FizzBuzz from First Priciples

Guy Gastineau

January 2023

Abstract

Explore solving the classic fizz buzz test from first principles in Haskell's type system. This is a silly, cheeky article introducing type level natural numbers (in a less efficient way than provided by base). We use DataKinds and TypeFamilies for almost everything including "printing". Not only will you learn how to define a very slow implementation of fizz buzz, but you will learn about the hidden, ghostly REPL that hides within the bowels of ghci.

Contents

Part I
Motivations

Most programmers have heard of the fizz buzz test even if they haven't had to write a program for it in an interview. It is very simple, and still Wikipedia thinks it has "value in coding interviews to analyze fundamental coding habits that may be indicative of overall coding ingenuity." I remain deeply skeptical of this claim. I thought its primary purpose was to identify candidates who simply can't program anything. Anyway, I suppose many engineers accidentally start from 0 or forget that the interval should be closed $[1, 100]$ (they only make it to 99) etc. Maybe this shows us that we don't pay enough attention when we think a task is too simple. Anyway fizz buzz ought to be simple enough; here is a simple, value-level fizz buzz in haskell:

```haskell
import Control.Monad (forM_)
import Data.Maybe (fromMaybe)

mkSound :: Int -> String -> Int -> Maybe String
mkSound m sound n = if n `mod` m == 0 then Just sound else Nothing
fizz, buzz :: Int -> Maybe String
fizz = mkSound 3 "Fizz"
buzz = mkSound 5 "Buzz"

appendMaybe :: Maybe String -> String -> Maybe String -> Maybe String
appendMaybe x        _   Nothing  = x
appendMaybe Nothing  _   y        = y
appendMaybe (Just x) sep (Just y) = Just (x <> sep <> y)

fizzBuzz :: IO ()
fizzBuzz = forM_ [1..100] fb
  where
    fb =  (putStrLn ∘) ∘ (fromMaybe ∘ show)
      <*> (flip appendMaybe " " ∘ fizz <*> buzz)
```

If you are not already familiar with fizz buzz, then maybe I should explain. For each number, n, in some interval, print "Fizz" if n is divisible by 3, "Buzz" if n is divisible by 5, "Fizz Buzz" if n is divisible by both 5 and 3 (*ie.* 15), or simply print the number if it is divisible by neither. As a programming "test", it is typically on the closed interval $[1, 100]$. Here is an example of output for the closed interval $[1, 15]$.

```
1
2
Fizz
4
Buzz
Fizz
7
8
```

```
Fizz
Buzz
11
Fizz
13
14
Fizz Buzz
```

But that is boring and too easy. Aren't we all tired of getting bogged down in the details of a value level language? Wouldn't it be better if we only worked with the world of types? I think we all know the answer to these serious questions. Think of all the problems this causes. We have to worry about performance in terms of complexity analysis. *Don't write slow, naive algorithms. Don't use too much memory.* Und so weiter... Well, we can completely avoid these issues by only using the compiler and never producing an executable. It is precisely this lofty goal that motivates me to torture myself and the world with this *research*.

Part II
Implementation

1 Shadow REPL or Printing During Compilation

How do we get started programming in haskell with no runtime costs? We first need to set up the *shadow REPL* in ghci. We'll need to import some entities from **GHC.TypeLits** in order to set up our *shadow REPL*, and we will define a Print type and finally a *function*[1] to use from within ghci itself. We will have another interesting hurdle. Our *shadow REPL* will only work if we can trick ghci into thinking we have a real value. This is why our **print** *function* exists at the value level. If GHC doesn't think there is some value to evaluate, then we can't get it to interpret our types interactively. Therefore, we introduce this forgetful **Proxy α** type. don't worry, even though it looks like we are using values, I promise you we will crash the compiler before any value can exist!

```
import GHC.TypeLits
  ( TypeError
  , ErrorMessage (..)
  , Symbol
  , AppendSymbol
  )

data Proxy α = Proxy

type family Print α :: k where
  Print α = TypeError ('ShowType α)

print :: forall α. Proxy (Print α)
print = Proxy :: Proxy (Print α)
```

The astute reader will realize that we are just making a custom type error that displays some type according to *ShowType*. This is really all we need to get into the shadow realms where GHC is an interpreted language with dynamic support. Feel free to try it yourself, by loading the literate source of this paper into ghci and using print with type applications. Here's an example session:

```
TERM=dumb ghci FizzBuzz.lhs
GHCi, version 9.2.5: https://www.haskell.org/ghc/  :? for help
Loaded GHCi configuration from /home/<yourname>/.ghci
[1 of 1] Compiling FizzBuzz         ( FizzBuzz.lhs, interpreted )
Ok, one module loaded.
ghci> import Prelude (Int, Double, Char)
ghci> :set -XTypeApplications
ghci> :set -XDataKinds
ghci> print @'[Int, Double, Char]
```

[1] It is actually just a value with an unused type variable. It is a *function* in the sense that we can pass it a type via the **TypeApplications** extension.

```
<interactive>:5:1: error:
    • '[Int, Double, Char]
    • When checking the inferred type
        it :: forall {k}. Proxy (TypeError ...)
```

And there you have it! Our type was `'[Int, Double, char]`, and that was displayed for us in the interactive error. Who knew this could be so easy?

2 Building the World

Anyone who is anyone knows that dependencies are bad. Like really bad. Posts on **r/rust** frequently thrash the language for using crates so freely. Don't rustaceans know that dependency chain attacks **ARE BAD?!?** Haskell also *suffers* from package fragmentation likely due to haskellers' obsession with abstraction. Didn't anyone ever tell you abstraction is also bad? Don't abstract! Pay attention to these tedious details that arise from incidental knowledge! So, *ahem*, we will avoid this problem by defining the world or, rather, all that we need of it.[2]

2.1 The Boolean Domain

We will be faced with some branching that is best suited to boolean logic. We will thus give the Haskell definition for the boolean domain, *{0,1}*. Syntactic sugar for a simple sum type will suffice.

```
data B = True | False
```

Then we need define the usual suspects ∧, ∨, *If*, and *Not*. Note well, *If* is strict in both of its arguments, so it only works as control flow when both branches are terminating. This makes it an especially shitty control flow operation.

```
type α ∧ β = And α β
infixl 3 ∧
type family And (α :: B) (β :: B) :: B where
  And 'True 'True = 'True
  And  _     _    = 'False

type α ∨ β = Or α β
infixl 3 ∨
type family Or (α :: B) (β :: B) :: B where
  Or 'True  _    = 'True
  Or  _    'True = 'True
  Or  _     _    = 'False

type family Not (p :: B) :: B where
  Not 'True  = 'False
  Not 'False = 'True
```

[2] Importing a few entities from **GHC.TypeLits** remains necessary for our shadow REPL. I have tried to avoid imports from **base** at all costs, but the world is an imperfect place

```
type family If (ρ :: B) (α :: k) (β :: k) :: k where
  If 'True  α _ = α
  If 'False _ β = β

type family Unless (ρ :: B) (msg :: ErrorMessage) :: Constraint where
  Unless 'True  _   = ()
  Unless 'False msg = TypeError msg
```

2.2 Computations that Might Fail

Just like humans, sometimes a computation fails in a way that doesn't need to end the world. To model this effect we may simply define a maybe type representing either a successful or a failed computation. We could pull its definition from **Data.Maybe**, but why would we do anything so reckless as depending on standard definitions from **base**? We will also need a way to get values out of Maybe. We can do this by providing a default value as a fallback when the computation wasn't successful.

```
data Maybe α = Just α | Nothing

type family RunMaybe (def :: k) (val :: Maybe k) :: k where
  RunMaybe def 'Nothing   = def
  RunMaybe _  ('Just val) = val
```

Now we may define an excellent helper that concatenates two *Maybe Symbol*s with a separator. Notice we only use the separator if both *Symbol*s were a successful computation. In the case that neither is a success, the result is a *Nothing'*.

```
type family AppendMaybeSymbol
  (α         :: Maybe Symbol)
  (separator :: Symbol)
  (β         :: Maybe Symbol) :: Maybe Symbol where
  AppendMaybeSymbol  'Nothing _           β       = β
  AppendMaybeSymbol   α       _          'Nothing = α
  AppendMaybeSymbol ('Just α) separator ('Just β) =
    'Just (AppendSymbol (AppendSymbol α separator) β)
```

2.3 A Natural Fugue

To complete our world, or what we need from it, we still need some kind of numbers. Fizz buzz is, after all, all about numbers. Luckily, we only need non-negative integrals, so the natural numbers should suffice. We will rely on induction for our formalization of arithmetic. This recursive structure reminds me of that timeless dance of voices chasing each other, theme-in-hand across an aural landscape sublime in its simple complications. I suppose one might even call it *piano* arithmetic.[3]

```
data Nat = S Nat | Z
```

[3] *Peano-arithmetic* - This is the worst pun I have ever conceived, and I am so sorry, dear reader. Moreover, while the Peano axioms involve induction, I think our approach is probably more similar to Grassman's, and I shouldn't (probably) falsely attribute this to Peano. Also, don't go look at pictures of his beard. You really don't want to look at it!

2.3.1 Addition and Multiplication

Well, that was simple enough, wasn't it? Of course, but now we will need to define recursive operations for our algebra. Addition is particularly easy. We simply recurse the left side until it is *Zero*.

$$n + m = \begin{cases} m & \text{if } n = 0 \\ n & \text{if } m = 0 \\ n' + succ(m) & \text{where } n = succ(n') \end{cases} \tag{1}$$

We include both of the zero identities under addition, because this is Haskell not Coq, and I don't hate myself enough to try proving the commutativity of addition to GHC. NB. **infixl 5 +** specifies that the **+** operator is a left-associative, infix operator with a precedence of 5.[4] Multiplication is simply a summation series guarded by case analysis

$$n \times m = \begin{cases} 0 & \text{if } n = 0 \\ \sum_{1}^{n} m & \text{otherwise} \end{cases} \tag{2}$$

acheived in Haskell using recursion and induction.

```
type n + m = Plus n m
infixl 5 +
type family Plus (n :: Nat) (m :: Nat) :: Nat where
  Plus 'Z     n = n
  Plus  n    'Z = n
  Plus ('S n) m = Plus n ('S m)

type n × m = Mult n m
infixl 6 ×
type family Mult (n :: Nat) (m :: Nat) :: Nat where
  Mult  'Z     _ = 'Z
  Mult   _    'Z = 'Z
  Mult ('S 'Z) m = m
  Mult ('S  n) m = m + n × m
```

2.3.2 Subtraction

Subtraction is a little dirty. Domain analysis would show that some pairs of inputs do not yeild valid elements of $\mathbb{N}$, *ie.* if the domain of subtraction is $\mathbb{N} \times \mathbb{N}$ then the range must be $\mathbb{Z}$. Instead of taking the time to model $\mathbb{Z}$ in our program, we will simply restrict the domain of subtraction such that the range is $\mathbb{N}$.

$$n - m = \begin{cases} Underflow! & \text{if } n < m \\ n & \text{if } m = 0 \\ n' - m' & \text{where } n = succ(n') \text{ and } m = succ(m') \end{cases} \tag{3}$$

[4] (Type level) function application has the highest precedence, so our operator will interfere with the application of neither constructors nor type families. Precedence for multiplication and division are higher as is customary, and logical operations and comparisons on natrual numbers will have lower precedence to allow for the most natural mixing of numbers in propositions.

We accomplish this simply by crashing on invalid inputs with a *Subtraction underflow!* message to the user in the *shadow REPL* when n is less than k.

```
type n - m = Subt n m
infixl 5 -
type family Subt (n :: Nat) (m :: Nat) :: Nat where
  Subt   n     'Z    = n
  Subt  'Z      m    = TypeError ('Text "Subtraction underflow!")
  Subt ('S n) ('S m) = Subt n m
```

2.3.3 Division and Modulus

The range of division is obviously $\supset \mathbb{N}$ for all but the most restrictive subsets of $\mathbb{N} \times \mathbb{N}$. Furthermore, we must account for attempted division by zero. We side-step the range/codomain issue by returning a product of the *quotient* and *remainder*. I believe this is the Euclidean remainder, but I am too lazy to check that right now. So, the range of our division function is the cartesian product $\mathbb{N} \times \mathbb{N}$. We define a generic product, and alias its accessors for semantic clarity as *Quotient* and *Remainder*. To handle division by zero, we send the user a message in a fashion similar to our subtraction underflow error above. Division as the product of quotient and remainder also makes an implementation of *modulo* and a *divides* predicate trivial. Our division function also requires keeping track of its recursion stack. It[5] is, after all, an identity for the *quotient*.

$$div(quo, n, m) = \begin{cases} DivideByZero! & \text{if } m = 0 \\ (quo, 0) & \text{if } n = 0 \\ div(succ(quo), n - m, m) & \text{if } n \geq m \\ (quo, n) & \text{otherwise} \end{cases} \tag{4}$$

```
data Prod α β = Prod α β

type family Fst (αβ :: Prod α β) :: α where
  Fst ('Prod α _) = α
type family Snd (αβ :: Prod α β) :: β where
  Snd ('Prod _ β) = β

type Quotient  αβ = Fst αβ
type Remainder αβ = Snd αβ

type n ÷ m = Divi Zero n m
infixl 6 ÷
type family Divi
  (quotient :: Nat)
  (n :: Nat)
  (m :: Nat) :: Prod Nat Nat where
  Divi _    _ 'Z = TypeError ('Text "Divide by zero!")
  Divi quo 'Z  _ = 'Prod quo 'Z
  Divi quo  n  m =
```

[5] the recursion stack/counter

```
    If (n ≥ m) (Divi ('S quo) (n - m) m)
               ('Prod quo n)

type Divides denom numer = Modulo numer denom == Zero

type n % m = Modulo n m
type family Modulo (n :: Nat) (m :: Nat) :: Nat where
  Modulo n m = Remainder (n ÷ m)
```

2.4 Building Intervals

Back to something less intimidating ... We can produce closed intervals on the natural numbers now as lists, and we can pass those to some FizzBuzz function. I haven't figured out how to relieve the invariant $n <= k$. When $n > k$, **Interval** loops endlessly, but I would prefer giving an empty list instead. I tried using **If**, but it suffered the same issue. Well, this is the price we pay for using such a dangerous language with so many footguns.

```
{- INVARIANT! `n < k` -}
type n ... m = Interval n m
infix 4 ...
type family Interval (n :: Nat) (k :: Nat) :: [Nat] where
  Interval n n = '[n]
  Interval n k = n ': Interval ('S n) k
```

3 Playing the Game

In order to satisfy the rules of fizz buzz we need the ability to print natural numbers that are neither divisible by 3 nor 5. We are only concerned with 3-digit numbers, since the canonical fizz buzz test for programmers works on the interval $(1, 100)$. We need to extract the digits from a number, and we may simply convert these digits to **Symbol**s and concatenate them. [6]

```
type OnesPlace n     = Remainder (n ÷ Ten)
type TensPlace n     = Quotient (Remainder (n ÷ Hundred) ÷ Ten)
type HundredsPlace n = Quotient (n ÷ Hundred)

type family ToSymbol (n :: Nat) :: Symbol where
  ToSymbol n =
    AppendSymbol
      (AppendSymbol
       (If (n ≥ Hundred) (Digit (HundredsPlace n)) "")
       (If (n ≥ Ten) (Digit (TensPlace n)) ""))
      (Digit (OnesPlace n))
```

Finally we can define a function that *vocalizes* a natural number according to the rules of our fizz buzz game. Recursive application on kind **[Nat]** results in a type level list of kind **Symbol**, that may be printed in the *shadow REPL*.

[6] For definitions of the number aliases and **Digit** see Appendix A

```
type family Sound (n :: Nat) :: Maybe Symbol where
  Sound n = AppendMaybeSymbol
            (If (Divides Three n) ('Just "Fizz") 'Nothing)
            " "
            (If (Divides Five  n) ('Just "Buzz") 'Nothing)

type family FizzBuzz (ns :: [Nat]) :: [Symbol] where
  FizzBuzz '[]       = '[]
  FizzBuzz (n ': ns) =
    RunMaybe (ToSymbol n) (Sound n) ': FizzBuzz ns
```

And now we may use our work to play fizz buzz in the GHCI interactive terminal. Remember, we are using the *shadow REPL*, so our resultant value will belong to the first bullet item in an interactive error. As promised, there are no values, there are only types and a compiler crash. Feel free to try other intervals. Just remember, we have made this **very** inefficient, so make some tea or something if you want to go big.

```
TERM=dumb ghci FizzBuzz.lhs
GHCi, version 9.2.5: https://www.haskell.org/ghc/  :? for help
Loaded GHCi configuration from /home/<yourname>/.ghci
[1 of 1] Compiling FizzBuzz         ( FizzBuzz.lhs, interpreted )
Ok, one module loaded.
ghci> :set -XTypeApplications
ghci> :set -XDataKinds
ghci> print @(FizzBuzz (Five × Three ... Five × Five))

<interactive>:3:1: error:
    • '["Fizz Buzz", "16", "17", "Fizz", "19", "Buzz", "Fizz", "22",
       "23", "Fizz", "Buzz"]
    • When checking the inferred type
        it :: forall {k}. Proxy (TypeError ...)
```

Appendix A Boring Code

A.1 Equality and Ordering

From one base definition, for *less than or equal to*, we can derive the rest of a total ordering on the natural numbers. This was too boring to jam into the prose of the rest of the paper.

```
type n ≤ k = LE n k
infix 4 ≤
type family LE (n :: Nat) (k :: Nat) :: B where
  LE  'Z      _     = 'True
  LE ('S _)  'Z     = 'False
  LE ('S n) ('S k) = LE n k

type n ≥ k = k ≤ n
infix 4 ≥

type n == k = n ≤ k ∧ k ≤ n
infix 4 ==

type n < k = Not (n ≥ k)
infix 4 <
type n > k = Not (n ≤ k)
infix 4 >
```

A.2 Writing Numbers

As a practical matter, we define aliases for the first ten natural numbers and a type family to convert naturals to symbols. We only need to handle up to the first 101 natural numbers. Our implementation can print the first 1,000, but we don't have infinite time to wait for the compiler. We certainly don't have time to work with even larger numbers, sheesh. Anyway, writing our Peano-encoded numbers is such a chore that we will resort to aliases and arithmetic. For example, **Five × Five** is much more succint than **S (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S (S Z))))))))))))))))))))))))**.

```
type Zero  = 'Z
type One   = 'S Zero
type Two   = 'S One
type Three = 'S Two
type Four  = 'S Three
type Five  = 'S Four
type Six   = 'S Five
type Seven = 'S Six
type Eight = 'S Seven
type Nine  = 'S Eight

type family Digit (n :: Nat) :: Symbol where
  Digit Zero  = "0"
```

```
  Digit One   = "1"
  Digit Two   = "2"
  Digit Three = "3"
  Digit Four  = "4"
  Digit Five  = "5"
  Digit Six   = "6"
  Digit Seven = "7"
  Digit Eight = "8"
  Digit Nine  = "9"

type Ten     = Two × Five
type Hundred = Ten × Ten
```

Part III
Bonus Program

4 Returning to the World of Values

I know I said we would ignore the world of values, and we have mostly done just that; however, it is *maybe* useful to come back down to earth. Now that we can compute our fizz buzz at compile time, we might as well try to turn the result into a real program[7]. In order to make this program more efficient than other implementations of value level fizzbuzz we should tell GHC to inline our function calls aggressively. Hopefully we can get it to spit out a really optimized function that just inlines all the printing calls. This is a similar kind of optimization we might expect to see from some unrolled loops generated by LLVM for Rust or CPP. To do this we will need a typeclass for printing lists of symbols.

```haskell
class PrintLn (n :: [Symbol]) where
  printLn :: IO ()

instance PrintLn '[] where
  printLn = return ()

instance (KnownSymbol n, PrintLn ns) => PrintLn (n ': ns) where
  {-# INLINE printLn #-}
  printLn = Prelude.putStrLn (symbolVal (Proxy @n)) >> printLn @ns

main :: IO ()
main = printLn @(FizzBuzz (One ... Hundred))
```

[7] Ewe, grouse!