Number theory in Haskell

Andrew Lelechenko
1@dxdy.ru

#kievfprog, Kiev, 18.11.2017

Why do math not in C?

"Haskell is slow, use C".

- Haskell is not slow, but C is fast.
  You cannot expect to beat hundred man years after lunch.
  Example: linear algebra in BLAS and LAPACK.
- Haskell is maintainable, C is not.
  Typical mathematical C library (especially *the fastest*^{TM}) comes without documentation, has arcane interface and unclear assumptions, emits uncontrolled side effects.

Why do math in Haskell?

- Purely functional language, similar to mathematics.
- Terse syntax, resembling mathematical formulas.
- Interactive mode of evaluation.
- Quick, but sound prototyping due to the polymorphic, but strong type system.

Haskell flaws for number crunching

- No manual memory management.
  Chris Done, Fast Haskell: Competing with C at parsing XML,
  http://chrisdone.com/posts/fast-haskell-c-parsing-xml
- No automatic vectorisation (SSE/AVX instructions).
  Use FFI.
  https://github.com/sergv/nn

Initially designed to cover Project Euler (https://projecteuler.net).
Inspired by leading PARI/GP library.

- Modular computations.
- Effective prime crunching algorithms.
- Toolbox of arithmetic functions.
- Gaussian integers.
- Recurrent relations.
- Riemann zeta function.
- Integer roots and logarithms.

Integer sqrt

Implement $\texttt{isqrt} :: \text{Integer} \to \text{Integer}$ such that

$$(\texttt{isqrt}\ n)^2 \le n < (1 + \texttt{isqrt}\ n)^2.$$

For example, $\texttt{isqrt}\ 4 = 2$, $\texttt{isqrt}\ 5 = 2$, $\texttt{isqrt}\ 9 = 3$.

- $\texttt{isqrt1}\ n = \texttt{floor}\ (\texttt{sqrt}\ (\texttt{fromInteger}\ \texttt{@Double}\ n))$

  $\texttt{isqrt1}\ 3037000502^2 = 3037000501$. Precision loss!

- $\texttt{isqrt2}\ n = \texttt{head}\ (\texttt{dropWhile}\ (\lambda r \to (r+1)^2 \le n)\ [\texttt{isqrt1}\ n..])$

  $\texttt{isqrt2}\ 2^{1024} = 2^{1024}$. Out of bounds!

Integer sqrt

Implement `isqrt :: Integer` $\to$ Integer such that

$$(\texttt{isqrt}\ n)^2 \leq n < (1 + \texttt{isqrt}\ n)^2.$$

- Heron algorithm:

```
heron n =
  head $
    dropWhile (\r -> r > step r) $
      iterate step n
  where
    step r = (r + n `quot` r) `quot` 2
```

  Valid, but slow.

- Karatsuba square root: divide-and-conquer algorithm, inspired by famous Karatsuba multiplication. Takes $O(n^{1.585})$ time.
  https://hal.inria.fr/inria-00072854/PDF/RR-3805.pdf

Anatoly Karatsuba (1937-2008)

- Research works in the field of analytic number theory, including trygonometric series and mean theorems. His results are mostly existence theorems, which do not provide exact constructions.
- Ironically, today he is widely known for his fast multiplication algorithm, invented by accident during his study in university.

Modular power

```haskell
powMod :: Integral a => a -> a -> a -> a
powMod x y m = (x ^ y) `mod` m
```

Intermediate value of x^y may be extremely huge, larger than physical memory. Can we do better?

```haskell
powMod :: Integral a => a -> a -> a -> a
powMod x y m =
  head $
    genericDrop y $
      iterate (\n -> n * x `mod` m) 1
```

This version performs y multiplications. Can we do better?

Binary algorithm

How (^) avoids linear number of multiplications?

	1	**2**	**4**	**8**
	x	x^2	x^4	x^8
$11 =$	1	1	0	1
	x	x^3	x^3	x^{11}
$13 =$	1	0	1	1
	x	x	x^5	x^{13}

Binary algorithm

```
(^) :: Integral a => a -> a -> a
x ^ y = f x y 1

f :: Integral a => a -> a -> a -> a
f x 0 z = 1
f x 1 z = x * z
f x y z
  | even y    = f (x * x) (y `quot` 2) z
  | otherwise = f (x * x) ((y - 1) `quot` 2) (x * z)
```

Example: $13 = 1101_2$.

$$x^{13} = \mathtt{f}\ x^1\ 13\ 1 = \mathtt{f}\ x^2\ 6\ x = \mathtt{f}\ x^4\ 3\ x = \mathtt{f}\ x^8\ 1\ x^5 = x^{13}$$

Let us steal the trick!

Polymorphic powMod

Append `mod` after every multiplication:

```
powMod :: Integral a => a -> a -> a -> a
powMod x y m = f x y 1
  where
    f :: Integral a => a -> a -> a -> a
    f x 0 z = 1
    f x 1 z = x * z `mod` m
    f x y z = f (x * x `mod` m) (y `quot` 2)
      (if odd y then (x * z `mod` m) else z)
```

Boxed and unboxed

- Unboxed type contains a primitive value. For instance, `Word#` is a machine-sized word, similar to its C counterpart.
- Boxed type contains:
  - Link to unboxed value.
  - Error message.
  - Unevaluated thunk.
  - Blackhole.
- Unboxed type is always monomorphic.
- Boxed type can be polymorphic.

Unboxed power

Can we write more efficient implementations for concrete types?

```
powModWord :: Word -> Word -> Word -> Word
powModWord (W# x) (W# y) (W# m) = W# (f x y 1##)
  where
    f :: Word# -> Word# -> Word# -> Word#
    f x 0## z = 1##
    f x 1## z = timesMod x z m
    f x y   z = f (timesMod x x m) (y `uncheckedShiftRL#`
        1#)
      (if odd# y then (timesMod x z m) else z)

timesMod :: Word# -> Word# -> Word# -> Word#
timesMod x y m = r
  where
    (# hi, lo #) = timesWord2# x y
    (#  q, r  #) = quotRemWord2# hi lo m

odd# :: Word# -> Bool
odd# w = isTrue# (word2Int# (w `and#` 1##))
```

One interface to rule them all

Specialize `powMod` for common use cases.

```
{-# RULES
  "powMod/Integer" powMod = powModInteger
  "powMod/Natural" powMod = powModNatural
  "powMod/Int"     powMod = powModInt
  "powMod/Word"    powMod = powModWord
  #-}
```

Mark `powMod` inlinable:

```
{-# INLINABLE powMod #-}
```

Thank you!

github.com/cartazio/arithmoi