The Power Of Const

Markus Hauck (@markus1189)

codecentric

Why This Talk

- why a whole talk about `Const`?
- one of the “funny” datatypes at first sight
- not immediately obvious what it is good for
- surprisingly, it is **many** useful usages
- this talk shows (some of) them

```haskell
newtype Const a b = Const { getConst :: a }
```

The Const Datatype

```haskell
newtype Const a b = Const { getConst :: a }
```

- two type parameters a and b
- b is a phantom type
- you can only ever get a value of type a out of it
- type level version of the `const` **function** that ignores one of the two arguments:

```haskell
const :: a -> b -> b
const x _ = x
```

The Rest Of This Talk

- `Functor` instance
- `Applicative` instance
- `Selective Applicative` instance

Functor

The Functor Instance

```haskell
instance Functor (Const a) where
  fmap f c@(Const a) = _
```

```
• Found hole: _ :: Const a b
...
• Relevant bindings include
    f :: a1 -> b
    c :: Const a a1
    a :: a
```

The Functor Instance

```haskell
instance Functor (Const a) where
  fmap _ (Const a) = Const a
```

```
ghci> c = Const @String @Int "can't touch me"

ghci> fmap (+1) c
Const "can't touch me"

ghci> fmap print c
Const "can't touch me"
```

- unpack and retag (change the phantom type)
- **discards** the function application
- ... but how is this useful?

Use of Const in Twan van Laarhoven Lenses

```haskell
type Lens a b = forall f. Functor f => (b -> f b) -> a -> f a
_1 :: Lens (a, b) a
_1 f (x1, y) = fmap (\x2 -> (x2, y)) (f x1)
```

- think: ’**a**’ somehow contains ’**b**’
- given:
  - functorial function modifying the **b**
  - and a “bigger” value of type **a**
- produce new **a** that has the modified **b** “inside”

Implementing Getter With Const

```
type Lens a b = forall f. Functor f => (b -> f b) -> a -> f a
_1 :: Lens (a, b) a
_1 f (x1, y) = fmap (\x2 -> (x2, y)) (f x1)

get :: Lens a b -> a -> b
get l x = getConst (l Const x)
```

```
  get _1 (42, 'a')
= getConst (_1 Const (42, 'a'))                  -- Def. of 'get'
= getConst (fmap (\x2 -> (x2, 'a')) (Const 42)) -- Def. of '_1'
= getConst (Const 42)                            -- Def. of 'fmap' for 'Const
= 42                                             -- The Answer!
```

Functor

- first example: Functor for `Const`
- useless at first glance
- nice for lenses

Applicative

The Applicative Instance

- probably one of my favorites
- opens up a lot of possibilities
- how? connects two very important concepts

The Applicative Instance

```
instance Applicative (Const m) where
  pure :: a -> Const m a
  pure = _pure

  (<*>) :: Const m (a -> b) -> Const m a -> Const m b
  Const f <*> Const v = _ap
```

The Applicative Instance

```
instance Monoid m => Applicative (Const m) where
  pure :: a -> Const m a
  pure _ = Const mempty

  (<*>) :: Const m (a -> b) -> Const m a -> Const m b
  Const f <*> Const v = Const (f <> v)
```

Using The Applicative Instance

- but what does it **mean**?
- we can use any function working with `Applicatives` and give them `Monoids`!

Using The Applicative Instance

```
traverse :: ... => (a -> Const m b) -> t a -> Const m (t b)
foldMap  :: ... => (a ->       m)   -> t a ->       m
```

❶

- by using `traverse` with `Const` we get `foldMap`
- that's why `Traversable` is enough to define a `Foldable` instance

Using The Applicative Instance

```
traverse :: ... => (a -> Const m b) -> t a -> Const m (t b)
foldMap  :: ... => (a ->       m)   -> t a ->       m
```

❶
- by using `traverse` with `Const` we get `foldMap`
- that's why `Traversable` is enough to define a `Foldable` instance

❷
- Use `Const` to statically analyze `Free Applicative` programs
- to accumulate monoidal value without performing actual effects

Using The Applicative Instance

```
traverse :: ... => (a -> Const m b) -> t a -> Const m (t b)
foldMap  :: ... => (a ->        m)   -> t a ->        m
```

1.
   - by using `traverse` with `Const` we get `foldMap`
   - that's why `Traversable` is enough to define a `Foldable` instance
2.
   - Use `Const` to statically analyze `Free Applicative` programs
   - to accumulate monoidal value without performing actual effects
3.
   - `Const` highlights the relation between Applicative and Monoid
   - use **everything** from Monoids and use with functions that require `Applicative`
   - Applicative laws state that instances have to be monoidal in their effects

Selective Applicative Functors

Selective Applicative Functors

ANDREY MOKHOV, Newcastle University, United Kingdom
GEORGY LUKYANOV, Newcastle University, United Kingdom
SIMON MARLOW, Facebook, United Kingdom
JEREMIE DIMINO, Jane Street, United Kingdom

90

Applicative functors and monads have conquered the world of functional programming by providing general and powerful ways of describing effectful computations using pure functions. Applicative functors provide a way to compose *independent effects* that cannot depend on values produced by earlier computations, and all of which are declared statically. Monads extend the applicative interface by making it possible to compose *dependent effects*, where the value computed by one effect determines all subsequent effects, dynamically.

This paper introduces an intermediate abstraction called *selective applicative functors* that requires all effects to be declared statically, but provides a way to select which of the effects to execute dynamically. We demonstrate applications of the new abstraction on several examples, including two industrial case studies.

CCS Concepts: • **Software and its engineering**; • **Mathematics of computing**;

Additional Key Words and Phrases: applicative functors, selective functors, monads, effects

ACM Reference Format:
Andrey Mokhov, Georgy Lukyanov, Simon Marlow, and Jeremie Dimino. 2019. Selective Applicative Functors. *Proc. ACM Program. Lang.* 3, ICFP, Article 90 (August 2019), 29 pages. https://doi.org/10.1145/3341694

https://www.staff.ncl.ac.uk/andrey.mokhov/selective-functors.pdf

This paper introduces an intermediate abstraction called selective applicative functors that requires all effects to be ***declared statically****, but provides a way to select which of the effects to* ***execute dynamically****.*

- (emphasis mine)
- `Applicative`: effects declared & executed statically
- offer some of the benefits of Arrows, less powerful (not this talk :/)

Selective Functors

```haskell
class Applicative f => Selective f where
  select :: f (Either a b) -> f (a -> b) -> f b
```

- (comes with some additional laws not shown here)
- what does it buy me?
- you can branch on **Bool**s that are inside an "effect"[1]

```haskell
ifS :: Selective f => f Bool -> f a -> f a -> f a
```

[1] "effect" with the usual caveats

The Selective Instance

- now the interesting part: how does the instance for `Const` work?
- with `Selective` we have two valid instances

```haskell
newtype Over m a = Over { getOver :: m }

newtype Under m a = Under { getUnder :: m }
```

- `Over` for static **over**-approximation
- `Under` for static **under**-approximation

```haskell
-- Reminder:
class Applicative f => Selective f where
  select :: f (Either a b) -> f (a -> b) -> f b

-- The two instances:
instance Monoid a => Selective (Over a) where
  select (Over (Const c)) (Over (Const t)) = Over (Const (c <> t))

instance Monoid a => Selective (Under a) where
  select (Under (Const c)) _ = Under (Const c)
```

- **Over** also goes into conditional branches
- **Under** ignores conditionally executed parts

The Selective Instance

- why is this so cool?
- DSL using **`FreeSelective`** can express conditional branching and do static analysis/transformation on programs
    - Build systems à la carte (Mokhov, Andrey, Neil Mitchell, and Simon Peyton Jones.)
- using `Over` and `Under` extremely useful to e.g. check properties of a program
    - does a certain effect **always** occur? Use `Under`
    - does a certain effect **never** occur: Use `Over`

Conclusion

Conclusion

```haskell
newtype Const a b = Const { getConst :: a }
```

- nonsense at first, but very useful
- many instances that are very useful:
  - Functor
  - Applicative
  - Selective Applicative
  - Monad? (exercise)
- good tool to understand monoidal aspects of Applicatives

Thanks!

What Is Next?

- we saw the instance for `Applicative`
- important bridge between `Monoid` and `Applicative`
- so what about the `Monad` instance?

The Monad Instance?

```haskell
import Data.Functor.Const

instance Monoid a => Monad (Const a) where
  c@(Const x) >>= f = f _
```

```
const-monad.hs:4:23: error:
    • Found hole: _ :: Const a b
...
    • Relevant bindings include
        f :: a1 -> Const a b (bound at const-monad.hs:4:19)
        x :: a (bound at const-monad.hs:4:12)
        c :: Const a a1 (bound at const-monad.hs:4:3)
...
```

- `mempty` would typecheck, but violate the left-identity law
- we simply can't "pretend" anymore
- `Const` does not have a `Monad` instance, but still manages to teach us
- somewhere between `Applicative` and `Monad`...