Why Types Matter

Slavomir Kaslev
slavomir.kaslev@gmail.com

March 11, 2017

QOTD

"Bad programmers worry about the code. Good programmers worry about data structures and their relationships." Linus Torvalds

Introduction

- Types were first proposed by Bertrand Russell in 1902 to resolve Russell's paradox in Gottlob Frege's naive set theory

Introduction

- Types were first proposed by Bertrand Russell in 1902 to resolve Russell's paradox in Gottlob Frege's naive set theory
- Since then types have been studied on their own and combined with Alonzo Church's λ-calculus provide the theoretical framework of modern functional languages

Introduction

- Types were first proposed by Bertrand Russell in 1902 to resolve Russell's paradox in Gottlob Frege's naive set theory
- Since then types have been studied on their own and combined with Alonzo Church's λ-calculus provide the theoretical framework of modern functional languages
- There are quite a few different flavors of type theories around

Introduction

- Types were first proposed by Bertrand Russell in 1902 to resolve Russell's paradox in Gottlob Frege's naive set theory
- Since then types have been studied on their own and combined with Alonzo Church's λ-calculus provide the theoretical framework of modern functional languages
- There are quite a few different flavors of type theories around
  - Simply typed lambda calculus (1940s), System F (1970s)
    - Haskell, OCaml, ML, ...

Introduction

- Types were first proposed by Bertrand Russell in 1902 to resolve Russell's paradox in Gottlob Frege's naive set theory
- Since then types have been studied on their own and combined with Alonzo Church's λ-calculus provide the theoretical framework of modern functional languages
- There are quite a few different flavors of type theories around
  - Simply typed lambda calculus (1940s), System F (1970s)
    - Haskell, OCaml, ML, ...
  - Martin-Löf Dependent Type Theory (1970s)
    - Agda, Coq, Idris, Lean, ..

Introduction

- Types were first proposed by Bertrand Russell in 1902 to resolve Russell's paradox in Gottlob Frege's naive set theory
- Since then types have been studied on their own and combined with Alonzo Church's λ-calculus provide the theoretical framework of modern functional languages
- There are quite a few different flavors of type theories around
  - Simply typed lambda calculus (1940s), System F (1970s)
    - Haskell, OCaml, ML, ...
  - Martin-Löf Dependent Type Theory (1970s)
    - Agda, Coq, Idris, Lean, ..
  - Homotopy Type Theory (2000s)

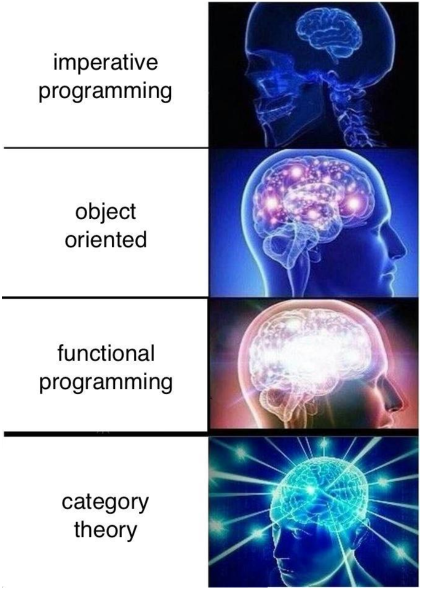
Introduction

- Types were first proposed by Bertrand Russell in 1902 to resolve Russell's paradox in Gottlob Frege's naive set theory
- Since then types have been studied on their own and combined with Alonzo Church's λ-calculus provide the theoretical framework of modern functional languages
- There are quite a few different flavors of type theories around
  - Simply typed lambda calculus (1940s), System F (1970s)
    - Haskell, OCaml, ML, ...
  - Martin-Löf Dependent Type Theory (1970s)
    - Agda, Coq, Idris, Lean, ..
  - Homotopy Type Theory (2000s)
  - ...

Introduction

- Types were first proposed by Bertrand Russell in 1902 to resolve Russell's paradox in Gottlob Frege's naive set theory
- Since then types have been studied on their own and combined with Alonzo Church's λ-calculus provide the theoretical framework of modern functional languages
- There are quite a few different flavors of type theories around
  - Simply typed lambda calculus (1940s), System F (1970s)
    - Haskell, OCaml, ML, ...
  - Martin-Löf Dependent Type Theory (1970s)
    - Agda, Coq, Idris, Lean, ..
  - Homotopy Type Theory (2000s)
  - ...
- Today, while the OOP world is catching up with functional programming, Haskell is catching up with dependent types

The Levels


imperative programming
object oriented
functional programming
category theory

What are Types

- Roughly speaking, types are specification of their possible values

What are Types

- Roughly speaking, types are specification of their possible values
- In typed programming languages each value a has a type A and we'll denote this as $a : A$

What are Types

- Roughly speaking, types are specification of their possible values
- In typed programming languages each value a has a type A and we'll denote this as $a : A$
- Instead of trying to define precisely what types are let's see what we can do with them

What are Types

- Roughly speaking, types are specification of their possible values
- In typed programming languages each value a has a type A and we'll denote this as $a : A$
- Instead of trying to define precisely what types are let's see what we can do with them
- For one, given the types $A : Type$ and $B : Type$ one can construct say $A \to B$, the type of functions from A to B

What are Types

- Roughly speaking, types are specification of their possible values
- In typed programming languages each value a has a type A and we'll denote this as $a : A$
- Instead of trying to define precisely what types are let's see what we can do with them
- For one, given the types $A : \mathit{Type}$ and $B : \mathit{Type}$ one can construct say $A \to B$, the type of functions from A to B
- $(\lambda(x : \mathit{Int}) \mapsto x + 1) : \mathit{Int} \to \mathit{Int}$

Algebraic Data Types

Algebraic Data Types

- The idea of algebraic data types is to introduce the operations $\oplus$ and $\otimes$ that given the types $A : Type$ and $B : Type$ construct new types namely:

Algebraic Data Types

- The idea of algebraic data types is to introduce the operations $\oplus$ and $\otimes$ that given the types $A : Type$ and $B : Type$ construct new types namely:
  - $A \oplus B$, disjoint union of A and B

Algebraic Data Types

- The idea of algebraic data types is to introduce the operations $\oplus$ and $\otimes$ that given the types $A : Type$ and $B : Type$ construct new types namely:
  - $A \oplus B$, disjoint union of A and B
  - $A \otimes B$, product of A and B

$A \otimes B$: Type product in Haskell

```haskell
data Pair = MkPair Int String
-- MkPair :: Int -> String -> Pair
```

$A \otimes B$: Type product in Haskell

```haskell
data Pair = MkPair Int String
-- MkPair :: Int -> String -> Pair

data R3 = MkR3 Float Float Float
-- MkR3 :: Float -> Float -> Float -> R3
```

$A \otimes B$: Type product in Haskell

```haskell
data Pair = MkPair Int String
-- MkPair :: Int -> String -> Pair

data R3 = MkR3 Float Float Float
-- MkR3 :: Float -> Float -> Float -> R3

data One = MkOne
-- MkOne :: One
```

$A \otimes B$: Type product in Haskell

```haskell
data Pair = MkPair Int String
-- MkPair :: Int -> String -> Pair

data R3 = MkR3 Float Float Float
-- MkR3 :: Float -> Float -> Float -> R3

data One = MkOne
-- MkOne :: One

data B x = B x x
-- B :: x -> x -> B x
```

$A \oplus B$: Disjoint union in Haskell

```haskell
data Bool = True | False
-- True, False :: Bool
```

$A \oplus B$: Disjoint union in Haskell

```
data Bool = True | False
-- True, False :: Bool

data Maybe x = Nothing | Some x
-- Nothing :: Maybe x
-- Some :: x -> Maybe x
```

$A \oplus B$: Disjoint union in Haskell

```haskell
data Bool = True | False
-- True, False :: Bool

data Maybe x = Nothing | Some x
-- Nothing :: Maybe x
-- Some :: x -> Maybe x

data List x = Nil | Cons x (List x)
-- Nil :: List x
-- Cons :: x -> List x -> List x
```

$A \oplus B$: Disjoint union in Haskell

```
data Bool = True | False
-- True, False :: Bool

data Maybe x = Nothing | Some x
-- Nothing :: Maybe x
-- Some :: x -> Maybe x

data List x = Nil | Cons x (List x)
-- Nil :: List x
-- Cons :: x -> List x -> List x

data BinTree x = Leaf x | Branch (BinTree x) (BinTree x)
-- Leaf :: x -> BinTree x
-- Branch :: BinTree x -> BinTree x -> BinTree x
```

Analytic Combinatorics

- Analytic combinatorics deals with counting combinatorial objects by means of their generating functions

Analytic Combinatorics

- Analytic combinatorics deals with counting combinatorial objects by means of their generating functions
  - See the book "Analytic Combinatorics" by Philippe Flajolet and Robert Sedgewick for an in-depth introduction

Analytic Combinatorics

- Analytic combinatorics deals with counting combinatorial objects by means of their generating functions
  - See the book "Analytic Combinatorics" by Philippe Flajolet and Robert Sedgewick for an in-depth introduction
- What is a generating function?

Analytic Combinatorics

- Analytic combinatorics deals with counting combinatorial objects by means of their generating functions
  - See the book "Analytic Combinatorics" by Philippe Flajolet and Robert Sedgewick for an in-depth introduction
- What is a generating function?
- Given a combinatorial class A and a size function $w : A \to \mathbb{N}$ we define A's ordinary generating function (OGF) as

$$A(x) = \sum_{a:A} x^{w(a)} = \sum_{n=0}^{\infty} a_n x^n$$

Analytic Combinatorics

- Analytic combinatorics deals with counting combinatorial objects by means of their generating functions
  - See the book "Analytic Combinatorics" by Philippe Flajolet and Robert Sedgewick for an in-depth introduction
- What is a generating function?
- Given a combinatorial class A and a size function $w : A \to \mathbb{N}$ we define A's ordinary generating function (OGF) as

$$A(x) = \sum_{a:A} x^{w(a)} = \sum_{n=0}^{\infty} a_n x^n$$

- The numbers a_n tell us how many objects in A are of size n

Symbolic Method: Finding generating functions

- Flajolet and Sedgewick propose a simple method of finding equation for the OGF of a given combinatorial construction expressed in their specification language

Symbolic Method: Finding generating functions

- Flajolet and Sedgewick propose a simple method of finding equation for the OGF of a given combinatorial construction expressed in their specification language
- In the special case of algebraic data types, the symbolic method uses the fact that if A, B, C are types and $A(x), B(x), C(x)$ are the corresponding OGFs then

$$C = A \oplus B \implies C(x) = A(x) + B(x)$$

and

$$C = A \otimes B \implies C(x) = A(x)B(x)$$

Symbolic Method: Finding generating functions

- Flajolet and Sedgewick propose a simple method of finding equation for the OGF of a given combinatorial construction expressed in their specification language
- In the special case of algebraic data types, the symbolic method uses the fact that if A, B, C are types and $A(x), B(x), C(x)$ are the corresponding OGFs then

$$C = A \oplus B \implies C(x) = A(x) + B(x)$$

and

$$C = A \otimes B \implies C(x) = A(x)B(x)$$

- For example

```
data Foo x = F0 | F1 x | F2 x (Foo x)
```

Symbolic Method: Finding generating functions

- Flajolet and Sedgewick propose a simple method of finding equation for the OGF of a given combinatorial construction expressed in their specification language
- In the special case of algebraic data types, the symbolic method uses the fact that if A, B, C are types and $A(x), B(x), C(x)$ are the corresponding OGFs then

$$C = A \oplus B \implies C(x) = A(x) + B(x)$$

and

$$C = A \otimes B \implies C(x) = A(x)B(x)$$

- For example

```
data Foo x = F0 | F1 x | F2 x (Foo x)
```

$$Foo(x) = 1 + x + xFoo(x)$$

Symbolic Method: Examples

```
data Empty
```

Symbolic Method: Examples

```
data Empty
```

$$E(x) = 0$$

Symbolic Method: Examples

```
data Empty
```

$$E(x) = 0$$

```
data One = One
```

Symbolic Method: Examples

```
data Empty
```

$$E(x) = 0$$

```
data One = One
```

$$O(x) = 1$$

Symbolic Method: Examples

```
data Empty
```

$$E(x) = 0$$

```
data One = One
```

$$O(x) = 1$$

```
data Bool = True | False
```

Symbolic Method: Examples

```
data Empty
```

$$E(x) = 0$$

```
data One = One
```

$$O(x) = 1$$

```
data Bool = True | False
```

$$B(x) = 1 + 1$$

Symbolic Method: Examples

```
data Empty
```

$$E(x) = 0$$

```
data One = One
```

$$O(x) = 1$$

```
data Bool = True | False
```

$$B(x) = 1 + 1$$

$$B(x) = 2$$

Symbolic Method: Examples

```
data Maybe x = None | Just x
```

Symbolic Method: Examples

```
data Maybe x = None | Just x
```

$$M(x) = 1 + x$$

Symbolic Method: Examples

```
data Maybe x = None | Just x
```

$$M(x) = 1 + x$$

```
data L x = Nil | Cons x (L x)
```

Symbolic Method: Examples

```
data Maybe x = None | Just x
```

$$M(x) = 1 + x$$

```
data L x = Nil | Cons x (L x)
```

$$L(x) = 1 + xL(x)$$

Symbolic Method: Examples

```
data Maybe x = None | Just x
```

$$M(x) = 1 + x$$

```
data L x = Nil | Cons x (L x)
```

$$L(x) = 1 + xL(x)$$

$$L(x) = \frac{1}{1 - x}$$

Symbolic Method: Examples

```
data Maybe x = None | Just x
```

$$M(x) = 1 + x$$

```
data L x = Nil | Cons x (L x)
```

$$L(x) = 1 + xL(x)$$

$$L(x) = \frac{1}{1 - x}$$

$$L(x) = 1 + x + x^2 + x^3 + x^4 + x^5 + x^6 + x^7 + \ldots$$

Symbolic Method: Examples

```
data T x = Leaf x | Branch (T x) (T x)
```

Symbolic Method: Examples

```
data T x = Leaf x | Branch (T x) (T x)
```

$$T(x) = x + T^2(x)$$

Symbolic Method: Examples

```haskell
data T x = Leaf x | Branch (T x) (T x)
```

$$T(x) = x + T^2(x)$$

$$T(x) = \frac{1 \pm \sqrt{1 - 4x}}{2}$$

Symbolic Method: Examples

```
data T x = Leaf x | Branch (T x) (T x)
```

$$T(x) = x + T^2(x)$$

$$T(x) = \frac{1 \pm \sqrt{1 - 4x}}{2}$$

$$T(x) = x + x^2 + 2x^3 + 5x^4 + 14x^5 + 42x^6 + 132x^7 + \ldots$$

Symbolic Method: Examples

```
data B x = B0 x | B1 x
data L x = Nil | Cons x (L x)
data Bits x = Bits (L (B x))
```

Symbolic Method: Examples

```
data B x = B0 x | B1 x
data L x = Nil | Cons x (L x)
data Bits x = Bits (L (B x))
```

$$B(x) = 2x$$
$$L(x) = 1 + xL(x)$$
$$Bits(x) = L(B(x))$$

Symbolic Method: Examples

```haskell
data B x = B0 x | B1 x
data L x = Nil | Cons x (L x)
data Bits x = Bits (L (B x))
```

$$
\begin{aligned}
B(x) &= 2x \\
L(x) &= 1 + xL(x) \\
Bits(x) &= L(B(x))
\end{aligned}
$$

$$Bits(x) = \frac{1}{1-2x}$$

Symbolic Method: Examples

```
data B x = B0 x | B1 x
data L x = Nil | Cons x (L x)
data Bits x = Bits (L (B x))
```

$$\begin{aligned} B(x) &= 2x \\ L(x) &= 1 + xL(x) \\ Bits(x) &= L(B(x)) \end{aligned}$$

$$Bits(x) = \frac{1}{1-2x}$$

$$Bits(x) = 1 + 2x + 4x^2 + 8x^3 + 16x^4 + 32x^5 + 64x^6 + 128x^7 + \ldots$$

Symbolic Method: Examples

```
data C x = C0 x | C1 x x
data L x = Nil | Cons x (L x)
data H = H0 (L (C x))
```

Symbolic Method: Examples

```
data C x = C0 x | C1 x x
data L x = Nil | Cons x (L x)
data H = H0 (L (C x))
```

$$
\begin{aligned}
C(x) &= x + x^2 \\
L(x) &= 1 + xL(x) \\
H(x) &= L(C(x))
\end{aligned}
$$

Symbolic Method: Examples

```
data C x = C0 x | C1 x x
data L x = Nil | Cons x (L x)
data H = H0 (L (C x))
```

$$
\begin{aligned}
C(x) &= x + x^2 \\
L(x) &= 1 + xL(x) \\
H(x) &= L(C(x))
\end{aligned}
$$

$$H(x) = \frac{1}{1 - x - x^2}$$

Symbolic Method: Examples

```
data C x = C0 x | C1 x x
data L x = Nil | Cons x (L x)
data H = H0 (L (C x))
```

$$
\begin{aligned}
C(x) &= x + x^2 \\
L(x) &= 1 + xL(x) \\
H(x) &= L(C(x))
\end{aligned}
$$

$$H(x) = \frac{1}{1 - x - x^2}$$

$$H(x) = 1 + x + 2x^2 + 3x^3 + 5x^4 + 8x^5 + 13x^6 + 21x^7 + \ldots$$

Symbolic Method: Examples

```
data F x = F0 x | F1 (F (G x))
data G x = G0 x | G1 x (G x)
```

Symbolic Method: Examples

```
data F x = F0 x | F1 (F (G x))
data G x = G0 x | G1 x (G x)
```

$$f(x) = x + f(g(x))$$
$$g(x) = x + xg(x)$$

Symbolic Method: Examples

```
data F x = F0 x | F1 (F (G x))
data G x = G0 x | G1 x (G x)
```

$$f(x) = x + f(g(x))$$
$$g(x) = x + xg(x)$$

$$f(x) = x + f\left(\frac{x}{1-x}\right)$$

Symbolic Method: Examples

```
data F x = F0 x | F1 (F (G x))
data G x = G0 x | G1 x (G x)
```

$$f(x) = x + f(g(x))$$
$$g(x) = x + xg(x)$$

$$f(x) = x + f(\frac{x}{1-x})$$

$$f(x) = \sum_{n=0}^{\infty} \frac{x}{1-nx}$$

Symbolic Method: Examples

```
data F x = F0 x | F1 (F (G x))
data G x = G0 x | G1 x (G x)
```

$$f(x) = x + f(g(x))$$
$$g(x) = x + xg(x)$$

$$f(x) = x + f(\frac{x}{1-x})$$

$$f(x) = \sum_{n=0}^{\infty} \frac{x}{1-nx}$$

$$f(x) = \psi(-\frac{1}{x}) + \gamma$$
$$\psi(x) = \frac{d}{dx} \ln \Gamma(x) = \frac{\Gamma'(x)}{\Gamma(x)}$$

Curry–Howard correspondence: Propositions as types

- The Curry–Howard correspondence is an observation that two seemingly unrelated formalism, namely typed λ-calculus and proof theory, are in fact closely related

Curry–Howard correspondence: Propositions as types

- The Curry–Howard correspondence is an observation that two seemingly unrelated formalism, namely typed λ-calculus and proof theory, are in fact closely related
- The general idea is that one can interpret a type A as a proposition in proof theory

Curry–Howard correspondence: Propositions as types

- The Curry–Howard correspondence is an observation that two seemingly unrelated formalism, namely typed λ-calculus and proof theory, are in fact closely related
- The general idea is that one can interpret a type A as a proposition in proof theory
- And a value $a : A$ of type A is interpreted as a proof of the corresponding proposition

Curry–Howard correspondence: Propositions as types

- The Curry–Howard correspondence is an observation that two seemingly unrelated formalism, namely typed λ-calculus and proof theory, are in fact closely related
- The general idea is that one can interpret a type A as a proposition in proof theory
- And a value $a : A$ of type A is interpreted as a proof of the corresponding proposition
- The function type $A \to B$ gets interpreted as the proposition $A \implies B$

Curry–Howard correspondence: Propositions as types

- The Curry–Howard correspondence is an observation that two seemingly unrelated formalism, namely typed λ-calculus and proof theory, are in fact closely related
- The general idea is that one can interpret a type A as a proposition in proof theory
- And a value $a : A$ of type A is interpreted as a proof of the corresponding proposition
- The function type $A \to B$ gets interpreted as the proposition $A \implies B$
- The types $A \oplus B$ and $A \otimes B$ get interpreted as $A \vee B$ and $A \wedge B$

The Rosetta Stone

Category Theory	Physics	Topology	Logic	Computation
object X	Hilbert space X	manifold X	proposition X	data type X
morphism $f\colon X \to Y$	operator $f\colon X \to Y$	cobordism $f\colon X \to Y$	proof $f\colon X \to Y$	program $f\colon X \to Y$
tensor product of objects: $X \otimes Y$	Hilbert space of joint system: $X \otimes Y$	disjoint union of manifolds: $X \otimes Y$	conjunction of propositions: $X \otimes Y$	product of data types: $X \otimes Y$

The Rosetta Stone: Homotopy Type Theory style

Types	Logic	Sets	Homotopy
A	proposition	set	space
$a : A$	proof	element	point
$B(x)$	predicate	family of sets	fibration
$b(x) : B(x)$	conditional proof	family of elements	section
$\mathbf{0}, \mathbf{1}$	$\bot, \top$	$\emptyset, \{\emptyset\}$	$\emptyset, *$
$A + B$	$A \vee B$	disjoint union	coproduct
$A \times B$	$A \wedge B$	set of pairs	product space
$A \to B$	$A \Rightarrow B$	set of functions	function space
$\sum_{(x:A)} B(x)$	$\exists_{x:A} B(x)$	disjoint sum	total space
$\prod_{(x:A)} B(x)$	$\forall_{x:A} B(x)$	product	space of sections
Id_A	equality $=$	$\{ (x,x) \mid x \in A \}$	path space A^I

Dependent Types

- The idea of dependent types is to allow types to depend on values

Dependent Types

- The idea of dependent types is to allow types to depend on values
- In simply typed λ-calculus values and types are separate

Dependent Types

- The idea of dependent types is to allow types to depend on values
- In simply typed λ-calculus values and types are separate
- In Haskell's core language, for example, values and types get mapped to two separate data structures: `data Expr` and `data Type` respectively

Dependent Types

- The idea of dependent types is to allow types to depend on values
- In simply typed λ-calculus values and types are separate
- In Haskell's core language, for example, values and types get mapped to two separate data structures: `data Expr` and `data Type` respectively
- In dependently typed languages those two are unified

Dependent Types

- The idea of dependent types is to allow types to depend on values
- In simply typed λ-calculus values and types are separate
- In Haskell's core language, for example, values and types get mapped to two separate data structures: `data Expr` and `data Type` respectively
- In dependently typed languages those two are unified
- The core language consist of a single data structure: `data Expr` that represents both values and types

Dependent Types II

- To allow values to appear at the type level, the function type $A \to B$ and the product type $A \otimes B$ get generalized to Π-types and Σ-types respectively

Dependent Types II

- To allow values to appear at the type level, the function type $A \to B$ and the product type $A \otimes B$ get generalized to Π-types and Σ-types respectively
- $\prod_{a:A} B(a)$ is analogous to the $\forall$ quantifier $\forall a : A, B(a)$ in logic
- $\sum_{a:A} B(a)$ is analogous to the $\exists$ quantifier $\exists a : A, B(a)$ in logic

Dependent Types II

- To allow values to appear at the type level, the function type $A \to B$ and the product type $A \otimes B$ get generalized to Π-types and Σ-types respectively
- $\prod_{a:A} B(a)$ is analogous to the $\forall$ quantifier $\forall a : A, B(a)$ in logic
- $\sum_{a:A} B(a)$ is analogous to the $\exists$ quantifier $\exists a : A, B(a)$ in logic
- The last ingredient is the equality type $Id_A : A \to A \to Type$ populated by proofs $a =_A b : Id_A(a, b)$

Dependent Types: Examples

Dependent Types: Examples

$$\prod_{a:\mathbb{N}} \prod_{b:\mathbb{N}} (a \leq b \to \sum_{n:\mathbb{N}} a + n =_{\mathbb{N}} b)$$

Dependent Types: Examples

$$\prod_{a:\mathbb{N}} \prod_{b:\mathbb{N}} (a \leq b \to \sum_{n:\mathbb{N}} a + n =_{\mathbb{N}} b)$$

$$Iso(A, B : Type) := \sum_{f:A \to B} \sum_{g:B \to A} (\prod_{a:A} g(f(a)) =_A a) \otimes (\prod_{b:B} f(g(b)) =_B b)$$

Challenge

- "Learn a new programming language every year" from the book "The Pragmatic Programmer"

Challenge

- "Learn a new programming language every year" from the book "The Pragmatic Programmer"
- "A language that doesn't affect the way you think about programming, is not worth knowing." Alan Perlis

Challenge

- "Learn a new programming language every year" from the book "The Pragmatic Programmer"
- "A language that doesn't affect the way you think about programming, is not worth knowing." Alan Perlis
- Make the next language you'll learn be dependently typed

Challenge

- "Learn a new programming language every year" from the book "The Pragmatic Programmer"
- "A language that doesn't affect the way you think about programming, is not worth knowing." Alan Perlis
- Make the next language you'll learn be dependently typed
  - Lean, lead by the author of the Z3 Theorem Prover

Challenge

- “Learn a new programming language every year” from the book “The Pragmatic Programmer”
- “A language that doesn’t affect the way you think about programming, is not worth knowing.” Alan Perlis
- Make the next language you’ll learn be dependently typed
  - Lean, lead by the author of the Z3 Theorem Prover
  - Idris

Challenge

- "Learn a new programming language every year" from the book "The Pragmatic Programmer"
- "A language that doesn't affect the way you think about programming, is not worth knowing." Alan Perlis
- Make the next language you'll learn be dependently typed
  - Lean, lead by the author of the Z3 Theorem Prover
  - Idris
  - Agda

Challenge

- "Learn a new programming language every year" from the book "The Pragmatic Programmer"
- "A language that doesn't affect the way you think about programming, is not worth knowing." Alan Perlis
- Make the next language you'll learn be dependently typed
  - Lean, lead by the author of the Z3 Theorem Prover
  - Idris
  - Agda
  - Coq

A taste of Lean

```
theorem prod_commutative (p q : Type) : p × q → q × p :=
λ hpq : p × q, prod.mk (prod.snd hpq) (prod.fst hpq)

theorem prod_commutative2 (p q : Type) : p × q → q × p :=
assume hpq : p × q,
have hp : p, from prod.fst hpq,
have hq : q, from prod.snd hpq,
show q × p, from prod.mk hq hp

theorem mul_cancel_left_or {a b c : ℤ} (H : a * b = a * c) : a = 0 ∨ b = c :=
have H2 : a * (b - c) = 0, by simp,
have H3 : a = 0 ∨ b - c = 0, from mul_eq_zero H2,
or.imp_or_right H3 (assume H4 : b - c = 0, sub_eq_zero H4)
```

Set theory Lean

```
def set (α : Type) := α → Prop

def mem (a : α) (s : set α) := s a

def union (s₁ s₂ : set α) : set α :=
{a | a ∈ s₁ ∨ a ∈ s₂}

def image (f : α → β) (s : set α) : set β :=
{b | ∃ a, a ∈ s ∧ f a = b}
```

Further reading

- "Analytic Combinatorics" by Philippe Flajolet and Robert Sedgewick
- "Physics, Topology, Logic and Computation: A Rosetta Stone" by John Baez and Mike Stay
- "Homotopy Type Theory: Univalent Foundations of Mathematics" by The Univalent Foundations Program
- "Proofs and Types" by Jean-Yves Girard
- "Constructive Mathematics and Computer Programming" by Per Martin-Löf
- "Tutorial: Theorem Proving in Lean" https://leanprover.github.io/tutorial/

Questions?