# Exploring type-directed, test-driven development

A case study using FizzBuzz

Franklin Chen
http://franklinchen.com/

June 7, 2014
Pittsburgh TechFest 2014

**Abstract**

An expressive static type system is one of the most joyful and powerful tools for prototyping, designing, and maintaining programs. In this performance-theatrical presentation, I will provide a taste of how to use types, in conjunction with tests, to drive iterative development of a particular program, the famous FizzBuzz problem. We will solve generalizations of this problem, changing and adding requirements, to illustrate the pleasures and benefits of "type thinking".

The Scala language will be used as the vehicle for this demonstration, but the techniques apply immediately to any industrial-strength statically typed language, such as Haskell, OCaml, F#, Rust, and most recently, Swift.

(Note: this presentation will use live human volunteers to play the roles of various programming concepts.)

## Outline

**Outline**

## Contents

# 1 Introduction

## 1.1 Goals

### Goals of this presentation

- Give a taste of a *practical* software development *process* that is:
  - *test*-driven
  - *type*-directed
- Show everything for real:
  - project build process
  - testing frameworks
  - all the code
- Use `FizzBuzz` because:
  - problem: easy to understand
  - modifications: easy to understand
  - fun!
- Encourage you to explore further
  - Have you heard the news about Swift?

## 1.2 Test-driven development (TDD)

**Test-driven development (TDD)**

- Think.
- Write a test that fails.
- Write code until test succeeds.
- Repeat, and refactor as needed.

**Is TDD dead?**
Short answer: No.

## 1.3 Type systems

**Type systems**

**What is a type system?**
For this presentation: a *syntactic* method for *proving* the absence of certain program behaviors.

**"Debating" types "versus" tests?**
Let's just use both!

## 1.4 Poor versus decent type systems

**Poor versus decent type systems**

**Poor type systems**

- (Developed using 1960s-1970s knowledge)
- C, C++, Objective C
- Java

**Decent type systems**

- (Developed using 1980s-1990s knowledge)
- ML (Standard ML, OCaml, F#): I first used for work in 1995
- Haskell: I first used for work in 1995
- Scala: first released in 2004
- Rust: not yet version 1.0
- Swift: announced by Apple on June 2, 2014!

# 2 Original FizzBuzz problem

## 2.1 Original FizzBuzz problem

**Original FizzBuzz problem**

**FizzBuzz defined**

Write a program that prints the numbers from 1 to 100.

But for multiples of three, print "Fizz" instead of the number.

And for the multiples of five, print "Buzz".

For numbers which are multiples of both three and five, print "FizzBuzz".

## 2.2 Starter Scala code: main driver

**Starter Scala code: main driver**

Scala: a modern *object-oriented* and *functional* language.

```
object Main extends App {
  // Will not compile yet!
  runToSeq(1, 100).foreach(println)
}
```

- Type-directed design: separate out effects (such as printing to terminal) from the real work.
- Type-directed feedback: compilation fails when something is not implemented yet.

## 2.3 The joys of continuous compilation and testing

**The joys of continuous compilation and testing**

SBT: build tool supporting Scala, Java...

**Winning features**

- Source file changes trigger smart recompilation!
- Source file changes trigger rerun of the tests that depend on changed code!

```
$ sbt
> ~testQuick
[info] Compiling 1 Scala source to ...
[error] ...Main.scala:16: not found: value runToSeq
[error]   runToSeq(1, 100) foreach println
[error]   ^
[error] one error found
```

**Write type-directed stub**

```scala
object Main extends App {
  runToSeq(1, 100).foreach(println)

  def runToSeq(start: Int, end: Int): Seq[String] = {
    ???
  }
}
```

**Write wanted type signature**
??? is convenient for stubbing.

- In Scala standard library
- Just performs: `throw new NotImplementedError`

## 2.4 Write acceptance test (simplified)

**Write acceptance test (simplified)**

specs2

Specs2: a fine testing framework for Scala, Java...

```scala
class MainSpec extends Specification { def is = s2"""
  ${Main.runToSeq(1, 16) ==== `strings for 1 to 16`}
  """

  val `strings for 1 to 16` = Seq(
    "1", "2", "Fizz", "4", "Buzz", "Fizz",
    "7", "8", "Fizz", "Buzz", "11", "Fizz",
    "13", "14", "FizzBuzz", "16"
  )
}
```

A realistic acceptance test would involve handling I/O, but an elegant technique for modularizing that, `scalaz-stream`, is outside the scope of this presentation.

**Test passes type check, but fails**

Incremental compilation/testing kicks in:

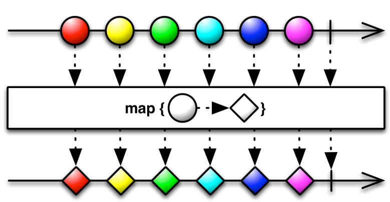
```
Waiting for source changes... (press enter to interrupt)
[info] MainSpec
[info]   x Main.runToSeq(1, 16) ==== ‘strings for 1 to 16
[error]  an implementation is missing (Main.scala:19)
```

**Outside-in: for a `FizzBuzz` unit**

Types are shapes to assemble logically.

```
  def runToSeq(start: Int, end: Int): Seq[String] = {
    start.to(end).map(FizzBuzz.evaluate)
  }
```

- `start.to(end): Seq[Int]`, where `Seq[_]` is a *type constructor* that, given a type `A`, returns a type of `Seq[A]`.
- For any value of type `Seq[A]`, `map: (A => B) => Seq[B]`.




- Therefore: need to implement function `FizzBuzz.evaluate: Int => String`.

## 2.5 Test-driven units

**Implement new `FizzBuzz` module**

A failing acceptance test drives *discovery* of

- A *unit*, `FizzBuzz`
- A function with a particular type, `Int => String`

```
object FizzBuzz {
  type Evaluator = Int => String

  val evaluate: Evaluator = { i =>
```

```
    ???
  }
}
```

***Types* are better than *comments* as *documentation*!**
Comments are not checkable, unlike types and tests.

**First part of *unit test*: example-based**

Manually write some *examples*.

```
class FizzBuzzSpec extends Specification { def is = s2"""
  ${FizzBuzz.evaluate(15) ==== "FizzBuzz"}
  ${FizzBuzz.evaluate(20) ==== "Buzz"}
  ${FizzBuzz.evaluate(6) ==== "Fizz"}
  ${FizzBuzz.evaluate(17) ==== "17"}
  """
}
```

## 2.6 Property-based tests

**The joy of property-based tests**

SCALA CHECK

ScalaCheck: a framework for writing *property-based* tests.

```
class FizzBuzzSpec extends Specification
    with ScalaCheck { def is = s2"""
  ${`Multiple of both 3 and 5 => "FizzBuzz"`} """
  def `Multiple of both 3 and 5 => "FizzBuzz"` =
    prop { i: Int => (i % 3 == 0 && i % 5 == 0) ==>
      { FizzBuzz.evaluate(i) ==== "FizzBuzz" }
    }
}
```

**Winning features**

- Auto-generates *random* tests for each property (100 by default).
- *Type-driven*: here, generates random `Int` values.

**Property-based tests (continued)**

The other three properties of interest:

```
def `Multiple of only 3 => "Fizz"` =
  prop { i: Int => (i % 3 == 0 && i % 5 != 0) ==>
    { FizzBuzz.evaluate(i) ==== "Fizz" }
  }

def `Multiple of only 5 => "Buzz"` =
  prop { i: Int => (i % 3 != 0 && i % 5 == 0) ==>
    { FizzBuzz.evaluate(i) ==== "Buzz" }
  }

def `Not a multiple of either 3 or 5 => number` =
  prop { i: Int => (i % 3 != 0 && i % 5 != 0) ==>
    { FizzBuzz.evaluate(i) ==== i.toString }
  }
```

## 2.7 Solving the FizzBuzz problem

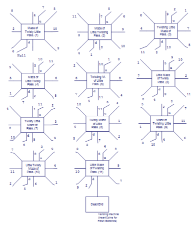
**A buggy and ugly solution**

```
// Buggy and ugly!
val evaluate: Evaluator = { i =>
  if (i % 3 == 0)
    "Fizz"
  else if (i % 5 == 0)
    "Buzz"
  else if (i % 3 == 0 && i % 5 == 0)
    "FizzBuzz"
  else
    i.toString
}
```

```
[info] FizzBuzzSpec
[info]   x FizzBuzz.evaluate(15) ==== "FizzBuzz"
[error]  'Fizz' is not equal to 'FizzBuzz'
            (FizzBuzzSpec.scala:14)
```

**Booleans are evil!**

**Maze of twisty little conditionals, all different**



- Too easy to write incorrect sequences of nested, combined conditionals.
- Overuse of Booleans is a type *smell*

### Why booleans are evil

### No help from type system

- Conditions can be arbitrary: depend on *any* combination of data.
- Multiple conditions: combinatorial explosion (two conditions led to four cases).
- Possibly overlapping conditions: order dependency subtleties.
- Possibly duplicated checking of the some condition.

### Pattern matching organizes information

```
val evaluate: Evaluator = { i =>
  (i % 3 == 0, i % 5 == 0) match {
    case (true,  false) => "Fizz"
    case (false, true)  => "Buzz"
    case (true,  true)  => "FizzBuzz"
    case (false, false) => i.toString
  }
}
```

### Winning features

- Visual *beauty* and clarity.
- No duplicated conditionals.
- No ordering dependency.
- *Type checker* verifies *full coverage* of cases.

**Example of non-exhaustive pattern matching**

```
val evaluate: Evaluator = { i =>
  (i % 3 == 0, i % 5 == 0) match {
    case (true,  false) => "Fizz"
    case (false, true)  => "Buzz"
    case (true,  true)  => "FizzBuzz"
    // case (false, false) => ???
  }
}
```

```
[warn] ...FizzBuzz.scala:46: match may not be exhaustive.
[warn] It would fail on the following input: (false, false)
[warn]     (i % 3 == 0, i % 5 == 0) match {
[warn]     ^
```

**Swift digression: pattern matching**

The same solution, in Swift:

```
typealias Evaluator = Int -> String

let evaluate: Evaluator = { i in
  switch (i % 3 == 0, i % 5 == 0) {
  case (true,  false): return "Fizz"
  case (false, true):  return "Buzz"
  case (true,  true):  return "FizzBuzz"
  case (false, false): return String(i)
  }
}
```

**Acceptance test passes**

```
[info] MainSpec
[info]   + Main.runToSeq(1, 16) ==== ‘strings for 1 to 16
```

**Done?**

No. Client wants more features.

# 3 FizzBuzz 2: user configuration

## 3.1 Adding new features

**Adding new features**

**Client wants to:**

- Choose two *arbitrary* divisors in place of 3 and 5
  - such as 4 and 7
- Choose other *arbitrary* words in place of "Fizz" and "Buzz"
  - such as "Moo" and "Quack"

## 3.2 Type-driven refactoring

**Type-driven refactoring**

**Types mean: refactoring is much more fun!**

- Add *new* tests.
- Change types and code: to make new tests *type check*.
- *Refactor* original code and tests: use new APIs.
- Keep passing the *old* tests.
- Delay writing code for new features.

**More features means more types**

Change `FizzBuzz.evaluate` to `Defaults.fizzBuzzer`:

```
def runToSeq(start: Int, end: Int): Seq[String] = {
  start.to(end).map(Defaults.fizzBuzzer)
}
```

Add new types to `FizzBuzz` module:

```
type Evaluator = Int => String
case class Config(pair1: (Int, String),
                  pair2: (Int, String))
type Compiler = Config => Evaluator

val compile: Compiler = {
  case Config((d1, w1), (d2, w2)) =>
    {  i =>
      ???
    }
}
```

**Extract original default configuration**

```
object Defaults {
  val fizzBuzzerConfig: Config =
    Config(3 -> "Fizz", 5 -> "Buzz")

  val fizzBuzzer: Evaluator =
    FizzBuzz.compile(fizzBuzzerConfig)

  // Useful to keep old implementation
  val oldFizzBuzzer: Evaluator = { i =>
    (i % 3 == 0, i % 5 == 0) match {
      case (true,  false) => "Fizz"
      case (false, true)  => "Buzz"
      case (true,  true)  => "FizzBuzz"
      case (false, false) => i.toString
    }
  }
}
```

**More types means more tests**

Write new property-based test over *arbitrary* user configurations:

```
val arbitraryConfig: Arbitrary[Config] = Arbitrary {
  for {
    (d1, d2, w1, w2) <-
      arbitrary[(Int, Int, String, String)]
  } yield Config(d1 -> w1, d2 -> w2)
}

def `Arbitrary pair of divisors: divisible by first` =
  arbitraryConfig { config: Config =>
    val evaluate = FizzBuzz.compile(config)
    val Config((d1, w1), (d2, _)) = config
    prop { i: Int => (i % d1 == 0 && i % d2 != 0) ==>
      { evaluate(i) ==== w1 }
    }
  }
```

## 3.3 Refining types

**Problem: coarse `Config` type**

```
[info]   ! Arbitrary divisor/word pair fizzBuzzers
[error]  ArithmeticException: :
```

```
A counter-example is 'Config((0,),(0,))':
java.lang.ArithmeticException: / by zero
(after 0 try)  (FizzBuzzSpec.scala:58)
```

- `0` as a divisor *crashes*!
- We discovered client's *underspecification*.
- Client says: meant to allow only divisors within `2` and `100`.

We need to:

- Add runtime *validation* when *constructing* `Config`.
- Refine `Config` random generator.

## 3.4 Validation

### Add (runtime) validation

*Runtime* precondition contract: Scala's `require` (throws exception on failure):

```scala
val DIVISOR_MIN = 2; val DIVISOR_MAX = 100

def validatePair(pair: (Int, String)) = pair match {
  case (d, _) =>
    require(d >= DIVISOR_MIN,
      s"divisor $d must be >= $DIVISOR_MIN")
    require(d <= DIVISOR_MAX,
      s"divisor $d must be <= $DIVISOR_MAX")
}

case class Config(pair1: (Int, String),
                  pair2: (Int, String)) {
  validatePair(pair1); validatePair(pair2)
}
```

### Data validation can be critical!

In real life, prefer non-exception type-based solution such as Scalaz validation.

### Digression: two ways to prevent Heartbleed

- Instead of C: use a dependently typed safe systems language such as ATS for compile-time TDD.
- Even with C: use good validation and testing practices.
  - A weaker type system is not an *excuse* to skip write tedious validation code or tests!

**Improve Config random generator**

```
val arbitraryConfig: Arbitrary[Config] =
  Arbitrary {
    for {
      d1 <- choose(DIVISOR_MIN, DIVISOR_MAX)
      d2 <- choose(DIVISOR_MIN, DIVISOR_MAX)
      w1 <- arbitrary[String]
      w2 <- arbitrary[String]
    } yield Config(d1 -> w1, d2 -> w2)
  }
```

**New test runs further, stills fails**

Refactor old code to `FizzBuzz.compile`, to pass old tests and new test.

```
  val compile: Compiler = {
    case Config((d1, w1), (d2, w2)) =>
      // Precompute, hence "compiler".
      val w = w1 + w2
      // Return an Evaluator.
      { i =>
        (i % d1 == 0, i % d2 == 0) match {
          case (true,  false) => w1
          case (false, true)  => w2
          case (true,  true)  => w
          case (false, false) => i.toString
        }
      }
  }
```

# 4 FizzBuzz 3: `FizzBuzzPop` and beyond

## 4.1 Generalize to more than two divisors

**Generalizing to more than two divisors**

**Client wants `FizzBuzzPop`!**

- Given three divisors (such as 3, 5, 7).
- Given three words (such as `"Fizz"`, `"Buzz"`, `"Pop"`).
- Compile to evaluator that given an integer prints:
  - either a string combining a subset of the three words, or

    - a numerical string if the integer is not a multiple of any of the three divisors

- Example: `21` should output `"FizzPop"`.

**Thought-driven development**

Software development is not primarily about *coding*, but *thinking*.

- Deep fact: solving a more general problem is often easier than solving the specific problem.
- There are four important numbers in the Universe:

  **0** emptiness

  **1** existence

  **2** other (relationship)

  **many** community

## 4.2 More features means more tests and types (again)

**More features means more tests**

Write new tests for new `Defaults.fizzBuzzPopper`:

```
def is = s2"""
${Defaults.fizzBuzzPopper(2) ==== "2"}
${Defaults.fizzBuzzPopper(21) ==== "FizzPop"}
${Defaults.fizzBuzzPopper(9) ==== "Fizz"}
${Defaults.fizzBuzzPopper(7) ==== "Pop"}
${Defaults.fizzBuzzPopper(35) ==== "BuzzPop"}
"""
```

Change configuration: to `Seq` of pairs, instead of just two:

```
val fizzBuzzPopperConfig: Config =
  Config(Seq(
    3 -> "Fizz", 5 -> "Buzz", 7 -> "Pop"
  ))
val fizzBuzzPopper: Evaluator =
  FizzBuzz.compile(fizzBuzzPopperConfig)
```

### More tests means more (or changed) types

```
[error] ...Defaults.scala:29: not enough arguments for
  method apply:
  (pair1: (Int, String), pair2: (Int, String))
  com.franklinchen.FizzBuzz.Config in object Config
[error] Unspecified value parameter pair2.
[error]     Config(Seq(
[error]           ^
```

Change *type* `Config` to allow a sequence of pairs:

```
case class Config(pairs: Seq[(Int, String)]) {
  pairs.foreach(validatePair)
}
```

Note how our iterative development process promotes *reuse* (here, of validation logic).

### Fix remaining type errors

Refactoring reveals need to implement case of more than two divisors.

```
val compile: Compiler = {
  case Config(Seq((d1, w1), (d2, w2))) =>
    val w = w1 + w2

    { i =>
      (i % d1 == 0, i % d2 == 0) match {
        case (true,  false) => w1
        case (false, true)  => w2
        case (true,  true)  => w
        case (false, false) => i.toString
      }
    }
  case _ => // TODO handle more than 2
    { i => ??? }
}
```

### General observations

- Return a sum of a subset of the configured words, if there is any divisor match.
- If there is *no* divisor match, return the numerical string.

**More computation means more types**

Compile each divisor to a "rule" that awaits input.

```
type Rule = Int => String

val buildRule: ((Int, String)) => Rule = {
  case (n, word) => { i =>
    if (i % n == 0) word else ""
  }
}
```

`FizzBuzz` **demo time!**

- Two volunteers: to play role of `Rule`.
- One volunteer: to combine sub-results.

## 4.3 Demo

**Demo explanation**

- Given a sequence of rules and an integer: apply all the rules to the integer, then combine the partial results.

**Assemble the types**

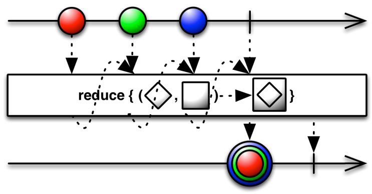
```
type Evaluator = Int    => String
type Compiler  = Config => Evaluator
type Rule      = Int    => String

val compile: Compiler = { case Config(pairs) =>
  val rules: Seq[Rule] = pairs.map(buildRule) // compile

  // Return an Evaluator.
  { i =>
    val words: Seq[String] = rules.map { rule => rule(i) }
    val combined: String = words.reduce { (x, y) => x + y }
    if (combined == "") i.toString else combined
  }
}
```

### A note on `reduce`

For any value of type `Seq[A]`, `reduce: ((A, A) => B) => B`.




Example: for `Seq[String]`, reduction with string concatenation `+` returns the concatenation of all the strings in the sequence.

### Test failure: coarse types again

```
nfo]   x Arbitrary pair of divisors: divisible by first
rror]  A counter-example is 'Config(List((8,), (32,)))'
(after 0 try)
rror]  A counter-example is '32405464'
(after 0 try - shrinked ('1150076488' -> '32405464'))
rror]  '32405464' is not equal to ''
(FizzBuzzSpec.scala:71)
```

### Demo time!

- Configuration: `Seq(3 -> "", 5 -> "Buzz")`
- Input: `2`
- Output: should be `""`
- Output was: `"2"`

### Property-based testing rescued us again!

### Be honest: would you have caught this bug manually?

- I didn't.
- I never wrote `FizzBuzzPop` examples testing empty strings.
- Property-based testing reveals *unexpected* corner cases.
  - (Empty "fizz" and "buzz" word strings).

## 4.4 `Option[A]` type

**An empty string is *not* equivalent to no string**

**Presence of something "empty" is *not* equivalent to no thing.**
Sending someone an empty email versus not sending any email.

Many programming languages get this wrong.

### `Option[A]` type

`Option[A]` is one of two possibilities:

- `None`
- `Some(a)` wraps a value `a` of type `A`.

For example, `Some("")` is not the same as `None`.

```
val fizzFor3     = Some("") // multiple of 3
val buzzFor3     = None     // not multiple of 5
val fizzbuzzFor3 = Some("") // fizzed ""

val fizzFor2     = None     // not multiple of 3
val buzzFor2     = None     // not multiple of 5
val fizzbuzzFor2 = None     // not multiple of any
```

### Cleaning up the types

```
// was: type Rule = Int => String
type Rule = Int => Option[String]
```

Useful type errors:

```
und    : String
quired: Option[String]
     word
     ^
und    : String("")
quired: Option[String]
     ""
     ^
und    : Seq[Option[String]]
quired: Seq[String]
     val words: Seq[String] = rules map { rule => rule(i) }
                                    ^
```

**Fix the type errors: our rule builder**

```
type Rule = Int => Option[String]

val buildRule: ((Int, String)) => Rule = {
  case (n, word) => { i =>
    (i % n == 0).option(word)
  }
}
```

**Demo time!**

- (Instructions: circle what you write to wrap it with `Some`)
- Configuration: `Seq(3 -> "", 5 -> "Buzz")`
- Input: `2`
- Output: should be `""`

**Fix the type errors: our compiler**

```
val compile: Compiler = { case Config(pairs) =>
  val rules: Seq[Rule] = pairs map buildRule

  { i =>
    val wordOptions: Seq[Option[String]] =
      rules.map { rule => rule(i) }
    val combinedOption: Option[String] =
      wordOptions.reduce(addOption)
    combinedOption.getOrElse(i.toString)
  }
}
```

- We need to write: `addOption`
- Scala standard library provides: `getOrElse`

**"Addition" for `Option[String]`**

```
def addOption(a1: Option[String],
              a2: Option[String])
    : Option[String] = (a1, a2) match {
  case (Some(s1), None)     => Some(s1)
  case (None,     Some(s2)) => Some(s2)
  case (Some(s1), Some(s2)) => Some(s1 + s2)
  case (None,     None)     => None
}
```

**Getting `A` back out of `Option[A]`**

**Do not lose information!**
`getOrElse` inspects the
and either

- returns the value `v` inside a `Some(v)`,
- or else returns the specific default value.

**Swift has two option types**
Swift calls them "optionals".

**Normal optional type**

- `A?` is a type for each type `A`.
- Must unwrap explicitly.

**Implicit unwrapped optional type**

- `A!` is a type for each type `A`.
- Unfortunate type hole:

> If you try to access an implicitly unwrapped optional when it does not contain a value, you will trigger a runtime error.

## 4.5 Transform information; don't destroy it

**Transform information; don't destroy it**
Did you notice? Our new code no longer uses `if`.

**Bug cause: destroying information, using `if`**

- Building a rule: `if (i % n == 0) word else ""`
- Obtaining a final string answer: `if (combined == "") i.toString else combined`

**Transforming information**

- To `Option[String]`: `(i % n == 0).option(word)`
- To `String`: `combinedOption.getOrElse(i.toString)`

**Type-directed design tip**
We could have saved trouble *up front*, by using precise *types*.

- Avoid `if`, when possible.
- Avoid `String` (but required at I/O boundaries of program).

# 5 Parallel FizzBuzz

### Parallelism

- Use of *map*: parallelizable; there are high-performance parallel collections for Scala.
- Use of *reduce*: parallelizable because of the monoid property:

  **`Option[String]` is a Monoid**

  - There is an identity element (`None`).
  - There is a binary associative operator (`addOption`).

  **Demo time!**

  - Configuration: `Seq(3 -> "Fizz", 5 -> "Buzz", 7 -> "Pop", 2 -> "Boom")`
  - Input: `42`
  - Output: `"FizzPopBoom"`

### Final parallelized code

```
val parallelCompile: Compiler = { case Config(pairs) =>
  val rules = pairs.toArray
    toArray.
    toPar.
    map(buildRule)

  { i: Int => rules.
    map { rule => rule(i) }.
    reduce(addOption).
    getOrElse(i.toString)
  }
}
```

**Coding style tip**
This level of conciseness is not always best: maybe too "clever"?

**Parallelism summary**
We discovered a theoretical speedup for generalized `FizzBuzz`:

- Sequential: $O(n)$
- Parallel: $O(\log n)$ (given $\log n$ processors, and omitting some technical subtleties)

Also, driver outer loop can be sped up:

- Sequential loop on 1 to $m$: $O(m)$
- Parallel loop: $O(1)$ (given $m$ processors)

# 6 Conclusion

### Conclusion

- *Tests* are useful.
- *Types* are useful.
- Tests and types work well together, to drive design and program evolution.
- Modern typed languages such as Scala promote fun, correct programming!
- It's a great time to be learning and using a modern typed language: there is excitement over Apple's Swift.

### Code, slides, article

- Go to `https://github.com/franklinchen/talk-on-type-directed-tdd-using-fizzbuzz`
- The article has more detail omitted in the presentation slides.
- The hyperlinks in all provided PDFs are clickable.

# Appendix

### Some free online courses on modern typed functional programming

- Using Scala: Functional Programming Principles in Scala on Coursera, taught by Martin Odersky (inventor of Scala)
- Using Haskell: Introduction to Functional Programming on EdX, taught by Erik Meijer (Haskell hero)

### Some articles

- The problem with Booleans and conditionals
- The problem with exceptions
- Principled validation using types
- Preventing Heartbleed by using a dependently typed language
- Twitter's use of monoids
- Side-by-side comparison of basic Scala and Swift constructs