# Functional Pearl: Implicit Configurations

Oleg Kiselyov and Chung-chieh Shan (2004)
http://bit.ly/implicit-configurations

As presented by Miikka Koskinen (@arcatan)
2016-09-27 Helsinki Haskell User Group

# The configurations problem

- Programs often have a set of run-time preferences
  - How to make the configuration easy and fast to access where needed?
  - How to not accidentally mix multiple sets of configurations?
  - How to solve this without mutable state? How to solve it in a way that suits Haskell?

# Strawman solution

- Running example: modular arithmetic, where modulus is the configurable value.

```haskell
newtype Modulus a = Modulus a deriving (Eq, Show)
newtype M a = M a deriving (Eq, Show)

unM (M a) = a

add :: Integral a => Modulus a -> M a -> M a -> M a
add (Modulus n) (M a) (M b) = M (mod (a + b) n)

test = unM (add (Modulus 5) (M 2) (M 4))
-- λ> test
-- 1
```

# Strawman solution 2

```haskell
data Conf1
data Conf2

newtype M' s a = M' { unM' :: a } deriving (Eq, Show)

class Modular s a where modulus :: s -> a
instance Modular Conf1 Int where modulus _ = 5
instance Modular Conf2 Int where modulus _ = 7

add' :: forall a s. (Integral a, Modular s a) =>
        M' s a -> M' s a -> M' s a
add' (M' a) (M' b) = M' ((a + b) `mod` (modulus (undefined :: s)))

calculation :: (Integral a, Modular s a) => M' s a
calculation = add' (M' 2) (M' 4)

test1 = unM' (calculation :: M' Conf1 Int)
test2 = unM' (calculation :: M' Conf2 Int)
-- λ> (test1, test2)
-- (1,6)
```

# Local type class instances?

- Type classes are easy to use!
- Can be implemented efficiently!
- The only problem: you can’t define type class instances at runtime? Or can you?

$$\textbf{newtype}\ M\ s\ a = M\ a \quad \textbf{deriving}\ (Eq, Show)$$

$$\textbf{class}\ Modular\ s\ a \mid s \to a\ \textbf{where}\ modulus :: s \to a$$

$$normalize :: (Modular\ s\ a,\ Integral\ a) \Rightarrow a \to M\ s\ a$$
$$normalize\ a :: M\ s\ a = M\ (mod\ a\ (modulus\ (\bot :: s)))$$

$$
\begin{array}{l}
\textbf{instance}\ (Modular\ s\ a,\ Integral\ a) \Rightarrow Num\ (M\ s\ a)\ \textbf{where} \\
\quad \begin{array}{ll}
M\ a + M\ b & = normalize\ (a + b) \\
M\ a - M\ b & = normalize\ (a - b) \\
M\ a \times M\ b & = normalize\ (a \times b) \\
negate\ (M\ a) & = normalize\ (negate\ a) \\
fromInteger\ i & = normalize\ (fromInteger\ i) \\
signum & = error\ \text{“Modular numbers are not signed”} \\
abs & = error\ \text{“Modular numbers are not signed”}
\end{array}
\end{array}
$$

$$test'_3 :: (Modular\ s\ a,\ Integral\ a) \Rightarrow s \to M\ s\ a$$
$$test'_3\ \_ = 3 \times 3 + 5 \times 5$$

# From types to numbers

**data** *Zero*; **data** *Twice s*; **data** *Succ s*; **data** *Pred s*

**class** *ReflectNum s* **where** *reflectNum* :: *Num a* $\Rightarrow s \to a$
**instance** *ReflectNum Zero* **where**
  *reflectNum* _ = 0
**instance** *ReflectNum s* $\Rightarrow$ *ReflectNum* (*Twice s*) **where**
  *reflectNum* _ = *reflectNum* ($\perp$ :: *s*) $\times$ 2
**instance** *ReflectNum s* $\Rightarrow$ *ReflectNum* (*Succ s*) **where**
  *reflectNum* _ = *reflectNum* ($\perp$ :: *s*) + 1
**instance** *ReflectNum s* $\Rightarrow$ *ReflectNum* (*Pred s*) **where**
  *reflectNum* _ = *reflectNum* ($\perp$ :: *s*) $-$ 1

# From numbers to types

$$
\begin{aligned}
&\mathit{reifyIntegral} :: \mathit{Integral}\ a \Rightarrow \\
&\qquad\qquad a \to (\forall s.\ \mathit{ReflectNum}\ s \Rightarrow s \to w) \to w \\
&\mathit{reifyIntegral}\ i\ k = \mathbf{case}\ \mathit{quotRem}\ i\ 2\ \mathbf{of} \\
&\quad (0, \ \ 0) \to k\ (\bot :: \mathit{Zero}) \\
&\quad (j, \ \ 0) \to \mathit{reifyIntegral}\ j\ (\lambda(\_ :: s) \to k\ (\bot :: \mathit{Twice}\ s)) \\
&\quad (j, \ \ 1) \to \mathit{reifyIntegral}\ j\ (\lambda(\_ :: s) \to k\ (\bot :: \mathit{Succ}\ (\mathit{Twice}\ s))) \\
&\quad (j, -1) \to \mathit{reifyIntegral}\ j\ (\lambda(\_ :: s) \to k\ (\bot :: \mathit{Pred}\ (\mathit{Twice}\ s)))
\end{aligned}
$$

$$
\begin{array}{l}
\textbf{data}\ \mathit{ModulusNum}\ s\ a\\[4pt]
\textbf{instance}\ (\mathit{ReflectNum}\ s,\ \mathit{Num}\ a) \Rightarrow\\
\qquad\qquad\quad \mathit{Modular}\ (\mathit{ModulusNum}\ s\ a)\ a\ \textbf{where}\\
\quad \mathit{modulus}\ \_ = \mathit{reflectNum}\ (\bot :: s)
\end{array}
$$

$$
\begin{array}{l}
\mathit{withIntegralModulus} :: \mathit{Integral}\ a \Rightarrow\\
\qquad\qquad\qquad\qquad\quad a \to (\forall s.\ \mathit{Modular}\ s\ a \Rightarrow s \to w) \to w\\
\mathit{withIntegralModulus}\ i\ k =\\
\quad \mathit{reifyIntegral}\ i\ (\lambda(\_ :: s) \to k\ (\bot :: \mathit{ModulusNum}\ s\ a))
\end{array}
$$

$$
\begin{array}{l}
\mathit{test}_3' :: (\mathit{Modular}\ s\ a,\ \mathit{Integral}\ a) \Rightarrow s \to M\ s\ a\\
\mathit{test}_3'\ \_ = 3 \times 3 + 5 \times 5\\[4pt]
\mathit{test}_3 = \mathit{withIntegralModulus}\ 4\ (\mathit{unM} \circ \mathit{test}_3')
\end{array}
$$

# Now let's run with it

- Reify lists of integers to type-level the same way.
- Now you can store lists of bytes. Storable marshals some values to lists of bytes.
- If you have an arbitrary value, you can get a StablePtr to it. StablePtr is Storable.
  - StablePtr: a reference to an expression that is guaranteed to be not GC'd
  - This means you can reify and reflect *anything*.

# You can use this

- Available as a library called *reflection*:
  http://hackage.haskell.org/package/reflection
  - Efficient implementation by Edward Kmett
  - 100% magical – see *Reflecting values to types* by Austin Sepp http://bit.ly/using-reflection

# Read the paper for:

- Details!
- Phantom types
- Run-time dispatch for fast performance
- Alternative solutions
- http://bit.ly/implicit-configurations

Photo: Kenneth Dellaquila. CC-BY-NC. Cropped. https://flic.kr/p/aHdJWM

# Questions?

ps. you should follow me on twitter, my handle is @arcatan