Introduction to Functional Programming

Allele Dev (@queertypes)

October 27, 2014

Contact Me!

- IRC: alcabrera @ freenode.net
- Github: cabrera
- Twitter: @queertypes
- Blog: `https://blog.cppcabrera.com/`

Overview

- What is functional programming?
- Why it matters
- Expressive types: modeling **exactly** what you mean
- Composition of a higher-order
- Making FP personal: how to make it yours

What is it?

- Expressions, not statements
- Immutability by default
- Functions are first class
- A focus on referential transparency
- Guided by the shape (types) of our data

Why?

- To develop a common vocabulary for abstractions and operations
- To build systems we can reason about (in the large, especially)
- To help against the struggles with concurrency/parallelism
- To bridge a clear gap between theory and practice (math!)
- To have fun - FUNctional programming is FUN!

A Sampling Tray of Languages

- Agda
- Clojure
- Coq
- Elm
- Erlang
- F#
- F*
- Haskell
- Idris
- Ocaml
- Purescript
- Racket
- Rust
- Scala
- Standard ML
- Swift

Today's Language

- Haskell will be my medium today

Expressions

- One of three things:

Expressions

- One of three things:
  - Variable/Value: An atom or a name: `1`, `x`, `"cat"`

Expressions

- One of three things:
  - Variable/Value: An atom or a name: `1`, `x`, `"cat"`
  - Abstraction: binding an expression to a name

Expressions

- One of three things:
  - Variable/Value: An atom or a name: `1`, `x`, `"cat"`
  - Abstraction: binding an expression to a name
    - `let x = 1`

Expressions

- One of three things:
  - Variable/Value: An atom or a name: `1`, `x`, `"cat"`
  - Abstraction: binding an expression to a name
    - `let x = 1`
  - Application: calling a function: `f 10`

Expressions

- One of three things:
  - Variable/Value: An atom or a name: `1`, `x`, `"cat"`
  - Abstraction: binding an expression to a name
    - `let x = 1`
  - Application: calling a function: `f 10`
- Maps directly to the lambda calculus!

Expressions

```
> 1 -- value
1
> let x = 1 -- abstraction
> x
1
> let f y = y + 1
> f 10  -- application
11
```

In Contrast to: Statements

```
>>> 1
1
>>> x = 1
>>> x
1
>>> def f(x): return x + 1
>>> f(1)
2
```

Immutability

- Once defined, values cannot be changed

Immutability

- Once defined, values cannot be changed
- The introduction of mutable state breaks reasoning

Immutability

- Once defined, values cannot be changed
- The introduction of mutable state breaks reasoning
- Solution: decouple definition from assignment

Immutability

- Once defined, values cannot be changed
- The introduction of mutable state breaks reasoning
- Solution: decouple definition from assignment
  - . . . some languages even disallow assignment!

Immutability

```
> x = 10
> x = 11
error: Multiple declarations of 'x'
```

Functions

- A function is a map from input to output
  - **Idempotent**: Given the same input, yield the same output
  - **Pure**: no side-effects; make effects explicit
  - **Effect**: an operation that changes the execution environment
    - Examples: mutating state, read/write from/to disk, logging, printing
- Functions are also values, and can be manipulated as such
- **Note**: Purity is optional
  - It is sufficient to have functions as first-class citizens to program functionally

Note on Purity

- Some languages track effects:
  - Haskell
  - Elm
  - Purescript
  - Agda
  - Coq
  - Idris
- Some languages do not:
  - Rust
  - Scala
  - Ocaml
  - Standard ML
  - F#
  - Swift
  - Erlang
  - Clojure
  - Racket

Referential Transparency

- The ability to reason by substitution
  - Replace an expression by its value at any time
  - Without changing meaning of program!
  - Possible because of idempotency + purity

Referential Transparency: How to Void Warranties

- null references

Referential Transparency: How to Void Warranties

- null references
- exceptions

Referential Transparency: How to Void Warranties

- null references
- exceptions
- type casting

Referential Transparency: How to Void Warranties

- null references
- exceptions
- type casting
- side effects

Referential Transparency: How to Void Warranties

- null references
- exceptions
- type casting
- side effects
- partiality (not handling all cases)

Referential Transparency: How to Void Warranties

- null references
- exceptions
- type casting
- side effects
- partiality (not handling all cases)
- general recursion (non-termination)

Referential Transparency by Example

- Let's walk through an example involving a map function

```
map' f xs = case xs of
    []     -> []
    (x:xs') -> f x : map' f xs'
```

Referential Transparency by Example

```
map' f (+1) [1,2,3]
```

Referential Transparency by Example

```
map' f (+1) [1,2,3]
(1+1) : map' (+1) [2,3]
```

Referential Transparency by Example

```
map' f (+1) [1,2,3]
(1+1) : map' (+1) [2,3]
(1+1) : (2+1) : map' (+1) [3]
```

Referential Transparency by Example

```
map' f (+1) [1,2,3]
(1+1) : map' (+1) [2,3]
(1+1) : (2+1) : map' (+1) [3]
(1+1) : (2+1) : (3+1) : map' (+1) []
```

Referential Transparency by Example

```
map' f (+1) [1,2,3]
(1+1) : map' (+1) [2,3]
(1+1) : (2+1) : map' (+1) [3]
(1+1) : (2+1) : (3+1) : map' (+1) []
(1+1) : (2+1) : (3+1) : []
```

Referential Transparency by Example

```
map' f (+1) [1,2,3]
(1+1) : map' (+1) [2,3]
(1+1) : (2+1) : map' (+1) [3]
(1+1) : (2+1) : (3+1) : map' (+1) []
(1+1) : (2+1) : (3+1) : []
(1+1) : (2+1) : 4 : []
```

Referential Transparency by Example

```
map' f (+1) [1,2,3]
(1+1) : map' (+1) [2,3]
(1+1) : (2+1) : map' (+1) [3]
(1+1) : (2+1) : (3+1) : map' (+1) []
(1+1) : (2+1) : (3+1) : []
(1+1) : (2+1) : 4 : []
(1+1) : (2+1) : [4]
```

Referential Transparency by Example

```
map' f (+1) [1,2,3]
(1+1) : map' (+1) [2,3]
(1+1) : (2+1) : map' (+1) [3]
(1+1) : (2+1) : (3+1) : map' (+1) []
(1+1) : (2+1) : (3+1) : []
(1+1) : (2+1) : 4 : []
(1+1) : (2+1) : [4]
(1+1) : 3 : [4]
```

Referential Transparency by Example

```
map' f (+1) [1,2,3]
(1+1) : map' (+1) [2,3]
(1+1) : (2+1) : map' (+1) [3]
(1+1) : (2+1) : (3+1) : map' (+1) []
(1+1) : (2+1) : (3+1) : []
(1+1) : (2+1) : 4 : []
(1+1) : (2+1) : [4]
(1+1) : 3 : [4]
(1+1) : [3,4]
```

Referential Transparency by Example

```
map' f (+1) [1,2,3]
(1+1) : map' (+1) [2,3]
(1+1) : (2+1) : map' (+1) [3]
(1+1) : (2+1) : (3+1) : map' (+1) []
(1+1) : (2+1) : (3+1) : []
(1+1) : (2+1) : 4 : []
(1+1) : (2+1) : [4]
(1+1) : 3 : [4]
(1+1) : [3,4]
2 : [3,4]
```

Referential Transparency by Example

```
map' f (+1) [1,2,3]
(1+1) : map' (+1) [2,3]
(1+1) : (2+1) : map' (+1) [3]
(1+1) : (2+1) : (3+1) : map' (+1) []
(1+1) : (2+1) : (3+1) : []
(1+1) : (2+1) : 4 : []
(1+1) : (2+1) : [4]
(1+1) : 3 : [4]
(1+1) : [3,4]
2 : [3,4]
[2,3,4]
```

Types

- The shapes of our data
- Checked before run-time/at compile-time
- In advanced languages, automatically inferred
- Rules as to what pieces can validly fit together
  - e.g., $1 +$ "1" does not make sense
- Can build proofs of correctness

Reading Types (in Haskell)

```haskell
x :: Int  -- x is an integer
y :: (Int, Int) -- y is a pair of integers
z :: [Int] -- z is a list of integers

-- f is a function taking an Int, returning an Int
f :: Int -> Int

-- id is a function taking any type "a"
-- and returning that "a"
-- only one possible implementation for id
id :: a -> a

-- only one possible implementation for swap
swap :: (a,b) -> (b,a)
```

Type Errors

```
-- type mismatch
> 1 + "1"
    No instance for (Num [Char]) arising from a use of ‘+’
    In the expression: 1 + "1"
    In an equation for ‘it’: it = 1 + "1"
```

Type Errors

```
-- too many arguments
> let f x = x + 1
> f 1 2
Couldn't match expected type ‘a0 -> s0’
            with actual type ‘Int’
The function ‘f’ is applied to two arguments,
but its type ‘Int -> Int’ has only one
```

More Types

```haskell
-- sum, or "one-of" types
data Weekday = M | T | W | R | F

-- pattern matching, compile-time checked!
nextDay :: Weekday -> Weekday
nextDay d = case d of
    M -> T
    T -> W
    W -> R
    R -> F
    F -> M
```

More Type Errors

```
-- pattern matching: didn't cover every case
nextDay' :: Weekday -> Weekday
nextDay' d = case d of
    M -> T
    T -> W
    W -> R
    R -> F

Warning: Pattern match(es) are non-exhaustive
In a case alternative: Patterns not matched: F
```

More Types

```haskell
-- product, or "each of" types: multiply possibilities!
data Switch = Switch Bool Bool

toggle :: Switch -> Switch
toggle (Switch True True) = Switch False False
toggle (Switch False True) = Switch True False
toggle (Switch True False) = Switch False True
toggle (Switch False False) = Switch True True
```

Type Errors

```
toggle' :: Switch -> Switch
toggle' (Switch True True) = Switch False False
toggle' (Switch False True) = Switch True False
toggle' (Switch True False) = Switch False True

Warning: Pattern match(es) are non-exhaustive
In an equation for ‘toggle’:
    Patterns not matched: Switch False False
```

More Types

```haskell
-- strong type alias: no longer mix up Ints and Ages!
newtype Age = Age Int

getOlder :: Age -> Age
getOlder (Age x) = Age (x + 1)
```

Type Error

```
-- can't mix Ages and Ints
let x = Age 1
let y = x + 1

No instance for (Num Age)
  arising from a use of ‘+’
In the expression: x + 1
In an equation for ‘y’: y = x + 1
```

Much More on Types

- We've barely scratched the surface of typed functional programming

Much More on Types

- We've barely scratched the surface of typed functional programming
- There's **much**, **much** more to play with

Much More on Types

- We've barely scratched the surface of typed functional programming
- There's **much**, **much** more to play with
- We'll not cover it today - let's carry on!

Review: What is Functional Programming?

- Expressions
- Immutability
- Functions
- Referential transparency
- Types

Going Higher Order

- Let's add a new tool: higher-order functions

Going Higher Order

- Let's add a new tool: higher-order functions
- Functions that take functions as arguments

Going Higher Order

- Let's add a new tool: higher-order functions
- Functions that take functions as arguments
- Key to abstracting away larger patterns

Going Higher Order

- Let's add a new tool: higher-order functions
- Functions that take functions as arguments
- Key to abstracting away larger patterns
    - Notably, for this talk, many forms of iteration

Going Higher Order: Map

- Map: apply a function to a collection of elements

```
map :: (a -> b) -> [a] -> [b]
map f xs = case xs of
    []     -> []
    (x:xs') -> f x : map

map (+1) [1,2,3]
=> [2,3,4]
```

Going Higher Order: Filter

- Filter: extract elements of interest from a collection

```
filter :: (a -> Bool) -> [a] -> [a]
filter p xs = case xs of
    []     -> []
    (x:xs') -> if p x
               then x : filter p xs'
               else filter p xs'

filter (<3) [1,2,3,4,5]
=> [1,2]
```

Going Higher Order: Fold

- Fold: summarize a collection by merging elements down to a value

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z xs = case xs of
    []     -> z
    (x:xs') -> f x (foldr f z xs')

foldr (+) 0 [1,2,3,4,5]
=> 15
```

Syntactic Note

```
(f x y) z
```

. . . is equivalent to:

```
f x y $ z
```

Fold Illustrated

```
foldr (+) 0 [1,2,3]
```

Fold Illustrated

```
foldr (+) 0 [1,2,3]
(+) 1 $ foldr (+) 0 [2,3]
```

Fold Illustrated

```
foldr (+) 0 [1,2,3]
(+) 1 $ foldr (+) 0 [2,3]
(+) 1 $ (+) 2 $ foldr (+) 0 [3]
```

Fold Illustrated

```
foldr (+) 0 [1,2,3]
(+) 1 $ foldr (+) 0 [2,3]
(+) 1 $ (+) 2 $ foldr (+) 0 [3]
(+) 1 $ (+) 2 $ (+) 3 $ foldr (+) 0 []
```

Fold Illustrated

```
foldr (+) 0 [1,2,3]
(+) 1 $ foldr (+) 0 [2,3]
(+) 1 $ (+) 2 $ foldr (+) 0 [3]
(+) 1 $ (+) 2 $ (+) 3 $ foldr (+) 0 []
(+) 1 $ (+) 2 $ (+) 3 0
```

Fold Illustrated

```
foldr (+) 0 [1,2,3]
(+) 1 $ foldr (+) 0 [2,3]
(+) 1 $ (+) 2 $ foldr (+) 0 [3]
(+) 1 $ (+) 2 $ (+) 3 $ foldr (+) 0 []
(+) 1 $ (+) 2 $ (+) 3 0
(+) 1 $ (+) 2 3
```

Fold Illustrated

```
foldr (+) 0 [1,2,3]
(+) 1 $ foldr (+) 0 [2,3]
(+) 1 $ (+) 2 $ foldr (+) 0 [3]
(+) 1 $ (+) 2 $ (+) 3 $ foldr (+) 0 []
(+) 1 $ (+) 2 $ (+) 3 0
(+) 1 $ (+) 2 3
(+) 1 5
```

Fold Illustrated

```
foldr (+) 0 [1,2,3]
(+) 1 $ foldr (+) 0 [2,3]
(+) 1 $ (+) 2 $ foldr (+) 0 [3]
(+) 1 $ (+) 2 $ (+) 3 $ foldr (+) 0 []
(+) 1 $ (+) 2 $ (+) 3 0
(+) 1 $ (+) 2 3
(+) 1 5
6
```

Going Higher Order: Fold in Action

```
sum xs = foldl (+) 0 xs
product xs = foldl (*) 1 xs
any xs = foldl (||) False xs
and xs = foldl (&&) True xs
```

Going Higher Order: Composition

- Compose: joining together functions to form a pipeline

```
compose :: (a -> b) -> (b -> c) -> a -> c
compose f g = g . f

filter (odd) . map (+1) $ [1,2,3,4]
=> [3, 5]
```

Notes on Higher Order

- Manual looping and primitive recursion are usually anti-patterns

Notes on Higher Order

- Manual looping and primitive recursion are usually anti-patterns
  - Fail to express the essence of the collective operation

Notes on Higher Order

- Manual looping and primitive recursion are usually anti-patterns
  - Fail to express the essence of the collective operation
  - Repeat low-level details

Notes on Higher Order

- Manual looping and primitive recursion are usually anti-patterns
  - Fail to express the essence of the collective operation
  - Repeat low-level details
- Other higher-order patterns exist: unfolds, lenses, bananas

Notes on Higher Order

- Manual looping and primitive recursion are usually anti-patterns
  - Fail to express the essence of the collective operation
  - Repeat low-level details
- Other higher-order patterns exist: unfolds, lenses, bananas
- There's a branch of math that studies laws of composition!

Notes on Higher Order

- Manual looping and primitive recursion are usually anti-patterns
  - Fail to express the essence of the collective operation
  - Repeat low-level details
- Other higher-order patterns exist: unfolds, lenses, bananas
- There's a branch of math that studies laws of composition!
- More to explore, more ways to apply than expressed here

Notes on Higher Order

- Manual looping and primitive recursion are usually anti-patterns
  - Fail to express the essence of the collective operation
  - Repeat low-level details
- Other higher-order patterns exist: unfolds, lenses, bananas
- There's a branch of math that studies laws of composition!
- More to explore, more ways to apply than expressed here
  - For example: map and fold can be generalized

Challenges of Functional Programming

- Many classical algorithms only specified imperatively

Challenges of Functional Programming

- Many classical algorithms only specified imperatively
- Many classical data structures only specified imperatively

Challenges of Functional Programming

- Many classical algorithms only specified imperatively
- Many classical data structures only specified imperatively
- Few side-effect-free functional libraries exist

Challenges of Functional Programming

- Many classical algorithms only specified imperatively
- Many classical data structures only specified imperatively
- Few side-effect-free functional libraries exist
    - More common with Haskell, Purescript, and Elm

Challenges of Functional Programming

- Many classical algorithms only specified imperatively
- Many classical data structures only specified imperatively
- Few side-effect-free functional libraries exist
    - More common with Haskell, Purescript, and Elm
- Culture

Challenges of Functional Programming

- Many classical algorithms only specified imperatively
- Many classical data structures only specified imperatively
- Few side-effect-free functional libraries exist
  - More common with Haskell, Purescript, and Elm
- Culture
  - Familiarity with functional programming uncommon

Challenges of Functional Programming

- Many classical algorithms only specified imperatively
- Many classical data structures only specified imperatively
- Few side-effect-free functional libraries exist
  - More common with Haskell, Purescript, and Elm
- Culture
  - Familiarity with functional programming uncommon
- JVM problems

Challenges of Functional Programming

- Many classical algorithms only specified imperatively
- Many classical data structures only specified imperatively
- Few side-effect-free functional libraries exist
  - More common with Haskell, Purescript, and Elm
- Culture
  - Familiarity with functional programming uncommon
- JVM problems
  - Lack of support for full tail call optimization requires workarounds

Make Functional Programming Yours

- Take what I've shared with you today and make it your own

Make Functional Programming Yours

- Take what I've shared with you today and make it your own
- What are your interests?

Make Functional Programming Yours

- Take what I've shared with you today and make it your own
- What are your interests?
    - Great UIs? Purescript, Elm, Clojurescript, FRP

Make Functional Programming Yours

- Take what I've shared with you today and make it your own
- What are your interests?
  - Great UIs? Purescript, Elm, Clojurescript, FRP
  - Theory? PLT, Type Systems, Type Theory

Make Functional Programming Yours

- Take what I've shared with you today and make it your own
- What are your interests?
  - Great UIs? Purescript, Elm, Clojurescript, FRP
  - Theory? PLT, Type Systems, Type Theory
  - Your own Languages? Pi Tutorial

Make Functional Programming Yours

- Take what I've shared with you today and make it your own
- What are your interests?
  - Great UIs? Purescript, Elm, Clojurescript, FRP
  - Theory? PLT, Type Systems, Type Theory
  - Your own Languages? Pi Tutorial
  - Ease maintenance? Proofs with types, property tests

Make Functional Programming Yours

- Take what I've shared with you today and make it your own
- What are your interests?
  - Great UIs? Purescript, Elm, Clojurescript, FRP
  - Theory? PLT, Type Systems, Type Theory
  - Your own Languages? Pi Tutorial
  - Ease maintenance? Proofs with types, property tests
  - DRY, Succinct programs? Recursion schemes, Category theory

Make Functional Programming Yours

- Take what I've shared with you today and make it your own
- What are your interests?
  - Great UIs? Purescript, Elm, Clojurescript, FRP
  - Theory? PLT, Type Systems, Type Theory
  - Your own Languages? Pi Tutorial
  - Ease maintenance? Proofs with types, property tests
  - DRY, Succinct programs? Recursion schemes, Category theory
  - Fast programs? Morte, Compiler optimizations for FP

Make Functional Programming Yours

- Take what I've shared with you today and make it your own
- What are your interests?
  - Great UIs? Purescript, Elm, Clojurescript, FRP
  - Theory? PLT, Type Systems, Type Theory
  - Your own Languages? Pi Tutorial
  - Ease maintenance? Proofs with types, property tests
  - DRY, Succinct programs? Recursion schemes, Category theory
  - Fast programs? Morte, Compiler optimizations for FP
  - Cryptography? Cryptol

Make Functional Programming Yours

- Take what I've shared with you today and make it your own
- What are your interests?
  - Great UIs? Purescript, Elm, Clojurescript, FRP
  - Theory? PLT, Type Systems, Type Theory
  - Your own Languages? Pi Tutorial
  - Ease maintenance? Proofs with types, property tests
  - DRY, Succinct programs? Recursion schemes, Category theory
  - Fast programs? Morte, Compiler optimizations for FP
  - Cryptography? Cryptol
  - Algorithms? Pearls, Data Structures 1 2

Make Functional Programming Yours

- Take what I've shared with you today and make it your own
- What are your interests?
  - Great UIs? Purescript, Elm, Clojurescript, FRP
  - Theory? PLT, Type Systems, Type Theory
  - Your own Languages? Pi Tutorial
  - Ease maintenance? Proofs with types, property tests
  - DRY, Succinct programs? Recursion schemes, Category theory
  - Fast programs? Morte, Compiler optimizations for FP
  - Cryptography? Cryptol
  - Algorithms? Pearls, Data Structures 1 2
  - Concurrency? Book, Erlang

Make Functional Programming Yours

- Take what I've shared with you today and make it your own
- What are your interests?
  - Great UIs? Purescript, Elm, Clojurescript, FRP
  - Theory? PLT, Type Systems, Type Theory
  - Your own Languages? Pi Tutorial
  - Ease maintenance? Proofs with types, property tests
  - DRY, Succinct programs? Recursion schemes, Category theory
  - Fast programs? Morte, Compiler optimizations for FP
  - Cryptography? Cryptol
  - Algorithms? Pearls, Data Structures 1 2
  - Concurrency? Book, Erlang
  - Systems? Server as Function, Mirage, sel4

Make Functional Programming Yours

- Take what I've shared with you today and make it your own
- What are your interests?
  - Great UIs? Purescript, Elm, Clojurescript, FRP
  - Theory? PLT, Type Systems, Type Theory
  - Your own Languages? Pi Tutorial
  - Ease maintenance? Proofs with types, property tests
  - DRY, Succinct programs? Recursion schemes, Category theory
  - Fast programs? Morte, Compiler optimizations for FP
  - Cryptography? Cryptol
  - Algorithms? Pearls, Data Structures 1 2
  - Concurrency? Book, Erlang
  - Systems? Server as Function, Mirage, sel4
  - Embedded? Ivory

Make Functional Programming Yours

- Take what I've shared with you today and make it your own
- What are your interests?
  - Great UIs? Purescript, Elm, Clojurescript, FRP
  - Theory? PLT, Type Systems, Type Theory
  - Your own Languages? Pi Tutorial
  - Ease maintenance? Proofs with types, property tests
  - DRY, Succinct programs? Recursion schemes, Category theory
  - Fast programs? Morte, Compiler optimizations for FP
  - Cryptography? Cryptol
  - Algorithms? Pearls, Data Structures 1 2
  - Concurrency? Book, Erlang
  - Systems? Server as Function, Mirage, sel4
  - Embedded? Ivory
  - Music? School of Music, Music suite

Make Functional Programming Yours

- Take what I've shared with you today and make it your own
- What are your interests?
  - Great UIs? Purescript, Elm, Clojurescript, FRP
  - Theory? PLT, Type Systems, Type Theory
  - Your own Languages? Pi Tutorial
  - Ease maintenance? Proofs with types, property tests
  - DRY, Succinct programs? Recursion schemes, Category theory
  - Fast programs? Morte, Compiler optimizations for FP
  - Cryptography? Cryptol
  - Algorithms? Pearls, Data Structures 1 2
  - Concurrency? Book, Erlang
  - Systems? Server as Function, Mirage, sel4
  - Embedded? Ivory
  - Music? School of Music, Music suite
  - Community? Meetups, Strangeloop, Online Examples 1 2, 3, Summer School

Make Functional Programming Yours

- Take what I've shared with you today and make it your own
- What are your interests?
  - Great UIs? Purescript, Elm, Clojurescript, FRP
  - Theory? PLT, Type Systems, Type Theory
  - Your own Languages? Pi Tutorial
  - Ease maintenance? Proofs with types, property tests
  - DRY, Succinct programs? Recursion schemes, Category theory
  - Fast programs? Morte, Compiler optimizations for FP
  - Cryptography? Cryptol
  - Algorithms? Pearls, Data Structures 1 2
  - Concurrency? Book, Erlang
  - Systems? Server as Function, Mirage, sel4
  - Embedded? Ivory
  - Music? School of Music, Music suite
  - Community? Meetups, Strangeloop, Online Examples 1 2, 3, Summer School
  - Jobs? 1, 2

Make it Fun(ctional)

- FP let's you abstract away from the low-level details
  - Focus on the problem at the right level
  - Model **exactly** what you need
- Less time debugging, more time crafting
- Types aid with communicating ideas with colleagues and friends

Resources (Books/Papers)

- Haskell: RWH, Pearls, ParConc, Web
- Scala: FP
- Ocaml: RWO, Ocaml Beginners
- Purescript: Intro
- Coq: SF, CPDT
- Agda: Thesis
- Idris: Tut
- F#: RWFP
- Clojure: Joy
- Erlang: ProgErl

Epilogue: Much More on Types

- Recursive types: represents Trees and the like at the type-level
- (Parametric) Polymorphism: c++/java generics, but generalized
- Higher-kinds: the shape of types
- Higher-sorts: the shape of kinds
- Effect tracking
- Safe coercions
- Linear types
- Gradual type systems
- Gradual effect systems
- Refinement types
- Dependent types
- Subtyping (inheritance, objects, and trade-offs in a type system)
- Typeful Techniques: phantoms, GADTs, nominal vs. structural, ...
- Iterative correct-by-construction development

Thank You!