevaluated but an expression is only evaluated once and shared upon subsequent references

In a “pure“ (effect-free) setting
⇒ this produces the same results as call-by-name (memoized version of call-by-name)
Unevaluated expressions are represented by thunks

https://en.wikipedia.org/wiki/Evaluation_strategy

Thunks

Thunks

Unevaluated expressions
in heap memory, built to
postpone the Evaluation

https://takenobu-hs.github.io/downloads/haskell_lazy_evaluation.pdf (pg. 68)

Thunks

Unevaluated expressions

in h

postpone the Evaluation

Suspended Computations

https://takenobu-hs.github.io/downloads/haskell_lazy_evaluation.pdf (pg. 68)

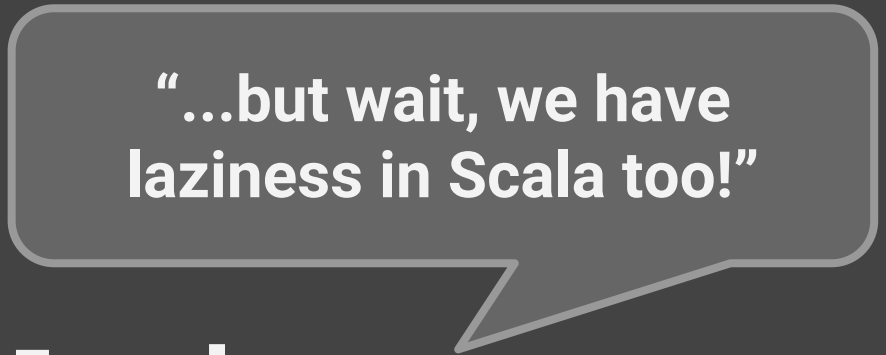
Thunks

GHC uses Thunks
to achieve **Laziness**

Thunks

GHC uses

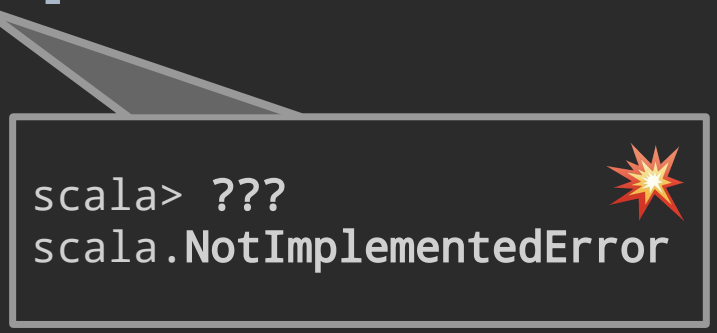
to achieve **Laziness**



http://scalapuzzlers.com/#pzzlr-012


PLEASE TELL ME MORE ABOUT
HOW SCALA IS LAZY
imgflip.com

```
λ> head [1, undefined]
1
```



```
λ> head [1, undefined]
1

λ> rest = drop 1 [1, undefined]
λ> :sp rest
rest = _
```

```
λ> head [1, undefined]
1

λ> rest = drop 1 [1, undefined]
λ> :sp rest
rest = _

λ> print $ head rest
[*** Exception: Prelude.undefined
```

```
λ> head [1, undefined]
1

λ> rest = drop 1 [1, undefined]
λ> :sp rest
rest = _

λ> print $ head rest
[*** Exception: Prelude.undefined
```

```
scala> val leroyJenkins = List(1, ???)
scala.NotImplementedError
```

```
scala> lazy val a = 1
a: Int = <lazy>

scala> lazy val b = ???
b: Nothing = <lazy>

scala> List(a, b)
```

```
scala> lazy val a = 1
a: Int = <lazy>

scala> lazy val b = ???
b: Nothing = <lazy>

scala> List(a, b)
scala.NotImplementedError
```

```
scala> val ethicList = 1 #:: ??? #:: Stream.empty
immutable.Stream[Int] = Stream(1, ?)

scala> ethicList.head
res0: Int = 1
```

```
scala> val ethicList = 1 #:: ??? #:: Stream.empty
immutable.Stream[Int] = Stream(1, ?)

scala> ethicList.head
res0: Int = 1

scala> ethicList.length
scala.NotImplementedError
```

```
λ> :set -XMonomorphismRestriction
λ> let ethicList = [1, undefined]

λ> head ethicList
1

λ> length ethicList
2

λ> :sp ethicList
ethicList = [1,_]
```

Thunks

Scala is a multi-paradigm language ⇒ **Imperative first**

https://www.quora.com/What-is-the-philosophy-of-functional-programming/answer/Tikhon-Jelvis?srid=Ezlj

Thunks

Does the order in which expressions are evaluated matter?

Scala is a [illegible] language ⇒ **Imperative first**

https://www.quora.com/What-is-the-philosophy-of-functional-programming/answer/Tikhon-Jelvis?srid=Ezlj

Thunks

Haskell is all about
evaluation and **simplification**

https://www.quora.com/What-is-the-philosophy-of-functional-programming/answer/Tikhon-Jelvis?srid=Ezlj

“Laziness allows you to express your thoughts concisely, letting the compiler figure out how to efficiently execute your code.”

Thanks!

Why make Haskell a non-strict language?

- Separation of concerns without time penalty: WYSIWYG
- Improved code reuse
- Infinite data structures
- Can make qualitative improvements to performance
- Can hurt performance in some other cases
- Makes code simpler
- Makes hard problems conceivable
- Allows for separation of concerns with regard to generating and processing data
- Laziness often introduces an overhead that leads programmers to hunt for places where they can make their code more strict
- The real benefit of laziness is in making the right things efficient enough
- Lazy evaluation allows us to write more simple, elegant code than we could in a strict environment

“**Hard work** pays off later.
Laziness pays off now!”
– Steven Wright