Programming with Math

Bartosz Milewski

@BartoszMilewski
BartoszMilewski.com

Why people hate types?

- Types limit our ability to reuse code
- Generic code is hard to write in strongly typed languages
- Working with types requires learning a new language
- Type errors are cryptic
- Proof: JavaScript has many more libraries than C++

On the other hand…

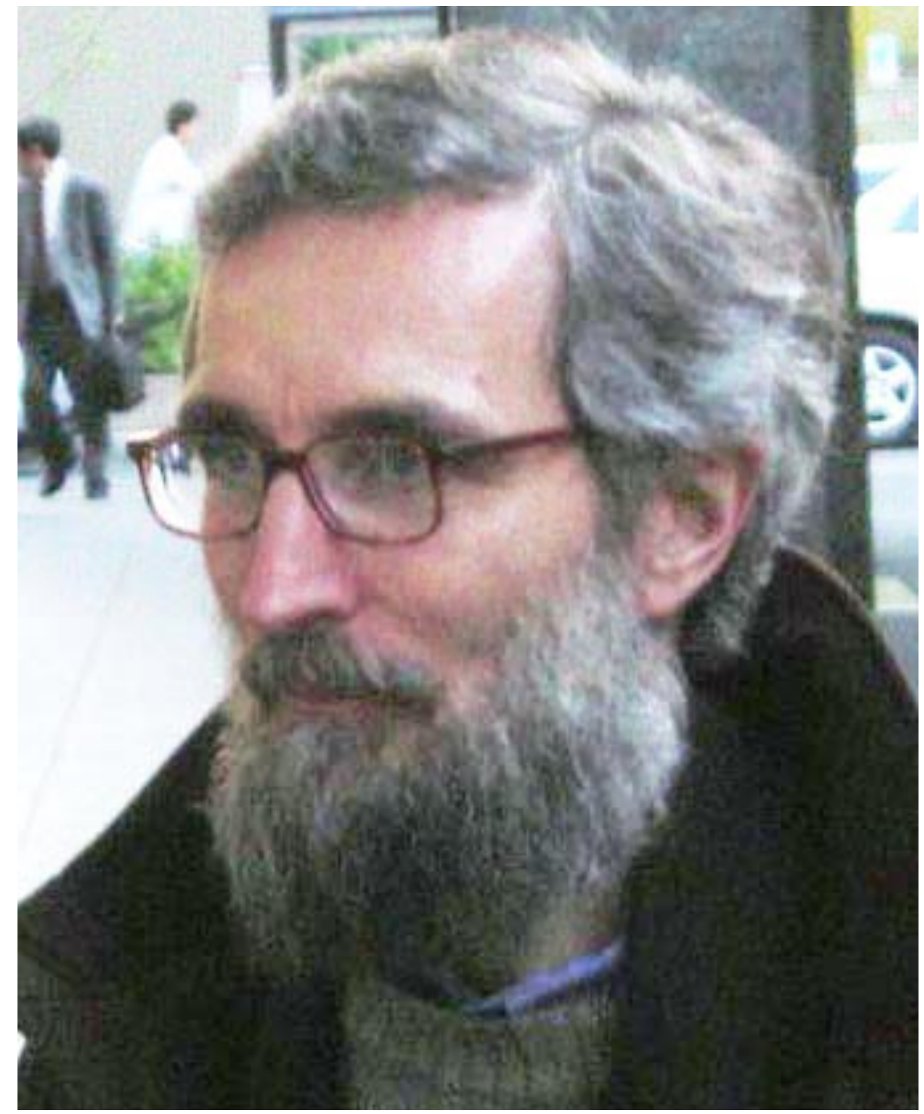
- Type checker detects lots of bugs at compile time
- Types provide up-to-date documentation
- Types can drive the design of software
- Proof: JavaScript libraries are notoriously low quality

The problem

- Most strongly typed languages have ad-hoc type systems
- Generics are added as an afterthought
- There is good theory, but language designer mostly ignore it
- Without good basics, it’s hard to build higher level abstractions

Systematic type theory

- Martin Löf type theory
  - Algebraic data types
  - Dependent types
- Intuitionistic logic
  - Propositions as types
- Category theory
  - Functors, monoids, monads



Per Martin-Löf

Abstracting over types

- Generic types:
  - Functions on types, Type constructors
  - Systematic construction of types = algebraic data types
  - Generic algorithms: polymorphic functions working on generic types

The power of composition

- Define primitive things
- Define ways of composing things
- The science of composition
  - Logic
  - Category theory
  - Type theory

Unit

- Logic. Truth value: $\top$
- Unit, `()`, **`void`** in C++/Java
- Set theory: Singleton set
- Introduction: `()`
- No elimination
- False: $\perp$ (Void, empty set)

Products

$$Logic\,.\,conjunction : a \land b$$

- Pairs, tuples, structs, records, (classes)
- Set theory: cartesian product of sets
- Introduction: `a -> b -> (a, b)`
- Elimination:
  - `fst :: (a, b) -> a`
  - `snd :: (a, b) -> b`

Sums

$$Logic\,.\,Alternative : a \vee b$$

- **`Either`**, tagged unions, class hierarchies

```
data Either a b =
  Left a | Right b
```

- Set theory: disjoint union
- Introduction:
  - **`Left  :: a -> Either a b`**
  - **`Right :: b -> Either a b`**
- Elimination: Pattern matching, case statement

```
case e of
  Left  x -> f x
  Right y -> g y
```

- Bool: 2 = 1 + 1

Exponentials

$$\boxed{Logic\,.\,Implication : a \Rightarrow b}$$

$$(a \wedge b) \wedge c \Rightarrow a \wedge (b \wedge c)$$

```
\x -> (fst (fst x), (snd (fst x), snd x))
```

- Function types, first class functions
- Set theory: set of functions **`a->b`**, exponential $\mathbf{b^a}$
- Introduction: lambda, **`\x -> expr x`**
- Elimination: evaluation, function application, **`f x`**

Algebraic identities

$$1 \times a \cong a$$

```
((), a) ~ a
```

$$a^{b+c} \cong a^b \times a^c$$

```
(b + c)-> a ~ (b -> a, c -> a)
```

$$a^2 \cong a^{1+1} \cong a \times a$$

```
Bool -> a ~(a, a)
```

$$a^{b \times c} \cong (a^c)^b$$

```
(b, c)-> a ~ b -> (c -> a)
```

Type functions

$$\Lambda a.1 + a$$

`Maybe a = Nothing | Just a`

- Type constructor **`Maybe`**
- Data constructors
  - **`Nothing`**: no argument, singleton output
  - **`Just`**: takes **`a`**, produces **`a`** (identity)
- Elimination: function of sum = pair of functions

Infinite products

$$Logic\,.\,universal\ \ quantification : \forall x\,.\,P(x)$$

- Polymorphic values, infinite products
- Intro: Provide a value for every possible type
- Elimination: Project one such value (and use it)

```
id :: forall a. a -> a
id x = x
```

Infinite sums

$$Logic\,.\,existential\ \ quantification : \exists x\,.\,P(x)$$

- Existential types, implementation hiding, pimpl pattern
- Intro: provide a value for any of the types
- Elimination: use this value without knowing the type

```
Expr = exists a. (a, a -> Int)
```

Existential types

- Intro: provide a value for any of the types
    - Infinitely many constructors, one for every type

```
data Expr = forall a. Expr a (a -> Int)

plus :: Expr
plus = Expr (3, 7) (\(x, y) -> x + y)
```

- Elimination: use this value without knowing the type
    - Use polymorphic function

```
eval :: Expr -> Int
eval (Expr x f) = f x
```

Recursion

- Advantage: Turing completeness
- Price: Non-termination
- Fixed points of algebraic equations

$$L(a) = 1 + a \times L(a)$$

```
List a = Nil | Cons a (List a)
```

Conclusions

- There is a strong foundation for types
- Generic programming is programming with types
- Generic types are functions on types
- Built from (algebraic) expressions on types
- Recursive data types are solutions to algebraic equations
- Algorithms are polymorphic functions