Dependent types in Haskell and bookkeeping

Andrew Lelechenko
1@dxdy.ru

F(café), Kiev, 20.06.2017

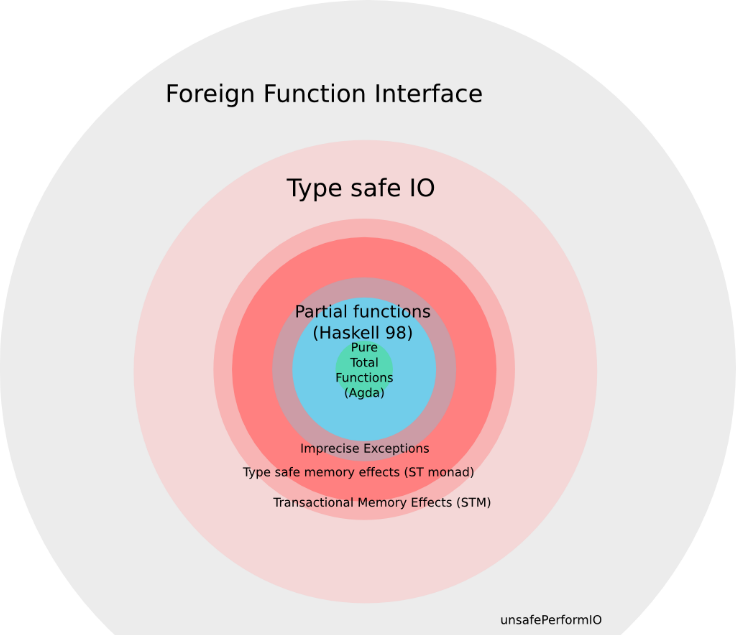
Why not Haskell?

Paul Hudak, John Hughes, Simon Peyton Jones, Philip Wadler,
A History of Haskell: Being Lazy With Class, HOPL-III, 2007.

- Haskell is lazy.
- Haskell is pure.
- Haskell has type classes.

Do you care?

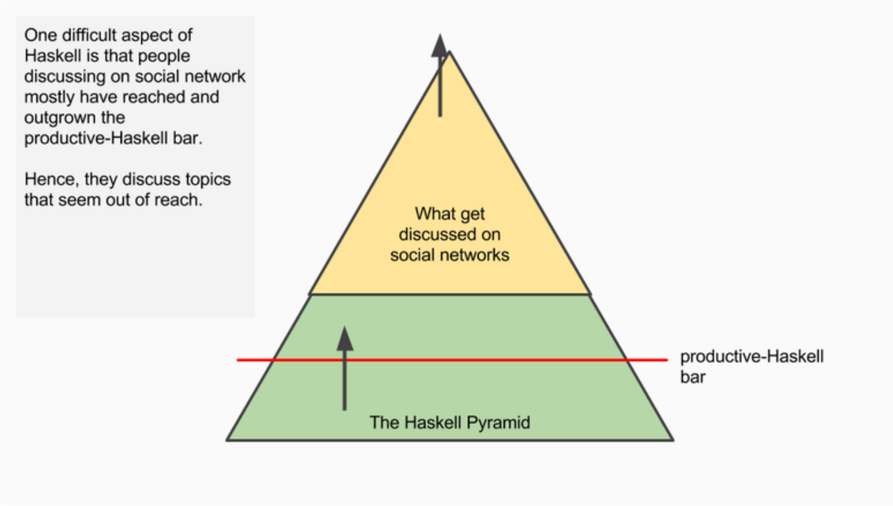
The Languages We Call Haskell © Don Stewart, 2010



So why Haskell?

- Haskell is a practical language. The number of active GitHub repos is only 25x less than in Java (http://githut.info).
- Haskell is easy to read, easy to reason about and easy to refactor. It is not so easy to write, however.
- Haskell offers ample performance for most applications and great options for parrallel/concurrent computations. See L. Petersen, T. A. Anderson, H. Liu, N. Glew, *Measuring the Haskell Gap,* IFL, 2013.
- Haskell encourages advanced type discipline, allowing to specify and check invariants in compile time. *Very important:* types are not about 'do not confuse `string` with `int`', types are propositions.

The Haskell Pyramid © Lucas Di Cioccio, 2017



Rounding is tricky

- There are many modes of rounding: CEILING / FLOOR, UP / DOWN, HALF_UP / HALF_DOWN, HALF_EVEN. Further we assume HALF_EVEN everywhere, which rounds towards the 'nearest neighbor' unless both neighbors are equidistant, in which case, round towards the even neighbor.
- Rounding does not distribute over arithmetic operations:

```
round(x, prec) + round(y, prec) /= round (x + y, prec)

round(1/3, 2) + round(1/3, 2) = 0.66
round(1/3 + 1/3, 2) = 0.67
```

- Roundings with different precisions do not commute:

```
round(round(x, pr1), pr2) /= round(round(x, pr2), pr1)

round(round(1.49, 1), 0) = round(1.50, 0) = 2.00
round(round(1.49, 0), 1) = round(1.00, 1) = 1.00
```

Decimal reinvented

Precision can be treated as a negative exponent.

```
data Decimal = Decimal
  { exp :: Integer, mantissa :: Integer }

instance Show Decimal where
  show d = show (fromInteger (mantissa d) /
                  10 ^ exp d)

foo = Decimal 2 12345 :: Decimal
-- show foo = "123.45"

instance Num Decimal where
  (Decimal exp1 mant1) + (Decimal exp2 mant2) = ?
```

How can we implement arithmetic operations on decimals with mixed exponents?

Addition of Decimals

- Cast both numbers to the highest precision, then add:
  $1.2 + 1.23 = 1.20 + 1.23 = 2.43$
  Decimal 1 12 + Decimal 2 123 = Decimal 2 243
  This is simply unlawful.
- Cast both numbers to the lowest precision, then add:
  $1.2 + 1.23 = 1.2 + 1.2 = 2.4$
  Decimal 1 12 + Decimal 2 123 = Decimal 1 24
  Precision decreases silently. And most of the time this is an error.
- Throw an error, if exponents are not equal. So much pure and strict, but such addition does not fit into type class `Num`.

Decimal re-reinvented

Goal: mismatch of precisions should be ruled out by construction.
Compare

```
data Decimal =
  Decimal { exp :: Integer, mantissa :: Integer }
```

vs.

```
newtype Decimal ( exp :: Nat ) =
  Decimal { mantissa :: Integer }
```

E. g.,

```
foo :: Decimal 2
foo = Decimal 12345
-- means 123.45, previously Decimal 2 12345

bar :: Decimal 3
bar = Decimal 12345
-- means 12.345, previously Decimal 3 12345
```

Why type-level `Nat` matters?

- $(+) :: \text{Decimal} \to \text{Decimal} \to \text{Decimal}$
  This type signature is a proposition: if you give me one Decimal, and then another Decimal, I'll return some other Decimal. Rather weak contract.
- $(+) :: \text{Decimal exp} \to \text{Decimal exp} \to \text{Decimal exp}$
  It says: if you give me two Decimals of equal precisions, I'll return you another Decimal with the same precision. Much better.

Exercises

- $\text{foo} :: \text{Decimal } 2 \to \text{Decimal } 3$
- $\text{bar} :: \text{Decimal } a \to \text{Decimal } 2$
- $\text{baz} :: \text{Decimal } a \to \text{Decimal } b \to \text{Decimal } (a + b)$
- $\text{qux} :: (a + b) \sim 5 \Rightarrow \text{Decimal } a \to \text{Decimal } b \to \text{Decimal } 5$
- $\text{quux} :: (a + b) \sim (c + d) \Rightarrow$
  $(\text{Decimal } a, \text{Decimal } b) \to (\text{Decimal } c, \text{Decimal } d)$

Demotion: from type level to value level

```
exp :: KnownNat exp => Decimal exp -> Integer
exp = natVal

instance KnownNat exp => Show (Decimal exp) where
  show d = show (fromInteger (mantissa d) / 10 ^ exp
      d)
```

We are ready to write Num instance:

```
instance KnownNat exp => Num (Decimal exp) where
  d1 + d2 = Decimal (mantissa d1 + mantissa d2)
  d1 - d2 = Decimal (mantissa d1 - mantissa d2)
  ...
```

Tired of boilerplate?

Actually we would like to write

$(+) = \text{coerce } (+)$

Instance of Num

What is the type of addition?

```
> :t (+)
forall a. Num a => a -> a -> a
```

Apply polymorphic function to a type:

```
instance KnownNat exp => Num (Decimal exp) where
  (+) = coerce ((+) @Integer)
  (-) = coerce ((-) @Integer)
  abs = coerce (abs @Integer)
  signum = coerce (signum @Integer)

  fromInteger m = let dec = Decimal (m * 10 ^ exp dec)
                  in  dec
```

Note the self-recurrent expression!

Exercise: implement multiplication.

Promotion: from value level to type level

```
makeDecimal :: Integer -> Integer -> Decimal ???
makeDecimal exp mantissa = Decimal mantissa
```

We would like to write promote exp instead of ???, but this is impossible in Haskell. Level of types is strictly above level of values.

Solution: hide exponent under a wrapper.

```
data SomeDecimal where
  SomeDecimal :: KnownNat exp =>
    Decimal exp -> SomeDecimal

makeDecimal :: Integer -> Integer -> SomeDecimal
makeDecimal exp mantissa = case someNatVal exp of
  Nothing                          -> error "neg precision"
  Just (SomeNat (_ :: Proxy t)) ->
    SomeDecimal (Decimal n :: Decimal t)
```

Further development

- We can put currency on type-level to prevent adding up UAHs to USDs:

  ```
  newtype Decimal ( exp :: Nat ) ( currency :: Symbol )
    = Decimal { mantissa :: Integer }
  ```

- We can define arithmetic over SomeDecimal, casting arguments to the same precision. E. g.,

```
cast :: (KnownNat a, KnownNat b) => Decimal a -> Decimal b
cast d = let r = Decimal (mantissa d * 10 ^ (exp r - exp d))
   in r

(+) :: SomeDecimal -> SomeDecimal -> SomeDecimal
SomeDecimal (d1::Decimal pr1) + SomeDecimal (d2::Decimal pr2) =
  case (Proxy :: Proxy pr1) `sameNat` (Proxy :: Proxy pr2) of
    Just Refl -> SomeDecimal (d1 + d2)
    Nothing  -> if exp d1 >= exp d2
                then SomeDecimal (cast d1 + d2)
                else SomeDecimal (d1 + cast d2)
```

Performance

For non-dependent type

```
data Decimal = Decimal {exp :: Integer, mantissa :: Integer}
```

function

```
add3 :: Decimal -> Decimal -> Decimal -> Decimal
```

consumes three pointers to data in heap, allocates new struct in heap and returns pointer to it.

For dependent type

```
newtype Decimal (exp :: Nat) = Decimal {mantissa :: Integer}
```

similar function

```
add3 :: KnownNat exp => Decimal exp -> Decimal exp ->
    Decimal exp -> Decimal exp
```

consumes just four Integers and returns one Integer.

Thank you!