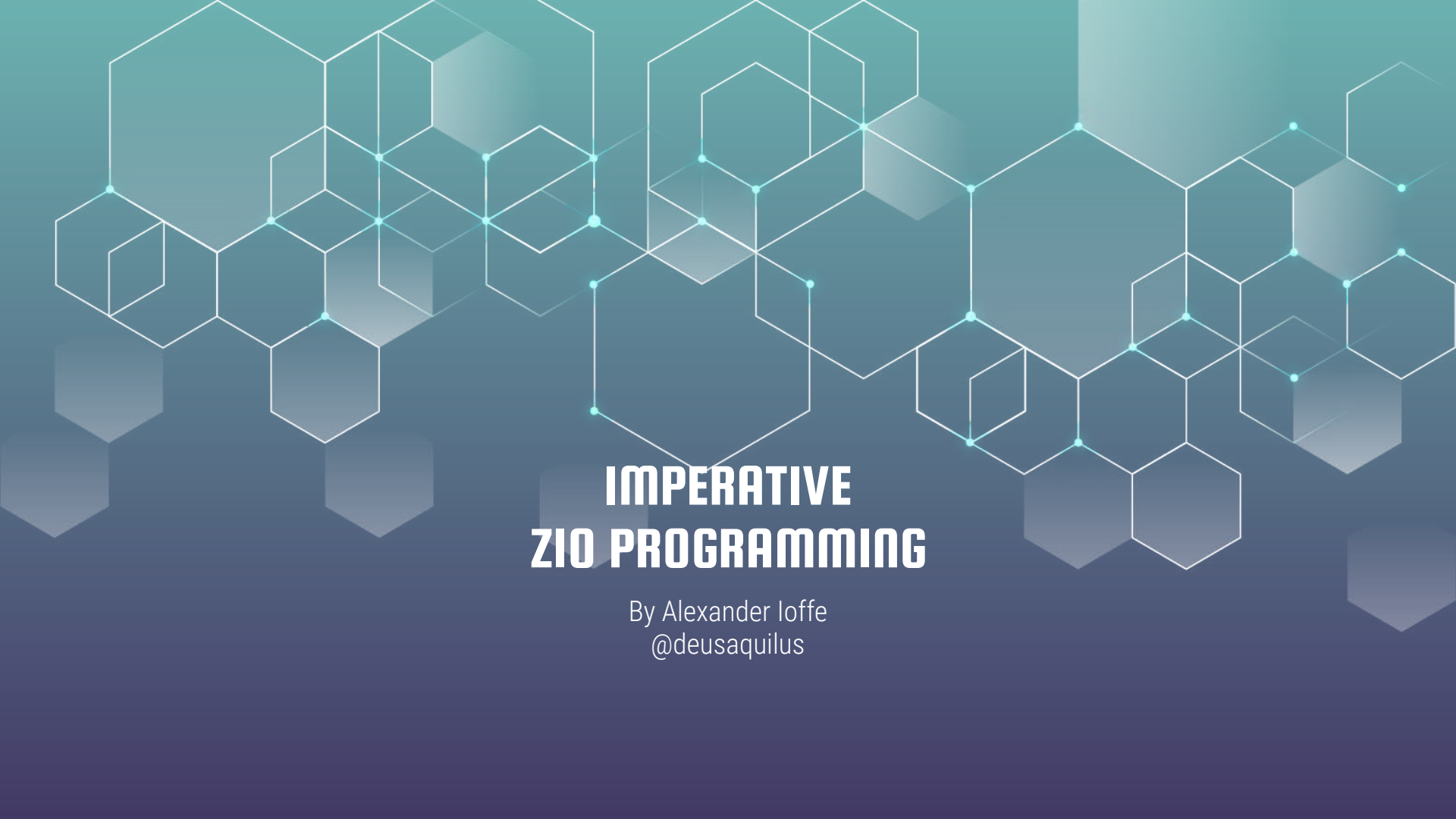


By Alexander Ioffe
@deusaquilus

Is Functional Programming Easy?

Is Functional Programming (With Effect Systems) Easy?

SEQUENCING

IMPERATIVE

```
val textA = read(fileA)
val textB = read(fileB)
write(fileC, textA + textB)
```

```
def read(file: File): String
def write(file: File, content: String): Unit
```

SEQUENCING

IMPERATIVE

```
val textA = read(fileA)
val textB = read(fileB)
write(fileC, textA + textB)
```

```
def read(file: File): String
def write(file: File, content: String): Unit
```

FUNCTIONAL

```
def read(file: File):
  ZIO[Any, Throwable, String]
def write(file: File, content: String):
  ZIO[Any, Throwable, Unit]
```

SEQUENCING

IMPERATIVE

```
val textA = read(fileA)
val textB = read(fileB)
write(fileC, textA + textB)
```

```
def read(file: File): String
def write(file: File, content: String): Unit
```

FUNCTIONAL

```
read(fileA).flatMap { textA =>
  read(fileB).flatMap { textB =>
    write(fileC, textA + textB)
  }
}
```

```
def read(file: File):
  ZIO[Any, Throwable, String]
def write(file: File, content: String):
  ZIO[Any, Throwable, Unit]
```

SEQUENCING | FOR COMPREHENSIONS TO THE RESCUE

IMPERATIVE

```
val textA = read(fileA)
val textB = read(fileB)
write(fileC, textA + textB)
```

```
def read(file: File): String
def write(file: File, content: String): Unit
```

FUNCTIONAL

```
for {
  textA <- read(fileA)
  textB <- read(fileB)
  _ <- write(fileC, textA + textB)
} yield ()
```

```
def read(file: File):
  ZIO[Any, Throwable, String]
def write(file: File, content: String):
  ZIO[Any, Throwable, Unit]
```

SEQUENCING | FOR COMPREHENSIONS TO THE RESCUE

IMPERATIVE

```
val textA = read(fileA)
val textB = read(fileB)
write(fileC, textA + textB)
```

```
def read(file: File): String
def write(file: File, content: String): Unit
```

FUNCTIONAL

```
for {
  textA <- read(fileA)
  textB <- read(fileB)
  _ <- write(fileC, textA + textB)
} yield ()
```


```
def read(file: File):
  ZIO[Any, Throwable, String]
def write(file: File, content: String):
  ZIO[Any, Throwable, Unit]
```

BRANCHING

IMPERATIVE

```
val rows: List[Row] = ...

val db = Database.open()
if (db.transactionsEnabled() && db.lazyFetchEnabled()) {
  db.bulkInsert(rows)
}
```

```
class Database {
  def transactionsEnabled(): Boolean
  def lazyFetchEnabled(): Boolean
  def bulkInsert(row: List[Row]): Unit
}
```

BRANCHING | FOR COMPREHENSIONS TO THE RESCUE?

IMPERATIVE

```scala
val rows: List[Row] = ...

val db = Database.open()
if (db.transactionsEnabled() && db.lazyFetchEnabled()) {
  db.bulkInsert(rows)
}
```

```scala
class Database {
  def transactionsEnabled(): Boolean
  def lazyFetchEnabled(): Boolean
  def bulkInsert(row: List[Row]): Unit
}
```

FUNCTIONAL

```scala
val rows: List[Row] = ...

for {
  db <- Database.open
  te <- db.transactionsEnabled()
  lf <- db.lazyFetchEnabled()
  _ <- {
    if (te && lf)
      db.bulkInsert(rows)
    else
      ZIO.unit
  }
} yield ()
```

Why am I fetching properties I might never use???

```scala
class Database {
  def transactionsEnabled(): ZIO[Any, Throwable, Boolean]
  def lazyFetchEnabled(): ZIO[Any, Throwable, Boolean]
  def bulkInsert(row: List[Row]): ZIO[Any, Throwable, Unit]
}
```

BRANCHING | FOR COMPREHENSIONS TO THE RESCUE?

IMPERATIVE

```scala
val rows: List[Row] = ...

val db = Database.open()
if (db.transactionsEnabled() && db.lazyFetchEnabled()) {
  db.bulkInsert(rows)
}
```

```scala
class Database {
  def transactionsEnabled(): Boolean
  def lazyFetchEnabled(): Boolean
  def bulkInsert(row: List[Row]): Unit
}
```

FUNCTIONAL

```scala
val rows: List[Row] = ...

Database.open().flatMap { db =>
  db.transactionsEnabled().flatMap { te =>
    if (te)
      db.lazyFetchEnabled().flatMap { lf =>
        if (lf)
          db.bulkInsert(rows)
        else
          ZIO.unit
      }
    else
      ZIO.unit
  }
}
```


```scala
class Database {
  def transactionsEnabled(): ZIO[Any, Throwable, Boolean]
  def lazyFetchEnabled(): ZIO[Any, Throwable, Boolean]
  def bulkInsert(row: List[Row]): ZIO[Any, Throwable, Unit]
}
```

LOOPING

IMPERATIVE

```
var line: String = file.readLine()

while (line != null) {
  buffer.append(line)
  line = file.readLine()
}
```

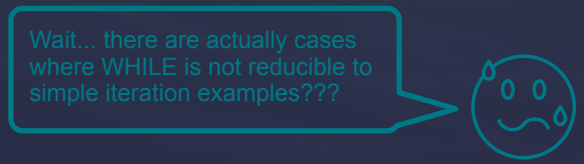
```
class File {
  def readLine(): String
  def close(): Unit
}
```

FUNCTIONAL

LOOPING | THE PROGRAM... OR YOUR MIND?

IMPERATIVE

```
var line: String = file.readLine()

while (line != null) {
  buffer.append(line)
  line = file.readLine()
}
```




```
class File {
  def readLine(): String
  def close(): Unit
}
```

FUNCTIONAL

```
file.readLine().flatMap { line0 =>
  var line = line0
  def whileFun(): ZIO[Any, Throwable, Unit] =
    if (line != null)
      buffer.append(line)
      file.readLine().flatMap { lineN =>
        line = lineN
        whileFun()
      }
    else
      ZIO.unit
  whileFun()
```

```
class File {
  def readLine(): ZIO[Any, Throwable, String]
  def close(): ZIO[Any, Throwable, Unit]
}
```

LOOPING | THE PROGRAM... OR YOUR MIND?

IMPERATIVE

```
var line: String = file.readLine()

def whileFun(): Unit =
  if (line != null) {
    buffer.append(line)
    line = file.readLine()
    whileFun()
  } else {
    ()
  }
```

```
class File {
  def readLine(): String
  def close(): Unit
}
```

FUNCTIONAL

```
file.readLine().flatMap { line0 =>
  var line = line0
  def whileFun(): ZIO[Any, Throwable, Unit] =
    if (line != null)
      buffer.append(line)
      file.readLine().flatMap { lineN =>
        line = lineN
        whileFun()
      }
    else
      ZIO.unit
  whileFun()
```

```
class File {
  def readLine(): ZIO[Any, Throwable, String]
  def close(): ZIO[Any, Throwable, Unit]
}
```

LOOPING

P.S. NOT ACTUALLY USING GLOBALLY MUTABLE STATE!

IMPERATIVE

```
def higherFunction(file: File) = {
  var line: String = file.readLine()

  def whileFun(): Unit =
    if (line != null) {
      buffer.append(line)
      line = file.readLine()
      whileFun()
    } else {
      ()
    }
}
```

```
class File {
  def readLine(): String
  def close(): Unit
}
```

FUNCTIONAL

```
def higherFunction(file: File) = {
  file.readLine().flatMap { line0 =>
    var line = line0
    def whileFun(): ZIO[Any, Throwable, Unit] =
      if (line != null)
        buffer.append(line)
        file.readLine().flatMap { lineN =>
          line = lineN
          whileFun()
        }
      else
        ZIO.unit
  whileFun()
}
```

```
class File {
  def readLine(): ZIO[Any, Throwable, String]
  def close(): ZIO[Any, Throwable, Unit]
}
```

ERROR HANDLING

EXPECT THE (MILDLY) UNEXPECTED

IMPERATIVE

```
var file: JsonFile = JsonFile.from(path)
try {
  file =
  file.readToJson()
} catch {
  case e: IOException       => handleIO(e)
  case e: DecodingException => handleDE(e)
} finally {
  file.close()
}
```

```
class JsonFile {
  def readToJson(): Json
  def close(): Unit
}
object JsonFile {
  def from(path: String): JsonFile
}
```

ERROR HANDLING

EXPECT THE (MILDLY) UNEXPECTED

IMPERATIVE

```
var file: JsonFile = JsonFile.from(path)
try {
  file =
  file.readToJson()
} catch {
  case e: IOException       => handleIO(e)
  case e: DecodingException => handleDE(e)
} finally {
  file.close()
}
```

```
class JsonFile {
  def readToJson(): Json
  def close(): Unit
}
object JsonFile {
  def from(path: String): JsonFile
}
```

FUNCTIONAL

```
succeed(JsonFile.from(path)).flatMap { jsonFile =>
  jsonFile.readToJson()
}.catchSome {
  case e: IOException       => handleIO(e)
  case e: DecodingException => handleDE(e)
}.ensuring {
  jsonFile.close().orDie
}
```

```
class JsonFile {
  def readToJson(): ZIO[Any, Throwable, Json]
  def close(): ZIO[Any, Throwable, Unit]
}
object JsonFile {
  def from(path: String): JsonFile
}
```

Once upon a time... IN JAVA 101

```
val textA = read(fileA)
val textB = read(fileB)
write(fileC, textA + textB)
```

```
if (
    db.transactionsEnabled() &&
    db.lazyFetchEnabled()
)
  db.bulkInsert(rows)
}
```

```
while (line != null) {
  buffer.append(line)
  line = file.readLine()
}
```

```
try {
  file = JsonFile.open(path)
  file.readToJson()
} catch {
  case e: IOException       => handleIO(e)
  case e: DecodingException => handleDE(e)
} finally {
  file.close()
}
```

```
val textA = read(fileA)
val textB = read(fileB)
write(fileC, textA + textB)
```

```
if (
    db.transactionsEnabled() &&
    db.lazyFetchEnabled()
)
  db.bulkInsert(rows)
}
```

```
while (line != null) {
  buffer.append(line)
  line = file.readLine()
}
```

```
val file = JsonFile.from(path)
try {
  file.readToJson()
} catch {
  case e: IOException       => handleIO(e)
  case e: DecodingException => handleDE(e)
} finally {
  file.close()
}
```

BUT THESE DON'T SCALE

```
val textA = read(fileA)
val textB = read(fileB)
write(fileC, textA + textB)
```

```
if (
    db.transactionsEnabled() &&
    db.lazyFetchEnabled()
)
  db.bulkInsert(rows)
}
```

```
while (line != null) {
  buffer.append(line)
  line = file.readLine()
}
```

```
val file = JsonFile.from(path)
try {
  file.readToJson()
} catch {
  case e: IOException       => handleIO(e)
  case e: DecodingException => handleDE(e)
} finally {
  file.close()
}
```

BUT THESE DON'T SCALE

SO WE NEED THESE...

```
for {
  textA <- read(fileA)
  textB <- read(fileB)
  _ <- write(fileC, textA + textB)
} yield ()
```

```
db.transactionsEnabled().flatMap { te =>
  if (te)
    db.lazyFetchEnabled().flatMap { lf =>
      if (lf)
        db.bulkInsert(rows)
      else
        ZIO.unit
    }
  else
    ZIO.unit
}
```

```
file.readLine().flatMap { line0 =>
  var line = line0
  def whileFun(): ZIO[Any, Throwable, Unit] =
    if (line != null)
      buffer.append(line)
      file.readLine().flatMap { lineN =>
        line = lineN
        whileFun()
      }
    else
      ZIO.unit
  whileFun()
```

```
succeed(JsonFile.from(path)).flatMap { jsonFile =>
  jsonFile.readToJson()
}.catchSome {
  case e: IOException       => handleIO(e)
  case e: DecodingException => handleDE(e)
}.ensuring {
  jsonFile.close().orDie
}
```

```
val textA = read(fileA)
val textB = read(fileB)
write(fileC, textA + textB)
```

```
if (
    db.transactionsEnabled() &&
    db.lazyFetchEnabled()
)
  db.bulkInsert(rows)
}
```

```
while (line != null) {
  buffer.append(line)
  line = file.readLine()
}
```

```
try {
  file = JsonFile.open(path)
  file.readToJson()
} catch {
  case e: IOException       => handleIO(e)
  case e: DecodingException => handleDE(e)
} finally {
  file.close()
}
```

BUT THESE DON'T SCALE

OR DO THEY?

```
for {
  textA <- read(fileA)
  textB <- read(fileB)
  _ <- write(fileC, textA + textB)
} yield ()
```

```
db.transactionsEnabled().flatMap { te =>
  if (te)
    db.lazyFetchEnabled().flatMap { lf =>
      if (lf)
        db.bulkInsert(rows)
      else
        ZIO.unit
    }
  else
    ZIO.unit
}
```

```
file.readLine().flatMap { line0 =>
  var line = line0
  def whileFun(): ZIO[Any, Throwable, Unit] =
    if (line != null)
      buffer.append(line)
      file.readLine().flatMap { lineN =>
        line = lineN
        whileFun()
      }
    else
      ZIO.unit
  whileFun()
```

```
succeed(JsonFile.from(path)).flatMap { jsonFile =>
  jsonFile.readToJson()
}.catchSome {
  case e: IOException       => handleIO(e)
  case e: DecodingException => handleDE(e)
}.ensuring {
  jsonFile.close().orDie
}
```

```
val textA = read(fileA)
val textB = read(fileB)
write(fileC, textA + textB)
```

```
if (
    db.transactionsEnabled() &&
    db.lazyFetchEnabled()
)
  db.bulkInsert(rows)
}
```

```
while (line != null) {
  buffer.append(line)
  line = file.readLine()
}
```

```
try {
  file = JsonFile.open(path)
  file.readToJson()
} catch {
  case e: IOException       => handleIO(e)
  case e: DecodingException => handleDE(e)
} finally {
  file.close()
}
```


```
for {
  textA <- read(fileA)
  textB <- read(fileB)
  _ <- write(fileC, textA + textB)
} yield ()
```

```
db.transactionsEnabled().flatMap { te =>
  if (te)
    db.lazyFetchEnabled().flatMap { lf =>
      if (lf)
        db.bulkInsert(rows)
      else
        ZIO.unit
    }
  else
    ZIO.unit
}
```

```
file.readLine().flatMap { line0 =>
  var line = line0
  def whileFun(): ZIO[Any, Throwable, Unit] =
    if (line != null)
      buffer.append(line)
      file.readLine().flatMap { lineN =>
        line = lineN
        whileFun()
      }
    else
      ZIO.unit
  whileFun()
```

```
succeed(JsonFile.from(path)).flatMap { jsonFile =>
  jsonFile.readToJson()
}.catchSome {
  case e: IOException       => handleIO(e)
  case e: DecodingException => handleDE(e)
}.ensuring {
  jsonFile.close().orDie
}
```

```
val textA = read(fileA)
val textB = read(fileB)
write(fileC, textA + textB)
```

```
if (
    db.transactionsEnabled() &&
    db.lazyFetchEnabled()
)
  db.bulkInsert(rows)
}
```

```
while (line != null) {
  buffer.append(line)
  line = file.readLine()
}
```

```
try {
  file = JsonFile.open(path)
  file.readToJson()
} catch {
  case e: IOException       => handleIO(e)
  case e: DecodingException => handleDE(e)
} finally {
  file.close()
}
```

Imperative coding is easy.

Functional code is scaleable.

Write imperative style, translate to Functional!

```
for {
  textA <- read(fileA)
  textB <- read(fileB)
  _ <- write(fileC, textA + textB)
} yield ()
```

```
db.transactionsEnabled().flatMap { te =>
  if (te)
    db.lazyFetchEnabled().flatMap { lf =>
      if (lf)
        db.bulkInsert(rows)
      else
        ZIO.unit
    }
  else
    ZIO.unit
}
```

```
file.readLine().flatMap { line0 =>
  var line = line0
  def whileFun(): ZIO[Any, Throwable, Unit] =
    if (line != null)
      buffer.append(line)
      file.readLine().flatMap { lineN =>
        line = lineN
        whileFun()
      }
    else
      ZIO.unit
  whileFun()
```

```
succeed(JsonFile.from(path)).flatMap { jsonFile =>
  jsonFile.readToJson()
}.catchSome {
  case e: IOException       => handleIO(e)
  case e: DecodingException => handleDE(e)
}.ensuring {
  jsonFile.close().orDie
}
```

```
val textA = read(fileA)
val textB = read(fileB)
write(fileC, textA + textB)
```

```
if (
    db.transactionsEnabled() &&
    db.lazyFetchEnabled()
)
  db.bulkInsert(rows)
}
```

```
while (line != null) {
  buffer.append(line)
  line = file.readLine()
}
```

```
try {
  file = JsonFile.open(path)
  file.readToJson()
} catch {
  case e: IOException       => handleIO(e)
  case e: DecodingException => handleDE(e)
} finally {
  file.close()
}
```

Monadless

```
for {
  textA <- read(fileA)
  textB <- read(fileB)
  _ <- write(fileC, textA + textB)
} yield ()
```

```
db.transactionsEnabled().flatMap { te =>
  if (te)
    db.lazyFetchEnabled().flatMap { lf =>
      if (lf)
        db.bulkInsert(rows)
      else
        ZIO.unit
    }
  else
    ZIO.unit
}
```

```
file.readLine().flatMap { line0 =>
  var line = line0
  def whileFun(): ZIO[Any, Throwable, Unit] =
    if (line != null)
      buffer.append(line)
      file.readLine().flatMap { lineN =>
        line = lineN
        whileFun()
      }
    else
      ZIO.unit
  whileFun()
```

```
succeed(JsonFile.from(path)).flatMap { jsonFile =>
  jsonFile.readToJson()
}.catchSome {
  case e: IOException       => handleIO(e)
  case e: DecodingException => handleDE(e)
}.ensuring {
  jsonFile.close().orDie
}
```

RE-WRITING SEQUENCES

FROM THIS... TO THIS!

IMPERATIVE

```
val textA = read(fileA)
val textB = read(fileB)
write(fileC, textA + textB)
```

```
def read(file: File): String
def write(file: File, content: String): Unit
```

FUNCTIONAL

```
read(fileA).flatMap { textA =>
  read(fileB).flatMap { textB =>
    write(fileC, textA + textB)
  }
}
```

```
def read(file: File):
  ZIO[Any, Throwable, String]
def write(file: File, content: String):
  ZIO[Any, Throwable, Unit]
```

RE-WRITING SEQUENCES

MAKE TO LOOK LIKE IMPERATIVE

IMPERATIVE

```
val textA = read(fileA)
val textB = read(fileB)
write(fileC, textA + textB)
```

```
def read(file: File): String
def write(file: File, content: String): Unit
```

```
val textA = read(fileA)
val textB = read(fileB)
write(fileC, textA + textB)
```

FUNCTIONAL

```
read(fileA).flatMap { textA =>
  read(fileB).flatMap { textB =>
    write(fileC, textA + textB)
  }
}
```

```
def read(file: File):
  ZIO[Any, Throwable, String]
def write(file: File, content: String):
  ZIO[Any, Throwable, Unit]
```

RE-WRITING SEQUENCES

...BUT TYPE LIKE FUNCTIONAL

IMPERATIVE

```
val textA = read(fileA)
val textB = read(fileB)
write(fileC, textA + textB)
```

```
def read(file: File): String
def write(file: File, content: String): Unit
```

```
val textA = read(fileA) :ZIO[..String]
val textB = read(fileB) :ZIO[..String]
write(fileC, textA + textB) :ZIO[..Unit]
```

FUNCTIONAL

```
read(fileA).flatMap { textA =>
  read(fileB).flatMap { textB =>
    write(fileC, textA + textB)
  }
}
```

```
def read(file: File):
  ZIO[Any, Throwable, String]
def write(file: File, content: String):
  ZIO[Any, Throwable, Unit]
```

RE-WRITING SEQUENCES

PRESTO-CHANGE-OH

IMPERATIVE

```
val textA = read(fileA)
val textB = read(fileB)
write(fileC, textA + textB)
```

```
def read(file: File): String
def write(file: File, content: String): Unit
```

```
val textA = run(read(fileA)):String
val textB = run(read(fileB)):String
run(write(fileC, textA + textB)):Unit
```

FUNCTIONAL

```
read(fileA).flatMap { textA =>
  read(fileB).flatMap { textB =>
    write(fileC, textA + textB)
  }
}
```

```
def read(file: File):
  ZIO[Any, Throwable, String]
def write(file: File, content: String):
  ZIO[Any, Throwable, Unit]
```

RE-WRITING SEQUENCES

THE RABIT IN THE HAT

IMPERATIVE

```
val textA = read(fileA)
val textB = read(fileB)
write(fileC, textA + textB)
```

```
def read(file: File): String
def write(file: File, content: String): Unit
```

```
val textA = run(read(fileA)):String
val textB = run(read(fileB)):String
run(write(fileC, textA + textB)):Unit
```

FUNCTIONAL

```
read(fileA).flatMap { textA =>
  read(fileB).flatMap { textB =>
    write(fileC, textA + textB)
  }
}
```

```
def read(file: File):
  ZIO[Any, Throwable, String]
def write(file: File, content: String):
  ZIO[Any, Throwable, Unit]
```

RE-WRITING SEQUENCES

RUN MACRO, WRAP IN A BOW

IMPERATIVE

```
val textA = read(fileA)
val textB = read(fileB)
write(fileC, textA + textB)
```

```
def read(file: File): String
def write(file: File, content: String): Unit
```

```
defer {
  val textA = run(read(fileA))
  val textB = run(read(fileB))
  run(write(fileC, textA + textB))
}
```

FUNCTIONAL

```
read(fileA).flatMap { textA =>
  read(fileB).flatMap { textB =>
    write(fileC, textA + textB)
  }
}
```

```
def read(file: File):
  ZIO[Any, Throwable, String]
def write(file: File, content: String):
  ZIO[Any, Throwable, Unit]
```

RE-WRITING SEQUENCES

RUN MACRO, WRAP IN A BOW

IMPERATIVE

```
val textA = read(fileA)
val textB = read(fileB)
write(fileC, textA + textB)
```

```
def read(file: File): String
def write(file: File, content: String): Unit
```

```
defer {
  val textA = run(read(fileA))
  val textB = run(read(fileB))
  run(write(fileC, textA + textB))
}:ZIO[Any, Throwable, Unit]
```

FUNCTIONAL

```
read(fileA).flatMap { textA =>
  read(fileB).flatMap { textB =>
    write(fileC, textA + textB)
  }
}
```

```
def read(file: File):
  ZIO[Any, Throwable, String]
def write(file: File, content: String):
  ZIO[Any, Throwable, Unit]
```

RE-WRITING SEQUENCES

THAT'S ALL FOLKS

IMPERATIVE

```
val textA = read(fileA)
val textB = read(fileB)
write(fileC, textA + textB)
```

```
def read(file: File): String
def write(file: File, content: String): Unit
```

```
defer {
  val textA = run(read(fileA))
  val textB = run(read(fileB))
  run(write(fileC, textA + textB))
}
```

FEEL IMPERATIVE

ACT FUNCTIONALLY

FUNCTIONAL

```
read(fileA).flatMap { textA =>
  read(fileB).flatMap { textB =>
    write(fileC, textA + textB)
  }
}
```

```
def read(file: File):
  ZIO[Any, Throwable, String]
def write(file: File, content: String):
  ZIO[Any, Throwable, Unit]
```

RE-WRITING SEQUENCES

P.S. BETTER LIKE THIS

IMPERATIVE

```
val textA = read(fileA)
val textB = read(fileB)
write(fileC, textA + textB)
```

```
def read(file: File): String
def write(file: File, content: String): Unit
```

```
defer {
  val textA = read(fileA).run
  val textB = read(fileB).run
  write(fileC, textA + textB).run
}
```

FEEL IMPERATIVE

ACT FUNCTIONALLY

FUNCTIONAL

```
read(fileA).flatMap { textA =>
  read(fileB).flatMap { textB =>
    write(fileC, textA + textB)
  }
}
```

```
def read(file: File):
  ZIO[Any, Throwable, String]
def write(file: File, content: String):
  ZIO[Any, Throwable, Unit]
```

RE-WRITING BRANCHING

START WITH IMPERATIVE STYLE

IMPERATIVE

```
val db = Database.open()
if (
    db.transactionsEnabled() &&
    db.lazyFetchEnabled()
) {
  db.bulkInsert(rows)
}
```

```
class Database {
  def transactionsEnabled(): Boolean
  def lazyFetchEnabled(): Boolean
  def bulkInsert(row: List[Row]): Unit
}
```

```
val db = Database.open()
if (
    db.transactionsEnabled() &&
    db.lazyFetchEnabled()
) {
  db.bulkInsert(rows)
}
```

FUNCTIONAL

```
Database.open().flatMap { db =>
  db.transactionsEnabled().flatMap { te =>
    if (te)
      db.lazyFetchEnabled().flatMap { lf =>
        if (lf)
          db.bulkInsert(rows)
        else
          ZIO.unit
      }
    else
      ZIO.unit
  }
}
```

```
class Database {
  def transactionsEnabled(): ZIO[Any, Throwable, Boolean]
  def lazyFetchEnabled(): ZIO[Any, Throwable, Boolean]
  def bulkInsert(row: List[Row]): ZIO[Any, Throwable, Unit]
}
```

RE-WRITING BRANCHING

START WITH IMPERATIVE STYLE... THEN DEFER AND RUN!

IMPERATIVE

```
val db = Database.open()
if (
    db.transactionsEnabled() &&
    db.lazyFetchEnabled()
) {
  db.bulkInsert(rows)
}
```

```
class Database {
  def transactionsEnabled(): Boolean
  def lazyFetchEnabled(): Boolean
  def bulkInsert(row: List[Row]): Unit
}
```

```
defer {
  val db = Database.open().run
  if (
      db.transactionsEnabled().run &&
      db.lazyFetchEnabled().run
  ) {
    db.bulkInsert(rows).run
  }
}
```

FEEL IMPERATIVE

ACT FUNCTIONALLY

FUNCTIONAL

```
Database.open().flatMap { db =>
  db.transactionsEnabled().flatMap { te =>
    if (te)
      db.lazyFetchEnabled().flatMap { lf =>
        if (lf)
          db.bulkInsert(rows)
        else
          ZIO.unit
      }
    else
      ZIO.unit
  }
}
```

```
class Database {
  def transactionsEnabled(): ZIO[Any, Throwable, Boolean]
  def lazyFetchEnabled(): ZIO[Any, Throwable, Boolean]
  def bulkInsert(row: List[Row]): ZIO[Any, Throwable, Unit]
}
```

RE-WRITING BRANCHING

START WITH IMPERATIVE STYLE... THEN DEFER AND RUN!

IMPERATIVE

```
val db = Database.open()
if (
    db.transactionsEnabled() &&
    db.lazyFetchEnabled()
) {
  db.bulkInsert(rows)
}
```

```
class Database {
  def transactionsEnabled(): Boolean
  def lazyFetchEnabled(): Boolean
  def bulkInsert(row: List[Row]): Unit
}
```

```
defer {
  val db = Database.open().run
  if (
      db.transactionsEnabled().run &&
      db.lazyFetchEnabled().run
  ) {
    db.bulkInsert(rows).run
  }
}
```

FEEL IMPERATIVE

ACT FUNCTIONALLY

FUNCTIONAL

```
Database.open().flatMap { db =>
  db.transactionsEnabled().flatMap { te =>
    if (te)
      db.lazyFetchEnabled().flatMap { lf =>
        if (lf)
          db.bulkInsert(rows)
        else
          ZIO.unit
      }
    else
      ZIO.unit
  }
}
```

```
class Database {
  def transactionsEnabled(): ZIO[Any, Throwable, Boolean]
  def lazyFetchEnabled(): ZIO[Any, Throwable, Boolean]
  def bulkInsert(row: List[Row]): ZIO[Any, Throwable, Unit]
}
```

RE-WRITING LOOPING

START WITH IMPERATIVE STYLE... THEN DEFER AND RUN!

IMPERATIVE

```
var line: String = file.readLine()
while (line != null) {
  buffer.append(line)
  line = file.readLine()
}
```

```
class File {
  def readLine(): String
  def close(): Unit
}
```

FEEL IMPERATIVE

ACT FUNCTIONALLY

FUNCTIONAL

```
file.readLine().flatMap { line0 =>
  var line = line0
  def whileFun(): ZIO[Any, Throwable, Unit] =
    if (line != null)
      buffer.append(line)
      file.readLine().flatMap { lineN =>
        line = lineN
        whileFun()
      }
    else
      ZIO.unit
  whileFun()
```

```
class File {
  def readLine(): ZIO[Any, Throwable, String]
  def close(): ZIO[Any, Throwable, Unit]
}
```

RE-WRITING LOOPING

START WITH IMPERATIVE STYLE... THEN DEFER AND RUN!

IMPERATIVE

```
var line: String = file.readLine()
while (line != null) {
  buffer.append(line)
  line = file.readLine()
}
```

```
class File {
  def readLine(): String
  def close(): Unit
}
```

```
var line: String = file.readLine()
while (line != null) {
  buffer.append(line)
  line = file.readLine()
}
```

FEEL IMPERATIVE

ACT FUNCTIONALLY

FUNCTIONAL

```
file.readLine().flatMap { line0 =>
  var line = line0
  def whileFun(): ZIO[Any, Throwable, Unit] =
    if (line != null)
      buffer.append(line)
      file.readLine().flatMap { lineN =>
        line = lineN
        whileFun()
      }
    else
      ZIO.unit
  whileFun()
```

```
class File {
  def readLine(): ZIO[Any, Throwable, String]
  def close(): ZIO[Any, Throwable, Unit]
}
```

RE-WRITING LOOPING

START WITH IMPERATIVE STYLE... THEN DEFER AND RUN!

IMPERATIVE

```
var line: String = file.readLine()
while (line != null) {
  buffer.append(line)
  line = file.readLine()
}
```

```
class File {
  def readLine(): String
  def close(): Unit
}
```

```
defer {
  var line: String = file.readLine().run
  while (line != null) {
    buffer.append(line)
    line = file.readLine().run
  }
}
```

FEEL IMPERATIVE

ACT FUNCTIONALLY

FUNCTIONAL

```
file.readLine().flatMap { line0 =>
  var line = line0
  def whileFun(): ZIO[Any, Throwable, Unit] =
    if (line != null)
      buffer.append(line)
      file.readLine().flatMap { lineN =>
        line = lineN
        whileFun()
      }
    else
      ZIO.unit
  whileFun()
```

```
class File {
  def readLine(): ZIO[Any, Throwable, String]
  def close(): ZIO[Any, Throwable, Unit]
}
```

RE-WRITING LOOPING

START WITH IMPERATIVE STYLE... THEN DEFER AND RUN!

IMPERATIVE

```
var line: String = file.readLine()
while (line != null) {
  buffer.append(line)
  line = file.readLine()
}
```

```
class File {
  def readLine(): String
  def close(): Unit
}
```

```
defer {
  val line0 = file.readLine().run
  var line: String = line0
  while (line != null) {
    buffer.append(line)
    val lineN = file.readLine().run
    line = lineN
  }
}
```

FEEL IMPERATIVE

ACT FUNCTIONALLY

FUNCTIONAL

```
file.readLine().flatMap { line0 =>
  var line = line0
  def whileFun(): ZIO[Any, Throwable, Unit] =
    if (line != null)
      buffer.append(line)
      file.readLine().flatMap { lineN =>
        line = lineN
        whileFun()
      }
    else
      ZIO.unit
  whileFun()
```

```
class File {
  def readLine(): ZIO[Any, Throwable, String]
  def close(): ZIO[Any, Throwable, Unit]
}
```

RE-WRITING LOOPING

START WITH IMPERATIVE STYLE... THEN DEFER AND RUN!

IMPERATIVE

```
var line: String = file.readLine()
while (line != null) {
  buffer.append(line)
  line = file.readLine()
}
```

```
class File {
  def readLine(): String
  def close(): Unit
}
```

```
defer {
  val line0 = file.readLine().run
  var line: String = line0
  while (line != null) {
    buffer.append(line)
    val lineN = file.readLine().run
    line = lineN
  }
}
```

FEEL IMPERATIVE

ACT FUNCTIONALLY

FUNCTIONAL

```
file.readLine().flatMap { line0 =>
  var line = line0
  def whileFun(): ZIO[Any, Throwable, Unit] =
    if (line != null)
      buffer.append(line)
      file.readLine().flatMap { lineN =>
        line = lineN
        whileFun()
      }
    else
      ZIO.unit
  whileFun()
```

```
class File {
  def readLine(): ZIO[Any, Throwable, String]
  def close(): ZIO[Any, Throwable, Unit]
}
```

RE-WRITING LOOPING

START WITH IMPERATIVE STYLE... THEN DEFER AND RUN!

IMPERATIVE

```
var line: String = file.readLine()
while (line != null) {
  buffer.append(line)
  line = file.readLine()
}
```

```
class File {
  def readLine(): String
  def close(): Unit
}
```

```
defer {
  val line0 = file.readLine().run
  var line: String = line0
  while (line != null) {
    buffer.append(line)
    val lineN = file.readLine().run
    line = lineN
  }
}
```

FEEL IMPERATIVE

ACT FUNCTIONALLY

FUNCTIONAL

```
file.readLine().flatMap { line0 =>
  var line = line0
  def whileFun(): ZIO[Any, Throwable, Unit] =
    if (line != null)
      buffer.append(line)
      file.readLine().flatMap { lineN =>
        line = lineN
        whileFun()
      }
    else
      ZIO.unit
  whileFun()
```

```
class File {
  def readLine(): ZIO[Any, Throwable, String]
  def close(): ZIO[Any, Throwable, Unit]
}
```

RE-WRITING ERROR HANDLING

EXPECT THE (MILDLY) UNEXPECTED

IMPERATIVE

```
val file = JsonFile.from(path)
try {
  file.readToJson()
} catch {
  case e: IOException       => handleIO(e)
  case e: DecodingException => handleDE(e)
} finally {
  file.close()
}
```

```
class JsonFile {
  def readToJson(): Json
  def close(): Unit
}
object JsonFile {
  def open(path: String): JsonFile
}
```

FUNCTIONAL

```
succeed(JsonFile.from(path))
.flatMap { file =>
  file.readToJson().catchSome {
  case e: IOException       => handleIO(e)
  case e: DecodingException => handleDE(e)
 }.ensuring {
  file.close().orDie
}
```

```
class JsonFile {
  def readToJson(): ZIO[Any, Throwable, Json]
  def close(): ZIO[Any, Throwable, Unit]
}
object JsonFile {
  def from(path: String): JsonFile
}
```

RE-WRITING ERROR HANDLING

EXPECT THE (MILDLY) UNEXPECTED

IMPERATIVE

```
val file = JsonFile.from(path)
try {
  file.readToJson()
} catch {
  case e: IOException       => handleIO(e)
  case e: DecodingException => handleDE(e)
} finally {
  file.close()
}
```

```
class JsonFile {
  def readToJson(): Json
  def close(): Unit
}
object JsonFile {
  def open(path: String): JsonFile
}
```

```
val file: JsonFile = JsonFile.from(path)
try {
  file.readToJson()
} catch {
  case e: IOException       => handleIO(e)
  case e: DecodingException => handleDE(e)
} finally {
  file.close()
}
```

FUNCTIONAL

```
succeed(JsonFile.from(path))
.flatMap { file =>
  file.readToJson().catchSome {
  case e: IOException       => handleIO(e)
  case e: DecodingException => handleDE(e)
 }.ensuring {
  file.close().orDie
}
```

```
class JsonFile {
  def readToJson(): ZIO[Any, Throwable, Json]
  def close(): ZIO[Any, Throwable, Unit]
}
object JsonFile {
  def from(path: String): JsonFile
}
```

RE-WRITING ERROR HANDLING

EXPECT THE (MILDLY) UNEXPECTED

IMPERATIVE

```
val file = JsonFile.from(path)
try {
  file.readToJson()
} catch {
  case e: IOException       => handleIO(e)
  case e: DecodingException => handleDE(e)
} finally {
  file.close()
}
```

```
class JsonFile {
  def readToJson(): Json
  def close(): Unit
}
object JsonFile {
  def open(path: String): JsonFile
}
```

```
defer {
  val file = JsonFile.from(path)
  try {
    file.readToJson().run
  } catch {
    case e: IOException       => handleIO(e).run
    case e: DecodingException => handleDE(e).run
  } finally {
    file.close().run
  }
}
```

FEEL IMPERATIVE →

→ ACT FUNCTIONALLY

FUNCTIONAL

```
succeed(JsonFile.from(path))
.flatMap { file =>
  file.readToJson().catchSome {
  case e: IOException       => handleIO(e)
  case e: DecodingException => handleDE(e)
 }.ensuring {
  file.close().orDie
}
```

```
class JsonFile {
  def readToJson(): ZIO[Any, Throwable, Json]
  def close(): ZIO[Any, Throwable, Unit]
}
object JsonFile {
  def from(path: String): JsonFile
}
```

RE-WRITING ERROR HANDLING

EXPECT THE (MILDLY) UNEXPECTED

IMPERATIVE

```scala
val file = JsonFile.from(path)
try {
  file.readToJson()
} catch {
  case e: IOException       => handleIO(e)
  case e: DecodingException => handleDE(e)
} finally {
  file.close()
}
```

```scala
class JsonFile {
  def readToJson(): Json
  def close(): Unit
}
object JsonFile {
  def open(path: String): JsonFile
}
```

```scala
defer {
  val file = JsonFile.from(path)
  try {
    file.readToJson().run
  } catch {
    case e: IOException       => handleIO(e).run
    case e: DecodingException => handleDE(e).run
  } finally {
    file.close().run
  }
}
```

FEEL IMPERATIVE →

→ ACT FUNCTIONALLY

FUNCTIONAL

```scala
succeed(JsonFile.from(path))
.flatMap { file =>
  file.readToJson().catchSome {
  case e: IOException       => handleIO(e)
  case e: DecodingException => handleDE(e)
 }.ensuring {
  file.close().orDie
}
```

```scala
class JsonFile {
  def readToJson(): ZIO[Any, Throwable, Json]
  def close(): ZIO[Any, Throwable, Unit]
}
object JsonFile {
  def from(path: String): JsonFile
}
```

```
defer {
  val textA = read(fileA).run
  val textB = read(fileB).run
  write(fileC, textA + textB).run
}
```

```
defer {
  val db = Database.open().run
  if (
      db.transactionsEnabled().run &&
      db.lazyFetchEnabled().run
  ) {
    db.bulkInsert(rows).run
  }
}
```

```
defer {
  val line0 = file.readLine().run
  var line: String = line0
  while (line != null) {
    buffer.append(line)
    val lineN = file.readLine().run
    line = lineN
  }
}
```

```
defer {
  val file = JsonFile.from(path)
  try {
    file.readToJson().run
  } catch {
    case e: IOException       => handleIO(e).run
    case e: DecodingException => handleDE(e).run
  } finally {
    file.close().run
  }
}
```


```
read(fileA).flatMap { textA =>
  read(fileB).flatMap { textB =>
    write(fileC, textA + textB)
  }
}
```

```
db.transactionsEnabled().flatMap { te =>
  if (te)
    db.lazyFetchEnabled().flatMap { lf =>
      if (lf)
        db.bulkInsert(rows)
      else
        ZIO.unit
    }
  else
    ZIO.unit
}
```

```
file.readLine().flatMap { line0 =>
  var line = line0
  def whileFun(): ZIO[Any, Throwable, Unit] =
    if (line != null)
      buffer.append(line)
      file.readLine().flatMap { lineN =>
        line = lineN
        whileFun()
      }
    else
      ZIO.unit
  whileFun()
```

```
succeed(JsonFile.from(path)).flatMap { jsonFile =>
  jsonFile.readToJson()
}.catchSome {
  case e: IOException       => handleIO(e)
  case e: DecodingException => handleDE(e)
}.ensuring {
  jsonFile.close().orDie
}
```

ISN'T THIS ASYNC-AWAIT ?

```
async {
  val textA = read(fileA).await
  val textB = read(fileB).await
  write(fileC, textA + textB).await
}
```

```
async {
  val db = Database.open().await
  if (
      db.transactionsEnabled().await &&
      db.lazyFetchEnabled().await
  ) {
    db.bulkInsert(rows).await
  }
}
```

```
async {
  val line0 = file.readLine().await
  var line: String = line0
  while (line != null) {
    buffer.append(line)
    val lineN = file.readLine().await
    line = lineN
  }
}
```

```
async {
  val file = JsonFile.from(path)
  try {
    file.readToJson().await
  } catch {
    case e: IOException       => handleIO(e).await
    case e: DecodingException => handleDE(e).await
  } finally {
    file.close().await
  }
}
```

Works for Collections!

```
defer {
  val p = List(joe, jack, jill).run
  val a = p.addresses.run
  (p, a)
}
```

Works for Queries!

```
defer {
  val p =
    query[Person].run
  val a =
    query[Address].filter(a.fk == p.id).run
  (p, a)
}
```

no

```
async {
  val textA = read(fileA).await
  val textB = read(fileB).await
  write(fileC, textA + textB).await
}
```

```
async {
  val db = Database.open().await
  if (
      db.transactionsEnabled().await &&
      db.lazyFetchEnabled().await
  ) {
    db.bulkInsert(rows).await
  }
}
```

```
async {
  val line0 = file.readLine().await
  var line: String = line0
  while (line != null) {
    buffer.append(line)
    val lineN = file.readLine().await
    line = lineN
  }
}
```

```
async {
  val file = JsonFile.from(path)
  try {
    file.readToJson().await
  } catch {
    case e: IOException       => handleIO(e).await
    case e: DecodingException => handleDE(e).await
  } finally {
    file.close().await
  }
}
```

Works for Collections!

```
defer {
  val p = List(joe, jack, jill).run
  val a = p.addresses.run
  (p, a)
}
```

Works for Queries!

```
defer {
  val p =
    query[Person].run
  val a =
    query[Address].filter(a.fk == p.id).run
  (p, a)
}
```

Wouldn't Everyone want this?

```
defer {
  val textA = read(fileA).run
  val textB = read(fileB).run
  write(fileC, textA + textB).run
}
```

```
defer {
  val db = Database.open().run
  if (
      db.transactionsEnabled().run &&
      db.lazyFetchEnabled().run
  ) {
    db.bulkInsert(rows).run
  }
}
```

```
defer {
  var line: String = file.readLine().run
  while (line != null) {
    buffer.append(line)
    line = file.readLine().run
  }
}
```

```
defer {
  val file = JsonFile.from(path)
  try {
    file.readToJson().run
  } catch {
    case e: IOException       => handleIO(e).run
    case e: DecodingException => handleDE(e).run
  } finally {
    file.close().run
  }
}
```

Wouldn't Everyone want this?

```
defer {
  val textA = read(fileA).run
  val textB = read(fileB).run
  write(fileC, textA + textB).run
}
```

```
defer {
  val db = Database.open().run
  if (
      db.transactionsEnabled().run &&
      db.lazyFetchEnabled().run
  ) {
    db.bulkInsert(rows).run
  }
}
```

```
defer {
  var line: String = file.readLine().run
  while (line != null) {
    buffer.append(line)
    line = file.readLine().run
  }
}
```

```
defer {
  val file = JsonFile.from(path)
  try {
    file.readToJson().run
  } catch {
    case e: IOException       => handleIO(e).run
    case e: DecodingException => handleDE(e).run
  } finally {
    file.close().run
  }
}
```

ZIO-DIRECT

```
defer {
  val textA = run(read(fileA))
  val textB = run(read(fileB))
  run(write(fileC, textA + textB))
}
```

```
defer {
  val db = Database.open().run
  if (
    db.transactionsEnabled().run &&
    db.lazyFetchEnabled().run
  ) {
    db.bulkInsert(rows).run
  }
}
```

```
defer {
  val line0 = file.readLine().run
  var line: String = line0
  while (line != null) {
    buffer.append(line)
    val lineN = file.readLine().run
    line = lineN
  }
}
```

```
defer {
  val file = JsonFile.from(path)
  try {
    file.readToJson().run
  } catch {
    case e: IOException       => handleIO(e).run
    case e: DecodingException => handleDE(e).run
  } finally {
    file.close().run
  }
}
```

SUPPORTS THESE

and...

```
defer {
  val textA = run(read(fileA))
  val textB = run(read(fileB))
  run(write(fileC, textA + textB))
}
```

```
defer {
  val db = Database.open().run
  if (
    db.transactionsEnabled().run &&
    db.lazyFetchEnabled().run
  ) {
    db.bulkInsert(rows).run
  }
}
```

```
defer {
  val line0 = file.readLine().run
  var line: String = line0
  while (line != null) {
    buffer.append(line)
    val lineN = file.readLine().run
    line = lineN
  }
}
```

```
defer {
  val file = JsonFile.from(path)
  try {
    file.readToJson().run
  } catch {
    case e: IOException       => handleIO(e).run
    case e: DecodingException => handleDE(e).run
  } finally {
    file.close().run
  }
}
```

FOR-LOOP

```
for (site <- getWebsites()) {
  postUpdate(site)
}
```

IMPERATIVE

```
getWebsites()
  .flatMap { websites =>
    ZIO.foreach(websites)(
      postUpdate(_)
    )
  }
```

TRANSLATION

```
defer {
  val textA = run(read(fileA))
  val textB = run(read(fileB))
  run(write(fileC, textA + textB))
}
```

```
defer {
  val db = Database.open().run
  if (
    db.transactionsEnabled().run &&
    db.lazyFetchEnabled().run
  ) {
    db.bulkInsert(rows).run
  }
}
```

```
defer {
  val line0 = file.readLine().run
  var line: String = line0
  while (line != null) {
    buffer.append(line)
    val lineN = file.readLine().run
    line = lineN
  }
}
```

```
defer {
  val file = JsonFile.from(path)
  try {
    file.readToJson().run
  } catch {
    case e: IOException       => handleIO(e).run
    case e: DecodingException => handleDE(e).run
  } finally {
    file.close().run
  }
}
```

FOR

```
defer {
  for (site <- getWebsites().run) {
    postUpdate(site).run
  }
}
```

IMPERATIVE

```
getWebsites()
  .flatMap { websites =>
    ZIO.foreach(websites)(
      postUpdate(_)
    )
  }
```

TRANSLATION

```
defer {
  val textA = run(read(fileA))
  val textB = run(read(fileB))
  run(write(fileC, textA + textB))
}
```

```
defer {
  val db = Database.open().run
  if (
    db.transactionsEnabled().run &&
    db.lazyFetchEnabled().run
  ) {
    db.bulkInsert(rows).run
  }
}
```

```
defer {
  val line0 = file.readLine().run
  var line: String = line0
  while (line != null) {
    buffer.append(line)
    val lineN = file.readLine().run
    line = lineN
  }
}
```

```
defer {
  val file = JsonFile.from(path)
  try {
    file.readToJson().run
  } catch {
    case e: IOException       => handleIO(e).run
    case e: DecodingException => handleDE(e).run
  } finally {
    file.close().run
  }
}
```

FOR

```
defer(Params(Collect.Parallel)) {
  for (site <- getWebsites().run) {
    postUpdate(site).run
  }
}
```

IMPERATIVE

```
getWebsites()
  .flatMap { websites =>
    ZIO.foreachPar(websites)(
      postUpdate(_)
    )
  }
```

TRANSLATION

Direct Programming tailored to ZIO

ZIO-DIRECT

1

ZIO-Tailored Features

2

Transparency of Rewrite Logic

3

$\{x \iff y\}$

Correctness

4

Error Channels

5

Using Refs

Direct Programming tailored to ZIO | ZIO-DIRECT

```
val out: ZIO[CustomerConfig & DistributorConfig, Throwable, (Customer, Distributor)] =
  defer {
    val custUrl: String = ZIO.service[CustomerConfig].run.url
    val distUrl: String = ZIO.service[DistributorConfig].run.url
    (parseCustomer(httpGet(custUrl).run), parseDistrubutor(httpGet(distUrl).run))
  }
```

```
def httpGet(url: String): ZIO[Any, Throwable, String]

def parseCustomer(customer: String): Customer
def parseDistrubutor(customer: String): Distributor

case class CustomerConfig(url: String)
case class DistributorConfig(url: String)
```

```scala
val out: ZIO[CustomerConfig & DistributorConfig, Throwable, (Customer, Distributor)] =
  defer {
    val custUrl: String = ZIO.service[CustomerConfig].run.url
    val distUrl: String = ZIO.service[DistributorConfig].run.url
    (parseCustomer(httpGet(custUrl).run), parseDistrubutor(httpGet(distUrl).run))
  }
```

```scala
def httpGet(url: String): ZIO[Any, Throwable, String]

def parseCustomer(customer: String): Customer
def parseDistrubutor(customer: String): Distributor

case class CustomerConfig(url: String)
case class DistributorConfig(url: String)
```

API

```scala
val out: ZIO[CustomerConfig & DistributorConfig, IOException, (Customer, Distributor)] =
  defer {
    val custUrl: String = ZIO.service[CustomerConfig].run.url
    val distUrl: String = ZIO.service[DistributorConfig].run.url
    (parseCustomer(httpGet(custUrl).run), parseDistrubutor(httpGet(distUrl).run))
  }
```

```scala
def httpGet(url: String): ZIO[Any, IOException, String]

def parseCustomer(customer: String): Customer
def parseDistrubutor(customer: String): Distributor

case class CustomerConfig(url: String)
case class DistributorConfig(url: String)
```

API

```
val out: ZIO[..., CustomerGetException | DistributorGetException, (...)] =
  defer {
    val custUrl: String = ZIO.service[CustomerConfig].run.url
    val distUrl: String = ZIO.service[DistributorConfig].run.url
    (parseCustomer(httpGetCustomer(custUrl).run), parseDistrubutor(httpGetDistributor(distUrl).run))
  }
```

```
def httpGetCustomer(url: String): ZIO[Any, CustomerGetException, String]
def httpGetDistributor(url: String): ZIO[Any, DistrubutorGetException, String]

def parseCustomer(customer: String): Customer
def parseDistrubutor(customer: String): Distributor

case class CustomerConfig(url: String)
case class DistributorConfig(url: String)
```

API

```scala
val out: ZIO[..., Exception, (...)] =
  defer(Params(TypeUnion.LeastUpper)) {
    val custUrl: String = ZIO.service[CustomerConfig].run.url
    val distUrl: String = ZIO.service[DistributorConfig].run.url
    (parseCustomer(httpGetCustomer(custUrl).run), parseDistrubutor(httpGetDistributor(distUrl).run))
  }
```

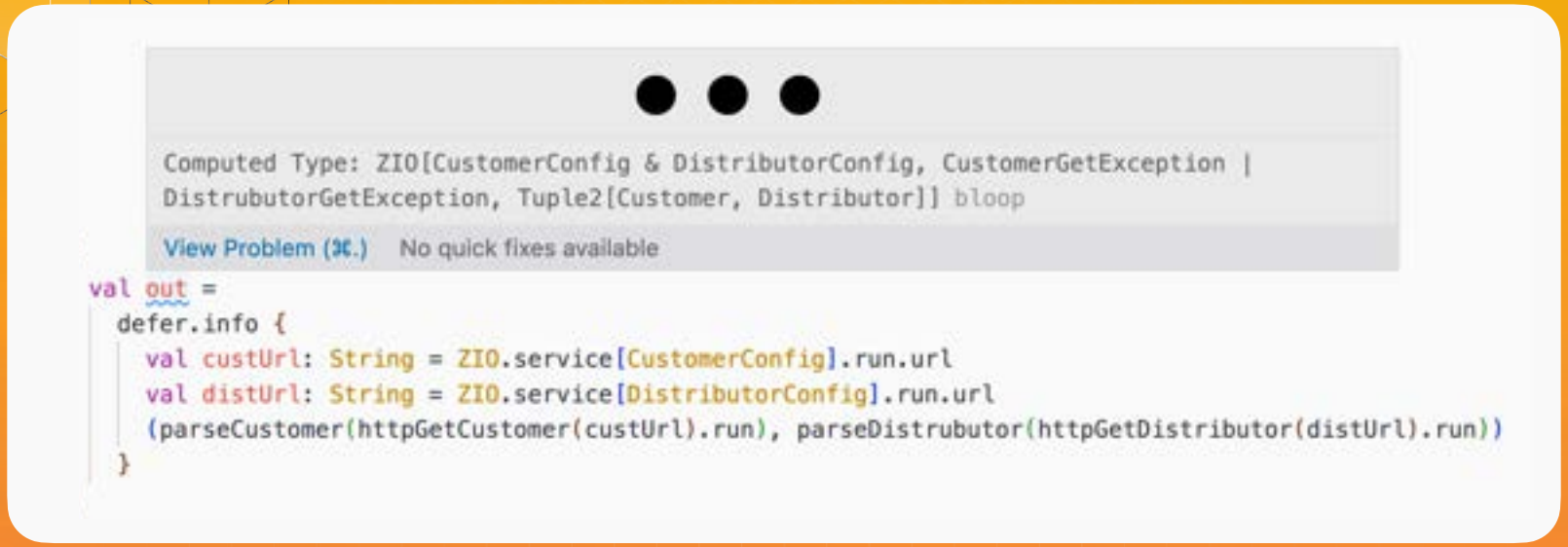
```scala
def httpGetCustomer(url: String): ZIO[Any, CustomerGetException, String]
def httpGetDistributor(url: String): ZIO[Any, DistrubutorGetException, String]

def parseCustomer(customer: String): Customer
def parseDistrubutor(customer: String): Distributor

case class CustomerConfig(url: String)
case class DistributorConfig(url: String)
```

API

Want to see it up close? | ZIO-DIRECT

```
val out: ZIO[CustomerConfig & DistributorConfig, CustomerGetException |
DistrubutorGetException, (Customer, Distributor)]
val out =
  defer {
    val custUrl: String = ZIO.service[CustomerConfig].run.url
    val distUrl: String = ZIO.service[DistributorConfig].run.url
    (parseCustomer(httpGetCustomer(custUrl).run), parseDistrubutor(httpGetDistributor(distUrl).run))
  }
```

Use defer.info | ZIO-DIRECT




```
root> compile
...
[info] -- Info: /Users/me/zio-direct/src/test/scala-3.x/zio/direct/examples/TailoredForZio.scala:40:8
[info] 40 |    val out =
[info]    |        ^
[info]    |Computed Type: ZIO[CustomerConfig & DistributorConfig, CustomerGetException | DistrubutorGetException, Tuple2[Customer, Distributor]]
```

Use defer.info | ZIO-DIRECT

```
val out: ZIO[CustomerConfig & DistributorConfig, CustomerGetException |
DistrubutorGetException, (Customer, Distributor)]

Computed Type: ZIO[CustomerConfig & DistributorConfig, CustomerGetException |
DistrubutorGetException, Tuple2[Customer, Distributor]] bloop

View Problem (⌘.)    No quick fixes available
```

```
val out =
  defer.info {
    val custUrl: String = ZIO.service[CustomerConfig].run.url
    val distUrl: String = ZIO.service[DistributorConfig].run.url
    (parseCustomer(httpGetCustomer(custUrl).run), parseDistrubutor(httpGetDistributor(distUrl).run))
  }
```

```
root> compile
...
[info] -- Info: /Users/me/zio-direct/src/test/scala-3.x/zio/direct/examples/TailoredForZio.scala:40:8
[info] 40 |    val out =
[info]    |        ^
[info]    |Computed Type: ZIO[CustomerConfig & DistributorConfig, CustomerGetException | DistrubutorGetException, Tuple2[Customer, Distributor]]
```

Use **defer.info** or defer.verbose | ZIO-DIRECT

```scala
val out =
  defer.info {
    val custUrl: String = ZIO.service[CustomerConfig].run.url
    val distUrl: String = ZIO.service[DistributorConfig].run.url
    (parseCustomer(httpGetCustomer(custUrl).run), parseDistrubutor(httpGetDistributor(distUrl).run))
  }
```

```
root> compile
=== Reconstituted Code ========
  ZIO.service[CustomerConfig].map(((sm: CustomerConfig) => {
    val runVal: CustomerConfig = sm
    runVal.url
  })).flatMap(((v: String) => {
    val custUrl: String = v
    ZIO.service[DistributorConfig].map(((`sm₂`: DistributorConfig) => {
      val `runVal₂`: DistributorConfig = `sm₂`
      `runVal₂`.url
    })).flatMap(((`v₂`: String) => {
      val distUrl: String = `v₂`
      ZIO.collectAll(Chunk.from(List.apply(httpGetCustomer(custUrl), httpGetDistributor(distUrl)))).map(((terms: Chunk[Any]) => {
        val iter: Iterator[Any] = terms.iterator
        {
          val `runVal₃`: String = iter.next
          val `runVal₄`: String = iter.next
          Tuple2.apply(parseCustomer(`runVal₃`), parseDistrubutor(`runVal₄`))
        }
      }))
    }))
  }))
```

Use **defer.info** or defer.verbose | ZIO-DIRECT

```scala
val out =
  defer.info {
    val custUrl: String = ZIO.service[CustomerConfig].run.url
    val distUrl: String = ZIO.service[DistributorConfig].run.url
    (parseCustomer(httpGetCustomer(custUrl).run), parseDistrubutor(httpGetDistributor(distUrl).run))
  }
```

```
root> compile
=== Reconstituted Code ========
ZIO.service[CustomerConfig].map(((sm: CustomerConfig) => {
  val runVal: CustomerConfig = sm
  runVal.url
})).flatMap(((v: String) => {
  val custUrl: String = v
  ZIO.service[DistributorConfig].map(((`sm₂`: DistributorConfig) => {
    val `runVal₃`: DistributorConfig = `sm₂`
    `runVal₃`.url
  })).flatMap(((`v₂`: String) => {
    val distUrl: String = `v₂`
    ZIO.collectAll(Chunk.from(List.apply(httpGet
      val iter: Iterator[Any] = terms.iterator
      {
        val `runVal₅`: String = iter.next
        val `runVal₆`: String = iter.next
        Tuple2.apply(parseCustomer(`runVal₅`),
      }
    }))
  }))
}))
```

```scala
ZIO.service[CustomerConfig].map { (sm: CustomerConfig) =>
  sm.url
}.flatMap { custUrl =>
  ...
}
```

Use **defer.info** or defer.verbose | ZIO-DIRECT

```scala
val out =
  defer.info {
    val custUrl: String = ZIO.service[CustomerConfig].run.url
    val distUrl: String = ZIO.service[DistributorConfig].run.url
    (parseCustomer(httpGetCustomer(custUrl).run), parseDistrubutor(httpGetDistributor(distUrl).run))
  }
```

```
root> compile
=== Reconstituted Code ========
ZIO.service[CustomerConfig].map(((sm: CustomerConfig) => {
  val runVal: CustomerConfig = sm
  runVal.url
})).flatMap(((v: String) => {
  val custUrl: String = v
  ZIO.service[DistributorConfig].map(((`sm₂`: DistributorConfig) => {
    val `runVal₂`: DistributorConfig = `sm₂`
    `runVal₂`.url
  })).flatMap(((`v₂`: String) => {
    val distUrl: String = `v₂`
    ZIO.collectAll(Chunk.from(List.apply(httpGet
      val iter: Iterator[Any] = terms.iterator
      {
        val `runVal₃`: String = iter.next
        val `runVal₄`: String = iter.next
        Tuple2.apply(parseCustomer(`runVal₃`),
      }
    }))
  }))
}))
```

```scala
ZIO.service[CustomerConfig].map { (sm: CustomerConfig) =>
  val runVal: CustomerConfig = sm
  sm.url
}.flatMap { (v: String) =>
  val custUrl: String = v
}
```

Use **defer.info** or defer.verbose | ZIO-DIRECT

```scala
val out =
  defer.verbose {
    val custUrl: String = ZIO.service[CustomerConfig].run.url
    val distUrl: String = ZIO.service[DistributorConfig].run.url
    (parseCustomer(httpGetCustomer(custUrl).run), parseDistrubutor(httpGetDistributor(distUrl).run))
  }
```

```
root> compile
=== Reconstituted Code ========
   ZIO.service[CustomerConfig].map(((sm: CustomerConfig) => {
     val runVal: CustomerConfig = sm
     runVal.url
   })).asInstanceOf[ZIO[_, _, String]].flatMap(((v: String) => {
     val custUrl: String = v
     ZIO.service[DistributorConfig].map((($sm2: DistributorConfig) => {
       val $runVal2: DistributorConfig = $sm2
       $runVal2.url
     })).asInstanceOf[ZIO[_, _, String]].flatMap((($v2: String) => {
       val distUrl: String = $v2
       ZIO.collectAll(Chunk.from(List.apply(httpGetCustomer(custUrl), httpGetDistributor(distUrl)))).map(((terms: Chunk[Any]) => {
         val iter: Iterator[Any] = terms.iterator
         {
           val $runVal3: String = iter.next.asInstanceOf[String]
           val $runVal4: String = iter.next.asInstanceOf[String]
           Tuple2.apply(parseCustomer($runVal3), parseDistrubutor($runVal4))
         }.asInstanceOf[Tuple2[Customer, Distributor]]
       })).asInstanceOf[ZIO[_, _, Tuple2[Customer, Distributor]]]
     }))
   }))
```

Use defer.info or **defer.verbose**

ZIO-DIRECT

```scala
val out =
  defer.verbose {
    val custUrl: String =
      ZIO.service[CustomerConfig].run.url
    val distUrl: String =
      ZIO.service[DistributorConfig].run.url
    (
      parseCustomer(httpGetCustomer(custUrl).run),
      parseDistrubutor(httpGetDistributor(distUrl).run)
    )
  }
```

```
=== Deconstructed Instructions ========
  IR.FlatMap(
      IR.Parallel(
        List(
          (
            IR.Monad(SCALA({ ZIO.service[CustomerConfig] }), Pipeline),
            SCALA_SYM( /*val runVal*/ )
          )
        ),
        IR.Pure(SCALA({ runVal.url }))
      ),
      Some(SCALA_SYM( /*val custUrl*/ )),
      IR.FlatMap(
        IR.Parallel(
          List(
            (
              IR.Monad(SCALA({ ZIO.service[DistributorConfig] }), Pipeline),
              SCALA_SYM( /*val runVal*/ )
            )
          ),
          IR.Pure(SCALA({ runVal.url }))
        ),
        Some(SCALA_SYM( /*val distUrl*/ )),
        IR.Parallel(
          List(
            (
              IR.Monad(SCALA({ httpGetCustomer(custUrl) }), Pipeline),
              SCALA_SYM( /*val runVal*/ )
            ),
            (
              IR.Monad(SCALA({ httpGetDistributor(distUrl) }), Pipeline),
              SCALA_SYM( /*val runVal*/ )
            )
          ),
          IR.Pure(SCALA({
            Tuple2.apply(parseCustomer(runVal), parseDistrubutor('runVal;'))
          }))
        )
      )
    )
```

Wait... how do you do that?

ZIO-DIRECT

```
val out =
  defer {
    val custUrl: String = ZIO.service[CustomerConfig].run.url
    val distUrl: String = ZIO.service[DistributorConfig].run.url
    (parseCustomer(httpGet(custUrl).run), parseDistrubutor(httpGet(distUrl).run))
  }
```

Mystery Challenge... spot the IOs!

ZIO-DIRECT

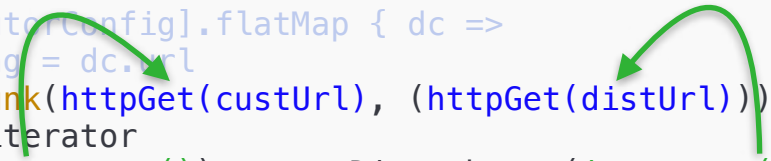
```
val out =
  defer {
    val custUrl: String = ZIO.service[CustomerConfig].run.url
    val distUrl: String = ZIO.service[DistributorConfig].run.url
    (parseCustomer(httpGet(custUrl).run), parseDistrubutor(httpGet(distUrl).run))
  }
```

```
ZIO.service[CustomerConfig].flatMap { cc =>
  val custUrl: String = cc.url
  ZIO.service[DistributorConfig].flatMap { dc =>
    val distUrl: String = dc.url
    ZIO.collectAll(Chunk(httpGet(custUrl), (httpGet(distUrl)))).map { col =>
      val iter = col.iterator
      (parseCustomer(iter.next()), parseDistrubutor(iter.next()))
    }
  }
}
```

Pull them out of the line body... | ZIO-DIRECT

```
val out =
  defer {
    val custUrl: String = ZIO.service[CustomerConfig].run.url
    val distUrl: String = ZIO.service[DistributorConfig].run.url
    (parseCustomer(httpGet(custUrl).run), parseDistrubutor(httpGet(distUrl).run))
  }
```

```
ZIO.service[CustomerConfig].flatMap { cc =>
  val custUrl: String = cc.url
  ZIO.service[DistributorConfig].flatMap { dc =>
    val distUrl: String = dc.url
    ZIO.collectAll(Chunk(httpGet(custUrl), (httpGet(distUrl)))).map { col =>
      val iter = col.iterator
      (parseCustomer(iter.next()), parseDistrubutor(iter.next()))
    }
  }
}
```



Collect, iterate, and splice back in! | ZIO-DIRECT

```
val out =
  defer {
    val custUrl: String = ZIO.service[CustomerConfig].run.url
    val distUrl: String = ZIO.service[DistributorConfig].run.url
    (parseCustomer(httpGet(custUrl).run), parseDistrubutor(httpGet(distUrl).run))
  }
```

```
ZIO.service[CustomerConfig].flatMap { cc =>
  val custUrl: String = cc.url
  ZIO.service[DistributorConfig].flatMap { dc =>
    val distUrl: String = dc.url
    ZIO.collectAll(Chunk(httpGet(custUrl), (httpGet(distUrl)))).map { col =>
      val iter = col.iterator
      (parseCustomer(iter.next()), parseDistrubutor(iter.next()))
    }
  }
}
```

ZIO-DIRECT

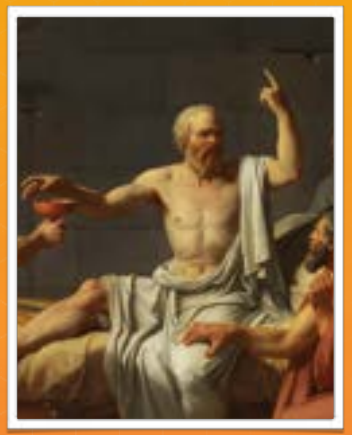
```
(parseCustomer(httpGet(custUrl).run), parseDistrubutor(httpGet(distUrl).run))
```

If you have no mutable data structures and no lazy actions... things in here cannot interact between each other... therefore, I can extract them!

$\{x \iff y\}$

CORRECTENESS

Things in defer blocks must be:
EAGER + IMMUTABLE
(unless they are wrapped in an effect)



yes... it's very important! | CORRECTNESS

```
defer(Params(Verify.None)) {
  val db = Database.open.run
  while (db.hasNextRow()) {
    if (db.lockNextRow())
      doSomethingWith(db.nextRow().run)
    else
      wait()
  }
}
```

```
class Database {
  def nextRow(): ZIO[Any, Throwable, Row]
  def hasNextRow(): Boolean
  def lockNextRow(): Boolean //somehow not an effect
}
```

API

yes... it's very important! | CORRECTNESS

```
defer(Params(Verify.None)) {
  val db = Database.open.run
  while (db.hasNextRow()) {
    if (db.lockNextRow())
      doSomethingWith(db.nextRow().run)
    else
      wait()
  }
}
```

```
class Database {
  def nextRow(): ZIO[Any, Throwable, Row]
  def hasNextRow(): Boolean
  def lockNextRow(): Boolean //somehow not an effect
}
```

API

```
Database.open.flatMap { db =>
  def whileFun(): ZIO[Any, Throwable, Unit] =
    if (db.hasNextRow())
      db.nextRow().flatMap { row =>
        // Too late to check if row is locked, we already READ IT!!
        if (db.lockNextRow()) doSomethingWith(row) else waitT()
        whileFun()
      }
    else
      ZIO.unit
  whileFun()
}
```

yes... it's very important! | CORRECTNESS

```
defer(Params(Verify.None)) {
  val db = Database.open.run
  while (db.hasNextRow()) {
    if (db.lockNextRow())
      doSomethingWith(db.nextRow().run)
    else
      wait()
  }
}
```

```
class Database {
  def nextRow(): ZIO[Any, Throwable, Row]
  def hasNextRow(): Boolean
  def lockNextRow(): Boolean //somehow not an effect
}
```

API

```
=== Reconstituted Code ========
  FunctionalObjectModel.Database.open.flatMap(((v: Database) => {
    val db: Database = v
    def whileFunc: ZIO[Any, Throwable, Unit] = if (db.hasNextRow) if (db.lockNextRow) db.nextRow.map(((sm: Row) => {
      val runVal: Row = sm
      FunctionalObjectModel.doSomethingWith(runVal)
    })) else zio.ZIO.succeed(FunctionalObjectModel.waitT).flatMap(((`v$2`: Unit) => whileFunc)) else zio.ZIO.succeed(())
    whileFunc
  }))
```

yes... it's very important!

CORRECTNESS

```
defer(Params(Verify.None)) {
  val db = Database.open.run
  while (ZIO.succeed(db.hasNextRow()).run) {
    if (ZIO.succeed(db.lockNextRow()).run)
      doSomethingWith(db.nextRow().run)
    else
      wait()
  }
}
```

```
class Database {
  def nextRow(): ZIO[Any, Throwable, Row]
  def hasNextRow(): Boolean
  def lockNextRow(): Boolean //somehow not an effect
}
```

API

Here's the thing you shouldn't do!

```
defer(Params(Verify.None)) {
  val db = Database.open.run
  (db.lockNextRow(), db.nextRow().run)
```

```
=== Reconstituted Code ========
   FunctionalObjectModel.Database.open.flatMap(((v: Database) => {
     val db: Database = v
     db.nextRow.map(((sm: Row) => {
       val runVal: Row = sm
       Tuple2.apply(db.lockNextRow, runVal)
     }))
   }))
```

```
val a = effectA.run
<expressions-using a>
val b = effectB.run
<expressions-using a, b>
val c = effectC.run
<expressions-using a, b, c>
```

```
effectA.flatMap { a =>
  <expressions-using a>
  effectB.flatMap { b =>
    <expressions-using a, b>
    effectC.flatMap {
      <expressions-using a, b, c>
    }
  }
}
```

```
val a = effectA.run
<expressions-using a>
val b = effectB.run
<expressions-using a, b>
val c = effectC.run
<expressions-using a, b, c>
```

```
effectA.flatMap { a =>
  <expressions-using a>
  effectB.flatMap { b =>
    <expressions-using a, b>
    effectC.flatMap {
      <expressions-using a, b, c>
    }
  }
}
```

```
val a = effectA.run
<expressions-using a>
val b = effectB.run
<expressions-using a, b>
<expressions-using effectC.run, effectD.run>
```

```
effectA.flatMap { a =>
  <expressions-using a>
  effectB.flatMap { b =>
    <expressions-using a, b>
    foreach(effectC, effectD) { list =>
      <expressions-using list.get(0), list.get(1)>
    }
  }
}
```

```
val a = effectA.run
<expressions-using a>
val b = effectB.run
<expressions-using a, b>
val c = effectC.run
<expressions-using a, b, c>
```

NO INTER-LINE INTERACTIONS

```
effectA.flatMap { a =>
  <expressions-using a>
  effectB.flatMap { b =>
    <expressions-using a, b>
    effectC.flatMap { c =>
      <expressions-using a, b, c>
    }
  }
}
```

```
val a = effectA.run
<expressions-using a>
val b = effectB.run
<expressions-using a, b>
<expressions-using effectC.run, effectD.run>
```

NO INTER-LINE INTERACTIONS

```
effectA.flatMap { a =>
  <expressions-using a>
  effectB.flatMap { b =>
    <expressions-using a, b>
    foreach(effectC, effectD) { list =>
      <expressions-using list.get(0), list.get(1)>
    }
  }
}
```

```
val a = effectA.run
<expressions-using a>
val b = effectB.run
<expressions-using a, b>
val c = effectC.run
<expressions-using a, b, c>
```

```
effectA.flatMap { a =>
  <expressions-using a>
  effectB.flatMap { b =>
    <expressions-using a, b>
    effectC.flatMap { c =>
      <expressions-using a, b, c>
    }
  }
}
```

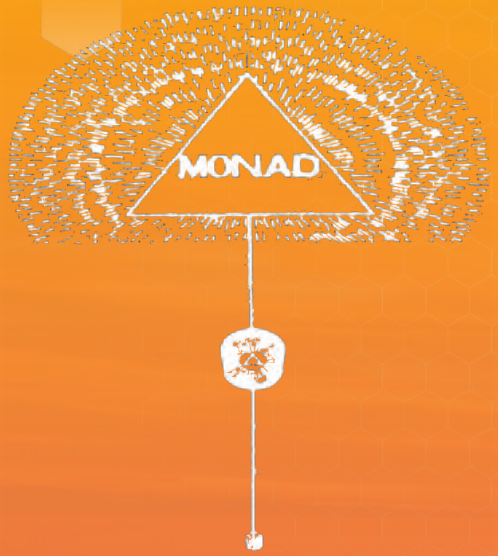
ALL INTERACTIONS HAPPEN DOWNWARD

```
val a = effectA.run
<expressions-using a>
val b = effectB.run
<expressions-using a, b>
<expressions-using effectC.run, effectD.run>
```

```
effectA.flatMap { a =>
  <expressions-using a>
  effectB.flatMap { b =>
    <expressions-using a, b>
    foreach(effectC, effectD) { list =>
      <expressions-using list.get(0), list.get(1)>
    }
  }
}
```

ALL INTERACTIONS HAPPEN DOWNWARD

SEQUENTIALITY =:= NESTING




```
effectA.flatMap { a =>
  <expressions-using a>
  effectB.flatMap { b =>
    <expressions-using a, b>
    effectC.flatMap { c =>
      <expressions-using a, b, c>
    }
  }
}
```

ALL INTERACTIONS HAPPEN DOWNWARD

```
effectA.flatMap { a =>
  <expressions-using a>
  effectB.flatMap { b =>
    <expressions-using a, b>
    ZIO.foreach(effectC, effectD) { list =>
      <expressions-using list.get(0), list.get(1)>
    }
  }
}
```

ALL INTERACTIONS HAPPEN DOWNWARD

```
for {
  a <- effectA.run
  _ = <expressions-using a>
  b <- effectB.run
  _ = <expressions-using a, b>
  c <- effectC.run
} yield <expressions-using a, b, c>
```

```
effectA.flatMap { a =>
  <expressions-using a>
  effectB.flatMap { b =>
    <expressions-using a, b>
    effectC.flatMap { c =>
      <expressions-using a, b, c>
    }
  }
}
```

ALL INTERACTIONS HAPPEN DOWNWARD

```
effectA.flatMap { a =>
  <expressions-using a>
  effectB.flatMap { b =>
    <expressions-using a, b>
    foreach(effectC, effectD) { list =>
      <expressions-using list.get(0), list.get(1)>
    }
  }
}
```

ALL INTERACTIONS HAPPEN DOWNWARD

- If the statement already transformed e.g. **defer { defer { ... } }**:
  - Chain it in a flatMap and continue
- If statement is a **try**:
  - Recurse on the body, then the catch clauses, then the finalizer if it exists.
  - Chain all of them in nested flatMaps (or maps if they have no run statements inside)
- If statement is a throw, or **unsafe { ... }** block:
  - Chain it in a flatMap (or map if it has run statements inside)
- If statement is an **if**, **for-loop**, or **match** statement
  - Recurse on the prefix/condition and then the body statement(s).
  - Chain both in flatMaps (or maps if they have no run statements inside)
- If statement is a **run** call:
  - Chain it in a flatMap
- If statement is a block e.g. **{ a; b; c }**
  - Chain each clause in a flatMap (or maps if it has no `run` statements inside)
- Otherwise if the statement has no vars, defs, classes mutable state, or any other forbidden construct:
  - Extract each run call into a ZIO. collect
  - Then splice in the iteration steps in each call-site.
  - Chain the whole thing in a flatMap

OTHER THINGS FORBIDDEN IN DEFER-CLAUSES

(Unless they are in the .run calls)

- lazy value declarations
- class declarations
- function declarations
- lambdas/closures
- implicit variables/functions

CORRECTNESS

Constraints

NO lazy value declarations
NO class declarations
NO function declarations
NO lambdas/closures
NO implicit variables/functions

Certainty

Cut-Points Work
Splicing Correct
Sequencing Correct
Generated Code Correct

Capability

Direct-Style Syntax
Language-Level Constructs

```
var x = 1
defer.info {
  ({ x = x + 1; x }, ZIO.succeed(x).run)
}
```

```
> runMain
(2, 1)
```

...but why not (2, 2) ???

```
defer.info {
  ({ x = x + 1; x }, ZIO.succeed(x).run)
}
```

```
var x: Int

Assignment is generally not allowed inside of defer calls,
because it can cause correctness problems with the
synthesized code if it directly reads the result
of a `run(...)` call or interacts with other
effects in `run(...)` clauses. Please use a ZIO Ref instead.
=========
For example, instead of this:
defer {
        val i = run(numCalls)
        while (i > 0) {
                println("Value:" + i)
```

Mutable Collections... Forbidden!

CORRECTNESS

```
defer.info {
  ({ buff.update(0, buff(0) + 1); buff(0) }, ZIO.succeed(buff(0)).run)
}
```

```
> runMain
(2, 1)
```

```
defer.info {
  ({ buff.update(0, buff(0) + 1); buff(0) }, ZIO.succeed(buff(0)).run)
}
```

```
var buff: ArrayBuffer[Int]

Detected the use of a mutable collection inside a defer clause.
Mutable collections can cause many potential issues as a result of defer-clause
rewrites so they are not allowed (Unless it is inside of a run-call).

=========
buff
 bloop
```

View Problem (⌘.) No quick fixes available

```
defer.info {
  (ZIO.succeed({ buff.update(0, buff(0) + 1); buff(0) }).run, ZIO.succeed(buff(0)).run)
}
```

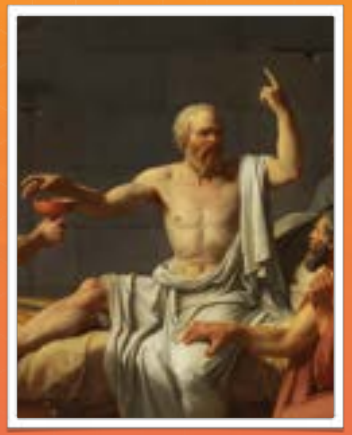
```
defer.info {
  (ZIO.succeed({ x = x + 1 }).run, ZIO.succeed(x).run)
}
```

CORRECTNESS

Things in defer blocks must be:

EAGER + IMMUTABLE

(unless they are wrapped in an effect!!!
...and called with .run)



ARGUMENT I | It is not Referentially Transparent

ARGUMENT I | It is not Referentially Transparent

Definition:
An expression shall called referentially transparent if it can be replaced with its corresponding value without changing the program's behavior.

Interpretation:
Ordering of value-terms should not matter.

ARGUMENT I | It is not Referentially Transparent

```
defer {
  val int = run(List(1, 2))
  val bool = run(List(true, false))
  (int, bool)
}
// List((1, true), (1, false), (2, true), (2, false))
```

```
defer {
  val bool = run(List(true, false))
  val int = run(List(1, 2))
  (int, bool)
}
// List((1, true), (2, true), (1, false), (2, false))
```

ARGUMENT I | It is not Referentially Transparent

```
defer.info {
  val int = run(List(1, 2))
  val bool = run(List(true, false))
  (int, bool)
}
// List((1, true), (1, false), (2, true), (2, false))
```

```
defer.info {
  val bool = run(List(true, false))
  val int = run(List(1, 2))
  (int, bool)
}
// List((1, true), (2, true), (1, false), (2, false))
```

```
List(1, 2).flatMap { int =>
  List(true, false).map { bool =>
    (int, bool)
  }
}
```

```
List(true, false).flatMap { int =>
  List(1, 2).map { bool =>
    (int, bool)
  }
}
```

ARGUMENT I | It is not Referentially Transparent

```
defer.info {
  val int = run(List(1, 2))
  val bool = run(List(true, false))
  (int, bool)
}
// List((1, true), (1, false), (2, true), (2, false))
```

```
defer.info {
  val bool = run(List(true, false))
  val int = run(List(1, 2))
  (int, bool)
}
// List((1, true), (2, true), (1, false), (2, false))
```

```
List(1, 2).flatMap { int =>
  List(true, false).map { bool =>
    (int, bool)
  }
}
```

```
List(true, false).flatMap { int =>
  List(1, 2).map { bool =>
    (int, bool)
  }
}
```

... but the things on the left hand side are PURE VALUES since they have no monad signature!

ARGUMENT I | It is not Referentially Transparent

```
defer {
  val int: Int = run(List(1, 2))
  val bool: Boolean = run(List(true, false))
  (int, bool)
}
// List((1, true), (1, false), (2, true), (2, false))
```

```
defer {
  val bool: Boolean = run(List(true, false))
  val int: Int = run(List(1, 2))
  (int, bool)
}
// List((1, true), (2, true), (1, false), (2, false))
```

```
List(1, 2).flatMap { int =>
  List(true, false).map { bool =>
    (int, bool)
  }
}
```

```
List(true, false).flatMap { int =>
  List(1, 2).map { bool =>
    (int, bool)
  }
}
```

... but the things on the left hand side are PURE VALUES since they have no monad signature!

No, they're just syntactic sugar for flatMap/map variables.

Same as the left-hand of a for comprehension. Look...

ARGUMENT I | It is not Referentially Transparent

```
defer {
  val int: Int = run(List(1, 2))
  val bool: Boolean = run(List(true, false))
  (int, bool)
}
// List((1, true), (1, false), (2, true), (2, false))
```

```
defer {
  val bool: Boolean = run(List(true, false))
  val int: Int = run(List(1, 2))
  (int, bool)
}
// List((1, true), (2, true), (1, false), (2, false))
```

```
List(1, 2).flatMap { int =>
  List(true, false).map { bool =>
    (int, bool)
  }
}
```

```
List(true, false).flatMap { int =>
  List(1, 2).map { bool =>
    (int, bool)
  }
}
```

```
for {
  int: Int <- List(1, 2)
  bool: Boolean <- List(true, false)
} yield (int, bool)
```

```
for {
  bool: Boolean <- List(true, false)
  int: Int <- List(1, 2)
} yield (int, bool)
```

ARGUMENT I | It is not Referentially Transparent

```
defer {
  val int: Int = run(List(1, 2))
  (int, run(List(true, false)))
}

// List((1, true), (1, false), (2, true), (2, false))
```

```
defer {
  val bool: Boolean = run(List(true, false))
  val int: Int = run(List(1, 2))
  (int, bool)
}
// List((1, true), (2, true), (1, false), (2, false))
```

```
List(1, 2).flatMap { int =>
  List(true, false).map { bool =>
    (int, bool)
  }
}
```

```
List(true, false).flatMap { int =>
  List(1, 2).map { bool =>
    (int, bool)
  }
}
```

No Equivalent

```
for {
  bool: Boolean <- List(true, false)
  int: Int <- List(1, 2)
} yield (int, bool)
```

ARGUMENT I | It is not Referentially Transparent... Take 613

```
defer {
  val int = List(1, 2).run
  val bool = List(true, false).run
  val bool1 = bool
  val int1 = int
  (int, bool)
}
// List((1, true), (1, false), (2, true), (2, false))
```

```
defer {
  val bool = List(true, false).run
  val int = List(1, 2).run
  val bool1 = bool
  val int1 = int
  (int, bool)
}
// List((1, true), (2, true), (1, false), (2, false))
```

```
List(1, 2).flatMap { int =>
  List(true, false).flatMap { bool =>
    val bool1 = bool
    val int1 = int
    (int, bool)
  }
}
```

```
List(true, false).flatMap { int =>
  List(1, 2).flatMap { bool =>
    val bool1 = bool
    val int1 = int
    (int, bool)
  }
}
```

ARGUMENT I | It is not Referentially Transparent... Take 613

```
defer {
  val int = List(1, 2)
  val bool = List(true, false)
  val bool1 = bool.run
  val int1 = int.run
  (int, bool)
}
// List((1, true), (2, true), (1, false), (2, false))
```

```
defer {
  val bool = List(true, false)
  val int = List(1, 2)
  val bool1 = bool.run
  val int1 = int.run
  (int, bool)
}
// List((1, true), (2, true), (1, false), (2, false))
```

```
val int = List(1, 2)
val bool = List(true, false)
bool.flatMap { bool =>
  int.map { int =>
    (int, bool)
  }
}
```

```
val bool = List(true, false)
val int = List(1, 2)
bool.flatMap { bool =>
  int.map { int =>
    (int, bool)
  }
}
```

ARGUMENT 2 | It is Lawless

Replying to @deusaquilus and 3 others

“Kinda-sorta `Kleisli` and `Cokleisli` but without laws and with unknown edge cases, in the name of direct-style syntax:” no, thank you.

2

ARGUMENT 2 | It is Lawless

IV.
ALL RUN(...) STATEMENTS INSIDE A DEFER SHALL BE RUN FROM TOP TO BOTTOM IF THEY ARE IN DIFFERENT LINES, AND FROM LEFT TO RIGHT IF THEY ARE ON THE SAME LINE.

V.
ALL PLAIN STATEMENTS INSIDE OF A DEFER SHALL BE RUN AFTER THE RUN(...) ON THE LINE ABOVE AND BEFORE THE RUN(...) ON THE LINE BELOW.

VI.
IN THE CASE THAT THERE ARE PLAIN STATEMENTS AND RUN(ZIO)S ON THE SAME LINE, THE RUN(ZIO)S CAN BE EXECUTED BEFORE THE PLAIN STATEMENTS IN SOME CIRCUMSTANCES. TO ENSURE CORRECTNESS IN THIS SCENARIO, MAKE SURE TO FOLLOW LAWS I, II, AND III.

I.
YE SHALL NOT USE MUTABLE THINGS INSIDE OF A DEFER UNLESS THE STATEMENT IS WRAPPED IN A RUN(...) OR UNSAFE(...) BLOCK.

II.
YE SHALL NOT USE LAZY THINGS INSIDE OF A DEFER UNLESS THE STATEMENT IS WRAPPED IN A RUN(...) OR UNSAFE(...) BLOCK.

III.
YE SHALL NOT INTERACT WITH EXTERNAL STATE INSIDE OF A DEFER UNLESS THE STATEMENT IS WRAPPED IN A RUN(...) OR UNSAFE(...) BLOCK.

ERROR CHANNELS

Because ZIO actually has them....

ERROR CHANNELS | Expecting the somewhat/somewhat-not unexpected

```scala
class Boom extends Exception("Boom!")
def pretendPureCode() = throw new Boom()
```

No Error

```scala
ZIO.succeed(pureCode)
```

```
< pureCode >
```

To Error Channel

```scala
ZIO.fail(new Boom())
```

```
zio.direct.BoomExamples$Boom: Boom!
  at zio.direct.BoomExamples$.main$$anonfun$1(BoomExamples.scala:22)
  at zio.ZIO$.fail$$anonfun$1(ZIO.scala:3021)
  at zio.ZIO$.failCause$$anonfun$1(ZIO.scala:3027)
  at zio.internal.FiberRuntime.runLoop(FiberRuntime.scala:986)
```

To Defect Channel

```scala
ZIO.succeed(pretendPureCode())
```

```
zio.direct.BoomExamples$Boom: Boom!
  at zio.direct.BoomExamples$.main$$anonfun$1(BoomExamples.scala:22)
  at zio.UnsafeVersionSpecific.implicitFunctionIsFunction$$anonfun$1(UnsafeVersionSpecific.scala:23)
  at zio.Unsafe$.unsafe(Unsafe.scala:37)
  at zio.ZIOCompanionVersionSpecific.succeed$$anonfun$1(ZIOCompanionVersionSpecific.scala:161)
  at zio.internal.FiberRuntime.runLoop(FiberRuntime.scala:831)
```

ERROR CHANNELS | ~~Expecting~~ Attempting the somewhat/somewhat-not unexpected

```scala
class Boom extends Exception("Boom!")
def pretendPureCode() = throw new Boom()
```

No Error

```scala
ZIO.succeed(pureCode)
```

```
< pureCode >
```

To Error Channel

```scala
ZIO.fail(new Boom())
```

```
zio.direct.BoomExamples$Boom: Boom!
  at zio.direct.BoomExamples$.main$$anonfun$1(BoomExamples.scala:22)
  at zio.ZIO$.fail$$anonfun$1(ZIO.scala:3021)
  at zio.ZIO$.failCause$$anonfun$1(ZIO.scala:3027)
  at zio.internal.FiberRuntime.runLoop(FiberRuntime.scala:986)
```

To Defect Channel

```scala
ZIO.succeed(pretendPureCode())
```

```
zio.direct.BoomExamples$Boom: Boom!
  at zio.direct.BoomExamples$.main$$anonfun$1(BoomExamples.scala:22)
  at zio.UnsafeVersionSpecific.implicitFunctionIsFunction$$anonfun$1(UnsafeVersionSpecific.scala:23)
  at zio.Unsafe$.unsafe(Unsafe.scala:37)
  at zio.ZIOCompanionVersionSpecific.succeed$$anonfun$1(ZIOCompanionVersionSpecific.scala:161)
  at zio.internal.FiberRuntime.runLoop(FiberRuntime.scala:831)
```

To Error Channel

```scala
ZIO.attempt(pretendPureCode())
```

```
zio.direct.BoomExamples$Boom: Boom!
  at zio.direct.BoomExamples$.main$$anonfun$1(BoomExamples.scala:22)
  at zio.ZIOCompanionVersionSpecific.attempt$$anonfun$1(ZIOCompanionVersionSpecific.scala:107)
  at zio.internal.FiberRuntime.runLoop(FiberRuntime.scala:967)
```

ERROR CHANNELS | ~~Expecting~~ Attempting the somewhat/somewhat-not unexpected

```scala
class Boom extends Exception("Boom!")
def pretendPureCode() = throw new Boom()
```

No Error

```scala
ZIO.succeed(pureCode)
```

```scala
defer { pureCode }
```

To Error Channel

```scala
ZIO.fail(new Boom())
```

```scala
defer { throw new Boom() }
```

To Defect Channel

```scala
ZIO.succeed(pretendPureCode())
```

```scala
defer { pretendPureCode() }
```

To Error Channel

```scala
ZIO.attempt(pretendPureCode())
```

```scala
???
```

ERROR CHANNELS

```
class Boom extends Exception("Boom!")
def pretendPureCode() = throw new Boom()
```

No Error

```
ZIO.succeed(pureCode)
```

```
defer { pureCode }
```

To Error Channel

```
ZIO.fail(new Boom())
```

```
defer { throw new Boom() }
```

To Defect Channel

```
ZIO.succeed(pretendPureCode())
```

```
defer { pretendPureCode() }
```

To Error Channel

```
ZIO.attempt(pretendPureCode())
```

```
defer { ZIO.attempt(pretendPureCode()).run }
```

ERROR CHANNELS

```
class Boom extends Exception("Boom!")
def pretendPureCode() = throw new Boom()
```

No Error

```
ZIO.succeed(pureCode)
```

```
defer { pureCode }
```

To Error Channel

```
ZIO.fail(new Boom())
```

```
defer { throw new Boom() }
```

To Defect Channel

```
ZIO.succeed(pretendPureCode())
```

```
defer { pretendPureCode() }
```

To Error Channel

```
ZIO.attempt(pretendPureCode())
```

```
defer { unsafe(pretendPureCode()) }
```

ERROR CHANNELS | A "~~Real~~ Cloud World" Example

```
object Database {
  def openConnection(): ZIO[Scope, Throwable, Connection]
}
object S3Object {
  def openInputStream(path: String): ZIO[Scope, Throwable, InputStream]
}
```

```
defer {
  try {
    val input = S3Object.openInputStream("foo/bar").run
    val reader = InputStreamReader(input)
    val conn = Database.openConnection().run
    val ps = conn.prepareStatement("INSERT ? INTO Someplace")
    ps.setClob(1, reader)
    ps.execute()
  } catch {
    case e: IOException  => handle(e).run
    case e: SQLException => handle(e).run
  }
}
```

ERROR CHANNELS | A "~~Real~~ Cloud World" Example

```
object Database {
  def openConnection(): ZIO[Scope, Throwable, Connection]
}
object S3Object {
  def openInputStream(path: String): ZIO[Scope, Throwable, InputStream]
}
```

Could Potentially Throw Something!

```
defer {
  try {
    val input = S3Object.openInputStream("foo/bar").run
    val reader = InputStreamReader(input)
    val conn = Database.openConnection().run
    val ps = conn.prepareStatement("INSERT ? INTO Someplace")
    ps.setClob(1, reader)
    ps.execute()
  } catch {
    case e: IOException  => handle(e).run
    case e: SQLException => handle(e).run
  }
}
```

ERROR CHANNELS | A "~~Real~~ Cloud World" Example

```
object Database {
  def openConnection(): ZIO[Scope, Throwable, Connection]
}
object S3Object {
  def openInputStream(path: String): ZIO[Scope, Throwable, InputStream]
}
```

```
defer {
  try {
    val input = S3Object.openInputStream("foo/bar").run
    val reader = unsafe(InputStreamReader(input))
    val conn = Database.openConnection().run
    val ps = unsafe(conn.prepareStatement("INSERT ? INTO Someplace"))
    unsafe {
      ps.setClob(1, reader)
      ps.execute()
    }
  } catch {
    case e: IOException  => handle(e).run
    case e: SQLException => handle(e).run
  }
}
```

ERROR CHANNELS | A "~~Real~~ Cloud World" Example

```scala
object Database {
  def openConnection(): ZIO[Scope, Throwable, Connection]
}
object S3Object {
  def openInputStream(path: String): ZIO[Scope, Throwable, InputStream]
}
```

```scala
val out: ZIO[Scope, Throwable, Unit] =
        defer {
          try {
            val input = S3Object.openInputStream("foo/bar").run
            val reader = unsafe(InputStreamReader(input))
            val conn = Database.openConnection().run
            val ps = unsafe(conn.prepareStatement("INSERT ? INTO Someplace"))
            unsafe {
              ps.setClob(1, reader)
              ps.execute()
            }
          } catch {
            case e: IOException  => handle(e).run
            case e: SQLException => handle(e).run
          }
        }
```

ERROR CHANNELS | A "~~Real~~ Cloud World" Example

```
object Database {
  def openConnection(): ZIO[Scope, Throwable, Connection]
}
object S3Object {
  def openInputStream(path: String): ZIO[Scope, Throwable, InputStream]
}
```

Or wrap the entire thing!

```
defer {
  try {
    unsafe {
      val input = S3Object.openInputStream("foo/bar").run
      val reader = InputStreamReader(input)
      val conn = Database.openConnection().run
      val ps = conn.prepareStatement("INSERT ? INTO Someplace")
      ps.setClob(1, reader)
      ps.execute()
    }
  } catch {
    case e: IOException  => handle(e).run
    case e: SQLException => handle(e).run
  }
}
```

ERROR CHANNELS | A "Real Cloud World" Example

Wrap in Unsafe

```
defer {
  try {
    unsafe {
      val input = S3Object.openInputStream("foo/bar").run
      val reader = InputStreamReader(input)
      val conn = Database.openConnection().run
      val ps = conn.prepareStatement("INSERT ? INTO Someplace")
      ps.setClob(1, reader)
      ps.execute()
    }
  } catch {
    case e: IOException  => handle(e).run
    case e: SQLException => handle(e).run
  }
}
```

Generated Code wraps ZIO.attempt

```
=== Reconstituted Code ========
   S3Object.openInputStream("foo/bar").flatMap(((v: InputStream) => {
     val input: InputStream = v
     val reader: InputStreamReader = new InputStreamReader(input)
     Database.openConnection.map(((`v2`: Connection) => {
       val conn: Connection = `v2`
       val ps: PreparedStatement = conn.prepareStatement("INSERT ? INTO Someplace")
       ps.setClob(1, reader)
       ps.execute
     }))
   })).catchSome(((tryLamParam: Throwable) => tryLamParam match {
     case e: IOException =>
       handle(e)
     case e: SQLException =>
       handle(`e2`)
   }))
```

```
=== Reconstituted Code ========
   ObjectModel.S3Object.openInputStream("foo/bar").flatMap(((v: InputStream) => {
     val input: InputStream = v
     ZIO.attempt(new InputStreamReader(input)).flatMap(((`v2`: InputStreamReader) => {
       val reader: InputStreamReader = `v2`
       ObjectModel.Database.openConnection.flatMap(((`v3`: Connection) => {
         val conn: Connection = `v3`
         ZIO.attempt(conn.prepareStatement("INSERT ? INTO Someplace")).flatMap(((`v4`: PreparedStatement) => {
           val ps: PreparedStatement = `v4`
           ZIO.attempt({
             ps.setClob(1, reader)
             ps.execute
           })
         }))
       }))
     }))
   })).catchSome(((tryLamParam: Throwable) => tryLamParam match {
     case e: IOException =>
       ObjectModel.handle(e)
     case e: SQLException =>
       ObjectModel.handle(`e2`)
   }))
```

USING REFS | Okay...

Looping with vars

```
var i: Int = 10
while (i > 0) {
  println(s"Currently: ${i}")
  i = i - 1
}
```

Looping with Refs

?

USING REFS | Let's pull some teeth!

Looping with vars

```
var i: Int = 10
while (i > 0) {
  println(s"Currently: ${i}")
  i = i - 1
}
```

Looping with Refs

```
val i: UIO[Ref[Int]] = Ref.make(10)
i.flatMap { i0 =>
  def whileLoop(): ZIO[Any, Nothing, Unit] =
    i0.get.flatMap { iv =>
      if (iv > 0) {
        println(s"Currently: ${iv}")
        i0.update(i => i - 1).flatMap { _ =>
          whileLoop()
        }
      } else {
        ZIO.unit
      }
    }
  whileLoop()
}
```

USING REFS | Let's pull some teeth!

Looping with vars

```
var i: Int = 10
while (i > 0) {
  println(s"Currently: ${i}")
  i = i - 1
}
```

Looping with Refs

```
val i: UIO[Ref[Int]] = Ref.make(10)
i.flatMap { i0 =>
  def whileLoop(): ZIO[Any, Nothing, Unit] =
    i0.get.flatMap { iv =>
      if (iv > 0) {
        println(s"Currently: ${iv}")
        i0.update(i => i - 1).flatMap { _ =>
          whileLoop()
        }
      } else {
        ZIO.unit
      }
    }
  whileLoop()
}
```

Be sure these are right...
or it runs forever!

USING REFS

Looping with vars

```
var i: Int = 10
while (i > 0) {
  println(s"Currently: ${i}")
  i = i - 1
}
```

Looping with Refs - Take 2

```
defer {
  val i: Ref[Int] = Ref.make(10).run
  while (i.get.run > 0) {
    println(s"Currently: ${i.get.run}")
    i.update(i => i - 1).run
  }
}
```

USING REFS

Looping with vars

```
var i: Int = 10
while (i > 0) {
  println(s"Currently: ${i}")
  i = i - 1
}
```

Looping with Refs - Take 2

```
defer.info {
  val i: Ref[Int] = Ref.make(10).run
  while (i.get.run > 0) {
    println(s"Currently: ${i.get.run}")
    i.update(i => i - 1).run
  }
}
```

```
=== Reconstituted Code ========
   Ref.make(10).flatMap(((v: Ref[Int]) => {
     val i: Ref[Int] = v
     def whileFunc: ZIO[Any, Nothing, Unit] = i.get.map(((sm: Int) => {
       val runVal: Int = sm
       runVal.>(0)
     })).flatMap((($v_2$: Boolean) => {
       val ifVar: Boolean = $v_2$
       if (ifVar) i.get.map((($sm_2$: Int) => {
         val $runVal_2$: Int = $sm_2$
         println(_root_.scala.StringContext.apply("Currently: ", "").s($runVal_2$))
       })).flatMap((($v_3$: Unit) => i.update((($i_2$: Int) => $i_2$.-(1))))).flatMap((($v_4$: Any) => whileFunc)) else zio.ZIO.succeed(())
     }))
     whileFunc
   }))
```

USING REFS

```
val arr = Array(3, 2, 8, 5, 7, 2, 3, 8, 9, 4, 5, 8, 2, 3, 4, 7, 6, 5, 9, 2, 3, 8, 4, 7, 5, 6, 2, 0, 8, 3)
quicksortImperative(arr)
println(arr.toList)
// List(0, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 4, 4, 4, 5, 5, 5, 5, 6, 6, 7, 7, 7, 8, 8, 8, 8, 8, 9, 9)
```

Quicksort with Mutable Indices

```
def quicksortImperative(a: Array[Int]): Unit = {
  def swap(i: Int, j: Int): Unit = {
    val t = a(i)
    a(i) = a(j)
    a(j) = t
  }
  def sort(l: Int, r: Int): Unit = {
    val pivot = a((l + r) / 2)
    var i = l
    var j = r
    while (i <= j) {
      while (a(i) < pivot) i += 1
      while (a(j) > pivot) j -= 1
      if (i <= j) {
        swap(i, j)
        i += 1
        j -= 1
      }
    }
    if (l < j) sort(l, j)
    if (j < r) sort(i, r)
  }
  sort(0, a.length - 1)
}
```

Quicksort with Mutable Indices

USING REFS

```
val arr = Array(3, 2, 8, 5, 7, 2, 3, 8, 9, 4, 5, 8, 2, 3, 4, 7, 6, 5, 9, 2, 3, 8, 4, 7, 5, 6, 2, 0, 8, 3)
runUnsafe(quicksortDefer(arr))
println(arr.toList)
// List(0, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 4, 4, 4, 5, 5, 5, 5, 6, 6, 7, 7, 7, 8, 8, 8, 8, 8, 9, 9)
```

Quicksort with Mutable Indices

```
def quicksortImperative(a: Array[Int]): Unit = {
  def swap(i: Int, j: Int): Unit = {
    val t = a(i)
    a(i) = a(j)
    a(j) = t
  }
  def sort(l: Int, r: Int): Unit = {
    val pivot = a((l + r) / 2)
    var i = l
    var j = r
    while (i <= j) {
      while (a(i) < pivot) i += 1
      while (a(j) > pivot) j -= 1
      if (i <= j) {
        swap(i, j)
        i += 1
        j -= 1
      }
    }
    if (l < j) sort(l, j)
    if (j < r) sort(i, r)
  }
  sort(0, a.length - 1)
}
```

Quicksort with Mutable Indices

```
def quicksortDefer(arr: Array[Int]): ZIO[Any, Nothing, Unit] = {
  def swap(i: Int, j: Int) =
    val temp = arr(i)
    arr(i) = arr(j)
    arr(j) = temp

  def sort(l: Int, r: Int): ZIO[Any, Nothing, Unit] =
    defer(Params(Verify.Lenient)) {
      val pivot = arr((l + r) / 2)
      val i = Ref.make(l).run
      val j = Ref.make(r).run
      while (i.get.run <= j.get.run)
        while (arr(i.get.run) < pivot) i.getAndUpdate(i => i + 1).run
        while (arr(j.get.run) > pivot) j.getAndUpdate(j => j - 1).run
        if (i.get.run <= j.get.run)
          swap(i.get.run, j.get.run)
          i.getAndUpdate(i => i + 1).run
          j.getAndUpdate(j => j - 1).run

      if (l < j.get.run)
        val jv = j.get.run
        sort(l, jv).run
      if (j.get.run < r)
        val iv = i.get.run
        sort(iv, r).run
    }
    sort(0, arr.length - 1)
  }
```

IN SUMMARY...

1. Introducing

ZIO-DIRECT is a ZIO-specific system based on the Monadless paradigm of imperative -> functional transformation.

2. Uses ZIO Features

ZIO-DIRECT Correctly interoperates with ZIO Environment and Error Types

3. Can Peek Inside

ZIO-DIRECT provides enhanced feedback with .info and .verbose so that you clearly know what code it is generating.

4. Correctness based on Constraints

ZIO-DIRECT constrains problematic Scala constructs in order to guarantee correctness.

5. Uses Error Channel

ZIO-DIRECT uses the ZIO error channel to mediate try-catch statements.

6. Works well with Mutable Refs

ZIO-DIRECT makes the use of ZIO Refs natural and ergonomic.

Future Directions

```
def httpGet[T](str: String): Future[T]
```

```
val q: Future[(Person, Address)] =
  defer[Future].from {
    val p = httpGet[Person]("http://people").run
    val a = httpGet[Address](s"http://address?owner=${p.id}").run
    (p, a)
  }
```

```
def people: List[Person]
```

```
val q: List[(Person, Address)] =
  defer[List].from {
    val p = people.run
    val a = p.addresses.run
    (p, a)
  }
```

```
def query[T]: Query[T]
```

```
val q: Query[(Person, Address)] =
  defer[Query].from {
    val p = query[Person].run
    val a = query[Address].join(a => a.fk == p.id).run
    (p, a)
  }
```

Thank You

Repo: github.com/zio/zio-direct

Example Usage: github.com/zio/zio-protoquill/tree/zio-direct