What is Functional Programming, anyway?

And why do we care?

Ricky Elrod
Youngstown State University

December 3, 2016

What does “functional programming” mean?

...and what are the implications?

- Functional programming is simply **programming with functions**.

What does “functional programming” mean?

...and what are the implications?

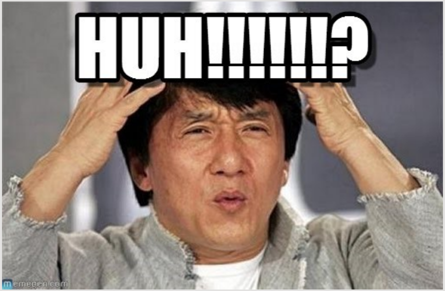
- Functional programming is simply **programming with functions**.
- But what is a function?
    - A function is **a relation mapping elements of one set to elements of another set**.
    - Just like in your high school algebra class!

Referential Transparency

- The central notion to the idea of functional programming is known as *referential transparency*.

Referential Transparency

- The central notion to the idea of functional programming is known as *referential transparency*.
- Referential transparency leads to *program compositionality*.


HUH!!!!!!?
memegen.com

Referential Transparency

- Think of some expression in a programming language of your choice.

Referential Transparency

- Think of some expression in a programming language of your choice.
- Now, mentally evaluate that expression and replace the expression in the code with the result of evaluating it.

Referential Transparency

- Think of some expression in a programming language of your choice.
- Now, mentally evaluate that expression and replace the expression in the code with the result of evaluating it.
- Does the behavior of the program change?
  - If no, then the expression is referentially transparent.
  - If yes, then the expression is not referentially transparent.

Referential Transparency

An abstract example

Program 1

```
val x = foobar(args)
val y1 = something(x)
val y2 = something(x)
```

Program 2

```
val y1 = something(foobar(args))
val y2 = something(foobar(args))
```

Referential Transparency

An abstract example

Program 1

```
val x = foobar(args)
val y1 = something(x)
val y2 = something(x)
```

Program 2

```
val y1 = something(foobar(args))
val y2 = something(foobar(args))
```

- If the two programs produce the same output, then `foobar` is referentially transparent.

Referential Transparency

An abstract example

Program 1

```
val x = foobar(args)
val y1 = something(x)
val y2 = something(x)
```

Program 2

```
val y1 = something(foobar(args))
val y2 = something(foobar(args))
```

- If the two programs produce the same output, then `foobar` is referentially transparent.
- In a *purely functional programming language*, every function is referentially transparent (i.e., pure).

So why do we care?

Compositionality - Frege's Principle

- Staying true to this central thesis of functional programming leads to composable programs.

So why do we care?

Compositionality - Frege’s Principle

- Staying true to this central thesis of functional programming leads to composable programs.
- Smaller programs can coherently be combined (composed) to make larger, more interesting programs, with little effort.

So why do we care?

Compositionality - Frege's Principle

- Staying true to this central thesis of functional programming leads to composable programs.
- Smaller programs can coherently be combined (composed) to make larger, more interesting programs, with little effort.

So why do we care?

Compositionality - Frege's Principle

- Staying true to this central thesis of functional programming leads to composable programs.
- Smaller programs can coherently be combined (composed) to make larger, more interesting programs, with little effort.
- We get fewer bugs because we can confidently determine program behavior by determining behavior of the smaller parts from which they are comprised.

So why do we care?

Compositionality - Frege's Principle

- Staying true to this central thesis of functional programming leads to composable programs.
- Smaller programs can coherently be combined (composed) to make larger, more interesting programs, with little effort.
- We get fewer bugs because we can confidently determine program behavior by determining behavior of the smaller parts from which they are comprised.
- Codebases scale infinitely and cleanly by composing more and more subprograms.

So why do we care?

Compositionality - Frege’s Principle

- Staying true to this central thesis of functional programming leads to composable programs.
- Smaller programs can coherently be combined (composed) to make larger, more interesting programs, with little effort.
- We get fewer bugs because we can confidently determine program behavior by determining behavior of the smaller parts from which they are comprised.
- Codebases scale infinitely and cleanly by composing more and more subprograms.
- We (force ourselves to) write deterministic algorithms. Reasoning is easier.

A quick review

before we move on...

- What is functional programming?

A quick review

before we move on...

- What is functional programming?
  - Functional programming is a means of programming in which expressions are refrerentially transparent.

A quick review

before we move on...

- What is functional programming?
  - Functional programming is a means of programming in which expressions are refrerentially transparent.
- What is referential transparency?

A quick review

before we move on...

- What is functional programming?
  - Functional programming is a means of programming in which expressions are refrerentially transparent.
- What is referential transparency?
  - The ability to replace an expression by its result.

Functional programming is a commitment to preserving referential transparency.

We have tools which help us to achieve this commitment.

Tool #1: Parametric Polymorphism

Parametric Polymorphism (a.k.a. “parametricity”)

- *Philip Wadler (1989) - “Theorems for Free”*: Write down the definition of a polymorphic function on a piece of paper. Tell me its type, but be careful not to let me see the function’s definition. I will tell you a theorem that the function satisfies. The purpose of this paper is to explain the trick.

Parametric Polymorphism (a.k.a. “parametricity”)

- Consider a function of this type: `int add10(int a)`
  - This function has $(2^{32})^{2^{32}} = 18,446,744,073,709,551,616$ possible implementations.

Parametric Polymorphism (a.k.a. "parametricity")

- Consider a function of this type: `int add10(int a)`
  - This function has $(2^{32})^{2^{32}} = 18,446,744,073,709,551,616$ possible implementations.
  - From the type alone, that is all we know about this function. :-(

Parametric Polymorphism (a.k.a. “parametricity”)

- Consider a function of this type: `int add10(int a)`
    - This function has $(2^{32})^{2^{32}} = 18,446,744,073,709,551,616$ possible implementations.
    - From the type alone, that is all we know about this function. :-(
    - From the name, we might form a suspicion that it adds 10 to its argument and returns the result.

Parametric Polymorphism (a.k.a. “parametricity”)

Another monomorphic example

- Consider `List<int> demo(List<int> xs)`
  - Does it add 6 to every element?

Parametric Polymorphism (a.k.a. “parametricity”)

Another monomorphic example

- Consider `List<int> demo(List<int> xs)`
  - Does it add 6 to every element?
  - Does it filter out and remove every prime number?

Parametric Polymorphism (a.k.a. “parametricity”)

Another monomorphic example

- Consider `List<int> demo(List<int> xs)`
  - Does it add 6 to every element?
  - Does it filter out and remove every prime number?
  - Who knows?

Parametric Polymorphism (a.k.a. “parametricity”)

Another monomorphic example

- Consider `List<int> demo(List<int> xs)`
  - Does it add 6 to every element?
  - Does it filter out and remove every prime number?
  - Who knows?
  - We can’t generate any theorem based on the type alone.

Parametric Polymorphism (a.k.a. “parametricity”)

A polymorphic example

- Consider `<A> List<A> demo(List<A> xs)`
  - *Theorem*: The list returned by `demo` will only ever contain elements which appeared in the input.

Parametric Polymorphism (a.k.a. “parametricity”)

A polymorphic example

- Consider `<A> List<A> demo(List<A> xs)`
  - *Theorem*: The list returned by `demo` will only ever contain elements which appeared in the input.
  - **Otherwise, it would not have compiled!**

Parametric Polymorphism (a.k.a. “parametricity”)

A polymorphic example

- Consider `<A> List<A> demo(List<A> xs)`
  - *Theorem*: The list returned by `demo` will only ever contain elements which appeared in the input.
  - **Otherwise, it would not have compiled!**
  - I can’t tell you what the function does, but I can certainly tell you a lot about things which it does **not** do!

Parametric Polymorphism (a.k.a. “parametricity”)

A polymorphic example

- Consider `<A> List<A> demo(List<A> xs)`
  - *Theorem*: The list returned by `demo` will only ever contain elements which appeared in the input.
  - **Otherwise, it would not have compiled!**
  - I can’t tell you what the function does, but I can certainly tell you a lot about things which it does **not** do!
  - And I didn’t have to put much effort into it, to be able to do that!

Tool #2: Treating programming language as if they are *total*

Fast And Loose Reasoning is Morally Correct

2006 - Danielsson, Hughes, Jansson, Gibbons

- Functional programmers often reason about programs as if they were written in a total language, expecting the results to carry over to non-total (partial) languages. We justify such reasoning.

ON A SCALE OF ONE TO EVEN
I LITERALLY CAN'T
memegenerator.net

So what does it mean?

- Consider `bool isOdd(int a) = ...`

So what does it mean?

- Consider `bool isOdd(int a) = ...`
- By “Fast and Loose Reasoning,” we can casually say “This function returns one of two values.”

So what does it mean?

- Consider `bool isOdd(int a) = ...`
- By “Fast and Loose Reasoning,” we can casually say “This function returns one of two values.”
- We can safely ignore implementations such as `bool isOdd(int a) = isOdd(a)`.

Fast and Loose Reasoning

- Many programming languages ship with things which let us escape the promises of the type system.
    - `null`

Fast and Loose Reasoning

- Many programming languages ship with things which let us escape the promises of the type system.
  - `null`
  - Exceptions

Fast and Loose Reasoning

- Many programming languages ship with things which let us escape the promises of the type system.
  - `null`
  - Exceptions
  - Type-casing

Fast and Loose Reasoning

- Many programming languages ship with things which let us escape the promises of the type system.
    - `null`
    - Exceptions
    - Type-casing
    - Type-casting

Fast and Loose Reasoning

- Many programming languages ship with things which let us escape the promises of the type system.
  - `null`
  - Exceptions
  - Type-casing
  - Type-casting
  - Side-effects

Fast and Loose Reasoning

- Many programming languages ship with things which let us escape the promises of the type system.
  - `null`
  - Exceptions
  - Type-casing
  - Type-casting
  - Side-effects
  - universal `equals`/`toString`/`hashCode`/etc.

Fast and Loose Reasoning

- Many programming languages ship with things which let us escape the promises of the type system.
    - `null`
    - Exceptions
    - Type-casing
    - Type-casting
    - Side-effects
    - universal `equals`/`toString`/`hashCode`/etc.
- We can discard these (and face **zero** penalty).

Fast and Loose Reasoning

- Many programming languages ship with things which let us escape the promises of the type system.
  - `null`
  - Exceptions
  - Type-casing
  - Type-casting
  - Side-effects
  - universal `equals`/`toString`/`hashCode`/etc.
- We **should** discard these (and face **zero** penalty).

Fast and Loose Reasoning

- Many programming languages ship with things which let us escape the promises of the type system.
  - `null`
  - Exceptions
  - Type-casing
  - Type-casting
  - Side-effects
  - universal `equals`/`toString`/`hashCode`/etc.
- We must discard these (and face **zero** penalty).

Tool #3: The lack of unit testing

Tool #3: The lack of unit testing

Yes, getting rid of unit testing is a useful tool.

Have I gotten your attention yet?

- The Problems with Unit Testing (Elrod, 2014)
  - Unit testing helps to convinces us of things that are likely untrue.

Have I gotten your attention yet?

- The Problems with Unit Testing (Elrod, 2014)
  - Unit testing helps to convinces us of things that are likely untrue.
  - Thus, they instill a false sense of confidence that our code works.

Have I gotten your attention yet?

- The Problems with Unit Testing (Elrod, 2014)
  - Unit testing helps to convinces us of things that are likely untrue.
  - Thus, they instill a false sense of confidence that our code works.
  - ...leading to bugs and surprises.

Property-based testing

- Consider again the function type: `<A> List<A> demo(List<A>)`
    - Recall: *Theorem*: The list returned by `demo` will only ever contain elements which appeared in the input.

Property-based testing

- Consider again the function type: `<A> List<A> demo(List<A>)`
  - Recall: *Theorem*: The list returned by `demo` will only ever contain elements which appeared in the input.
  - But how do we narrow down the ambiguity?

Property-based testing

- Consider again the function type: `<A> List<A> demo(List<A>)`
  - Recall: *Theorem*: The list returned by `demo` will only ever contain elements which appeared in the input.
  - But how do we narrow down the ambiguity?
  - We write unit tests to convince ourselves that our suspicion is right.

Property-based testing

- Consider again the function type: `<A> List<A> demo(List<A>)`
  - Recall: *Theorem*: The list returned by `demo` will only ever contain elements which appeared in the input.
  - But how do we narrow down the ambiguity?
  - ~~We write unit tests to convince ourselves that our suspicion is right.~~

Property-based testing

- Consider again the function type: `<A> List<A> demo(List<A>)`
  - Recall: *Theorem*: The list returned by `demo` will only ever contain elements which appeared in the input.
  - But how do we narrow down the ambiguity?
  - ~~We write unit tests to convince ourselves that our suspicion is right.~~
  - We write a comment above the code:

```
/* This function definitely reverses its
   input list! */
```

Property-based testing

- Consider again the function type: `<A> List<A> demo(List<A>)`
  - Recall: *Theorem*: The list returned by `demo` will only ever contain elements which appeared in the input.
  - But how do we narrow down the ambiguity?
  - ~~We write unit tests to convince ourselves that our suspicion is right.~~
  - ~~We write a comment above the code:~~

```
/* This function definitely reverses its
   input list! */
```

Property-based testing

- Consider again the function type: `<A> List<A> demo(List<A>)`
  - Recall: *Theorem*: The list returned by `demo` will only ever contain elements which appeared in the input.
  - But how do we narrow down the ambiguity?
  - ~~We write unit tests to convince ourselves that our suspicion is right.~~
  - ~~We write a comment above the code:~~

    ```
    /* This function definitely reverses its
       input list! */
    ```

  - We write true, testable statements *about* the code. Properties that we claim it exhibits.

Property-based testing

Program 1

```
// property> demo(List.empty) == List.empty
//
// property> x => demo(demo(x)) == x
//
// property> (x, y) => demo(x.append(y))
//       == demo(y).append(demo(x))

<A> List<A> demo(List<A> xs) {
  // ...
}
```

Property-based testing

- Once those properties are written, the computer can *generate random test cases* to ensure they are met.

Property-based testing

- Once those properties are written, the computer can *generate random test cases* to ensure they are met.
- The computer’s test cases are better than yours.

Property-based testing

- Once those properties are written, the computer can *generate random test cases* to ensure they are met.
- The computer's test cases are better than yours.
- If a test case fails, the computer can tell us which inputs it tried and failed with.

Property-based testing

- Once those properties are written, the computer can *generate random test cases* to ensure they are met.
- The computer’s test cases are better than yours.
- If a test case fails, the computer can tell us which inputs it tried and failed with.
- This method of testing has been popularized by Claessen and Hughes in their *QuickCheck* tool and corresponding paper.

Property-based testing

- Once those properties are written, the computer can *generate random test cases* to ensure they are met.
- The computer's test cases are better than yours.
- If a test case fails, the computer can tell us which inputs it tried and failed with.
- This method of testing has been popularized by Claessen and Hughes in their *QuickCheck* tool and corresponding paper.
- It subsumes unit testing.

Tool #4: Types As Documentation

Types As Documentation

What theorems do these functions give us for free?

- `<A> A blah(A x)`

Types As Documentation

What theorems do these functions give us for free?

- `<A> A blah(A x)`
- `<A, B> List<B> blah2(List<A> x, Func<A, B> f)`

Types As Documentation

What theorems do these functions give us for free?

- `<A> A blah(A x)`
- `<A, B> List<B> blah2(List<A> x, Func<A, B> f)`
- `<A, B> List<B> blah3(List<A> x, Func<A, List<B>» f)`

Types As Documentation

- Types, used properly, are documentation.

Types As Documentation

- Types, used properly, are documentation.
- **Reliable** documentation, that doesn’t go out of date.

Types As Documentation

- Types, used properly, are documentation.
- **Reliable** documentation, that doesn’t go out of date.
- **Dense** documentation.

Types As Documentation

- Types, used properly, are documentation.
- **Reliable** documentation, that doesn’t go out of date.
- **Dense** documentation.
- Like *comments* except condensed, machine-checked, and without the human-added falsehoods and lies.

Tool #5: Types As Theorems; Programs as Proofs
(Curry-Howard Correspondence)

Tool #6: Mathematical correspondences

(Curry-Howard-Lambek Correspondence; category theory)

Tool #7: Data types

Data Types

Example: The Option Type

- The Option (or “Optional” or “Maybe”) type is a list with at-most one element.

Data Types

Example: The Option Type

- The Option (or “Optional” or “Maybe”) type is a list with at-most one element.
- Every operation we can perform on lists (`map`, `flatMap`, etc.) can be performed on `Option`.

Data Types

Example: The Option Type

- The Option (or “Optional” or “Maybe”) type is a list with at-most one element.
- Every operation we can perform on lists (`map`, `flatMap`, etc.) can be performed on `Option`.
- Like `List<A>`, it is polymorphic over its element: `Option<A>`.

Data Types

Example: The Option Type

- The Option (or “Optional” or “Maybe”) type is a list with at-most one element.
- Every operation we can perform on lists (`map`, `flatMap`, etc.) can be performed on `Option`.
- Like `List<A>`, it is polymorphic over its element: `Option<A>`.
- Haskell code: `data Maybe a = Just a | Nothing`

Data Types

Example: The Option Type

- The Option (or “Optional” or “Maybe”) type is a list with at-most one element.
- Every operation we can perform on lists (`map`, `flatMap`, etc.) can be performed on `Option`.
- Like `List<A>`, it is polymorphic over its element: `Option<A>`.
- Haskell code: `data Maybe a = Just a | Nothing`
- Used for indicating no useful value has come back from a computation.

Data Types

Example: The Option Type

- The Option (or “Optional” or “Maybe”) type is a list with at-most one element.
- Every operation we can perform on lists (`map`, `flatMap`, etc.) can be performed on `Option`.
- Like `List<A>`, it is polymorphic over its element: `Option<A>`.
- Haskell code: `data Maybe a = Just a | Nothing`
- Used for indicating no useful value has come back from a computation.
- It’s basically `null`, except type-safe!

Data Types

Example: The Option Type

```
head :: List a -> Maybe a
head EmptyList = Nothing
head NonEmptyList x xs = Just x

-- Ever seen an ArrayOutOfBoundsException?
index :: Array a -> Int -> Maybe a
index arr n =
  if length arr >= (n - 1)
  then Just ...
  else Nothing

-- and so on.
```

Tool #7:
Commitment to all of the above.
(Because they are better than the dysfunctional programming you are doing now.)

Software Engineering Goals

- Fix bugs independently of creating new ones.

Software Engineering Goals

- Fix bugs independently of creating new ones.
- Introduce features without breaking old ones.

Software Engineering Goals

- Fix bugs independently of creating new ones.
- Introduce features without breaking old ones.
- Be able to have many projects with little-to-no maintenance.

Software Engineering Goals

- Fix bugs independently of creating new ones.
- Introduce features without breaking old ones.
- Be able to have many projects with little-to-no maintenance.
- Reliably, efficiently, correctly determine what problem existing code solves.

Commonly heard quotes, distracting from goals

- “How’s the Haskell Programmer ivory tower?”

Commonly heard quotes, distracting from goals

- “How’s the Haskell Programmer ivory tower?”
- “Why do you hate <technology/language>?”

Commonly heard quotes, distracting from goals

- “How’s the Haskell Programmer ivory tower?”
- “Why do you hate <technology/language>?”
- “All tools have a purpose!”

Commonly heard quotes, distracting from goals

- “How’s the Haskell Programmer ivory tower?”
- “Why do you hate <technology/language>?”
- “All tools have a purpose!”
- “The learning curve is too high!”

Commonly heard quotes, distracting from goals

- “How’s the Haskell Programmer ivory tower?”
- “Why do you hate <technology/language>?”
- “All tools have a purpose!”
- “The learning curve is too high!”
- “Why are you so extremist?”

Contact

- ricky@elrod.me
- github: @relrod
- twitter: @relrod6
- freenode IRC: relrod (see also: #haskell, #scalaz)