Type-Directed TDD in Rust

A case study using FizzBuzz

Franklin Chen
http://franklinchen.com/

July 21, 2014
Pittsburgh Code and Supply

Outline

Goals of this presentation

- Give a taste of a practical software development process that is:
  - test-driven
  - type-directed
- Show everything for real (using Rust):
  - project build process
  - testing frameworks
  - all the code
- Use `FizzBuzz` because:
  - problem: easy to understand
  - modifications: easy to understand
  - fun!
- Encourage you to explore a modern typed language; now is the time!
  - Recently, Apple ditched Objective C for its new language Swift!

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.

Type systems

What is a type system?

A syntactic method to prove that bad things can't happen.

"Debating" types "versus" tests?

- Let's use both types and tests!
- But: use a good type system, not a bad one.

Some decent practical typed languages

- OCaml: 20 years old
- Haskell: 20 years old
- Scala: 10 years old
- Swift: $<$ 2 months old
- Rust (still not at 1.0!)

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
- Swift: announced by Apple on June 2, 2014!
- Rust: not yet version 1.0

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”.

Starter code: main driver

Rust: a modern systems programming language for efficiency and safety in time, space, and concurrency.

```
fn main() {
  // Will not compile yet!
  for result in run_to_seq(1i, 100).iter() {
    println!("{}", result)
  }
}
```

- 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.

Compiling and testing with Cargo

Cargo: build tool for Rust

Features

- Library dependency tracking.
- `cargo build`
- `cargo test`

My wish list, based on Scala SBT

- Triggered compilation and testing
- Interactive REPL

First compilation failure

src/main.rs:

```
$ cargo build
src/main.rs:16:19:
error: unresolved name ‘run_to_seq‘
```

Write type-directed stub

```
fn main() {
  for result in run_to_seq(1i, 100).iter() {
    println!("{}", result)
  }
}

fn run_to_seq(start: int, end: int) -> Vec<String> {
  fail!()
}
```

Write wanted type signature

`fail!` is convenient for stubbing.

- In Rust standard library
- Causes whole task to fail

Write acceptance test (simplified)

```
#[test]
fn test_1_to_16() {
  let expected = vec![
    "1", "2", "Fizz", "4", "Buzz", "Fizz",
    "7", "8", "Fizz", "Buzz", "11", "Fizz",
    "13", "14", "FizzBuzz", "16",
    ]
    .iter()
    .map(|&s| s.to_string())
    .collect();
  assert_eq!(run_to_seq(1, 16), expected)
}
```

TDD in Swift

```
class MainSpec: XCTestCase {
  func test1to16() {
    let expected: [String] = [
      "1", "2", "Fizz", "4", "Buzz", "Fizz",
      "7", "8", "Fizz", "Buzz", "11", "Fizz",
      "13", "14", "FizzBuzz", "16"
    ]

    XCTAssert(Main.runToSeq(1, 16) == expected)
  }
}
```

Test passes type check, but fails

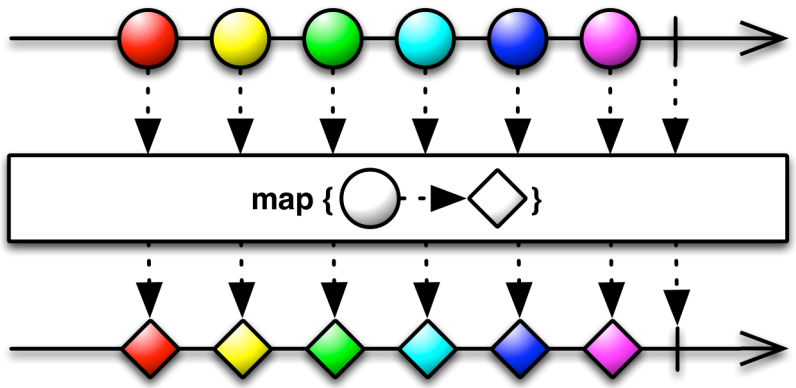
```
$ cargo test
task 'test::test_1_to_16' failed at 'write run_to_seq',
...src/main.rs:37
```

Outside-in: for a `fizzbuzz` module

Types are shapes to assemble logically.

```
fn run_to_seq(start: int, end: int) -> Vec<String> {
  range_inclusive(start, end)
    .map(fizzbuzz::evaluate)
    .collect()
}
```

- `range(include, exclude)` returns an iterator.
- `map` takes an iterator of one type to an iterator of another:



- Therefore: need to implement function `fizzbuzz::evaluate: int -> String`.

Swift version of driver

```
func runToSeq(i: Int, j: Int) -> [String] {
  return Array(i...j).map(Defaults.fizzBuzzer)
}
```

Implement new `fizzbuzz` module

A failing acceptance test drives discovery of

- A unit, `fizzbuzz`
- A function with a particular type, `int -> String`

```
pub fn evaluate(i: int) -> String {
  fail!()
}
```

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.

```
#[test] fn test_15() {
  assert_eq!(evaluate(15), "FizzBuzz".to_string())
}

#[test] fn test_20() {
  assert_eq!(evaluate(20), "Buzz".to_string())
}

#[test] fn test_6() {
  assert_eq!(evaluate(6), "Fizz".to_string())
}

#[test] fn test_17() {
  assert_eq!(evaluate(17), "17".to_string())
}
```

The joy of property-based tests

QuickCheck for Rust: a framework for writing property-based tests.

```
#[quickcheck]
fn multiple_of_both_3_and_5(i: int) -> TestResult {
  if i % 3 == 0 && i % 5 == 0 {
    TestResult::from_bool(evaluate(i) ==
                          "FizzBuzz".to_string())
  } else {
    TestResult::discard()
  }
}
```

Winning features

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

Property-based testing for Swift?

I hope someone writes a property-based testing framework for Swift!

Property-based tests (continued)

```
#[quickcheck]
fn multiple_of_only_3(i: int) -> TestResult {
  if i % 3 == 0 && i % 5 != 0 {
    TestResult::from_bool(evaluate(i) == "Fizz".to_string())
  } else {
    TestResult::discard()
  }
}

#[quickcheck]
fn not_multiple_of_3_and_5(i: int) -> TestResult {
  if i % 3 != 0 && i % 5 != 0 {
    TestResult::from_bool(evaluate(i) == i.to_string())
  } else {
    TestResult::discard()
  }
}
```

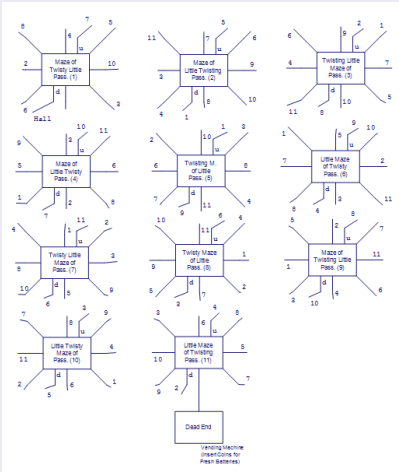
A buggy and ugly solution

```
// Buggy and ugly!
if i % 3 == 0 {
  "Fizz".to_string()
} else if i % 5 == 0 {
  "Buzz".to_string()
} else if i % 3 == 0 && i % 5 == 0 {
  "FizzBuzz".to_string()
} else {
  i.to_string()
}
```

```
$ cargo test
task 'fizzbuzz::test::test_15' failed at
  'assertion failed: `(left == right) && (right == left)`
  (left: `Fizz`, right: `FizzBuzz`)', .../src/fizzbuzz.rs:21
```

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

```
pub fn evaluate(i: int) -> String {
  match (i % 3 == 0, i % 5 == 0) {
    (true,  false) => "Fizz".to_string(),
    (false, true)  => "Buzz".to_string(),
    (true,  true)  => "FizzBuzz".to_string(),
    (false, false) => i.to_string(),
  }
}
```

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

```
pub fn evaluate(i: int) -> String {
  match (i % 3 == 0, i % 5 == 0) {
    (true,  false) => "Fizz".to_string(),
    (false, true)  => "Buzz".to_string(),
    (true,  true)  => "FizzBuzz".to_string(),
    // (false, false) => i.to_string(),
  }
}
```

```
$ cargo test
.../src/fizzbuzz.rs:16:5: 21:6 error:
non-exhaustive patterns: `(false, false)` not covered
```

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

```
$ cargo test
test test::test_1_to_16 ... ok
```

Done?

No. Client wants more 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"`

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`:

```
mod defaults;

fn run_to_seq(start: int, end: int) -> Vec<String> {
  range_inclusive(start, end)
    .map(defaults::fizzbuzzer)
    .collect()
}
```

Add new types to FizzBuzz module:

```
pub struct Config(pub Pair, pub Pair);

pub fn evaluate(Config((d1, w1), (d2, w2)): Config, i: int)
                  -> String {
  fail!()
}
```

New default configuration

```
// Rust limitation: cannot be static variable
// because Config stores String.
fn fizzbuzzer_config() -> Config {
  Config((3, "Fizz".to_string()),
         (5, "Buzz".to_string()))
}

pub fn fizzbuzzer(i: int) -> String {
  fizzbuzz::evaluate(fizzbuzzer_config(), i)
}
```

More types means more tests

Write new property-based test over arbitrary user configurations:

```
#[quickcheck]
fn d1_but_not_d2(p1: Pair,
                 p2: Pair,
                 i: int) -> TestResult {
  let (d1, w1) = p1.clone();
  let (d2, _) = p2;

  let config = Config(p1, p2);

  if i % d1 == 0 && i % d2 != 0 {
    TestResult::from_bool(fizzbuzzer(i) == w1)
  } else {
    TestResult::discard()
  }
}
```

Problem: coarse `Config` type

```
$ cargo build
task 'fizzbuzz::test::d1_but_not_d2' failed at
  '[quickcheck] TEST FAILED (runtime error).
  Arguments: ((0, ), (0, ), 0)'
```

- `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.

Add (runtime) validation

Runtime precondition contract: Rust's `assert!` (very primitive; fails a task on failure):

```
static DIVISOR_MIN: int = 2; static DIVISOR_MAX: int = 100;

fn validate_pair(&(d, _): &Pair) {
  assert!(d >= DIVISOR_MIN,
          "divisor {} must be >= {}", d, DIVISOR_MIN);
  assert!(d <= DIVISOR_MAX,
          "divisor {} must be <= {}", d, DIVISOR_MAX);
}

impl Config {
  pub fn new(pair1: Pair, pair2: Pair) -> Config {
    validate_pair(&pair1); validate_pair(&pair2);
    Config(pair1, pair2)
  }
}
```

A note on error handling

- Rust does not have exceptions!
  - Exceptions are evil because they escape the type system.
- Rust task failures are brutal.
- Outside scope of this presentation: principled type-based error handling using `Result<T, E>`:

Data validation can be critical!

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

```
#[quickcheck]
fn d1_but_not_d2(p1: Pair, p2: Pair, i: int) -> TestResult {
  let (d1, w1) = p1.clone();
  let (d2, _) = p2;

  if (d1 >= DIVISOR_MIN && d1 <= DIVISOR_MAX)
    && (d2 >= DIVISOR_MIN && d2 <= DIVISOR_MAX) {
    let config = Config::new(p1, p2);

    if i % d1 == 0 && i % d2 != 0 {
      TestResult::from_bool(evaluate(config, i) == w1)
    } else { TestResult::discard() }
  } else { TestResult::discard() }
}
```

New test runs further, stills fails

Refactor old code to `fizzbuzz::evaluate`, to pass old tests and new test.

```
pub fn evaluate(Config((d1, w1), (d2, w2)): Config, i: int)
                  -> String {
  match (i % d1 == 0, i % d2 == 0) {
    (true,  false) => w1,
    (false, true)  => w2,
    (true,  true)  => w1 + w2,
    (false, false) => i.to_string(),
  }
}
```

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"`).
- Create 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

More features means more tests

Write new tests for new `defaults::fizzbuzzpopper`:

```
#[test] fn test_fizzbuzzpopper_2() {
  assert_eq!(fizzbuzzpopper(2), "2".to_string())
}
#[test] fn test_fizzbuzzpopper_21() {
  assert_eq!(fizzbuzzpopper(21), "FizzPop".to_string())
}
#[test] fn test_fizzbuzzpopper_9() {
  assert_eq!(fizzbuzzpopper(9), "Fizz".to_string())
}
#[test] fn test_fizzbuzzpopper_7() {
  assert_eq!(fizzbuzzpopper(7), "Pop".to_string())
}
#[test] fn test_fizzbuzzpopper_35() {
  assert_eq!(fizzbuzzpopper(35), "BuzzPop".to_string())
}
```

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

More tests means more (or changed) types

```
error: this function takes 2 parameters
  but 1 parameter was supplied
Config(vec![(3, "Fizz".to_string()),
            (5, "Buzz".to_string())])
```

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

```
pub struct Config(pub Vec<Pair>);

impl Config {
  pub fn new(pairs: Vec<Pair>) -> Config {
    for pair in pairs.iter() {
      validate_pair(pair);
    }
    Config(pairs)
  }
}
```

Note how our iterative development process promotes reuse (here, of

Fix remaining type errors

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

```
pub fn evaluate(Config(pairs): Config, i: int)
                    -> String {
  // Can crash! And incorrect except for 2 pairs.
  let (d1, ref w1) = pairs[0];
  let (d2, ref w2) = pairs[1];

  match (i % d1 == 0, i % d2 == 0) {
    (true,  false) => (*w1).clone(),
    (false, true)  => (*w2).clone(),
    (true,  true)  => *w1 + *w2,
    (false, false) => i.to_string(),
  }
}
```

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

Associate each divisor with a "rule" that awaits input.

```
fn rule(&(n, ref word): &Pair, i: int) -> String {
  if i % n == 0 {
    (*word).clone()
  } else {
    String::new()
  }
}
```

FizzBuzz demo time!

- Two volunteers: each to play role of `Rule`.
- One "manager" to combine two sub-results.

Demo explanation

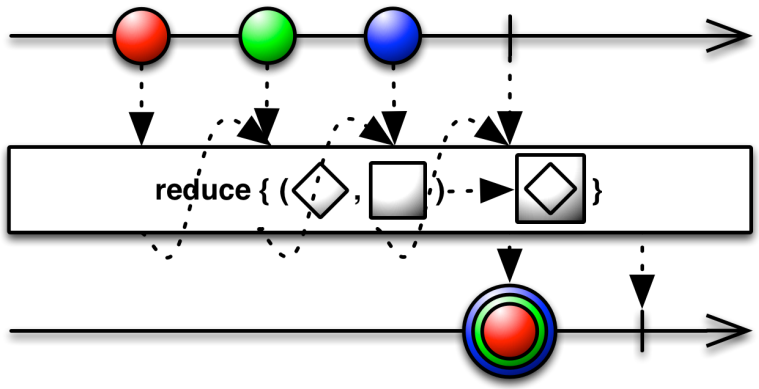
- Given a sequence of rules and an integer: apply all the rules to the integer, then combine the partial results.

Assemble the types

```
pub fn evaluate(Config(pairs): Config, i: int)
                  -> String {
  let combined: String = pairs.iter()
            .map(|pair| rule(pair, i))
            .fold(String::new(),
                  |result, s| result + s);
  if combined.is_empty() {
    i.to_string()
  } else {
    combined
  }
}
```

A note on `fold`

For any value of type `Iterator<A>`, we can apply `fold: (B, |B, A| -> B) -> B`.



Example: for `Vec<String>`, fold with string concatenation `+` returns the concatenation of all the strings in the vector.

Test failure: coarse types again

```
$ cargo test
task 'fizzbuzz::test::d1_but_not_d2' failed at
  '[quickcheck] TEST FAILED.
  Arguments: ((2, ), (3, ), 2)'
```

Demo time!

- Configuration: `vec![(3, ""), (5, "Buzz")]`
- Input: `9` (note: divisible by `2`)
- Output: should be `""` (because of the part divisible by `3`)
- Output was: `"9"`

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).

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(String::empty()) is not the same as None.

```
let fizzFor3     = Some(String::new()) // multiple of 3
let buzzFor3     = None                // not multiple of 5
let fizzbuzzFor3 = Some(String::new()) // fizzed ""

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

Cleaning up the types

```
// rule was: (&Pair, int) -> String
//      now: (&Pair, int) -> Option<String>
```

Useful type errors:

```
mismatched types: expected ‘Option<String>‘ but found
                                                ‘String‘
  if i % n == 0 {
    (*word).clone()
  } else {
    String::new()
  }

failed to find an implementation of trait Str for
                                        Option<String>
  result + s
```

Fix the type errors: our rule builder

```
fn rule(&(n, ref word): &Pair, i: int) -> Option<String> {
  if i % n == 0 {
    Some((*word).clone())
  } else {
    None
  }
}
```

Demo time!

- (Instructions: for result `Some(s)`, hold up the string, else don't hold up anything)
- Configuration: `vec![(3, ""), (5, "Buzz")]`
- Input: `9`
- Output: now correctly is `""`

Fix the type errors: our compiler

```
pub fn evaluate(Config(pairs): Config, i: int)
                  -> String {
  let combined: Option<String> = pairs.iter()
              .map(|pair| rule(pair, i))
              .fold(None, add_option);
  combined.unwrap_or_else(|| i.to_string())
}
```

- We need to write: `addOption`
- Rust standard library provides: `unwrap_or_else`

"Addition" for Option[String]

```
fn add_option(a1: Option<String>, a2: Option<String>)
                -> Option<String> {
  match (a1, a2) {
    (Some(s1), None)     => Some(s1),
    (None,     Some(s2)) => Some(s2),
    (Some(s1), Some(s2)) => Some(s1 + s2),
    (None,     None)     => None,
  }
}
```

Getting `A` back out of `Option<A>`

Do not lose information!

`unwrap_or_else` inspects the and either

- returns the value `v` inside a `Some(v)`,
- or else returns the value from a closure.

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.*

The same thing, in Swift

```
func addOption(a1: String?, a2: String?) -> String? = {
  switch (a1, a2) {
  case (let .Some(s1), .None):          return .Some(s1)
  case (.None, let .Some(s2)):          return .Some(s2)
  case (let .Some(s1), let .Some(s2)): return .Some(s1+s2)
  case (.None, .None):                  return .None
  }
}
```

Transform information; don't destroy it

Our complete code only uses `if` in one place.

Bug cause: destroying information, using `if`

- `if i % n == 0 { word } else { String::new() }`
- `if combined.is_empty() {i.to_string()} else {combined}`

Transforming information

- To `Option[String]`:
  `if i % n == 0 { Some((*word).clone()) } else { None }`
- Back to `String`: `combined.unwrap_or_else(|| i.to_string())`

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).

Parallelism

Some easy parallelism possible (not yet in Rust standard libraries):

- Use of map.
- Use of fold: parallelizable because of the monoid property:

`Option<String>` is a Monoid

- There is an identity element (`None`).
- There is a binary associative operator (`add_option`).
- Fantastically important in practice!

Final (hypothetical) parallelized code

```
pub fn evaluate(Config(pairs): Config, i: int)
                  -> String {
  pairs.par
       .iter()
       .map(|pair| rule(pair, i))
       .reduce(add_option)
       .unwrap_or_else(|| i.to_string())
}
```

Coding style tip

This level of conciseness is not always best: maybe too "clever"?

Final demo!

Demo time!

- Configuration:
  `vec![(3, "Fizz"), (5, "Buzz"), (7, "Pop"), (2, "Boom")]`
- Tree of volunteers to simulate concurrency:
  - Four at leaves.
  - Two "middle managers" each handling two leaves.
  - One top-level manager handling two middle managers.
- Input: `42`
- Output: `"FizzPopBoom"`

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)

Current (July 2014) Rust limitations

- Rust closures: still limited (work in progress!!).
- Scala/Swift/Haskell/etc. have unrestricted closures: less complex types, easy staged compilation.
- Needs standard libraries for parallelism, using concurrency primitives such as `spawn`.
- Need faster compiler, build system.
- Need better test frameworks.

Bonus: the final code in Swift

```
typealias Evaluator = Int -> String
typealias Config = [(Int, String)]
typealias Compiler = Config -> Evaluator
typealias Rule = Int -> String?

let buildRule: ((Int, String)) -> Rule = { n, word in
  { i in return (i % n == 0) ? word : nil } }
}

let compile: Compiler = { pairs in
  let rules: [Rule] = pairs.map(buildRule)
  return { i in
    let wordOptions = rules.map { rule in rule(i) }
    let combinedOption = wordOptions.reduce(nil, addOption)
    if let combined = combinedOption { return combined }
                                     else { return String(i) }
  }
}
```

Future work

- Asynchronous
- Real-time
- Interactive

Conclusion

- Tests are great.
- Types are great.
- Tests and types work hand in hand, driving design and program evolution.
- Modern typed languages such as Rust promote fun, correct programming!
- It's a great time to be learning and using a modern typed language.

Code, slides, article

- https://github.com/franklinchen/type-directed-tdd-rust
- The article has more detail omitted in the presentation.
- The hyperlinks in all provided PDFs are clickable.
- Scala: https://github.com/franklinchen/talk-on-type-directed-tdd-using-fizzbuzz
- Swift: https://github.com/franklinchen/fizzbuzz-swift

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)