

Effects: to be or not to be?

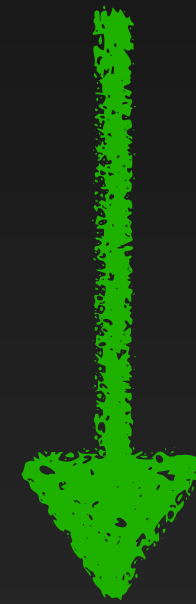
Adam Warski, May 2023

@adamwarski / @softwaremill.social / softwaremill.com



What's the problem?

```
function prepareLaunch (passengers, heading)
```



```
def prepareLaunch (  
  passengers: List[Passenger],  
  heading: Double): LaunchParameters
```



static types

Generate code
(typeclasses/implicits)

IDE completions

Eliminate whole classes
of bugs

Refactor with
confidence

Less mental burden

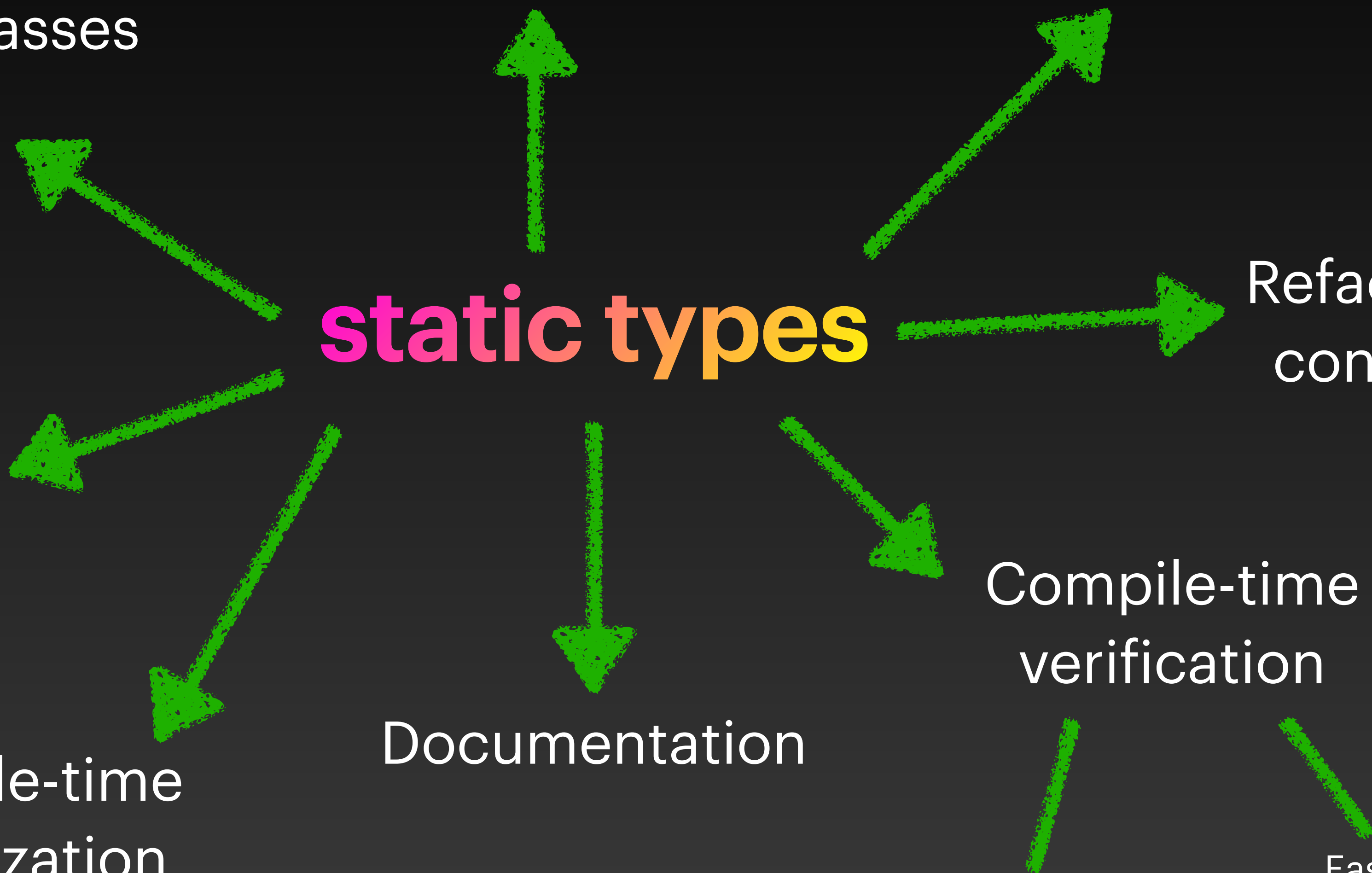
Compile-time
verification

Compile-time
optimization

Documentation

Less tests

Easier review of
AI-generated code



What's the problem?

- Can we extend the same to **effects**?
- Again: adding some more **structure**
- Will the **benefits** still outweigh the costs?



A function has a side effect if

(...) it has an observable effect other than returning a value to the caller

[https://en.wikipedia.org/wiki/Side_effect_\(computer_science\)](https://en.wikipedia.org/wiki/Side_effect_(computer_science))

A function has a side effect if

**apart from returning a value, changes
observable behaviour of the system**

side effect

vs

effect

Unwanted



Undesirable

Wanted



The purpose of calling a method

Exceptions

I/O

Write a global variable

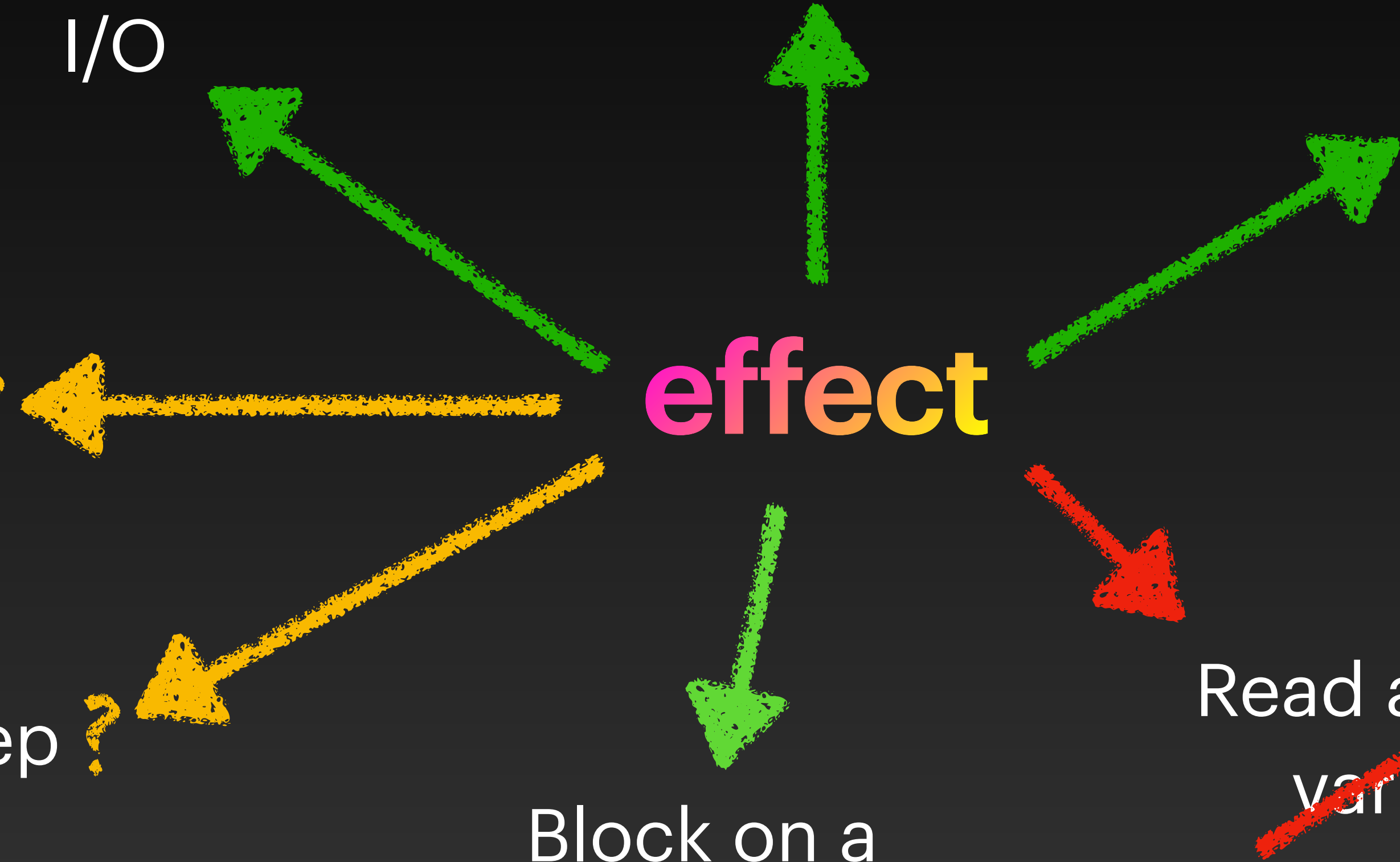
effect

Start a thread?

Sleep?

Block on a queue

~~Read a global variable~~



effect-free \neq pure



Mathematical
function

An effect system: the what?

Formal system that describes effects of programs

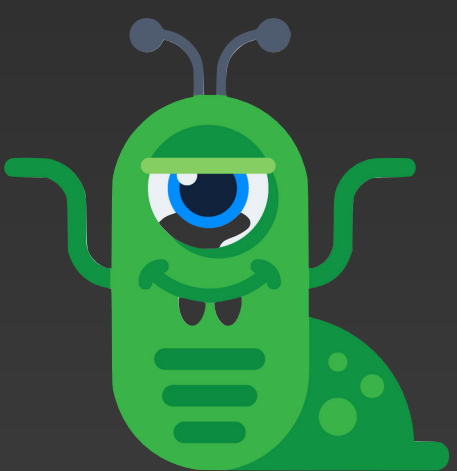
Guard rails for expressing program's logic involving effects

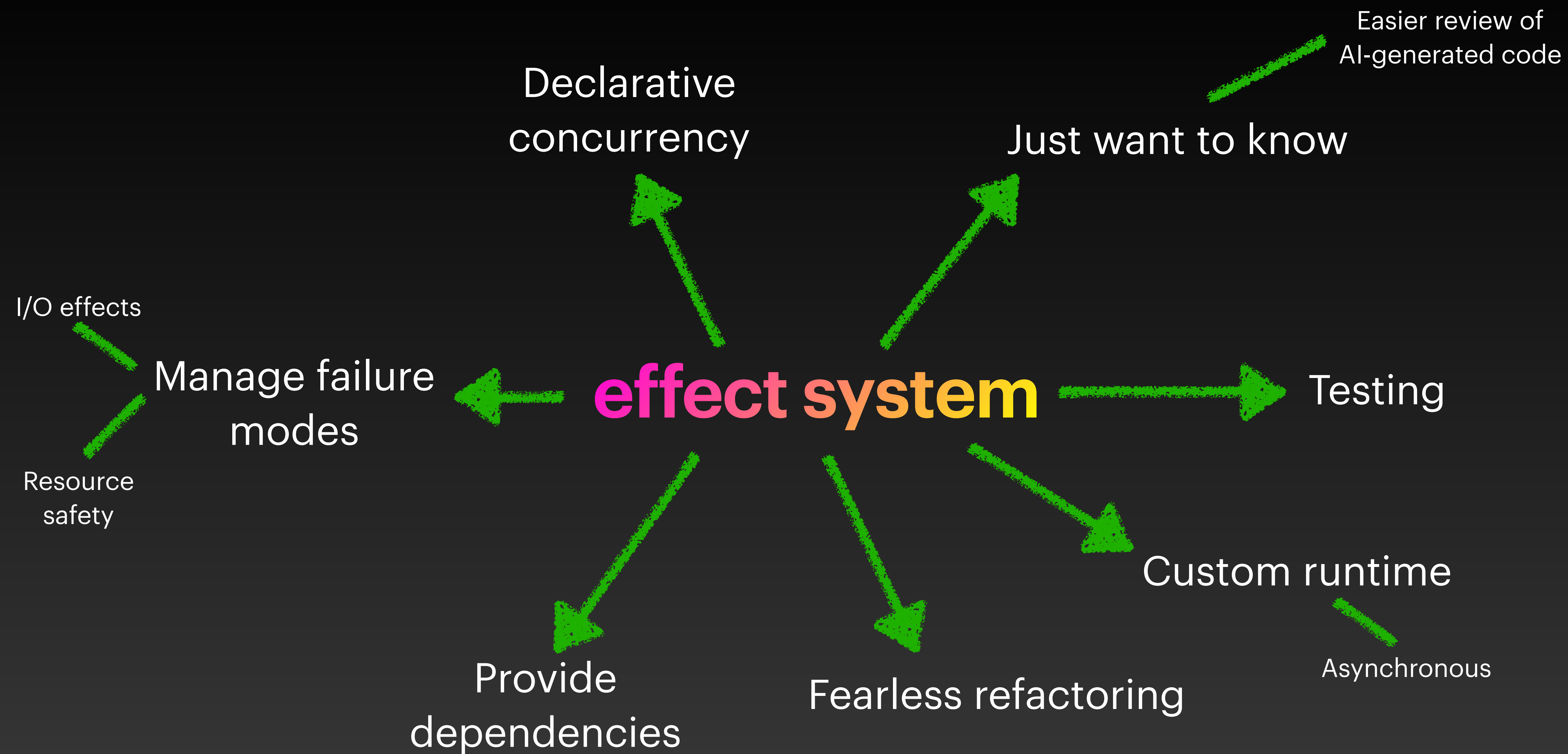
An effect system: the why?

RPC fallacy

- Make RPC calls look & behave the same as **local calls**
- But: **latency**
- But: different **failure modes**
 - intermittent failures
 - timeouts
 - partial failures

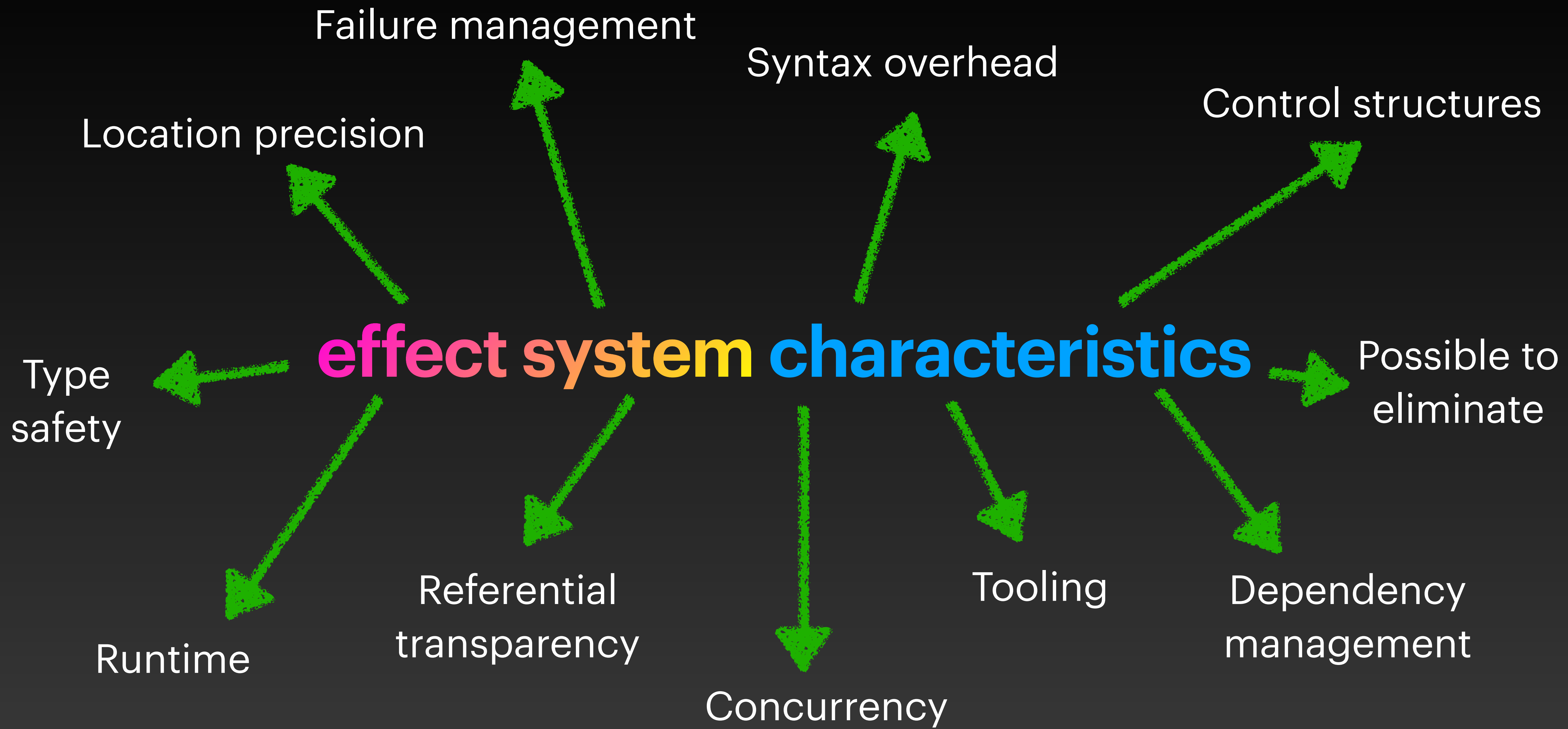
See also: distributed systems fallacy #1, "*the network is reliable*"





An effect system: the how?

Extend the type system



effect system characteristics

Failure management

Syntax overhead

Control structures

Possible to eliminate

Dependency management

Tooling

Concurrency

Referential transparency

Runtime

Type safety

Location precision

Checked exceptions

```
def goHome(): Unit throws IOException,
    WrongFuelException = {
    val passengers = fetchPassengers()

    val params = prepareLaunch(
        passengers, Headings.Home)

    if (params.farAway) {
        attachBoosterRockets()
    }

    params
    .rocketStages
    .forEach { stage => fuelUp(stage) }

    pressBigRedButton()
}
```

```
def fetchPassengers: List[Passenger]
    throws IOException

def prepareLaunch(
    passengers: List[Passenger],
    heading: Double): LaunchParams

def attachBoosterRockets(): Unit
    throws IOException

def fuelUp(stage: Stage): Unit
    throws WrongFuelException

def pressBigRedButton(): Unit
    throws IOException
```


Checked exceptions

```
def goHome(): Unit throws IOException,
    WrongFuelException = {

    val passengers = fetchPassengers()

    val params = prepareLaunch(
        passengers, Headings.Home)

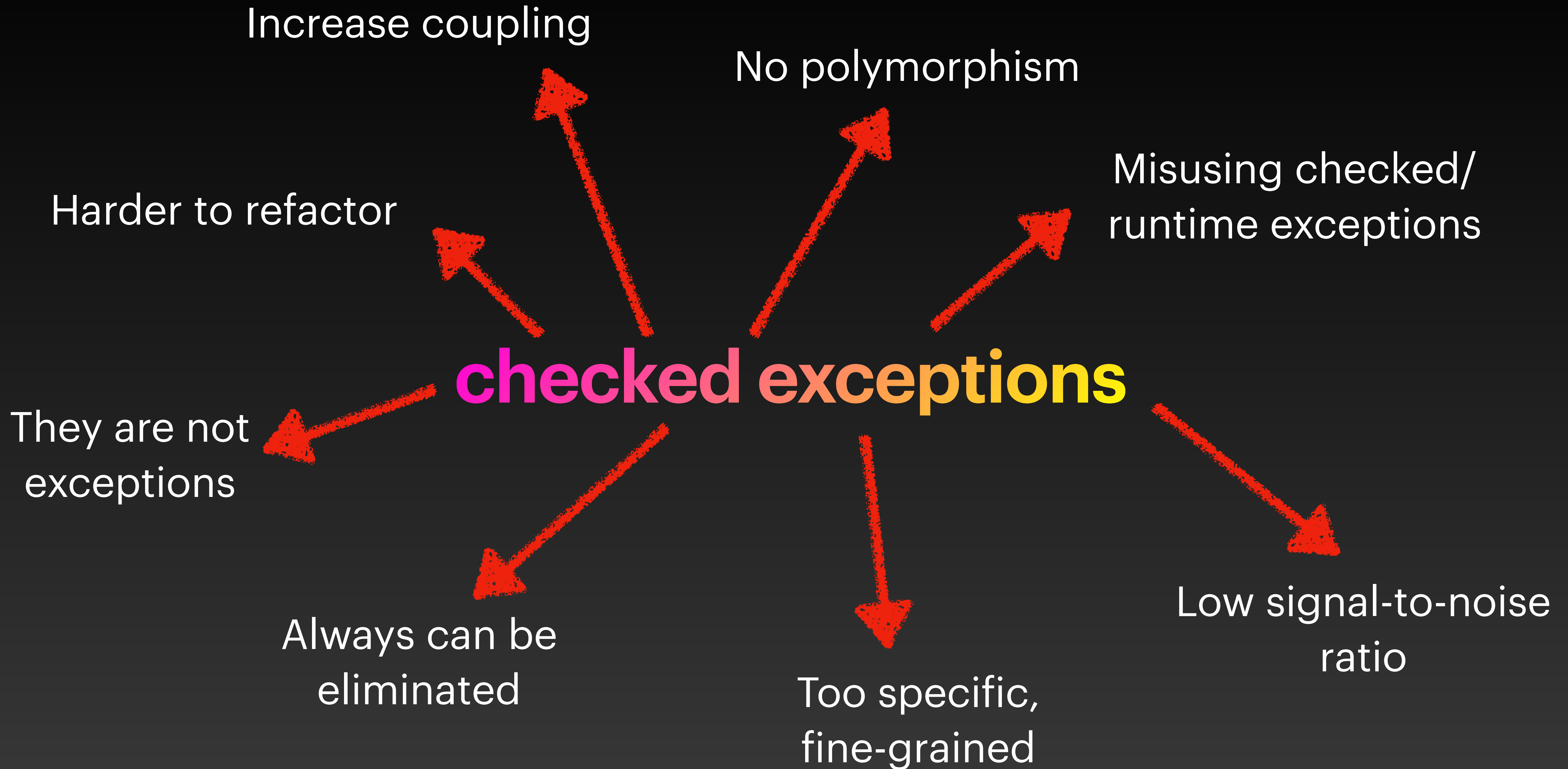
    if (params.farAway) {
        attachBoosterRockets()
    }

    params
        .rocketStages
        .map { stage => fuelUp(stage) }

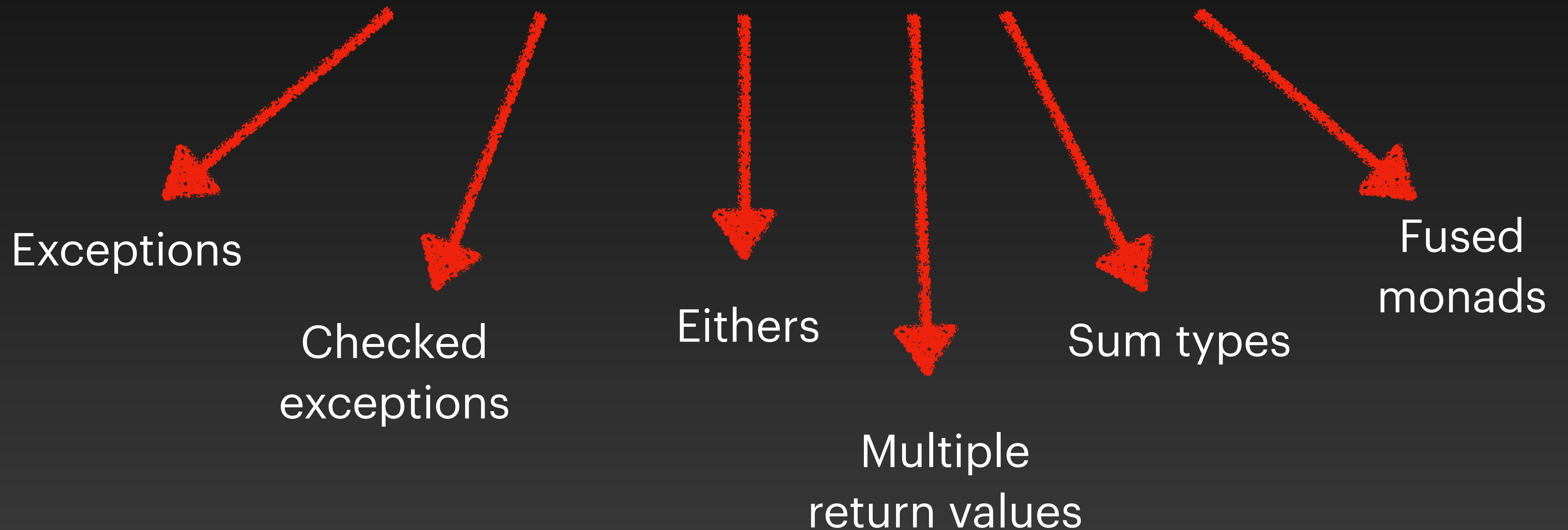
    pressBigRedButton()
}
```



Location precision	Low
Syntax overhead	Low
Type safety	Yes
Referential transparency	No
Concurrency	No
Control structures	Built-in
Failure management	Some
Resource safety	Some
Runtime	Built-in
Eliminatable	Yes
Dependency management	No



error handling
+
statically-typed languages



Future

```
def goHome(): Future[Unit] =
  fetchPassengers
    .flatMap { passengers =>
      val params = prepareLaunch(
        passengers, Headings.Home)

      def attach = if (params.farAway) {
        attachBoosterRockets
      } else Future.unit

      def fuel = Future.sequence(
        params.rocketStages.map(fuelUp))

      attach
        .flatMap(_ => fuel)
        .flatMap(_ => pressBigRedButton)
    }
}
```

```
def fetchPassengers
  : Future[List[Passenger]]
```

```
def prepareLaunch(
  passengers: List[Passenger],
  heading: Double): LaunchParams
```

```
def attachBoosterRockets(): Future[Unit]
```

```
def fuelUp(stage: Stage): Future[Unit]
```

```
def pressBigRedButton(): Future[Unit]
```

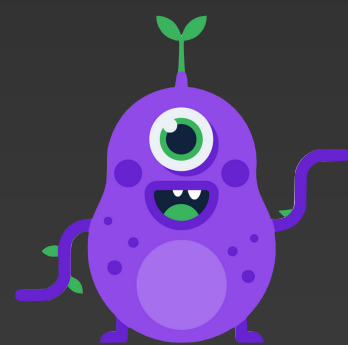
Future

```
def goHome(): Future[Unit] =
  fetchPassengers
  .flatMap { passengers =>
    val params = prepareLaunch(
      passengers, Headings.Home)

    def attach = if (params.farAway) {
      attachBoosterRockets
    } else Future.unit

    def fuel = Future.sequence(
      params.rocketStages.map(fuelUp))

    attach
      .flatMap(_ => fuel)
      .flatMap(_ => pressBigRedButton)
  }
```



Location precision	High
Syntax overhead	High
Type safety	Yes
Referential transparency	No
Concurrency	Some
Control structures	Custom
Failure management	Some
Resource safety	Some
Runtime	Custom
Eliminatable	Preferably not
Dependency management	No

ZIO

```
def goHome(): ZIO[Exception, Unit] =
  fetchPassengers
    .flatMap { passengers =>
      val params = prepareLaunch(
        passengers, Headings.Home)

      val attach = if (params.farAway) {
        attachBoosterRockets
      } else ZIO.unit

      val fuel = ZIO.foreachPar(
        params.rocketStages)(fuelUp)

      attach
        .flatMap(_ => fuel)
        .flatMap(_ => pressBigRedButton)
    }

def fetchPassengers
  : ZIO[IOException, List[Passenger]]

def prepareLaunch(
  passengers: List[Passenger],
  heading: Double): LaunchParams

def attachBoosterRockets()
  : ZIO[IOException, Unit]

def fuelUp(stage: Stage)
  : ZIO[WrongFuelException, Unit]

def pressBigRedButton()
  : ZIO[IOException, Unit]
```


ZIO

```
def goHome(): ZIO[Exception, Unit] =
  fetchPassengers
    .flatMap { passengers =>
      val params = prepareLaunch(
        passengers, Headings.Home)

      val attach = if (params.farAway) {
        attachBoosterRockets
      } else ZIO.unit

      val fuel = ZIO.foreachPar(
        params.rocketStages)(fuelUp)

      attach
        .flatMap(_ => fuel)
        .flatMap(_ => pressBigRedButton)
    }
}
```



Location precision	High
Syntax overhead	High
Type safety	Yes
Referential transparency	Yes
Concurrency	Yes
Control structures	Custom
Failure management	Yes
Resource safety	Yes
Runtime	Custom
Eliminatable	Preferably not
Dependency management	Yes

ZIO + ZIO-direct

```
def goHome(): ZIO[Any, Exception, Unit] =
  defer {
    val passengers = fetchPassengers.run
    val params = prepareLaunch(
      passengers, Headings.Home)

    if (params.farAway) {
      attachBoosterRockets.run
    }

    ZIO.foreachPar(
      params.rocketStages)(fuelUp)
      .run

    pressBigRedButton.run
  }
```

```
def fetchPassengers
  : ZIO[IOException, List[Passenger]]

def prepareLaunch(
  passengers: List[Passenger],
  heading: Double): LaunchParams

def attachBoosterRockets()
  : ZIO[IOException, Unit]

def fuelUp(stage: Stage)
  : ZIO[WrongFuelException, Unit]

def pressBigRedButton()
  : ZIO[IOException, Unit]
```


ZIO + ZIO-direct

```
def goHome(): ZIO[Any, Exception, Unit] =
  defer {
    val passengers = fetchPassengers.run
    val params = prepareLaunch(
      passengers, Headings.Home)

    if (params.farAway) {
      attachBoosterRockets.run
    }

    ZIO.foreachPar(
      params.rocketStages)(fuelUp)
      .run

    pressBigRedButton.run
  }
```



Location precision	High
Syntax overhead	Medium
Type safety	Yes
Referential transparency	Mixed
Concurrency	Yes
Control structures	Mostly custom
Failure management	Yes
Resource safety	Yes
Runtime	Custom
Eliminatable	Preferably not
Dependency management	No

Abilities (Unison)

```
goHome : '{IO} Unit
goHome =
  passengers = !fetchPassengers
  params = prepareLaunch(
    passengers, Headings.Home)

  if (params.farAway)
    then !attachBoosterRockets
    else ()

  map (stage -> !fuelUp(stage))
    params.rocketStages

  !pressBigRedButton
```

```
fetchPassengers : '{IO} List Passenger
-- sugar for: () {IO}-> List Passenger

prepareLaunch : List Passenger ->
  Double -> LaunchParams

attachBoosterRockets : '{IO} Unit

fuelUp(stage: Stage) : '{IO} Unit

pressBigRedButton() : '{IO} Unit
```

Abilities (Unison)

```
goHome : '{IO} Unit
goHome =
  passengers = !fetchPassengers
  params = prepareLaunch(
    passengers, Headings.Home)

  if (params.farAway)
    then !attachBoosterRockets
    else ()

  map (stage -> !fuelUp(stage))
    params.rocketStages

  !pressBigRedButton
```



Location precision	High
Syntax overhead	Medium
Type safety	Yes
Referential transparency	Mixed
Concurrency	Yes
Control structures	Built-in
Failure management	Some
Resource safety	No
Runtime	Built-in
Eliminatable	IO - No, other - Yes
Dependency management	Yes

Loom-based library + capabilities

```
def goHome(): IO[Unit] = {  
  val passengers = fetchPassengers()  
  
  val params = prepareLaunch(  
    passengers, Headings.Home)  
  
  if (params.farAway) {  
    attachBoosterRockets()  
  }  
  
  params  
    .rocketStages  
    .map { stage => fuelUp(stage) }  
  
  pressBigRedButton()  
}
```

```
def fetchPassengers: IO[List[Passenger]]  
// def fetchPassengers(  
//   using IO): List[Passenger]  
  
def prepareLaunch(  
  passengers: List[Passenger],  
  heading: Double): LaunchParams  
  
def attachBoosterRockets(): IO[Unit]  
  
def fuelUp(stage: Stage): IO[Unit]  
  
def pressBigRedButton(): IO[Unit]
```

Loom-based library + capabilities

```
def goHome(): IO[Unit] = {  
  val passengers = fetchPassengers()  
  
  val params = prepareLaunch(  
    passengers, Headings.Home)  
  
  if (params.farAway) {  
    attachBoosterRockets()  
  }  
  
  params  
  .rocketStages  
  .map { stage => fuelUp(stage) }  
  
  pressBigRedButton()  
}
```



Location precision	Low
Syntax overhead	Low
Type safety	Yes
Referential transparency	No
Concurrency	Yes
Control structures	Built-in
Failure management	Some
Resource safety	?
Runtime	Built-in
Eliminatable	IO - Preferably not, other - Yes
Dependency management	No

Location precision (high)

2-of-3?

1-of-3?

Referential transparency (yes)

Syntax overhead (low)

Decisions, decisions ...

- How important is the lack **syntax overhead** for you?
- What can we learn from **checked exceptions**?
- How representing **computations as values** fits your programming style?
- Is the JVM **runtime** a good fit, or do you need more control / performance?





IO.pure("Thank you!")

@adamwarski / @softwaremill.social / softwaremill.com

