Functional Pearl: Implicit Configurations —or, Type Classes Reflect the Values of Types

Oleg Kiselyov and Chung-chieh Shan

22 September 2004

1 Three sides of the same coin?

Local type-class instances

Dependent types

Implicit parameters

1 Three sides of the same coin?

Local type-class instances

```
let instance Show (IO a)     where show _ = "<action>"
    instance Show (a -> b)   where show _ = "<function>"
in show (getChar, putChar)
```

But then principal types are broken.

```
f a = let instance Show (IO a)     where show _ = "<action>"
          instance Show (a -> b)   where show _ = "<function>"
      in show a
f :: IO a -> String
f :: (a -> b) -> String
```

We propose a restriction that seems to recover principal types.

Dependent types

Implicit parameters

1 Three sides of the same coin?

Local type-class instances

Dependent types

```
data Vector :: Nat -> * -> * where
  Nil  :: Vector Zero a
  Cons :: a -> Vector n a -> Vector (Succ n) a
```

But Haskell doesn't have dependent types.

We further the Haskell tradition of emulating dependent types using type classes.

Implicit parameters

1 Three sides of the same coin?

Local type-class instances

Dependent types

Implicit parameters

```
pretty :: (?width :: Int) => Doc -> String
pretty = ... f ...
f      = ... g ...
g      = ... if i >= ?width then ...
```

But pitfalls abound in the presence of multiple implicit parameters.

We avoid these pitfalls by relying on the coherence of type classes.

The configurations problem

—is to propagate run-time preferences throughout a program, allowing multiple concurrent configuration sets to coexist safely under statically guaranteed separation.

The configurations problem

—is to propagate run-time preferences throughout a program, allowing multiple concurrent configuration sets to coexist safely under statically guaranteed separation.

Applications:

- Page width for pretty printing
- Modulus in modular arithmetic
- Numerical tolerance, rounding mode
- Currency conversion rate, time zone
- Connection handle in external API
- First-class implicit modules

Desiderata:

- Keep out of the way: do not change code that does not use parameters.
- Parameters should be strongly typed and fast to access, like lexical variables.
- Configuration data may become known only at run time.
- Propagate multiple parameters together.
- Multiple configurations should coexist with static separation.
- The solution should be available today.

The configurations problem

—is to propagate run-time preferences throughout a program, allowing multiple concurrent configuration sets to coexist safely under statically guaranteed separation.

Applications:

- Page width for pretty printing
- Modulus in modular arithmetic
- Numerical tolerance, rounding mode
- Currency conversion rate, time zone
- Connection handle in external API
- First-class implicit modules

Desiderata:

- Keep out of the way: do not change code that does not use parameters.
- Parameters should be strongly typed and fast to access, like lexical variables.
- Configuration data may become known only at run time.
- Propagate multiple parameters together.
- Multiple configurations should coexist with static separation.
- The solution should be available today.

2 Motivating example: modular arithmetic

$$3 \times 3 + 5 \times 5 \;=\; 9 + 25 \;=\; 1 + 1 \;=\; 2 \pmod 4$$

Getting started

```
add :: Integral a => a -> a -> a -> a
mul :: Integral a => a -> a -> a -> a

add m a b = mod (a + b) m
mul m a b = mod (a * b) m
```

Getting started

```
add :: Integral a => a -> a -> a -> a
mul :: Integral a => a -> a -> a -> a

add m a b = mod (a + b) m
mul m a b = mod (a * b) m
```

```
▶ add 4 (mul 4 3 3) (mul 4 5 5)
2
▶ add 4 (mul 4 3 3) (mul 5 5 5)
1
```

Phantom types for parameter threads (cf. runST)

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

add :: Integral a => Modulus s a -> M s a -> M s a -> M s a
mul :: Integral a => Modulus s a -> M s a -> M s a -> M s a

add (Modulus m) (M a) (M b) = M (mod (a + b) m)
mul (Modulus m) (M a) (M b) = M (mod (a * b) m)
```

Phantom types for parameter threads (cf. runST)

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

```
add :: Integral a => Modulus s a -> M s a -> M s a -> M s a
mul :: Integral a => Modulus s a -> M s a -> M s a -> M s a
```

```
add (Modulus m) (M a) (M b) = M (mod (a + b) m)
mul (Modulus m) (M a) (M b) = M (mod (a * b) m)
```

Use rank-2 polymorphism to enforce parameter thread like state threads.

```
withModulus :: a -> (forall s. Modulus s a -> w) -> w
withModulus m k = k (Modulus m)
```

Phantom types for parameter threads (cf. runST)

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

```
add :: Integral a => Modulus s a -> M s a -> M s a -> M s a
mul :: Integral a => Modulus s a -> M s a -> M s a -> M s a
```

```
add (Modulus m) (M a) (M b) = M (mod (a + b) m)
mul (Modulus m) (M a) (M b) = M (mod (a * b) m)
```

Use rank-2 polymorphism to enforce parameter thread like state threads.

```
withModulus :: a -> (forall s. Modulus s a -> w) -> w
withModulus m k = k (Modulus m)
```

```
▶ withModulus 4 (\m ->
    unM $ add m (mul m (M 3) (M 3)) (mul m (M 5) (M 5)))
2
▶ withModulus 4 (\m -> withModulus 5 (\m' ->
    unM $ add m (mul m (M 3) (M 3)) (mul m' (M 5) (M 5))))
Inferred type is less polymorphic than expected
Quantified type variable s escapes
```

This static separation is not present in OpenSSL.

Multiple configurations

Multiple moduli can coexist in the same scope, as long as they stay separate!

```
▶ withModulus 4 (\m -> withModulus 5 (\m' ->
    (unM (mul m (M 3) (M 3)), unM (mul m' (M 5) (M 5)))))
(1,0)
```

We want the phantom type to not just check threads but also carry threads ...

Type classes for modulus passing

```
class Modular s a | s -> a where modulus :: s -> a
newtype M s a        = M {unM::a} deriving (Eq, Show)

add :: (Integral a, Modular s a) => M s a -> M s a -> M s a
mul :: (Integral a, Modular s a) => M s a -> M s a -> M s a

add (M a) (M b) :: M s a = M (mod (a + b) (modulus (⊥ :: s)))
mul (M a) (M b) :: M s a = M (mod (a * b) (modulus (⊥ :: s)))
```

Carry (piggyback) the modulus on the phantom type s.

Modular arithmetic in the Num class

```
class Modular s a | s -> a where modulus :: s -> a
newtype M s a       = M {unM::a} deriving (Eq, Show)

instance (Integral a, Modular s a) => Num (M s a) where
  M a + M b = M (mod (a + b) (modulus (⊥ :: s)))
  M a * M b = M (mod (a * b) (modulus (⊥ :: s)))
            ⋮
```

Modular arithmetic in the `Num` class

```
class Modular s a | s -> a where modulus :: s -> a
newtype M s a        = M {unM::a} deriving (Eq, Show)

instance (Integral a, Modular s a) => Num (M s a) where
  M a + M b = M (mod (a + b) (modulus (⊥ :: s)))
  M a * M b = M (mod (a * b) (modulus (⊥ :: s)))
            ⋮

data Four
instance Modular Four Int where modulus _ = 4
```

```
▶ M 3 * M 3 + M 5 * M 5 :: M Four Int
M 2
▶ 3 * 3 + 5 * 5 :: M Four Int
M 2
```

Modular arithmetic in the `Num` class

```
class Modular s a | s -> a where modulus :: s -> a
newtype M s a         = M {unM::a} deriving (Eq, Show)

instance (Integral a, Modular s a) => Num (M s a) where
  M a + M b = M (mod (a + b) (modulus (⊥ :: s)))
  M a * M b = M (mod (a * b) (modulus (⊥ :: s)))
            ⋮

data Four
instance Modular Four Int where modulus _ = 4
```

```
▶ M 3 * M 3 + M 5 * M 5 :: M Four Int
M 2
▶ 3 * 3 + 5 * 5 :: M Four Int
M 2
```

But what about run-time moduli?

```
withModulus :: a -> (forall s. Modular s a => s -> w) -> w
withModulus m k = k (⊥ :: ???)
```


terms :: types

```haskell
withModulus :: a -> (forall s. Modular s a => s -> w) -> w
withModulus m k = k (⊥ :: ???)
```

```
withModulus :: a -> (forall s. Modular s a => s -> w) -> w
withModulus m k = k (⊥ :: ???)
```

Using **local instances**, we'd say something like . . .

```
withModulus (m::a) k = let instance Modular () a where
                             modulus _ = m
                       in k (⊥ :: ())
```

```
withModulus :: a -> (forall s. Modular s a => s -> w) -> w
withModulus m k = k (⊥ :: ???)
```

Using **local instances**, we'd say something like ...

```
withModulus (m::a) k = let instance Modular () a where
                             modulus _ = m
                       in k (⊥ :: ())
```

... but for one thing, local instances break principal types.

Using **dependent types**, we'd say something like ...

```
instance Modular (m::a) a where modulus _ = m
withModulus m k = k (⊥ :: m)
```

```
withModulus :: a -> (forall s. Modular s a => s -> w) -> w
withModulus m k = k (⊥ :: ???)
```

Using **local instances**, we'd say something like . . .

```
withModulus (m::a) k = let instance Modular () a where
                             modulus _ = m
                       in k (⊥ :: ())
```

. . . but for one thing, local instances break principal types.

Using **dependent types**, we'd say something like . . .

```
instance Modular (m::a) a where modulus _ = m
withModulus m k = k (⊥ :: m)
```

. . . but for one thing, Haskell doesn't have dependent types.

Using **implicit parameters**, we'd say something like . . .

```
withModulus :: a -> (?modulus :: a => w) -> w
withModulus m k = let ?modulus = m in k
```

```
withModulus :: a -> (forall s. Modular s a => s -> w) -> w
withModulus m k = k (⊥ :: ???)
```

Using **local instances**, we'd say something like . . .

```
withModulus (m::a) k = let instance Modular () a where
                           modulus _ = m
                       in k (⊥ :: ())
```

. . . but for one thing, local instances break principal types.

Using **dependent types**, we'd say something like . . .

```
instance Modular (m::a) a where modulus _ = m
withModulus m k = k (⊥ :: m)
```

. . . but for one thing, Haskell doesn't have dependent types.

Using **implicit parameters**, we'd say something like . . .

```
withModulus :: a -> (?modulus :: a => w) -> w
withModulus m k = let ?modulus = m in k
```

. . . but for one thing, a `Num` instance cannot rely on implicit parameters.

Instead, we roll up our sleeves . . .

3 An infinite family of phantom types

See Dylan Thurston's messages to the Haskell mailing list (2001).

```
withModulus :: a -> (forall s. Modular s a => s -> w) -> w
withModulus m k = k (⊥ :: ???)
```

3 An infinite family of phantom types

See Dylan Thurston's messages to the Haskell mailing list (2001).

```
withModulus :: a -> (forall s. Modular s a => s -> w) -> w
withModulus m k = k (⊥ :: Succ (Succ (Succ (Succ Zero))))
```

3 An infinite family of phantom types

See Dylan Thurston's messages to the Haskell mailing list (2001).

```
withModulus :: a -> (forall s. Modular s a => s -> w) -> w
withModulus m k = k (⊥ :: Twice (Twice (Succ Zero)))
```

3 An infinite family of phantom types

See Dylan Thurston's messages to the Haskell mailing list (2001).

```
withModulus :: a -> (forall s. Modular s a => s -> w) -> w
withModulus m k = k (⊥ :: Twice (Twice (Succ Zero)))
```

Introduce a family of types: recipes for building a value.

```
data Zero
data Twice s
data Succ s
data Pred s
```

Type classes reflect the values of types

```
class                        ReflectNum s where
                             reflectNum :: Num a => s -> a

instance                     ReflectNum Zero where
                             reflectNum _ = 0

instance ReflectNum s => ReflectNum (Twice s) where
                             reflectNum _ = reflectNum (⊥::s) * 2

instance ReflectNum s => ReflectNum (Succ s) where
                             reflectNum _ = reflectNum (⊥::s) + 1

instance ReflectNum s => ReflectNum (Pred s) where
                             reflectNum _ = reflectNum (⊥::s) - 1
```

`ReflectNum` is a parser, or an interpreter.
It turns a static formula into a dynamic value.

```
▶ reflectNum (⊥ :: Twice (Twice (Succ Zero)))
4
```

Polymorphic recursion reifies the types of values

```
reifyIntegral :: Integral a =>
  a -> (forall s. ReflectNum s => s -> w) -> w

reifyIntegral i k = case quotRem i 2 of
  (0, 0) -> k (⊥ :: Zero)
  (j, 0) -> reifyIntegral j (\_::s -> k (⊥ :: Twice s))
  (j, 1) -> reifyIntegral j (\_::s -> k (⊥ :: Succ (Twice s)))
  (j,-1) -> reifyIntegral j (\_::s -> k (⊥ :: Pred (Twice s)))
```

`reifyIntegral` is an unparser, or an uninterpreter.
It turns a dynamic value into a static formula.

```
▶ reifyIntegral (-20) reflectNum
-20
```

Tying it all together

```
data ModulusNum s a
instance (ReflectNum s, Num a) => Modular (ModulusNum s a) a where
                                  modulus _ = reflectNum (⊥::s)

withModulus :: Integral a =>
                 a -> (forall s. Modular s a => s -> w) -> w
withModulus i k = reifyIntegral i
                    (\_::s -> k (⊥ :: ModulusNum s a))
```

Tying it all together

```
data ModulusNum s a
instance (ReflectNum s, Num a) => Modular (ModulusNum s a) a where
                                  modulus _ = reflectNum (⊥::s)

withModulus :: Integral a =>
                 a -> (forall s. Modular s a => s -> w) -> w
withModulus i k = reifyIntegral i
                    (\_::s -> k (⊥ :: ModulusNum s a))
```

```
▶ withModulus 4 (\_::s -> unM (3 * 3 + 5 * 5 :: M s Int))
2
```

Yay!

"Type classes reflect the values of types":

Types cannot depend on terms in Haskell. Nevertheless, types *can* depend on *types that reflect associated terms*!

4 Interim summary

Dylan Thurston's messages to the Haskell mailing list (2001):

- ✓ Modular numbers belong in the `Num` class, just like regular numbers.
- ✓ Pass implicit dictionary arguments whose types belong to the `Integral` class.

Our contributions:

- ▶ Pass implicit dictionary arguments whose types belong to the `Storable` class.
- Pass implicit dictionary arguments of any type.
- Solve the configurations problem, and survey existing solutions.
- Achieve more intuitive notation and faster run-time performance.
- Inform language design: recover principal types for local type-class instances and coherence for implicit parameters.

5 Beauty is in the eye of the beholder

```
...

Thank you very much for submitting to ICFP. I am sorry to have to
disappoint you.

Best wishes,

Kathleen Fisher
ICFP 2004 Program Chair

*=---=*---=*---=*---=*---=*---=*---=*---=*---=*---=*---=*---=*---=*---=*---=*=

First reviewer's review:

          >>> Summary of the submission <<<

The paper show how you can emulate locally defined instances in Haskell by
using polymorphic recursion and an ingenious (or ugly, depending on your point
of view) trick.

...
```

The ingenious/ugly trick

- Pass implicit dictionary arguments whose types belong to the `Storable` class.

  Marshall into bytes, pass them as a list of numbers, then unmarshall.

- Pass implicit dictionary arguments of any type.

  Arbitrary types must be passed *by reference*, not by value:
  `StablePtr` in the foreign function interface is `Storable`.

$$
\begin{tikzcd}
\text{Arbitrary type} \arrow[r, bend left, "\text{New } \texttt{StablePtr}"] & \texttt{Storable}\text{ type} \arrow[l, bend left, "\text{Free } \texttt{StablePtr}"] \arrow[r, bend left, "\text{Marshall into bytes}"] & \texttt{Integral}\text{ type} \arrow[l, bend left, "\text{Unmarshall from bytes}"]
\end{tikzcd}
$$

Even polymorphic functions and abstract data types!

6 Back to the configurations problem

The key idea:

> Piggyback values onto types. Propagate configurations by unification.

Applications:

• Page width for pretty printing	`Doc s`
• Modulus in modular arithmetic	`M s Int`
• Numerical tolerance, rounding mode	`Double s`
• Currency conversion rate, time zone	`Money s; Time s`
• Connection handle in external API	`Jobject s; Jclass s; JString s`
• First-class implicit modules	`...`

Desiderata achieved, thanks to Haskell's advanced type system!

Flexible propagation for intuitive notation

Piggyback wisely.

Rather than

```
  withModulus :: a -> (forall s. Modular s a => s -> w) -> w
▶ withModulus 4 (\_::s -> unM (3 * 3 + 5 * 5 :: M s Int))
2
```

say instead

```
  withModulus :: a -> (forall s. Modular s a => M s w) -> w
▶ withModulus 4 (3 * 3 + 5 * 5)
2
```

(which is not `mod (3 * 3 + 5 * 5) 4`).

The direction of type unification and overloading resolution is independent of the direction of argument passing!

Run-time dispatch for fast performance

Dictionaries can contain not just numerical constants but also functions.

Specialize for special configuration data:

- Powers of 2; composite moduli (*residue number system*)
- Connection handles to local host
- …

Different representations and implementations are assured static separation!

OpenSSL implements some specializations unsafely, and skips others nervously.

7 Informing language design

Local type-class instances

Roughly, require each local instance head to mention an eigenvariable at that scope.

```
withModulus (m::a) k = let instance Modular () a where
                               modulus _ = m
                           in k (⊥ :: ())
```

```
data Any = forall s. Any s
withModulus (m::a) k = let Any (_::s) = Any ()
                           instance Modular s a where
                             modulus _ = m
                           in k (⊥ :: s)
```

Intuition 1: avoid overlapping instances, to ensure coherence.

Intuition 2: restore principle types by disallowing problematic examples.

```
f a = let instance Show (IO a)   where show _ = "<action>"
          instance Show (a -> b) where show _ = "<function>"
      in show a
```

Dependent types

Instead of compile-time terms, we use run-time types.

Dependent types

Instead of compile-time terms, we use run-time types.

Implicit parameters

Beware multiple implicit parameters *of the same (name and) type*:

1. Inlining becomes unsound.
2. Type signatures can affect term behavior.
3. The monomorphism restriction blocks generalizing over implicit parameters.
4. Implicit parameter constraints cannot appear in a class or instance context.

(The monomorphism restriction is not to blame.)

Hence, prohibit such “overlapping” multiple implicit parameters
—we end up with type classes again!

Intuition 1: avoid overlapping instances, to ensure coherence.

Intuition 2: carry configuration through static types, not dynamic scopes.

8 Conclusion

A new, satisfactory solution to the configurations problem

- higher start-up cost
- multiple configurations
- intuitive notation
- supports run-time specialization
- local type-class instances