Haskell for mathematical libraries

Andrew Lelechenko

Barclays, London

Lambda Days 2020, Kraków, 13.02.2020

Reproducibility crisis

- **Problem:** Many scientific studies are difficult or impossible to replicate or reproduce.
- According to 2016 poll by *Nature*, 70% of 1500 participants failed to reproduce other scientist's experiment.
- **Even worse:** 50% failed to reproduce their own experiment.
- Social sciences and medicine are most susceptible *(kinda expected)*.
- But are computer sciences and mathematics secure?

Reproducibility crisis — CS

C. Collberg, T. Proebsting, A. M. Warren,
Repeatability and benefaction in computer systems research, 2015.

- Collberg et al. conducted an exploration of ~ 400 papers from ACM conferences and journals.
- For 32.3% they were able to obtain the code and build it easily.
- For 48.3% they managed to build the code, but it may have required extra effort.
- For 54.0% either they managed to build the code or the authors stated the code would build with reasonable effort.

Reproducibility crisis — mathematics

abc conjecture

Let a, b, c be coprime positive integers such that $a + b = c$. Then the product of distinct prime factors of $a \cdot b \cdot c$ is usually not much smaller than c:

$$c > \mathrm{rad}(abc)^{1+\varepsilon}.$$

- In 2012 Shinichi Mochizuki outlined a proof on 500 pages.
- In 2015–2016 several workshops tried to grasp his ideas.
- In 2017 Go Yamashita published 300 pages of explanations.
- In 2018 Peter Scholze and Jakob Stix identified a gap.
- Mochizuki claimed that they misunderstood vital aspects of the theory and made invalid simplifications.

Concrete math

If I can give an abstract proof of something, I'm reasonably happy. But if I can get a concrete, computational proof and actually produce numbers I'm much happier. I'm rather an addict of doing things on the computer.

John Milnor

As a computational and experimental pure mathematician my main goal is: insight. *. . . This is leading us towards an* Experimental Mathodology *as a philosophy and in practice.*

Jonathan M. Borwein,
Æsthetics for the working mathematician, 2001.

Haskell for the working mathematician

Andrew Lelechenko

Barclays, London

Lambda Days 2020, Kraków, 13.02.2020

Boolean blindness

Antipattern: use `Bool` to represent any type with two values:

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

Example:

```
filter ::  (a -> Bool) -> [a] -> [a]
```

Does `True` mean to keep or to discard here?

Pattern: use a domain-specific type.

```
data Action = Keep | Discard
```

Int blindness and nominal typing

Antipattern:

```
substring ::  Int -> Int -> String -> String
```

Is the second `Int` an offset or a count?

Pattern: use new types to wrap `Int`s and name them differently.

```
newtype Offset = Offset Int
newtype Count = Count Int
substring ::  Offset -> Count -> String -> String
```

Question: would named arguments help?

Int blindness and nominal typing — 2

Answer: not really. Named arguments do not prohibit invalid operations:

- `Count + Count = Count`
- `Offset + Count = Offset`
- `Count + Offset = ?`
- `Offset + Offset = ?`

Real world example:

```
montgomeryFactorisation ::
  Integer -> Word -> Word -> Integer -> Maybe Integer
```

Too many types!

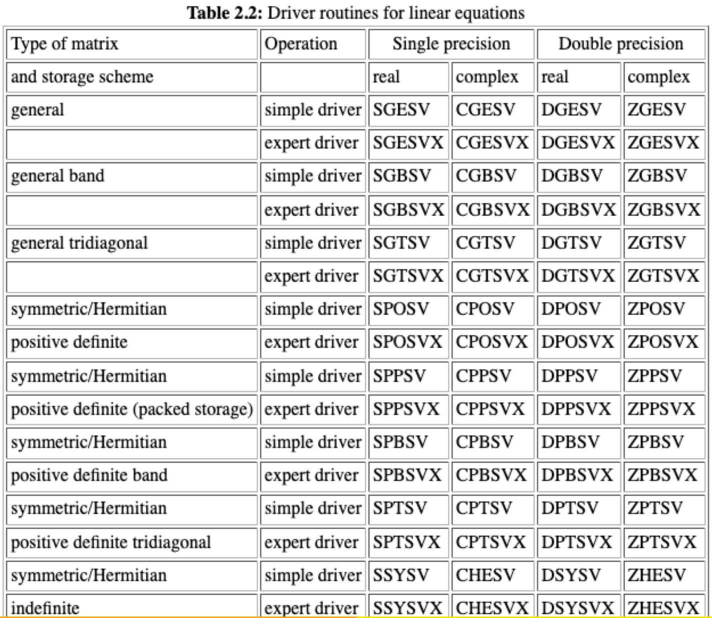


Table 2.2: Driver routines for linear equations

Type of matrix	Operation	Single precision		Double precision	
and storage scheme		real	complex	real	complex
general	simple driver	SGESV	CGESV	DGESV	ZGESV
	expert driver	SGESVX	CGESVX	DGESVX	ZGESVX
general band	simple driver	SGBSV	CGBSV	DGBSV	ZGBSV
	expert driver	SGBSVX	CGBSVX	DGBSVX	ZGBSVX
general tridiagonal	simple driver	SGTSV	CGTSV	DGTSV	ZGTSV
	expert driver	SGTSVX	CGTSVX	DGTSVX	ZGTSVX
symmetric/Hermitian	simple driver	SPOSV	CPOSV	DPOSV	ZPOSV
positive definite	expert driver	SPOSVX	CPOSVX	DPOSVX	ZPOSVX
symmetric/Hermitian	simple driver	SPPSV	CPPSV	DPPSV	ZPPSV
positive definite (packed storage)	expert driver	SPPSVX	CPPSVX	DPPSVX	ZPPSVX
symmetric/Hermitian	simple driver	SPBSV	CPBSV	DPBSV	ZPBSV
positive definite band	expert driver	SPBSVX	CPBSVX	DPBSVX	ZPBSVX
symmetric/Hermitian	simple driver	SPTSV	CPTSV	DPTSV	ZPTSV
positive definite tridiagonal	expert driver	SPTSVX	CPTSVX	DPTSVX	ZPTSVX
symmetric/Hermitian	simple driver	SSYSV	CHESV	DSYSV	ZHESV
indefinite	expert driver	SSYSVX	CHESVX	DSYSVX	ZHESVX

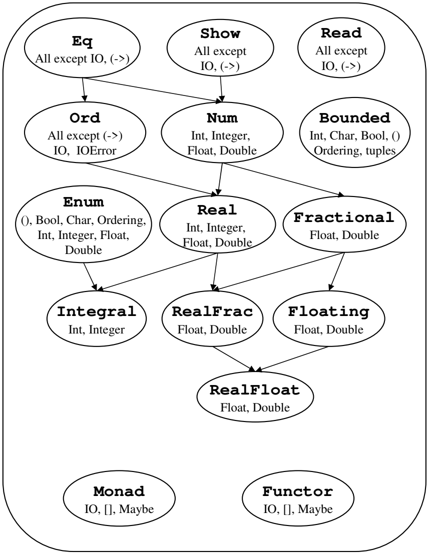
A taste of abstract algebra

- A set with an associative operation $\cdot$ such that $a \cdot (b \cdot c) = (a \cdot b) \cdot c$ is called a *semigroup*.
- If there is a neutral element, it is a *monoid*.
- And if the operation is invertible, it is a *group*.
- And if it commutes (so that $a \cdot b = b \cdot a$), it is an *abelian group*.

- A set with two sufficiently good operations may appear to be a *semiring*, a *ring*, a *rng* or *rig*.
- And if these operations are extraordinary good and fit for each other, it may even play out to be a *domain* or a *field*.

Type classes for the rescue

- **Haskell solution:** Type classes for ad-hoc polymorphism.
- **Problem:** Vanilla numeric classes are notoriously unfit for mathematics.
- **Haskell solution:** Use `algebra` package, which offers a hierarchy of 100+ numeric classes, carefully reflecting algebraic structures.
- **Any problems?**



The Haskell 2010 Report, S. Marlow (ed.), 2010

Linear algebra

Problem: The determinant is defined only for square matrices. Multiplication is defined only if the width of the first argument matches the height of the second one.

```
det :: Matrix -> Maybe Double
mul :: Matrix -> Matrix -> Maybe Matrix
```

Haskell solution: Parametrize matrices by phantom type-level numbers. E. g., `Matrix 3 3`, `Matrix 2 4`.

```
det :: Matrix n n -> Double
mul :: Matrix k l -> Matrix l m -> Matrix k m
```

Modular arithmetic in cryptography

Problem: In modular arithmetic all values are reduced by modulo. Values with different moduli are incompatible.

```
newtype Mod = Mod Int Int

(+) :: Mod -> Mod -> Maybe Mod
Mod n mod + Mod n' mod'
  | mod == mod' = Just (Mod ((n + n') `rem` mod)) mod
  | otherwise   = Nothing

Mod 4 7 + Mod 5 7 = ?
```

Haskell solution: Parametrize modular values by phantom type-level numbers.

```
data Mod (mod :: Nat) = Mod Int
(+) :: Mod m -> Mod m -> Mod m
```

Singleton values

Definition

An integer r is a square root of n modulo m when $r^2 \equiv n \pmod{m}$.

```
sqrtMod :: Mod m -> Maybe (Mod m)
sqrtMod (Mod 4 :: Mod 5) = Just (3 :: Mod 5)
```

Problem: The algorithm requires prime factorisation of m, which is expensive to compute each time we need `sqrtMod`:

```
factorise :: Int -> [(Int, Int)]
factorise 60 = [(2, 2), (3, 1), (5, 1)]
```

Never forget to take a leverage of nominal types!

```
factorise :: Int -> [(Prime, Power)]
```

Singleton values — 2

```
sqrtMod ::  Mod m -> Maybe (Mod m)
```

```
sqrtMod ::  [(Prime, Power)] -> Mod m -> Maybe (Mod m)
```

Haskell solution: Use singleton types, which establish a bijection between a type-level index and its property, represented at the term level. Define

```
newtype SFactors m = SFactors [(Prime, Power)]
```

and ensure by smart constructors that it has a single inhabitant.

```
sqrtMod ::  SFactors m -> Mod m -> Maybe (Mod m)
```

Lazy factorization

- Prime factorisation is very expensive:

```
factorise :: Int -> [(Prime, Power)]
factorise 60 = [(2, 2), (3, 1), (5, 1)]
```

- **Problem:** What if further computations depend only on the first factor? Should we expose more helpers?

```
firstFactor :: Int -> (Prime, Power, Int)
```

- **Haskell solution:** In a lazy language an output list is computed only on demand. If we consume only its head, other elements are not computed at all. This helps to keep API neat and concise.

```
firstFactor = head . factorise
```

Recurrent sequences

Problem: Here are Fibonacci numbers:

$$F_0 = 0$$
$$F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2}$$

Naive solution:

```
fib :: Int -> Int
fib n = if n < 2 then n else fib (n-1) + fib (n-2)
```

Haskell solution: Use a lazy list as a cache:

```
fibs :: [Int]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

Riemann zeta function

Problem: *Computational strategies for the Riemann zeta function* by Borwein et al., 2000:

$$-\frac{(1-2^{-2m-1})2\zeta(2m+1)}{(\pi i)^{2m}} = \sum_{k=1}^{m-1} \frac{(1-4^{-k})\zeta(2k+1)}{(\pi i)^{2k}(2m-2k)!} + \\ + \frac{1}{(2m)!}\left\{\log 2 - \frac{1}{2m} + \sum_{n=1}^{\infty} \frac{\zeta(2n)}{4^n(n+m)}.\right\}$$

Haskell solution: There is a generic approach to memoize recurrent sequences, using fix-point combinator and higher-order functions. Moreover, one can split such computation into an actual computation and memoization layer, which can be composed independently.

Fix-point combinator

This is a fix-point combinator:

```
fix :: (a -> a) -> a
fix f = f (fix f)
```

These are our naïve Fibonacci numbers:

```
fib :: Int -> Int
fib n = if n < 2 then n else fib (n-1) + fib (n-2)
```

These are Fibonacci numbers with recursion factored out:

```
fibF :: (Int -> Int) -> Int -> Int
fibF f n = if n < 2 then n else f (n-1) + f (n-2)
```

Conclusion

- **Problem:** mathematical objects are pure, immutable and lazy.
- **Solution:** Haskell is pure, immutable and lazy.

Thank you!

`https://github.com/Bodigrim/{arithmoi,chimera,mod,poly}`