

Singletons and You

Justin Le <https://blog.jle.im> (justin@je.im)

Lambdaconf 2017, May 27, 2017

Preface

Slide available at <https://talks.jle.im/lambdaconf-2017/singletons/singleton-slides.html>.

Preface

Slide available at <https://talks.jle.im/lambdaconf-2017/singletons/singleton-slides.html>.

GHC extensions (potentially) used:

```
{-# LANGUAGE GADTs                #-}  
{-# LANGUAGE KindSignatures       #-}  
{-# LANGUAGE LambdaCase          #-}  
{-# LANGUAGE RankNTypes          #-}  
{-# LANGUAGE RankNTypes          #-}  
{-# LANGUAGE ScopedTypeVariables #-}  
{-# LANGUAGE TemplateHaskell      #-}  
{-# LANGUAGE TypeFamilies         #-}  
{-# LANGUAGE TypeInType           #-}  
{-# LANGUAGE TypeOperators        #-}  
{-# LANGUAGE UndecidableInstances #-}
```

```
import Data.Kind           -- to get type Type = *  
import Data.Singletons
```

Safety with Phantom Types

```
data DoorState = Opened | Closed | Locked
  deriving (Show, Eq)
```

```
data Door (s :: DoorState) = UnsafeMkDoor
```

```
-- alternatively
```

```
data Door :: DoorState -> Type where
  UnsafeMkDoor :: Door s
```

Other similar examples

- ▶ State machines (socket connections, file handles, opened/closed)
- ▶ Refinement types
- ▶ “Tagged” types (santized/unsantized strings)

Phantom types in action

```
closeDoor :: Door 'Opened -> Door 'Closed  
closeDoor UnsafeMkDoor = UnsafeMkDoor
```

Phantom types in action

```
closeDoor :: Door 'Opened -> Door 'Closed
closeDoor UnsafeMkDoor = UnsafeMkDoor

openDoor  :: Door 'Closed -> Door 'Opened
openDoor  UnsafeMkDoor = UnsafeMkDoor
```

Phantom types in action

```
doorStatus :: Door s -> DoorState  
doorStatus = -- ????
```

We have a problem.

Phantom types in action

```
doorStatus :: Door s -> DoorState  
doorStatus = -- ????
```

We have a problem.

```
doorStatus :: Door s -> DoorState  
doorStatus UnsafeMkDoor = -- s ???
```

More Problems

```
initializeDoor :: DoorStatus -> Door s
initializeDoor = \case
    Opened -> UnsafeMkDoor
    Closed -> UnsafeMkDoor
    Locked -> UnsafeMkDoor
```

More Problems

```
initializeDoor :: DoorStatus -> Door s
initializeDoor = \case
    Opened -> UnsafeMkDoor
    Closed -> UnsafeMkDoor
    Locked -> UnsafeMkDoor
```

Neat, but does this work?

More Problems

```
ghci> :t initializeDoor Opened :: Door 'Closed
initializeDoor Opened :: Door 'Closed
```

Oops.

The Fundamental Issue in Haskell

- ▶ In Haskell, types only exist at *compile-time*. They are **erased** at runtime.

The Fundamental Issue in Haskell

- ▶ In Haskell, types only exist at *compile-time*. They are **erased** at runtime.
- ▶ This is a good thing for performance! Types incur no runtime overhead!

The Fundamental Issue in Haskell

- ▶ In Haskell, types only exist at *compile-time*. They are **erased** at runtime.
- ▶ This is a good thing for performance! Types incur no runtime overhead!
- ▶ But it makes functions like `doorStatus` fundamentally unwritable without fancy typeclasses.

The Fundamental Issue in Haskell

- ▶ In Haskell, types only exist at *compile-time*. They are **erased** at runtime.
- ▶ This is a good thing for performance! Types incur no runtime overhead!
- ▶ But it makes functions like `doorStatus` fundamentally unwritable without fancy typeclasses.
- ▶ ... or does it?

The Singleton Pattern

```
data SingDS :: DoorStatus -> Type where  
    SOpened  :: SingDS 'Opened  
    SClosed  :: SingDS 'Closed  
    SLocked  :: SingDS 'Locked
```

Creates three constructors:

```
SOpened  :: SingDS 'Opened  
SClosed  :: SingDS 'Closed  
SLocked  :: SingDS 'Locked
```

The Singleton Pattern

- ▶ A **singleton** is a type that has exactly one inhabited value.

The Singleton Pattern

- ▶ A **singleton** is a type that has exactly one inhabited value.
- ▶ There is only one value of type `SingDS 'Opened`, and only one value of type `SingDS 'Closed`.

The Singleton Pattern

- ▶ A **singleton** is a type that has exactly one inhabited value.
- ▶ There is only one value of type `SingDS 'Opened`, and only one value of type `SingDS 'Closed`.
- ▶ The constructor that a `SingDS s` uses reveals to us what `s` is.

The Singleton Pattern

With our new singletons, we can essentially **pattern match** on types:

```
showSingDS :: SingDS s -> String
showSingDS = \case
    SOpened -> "Opened"
    SClosed -> "Closed"
    SLocked -> "Locked"
```

The Singleton Pattern

With our new singletons, we can essentially **pattern match** on types:

```
showSingDS :: SingDS s -> String
showSingDS = \case
    SOpened -> "Opened"
    SClosed -> "Closed"
    SLocked -> "Locked"
```

Alone like this, it's a bit boring. We didn't need GADTs for this.

Door Status

```
doorStatus' :: SingDS s -> Door s -> DoorState
doorStatus' = \case
  SOpened -> \_ -> "Door is opened"
  SClosed -> \_ -> "Door is closed"
  SLocked -> \_ -> "Door is locked"
```

- ▶ GADT-ness allows us to enforce that the `s` in `SingDS s` is the same as the `s` in our `Door`.

Door Status

```
doorStatus' :: SingDS s -> Door s -> DoorState
doorStatus' = \case
    SOpened -> \_ -> "Door is opened"
    SClosed -> \_ -> "Door is closed"
    SLocked -> \_ -> "Door is locked"
```

- ▶ GADT-ness allows us to enforce that the `s` in `SingDS s` is the same as the `s` in our `Door`.
- ▶ Singleton property means that `SingDS s` has a one-to-one correspondence with its constructors.

Door Status

```
doorStatus' :: SingDS s -> Door s -> DoorState
doorStatus' = \case
  SOpened -> \_ -> "Door is opened"
  SClosed -> \_ -> "Door is closed"
  SLocked -> \_ -> "Door is locked"
```

- ▶ GADT-ness allows us to enforce that the `s` in `SingDS s` is the same as the `s` in our `Door`.
- ▶ Singleton property means that `SingDS s` has a one-to-one correspondence with its constructors.
- ▶ Pattern matching on that single constructor reveals to us the type of `Door`.

Implicit Passing

```
class SingDSI s where
    singDS :: SingDSI s

instance SingDSI 'Opened where
    singDS = SOpened
instance SingDSI 'Closed where
    singDS = SClosed
instance SingDSI 'Locked where
    singDS = SLocked
```

Implicit Passing

```
class SingDSI s where
  singDS :: SingDSI s

instance SingDSI 'Opened where
  singDS = SOpened
instance SingDSI 'Closed where
  singDS = SClosed
instance SingDSI 'Locked where
  singDS = SLocked

doorStatus :: SingDSI s => Door s -> DoorState
doorStatus = doorStatus' singDS
```

Implicit Passing

```
class SingDSI s where
    singDS :: SingDSI s

instance SingDSI 'Opened where
    singDS = SOpened
instance SingDSI 'Closed where
    singDS = SClosed
instance SingDSI 'Locked where
    singDS = SLocked

doorStatus :: SingDSI s => Door s -> DoorState
doorStatus = doorStatus' singDS

ghci> doorStatus (UnsafeMkDoor :: Door 'Locked)
Door is locked!
```

Initialize Door

```
initializeDoor' :: SingDS s -> Door s  
initializeDoor' _ _ = UnsafeMkDoor
```

Initialize Door

```
initializeDoor' :: SingDS s -> Door s
initializeDoor' _ _ = UnsafeMkDoor

ghci> :t initializeDoor' SOpened
initializeDoor' SOpened :: Door 'Opened
ghci> :t initializeDoor' SClosed
initializeDoor' SClosed :: Door 'Closed
```

Initialize Door

Implicit passing style:

```
initializeDoor :: SingDSI s => Door s  
initializeDoor = initializeDoor' singDS
```

SingDS vs. SingDSI

- ▶ Really, $\text{SingDS } s \rightarrow$ is the same as $\text{SingDSI } s \Rightarrow$

SingDS vs. SingDSI

- ▶ Really, `SingDS s ->` is the same as `SingDSI s =>`
- ▶ The two are the same way of providing the same information to the compiler, and at runtime.

SingDS vs. SingDSI

- ▶ Really, `SingDS s ->` is the same as `SingDSI s =>`
- ▶ The two are the same way of providing the same information to the compiler, and at runtime.
- ▶ We can use the two styles interchangeably.

SingDS vs. SingDSI

- ▶ Really, `SingDS s ->` is the same as `SingDSI s =>`
- ▶ The two are the same way of providing the same information to the compiler, and at runtime.
- ▶ We can use the two styles interchangeably.
- ▶ One is **explicitly passing the type**, the other is **explicitly passing the type**.

Ditching the phantom

Sometimes we don't care about what the status of our door is, and we want the type system to relax.

Ditching the phantom

Sometimes we don't care about what the status of our door is, and we want the type system to relax.

This is essentially the same as saying that the status of our door is a runtime property that we do not want to (or sometimes can't) check at compile-time.

Ditching the phantom

```
data SomeDoor :: Type where  
  MkSomeDoor :: SingDS s => Door s -> SomeDoor
```

Ditching the phantom

```
data SomeDoor :: Type where
    MkSomeDoor :: SingDS s => Door s -> SomeDoor

ghci> let myDoor = MkSomeDoor (initializeDoor SOpened)
ghci> :t myDoor
myDoor :: SomeDoor
ghci> case myDoor of MkSomeDoor d -> doorStatus d
Door is opened.
```

Runtime-deferred types

```
initializeSomeDoor :: DoorStatus -> SomeDoor
initializeSomeDoor = \case
    Opened -> SomeDoor (initialiseDoor' SOpened)
    Closed -> SomeDoor (initialiseDoor' SClosed)
    Locked -> SomeDoor (initialiseDoor' SLocked)
```


Runtime-deferred types

```
initializeSomeDoor :: DoorStatus -> SomeDoor
initializeSomeDoor = \case
    Opened -> SomeDoor (initialiseDoor' SOpened)
    Closed -> SomeDoor (initialiseDoor' SClosed)
    Locked -> SomeDoor (initialiseDoor' SLocked)

ghci> let myDoor = initializeSomeDoor Locked
ghci> :t myDoor
myDoor :: SomeDoor
ghci> case myDoor of MkSomeDoor d -> doorStatus d
Door is locked.
```

The Singletons Library

The singletons library provides a unified framework for creating and working with singletons for different types (not just `DoorStatus`), and also for functions on those types.

<http://hackage.haskell.org/package/singletons>

The singletons way

```
$(singletons [d|  
  data DoorState = Opened | Closed | Locked  
    deriving (Show, Eq)  
  |])
```

The singletons way

```
$(singletons [d|  
  data DoorState = Opened | Closed | Locked  
    deriving (Show, Eq)  
  |])
```

This creates three types and three constructors:

```
-- not the actual code, but essentially what happens  
data Sing :: DoorState -> Type where  
  SOpened  :: Sing 'Opened  
  SClosed  :: Sing 'Closed  
  SLocked  :: Sing 'Locked
```

Sing is a poly-kinded type constructor (family):

The singletons way

And also

```
instance SingI 'Opened where  
    sing = SOpened  
instance SingI 'Closed where  
    sing = SClosed  
instance SingI 'Locked where  
    sing = SLocked
```

(SingI is a poly-kinded typeclass)

Examples

```
STrue :: Sing 'True
```

```
SJust SFalse :: Sing ('Just 'True)
```

```
SOpened `SCons` SClosed `SCons` SNil :: Sing '[ 'Opened, 'Closed
```

```
ghci> sing :: Sing 'True'
```

```
STrue
```

Other stuff created from the library

Some other convenient features:

```
ghci> fromSing SOpened  
Opened
```

```
ghci> let s = toSing Opened
```

```
ghci> :t s
```

```
s :: SomeSing DoorStatus
```

```
ghci> case s of
```

```
    SomeSing SOpened -> "Opened."
```

```
    SomeSing SClosed -> "SClosed."
```

```
    SomeSing SLocked -> "SLocked."
```

Non-trivial type logic

```
knock :: Door s -> IO ()  
knock = -- ??
```

We want to allow the user to knock on a closed or locked door, but not an opened door.

Non-trivial type logic

```
knock :: Door s -> IO ()  
knock = -- ??
```

We want to allow the user to knock on a closed or locked door, but not an opened door.

We can do this simple case using pattern matching, but it's not always feasible or scalable. We want to define a type relationship that can be used by potentially many functions.

Singletons to the Rescue

```
$(singletons [d|  
  canKnock :: DoorState -> Bool  
  canKnock Opened = False  
  canKnock Closed = True  
  canKnock Locked = True  
|])
```

Singletons to the Rescue

```
$(singletons [d|  
  canKnock :: DoorState -> Bool  
  canKnock Opened = False  
  canKnock Closed = True  
  canKnock Locked = True  
|])  
  
knock :: (CanKnock s ~ True) => Door s -> IO ()  
knock _ = putStrLn "knock knock!"
```

Singletons to the Rescue

```
$(singletons [d|
  canKnock :: DoorState -> Bool
  canKnock Opened = False
  canKnock Closed = True
  canKnock Locked = True
|])

knock :: (CanKnock s ~ True) => Door s -> IO ()
knock _ = putStrLn "knock knock!"

ghci> knock (initializeDoor SOpened)
Compile Error!!!!
ghci> knock (initializeDoor SClosed)
knock knock!
```

Singletons to the Rescue

```
tryKnock' :: Sing s -> Door s -> IO ()
tryKnock' s = case sCanKnock s of
    STrue  -> knock
    SFalse -> \_ -> putStrLn "Cannot knock door!"

tryKnock :: SingI s => Door s -> IO ()
tryKnock = tryKnock' sing
```

Singletons to the Rescue

```
tryKnock' :: Sing s -> Door s -> IO ()
tryKnock' s = case sCanKnock s of
    STrue  -> knock
    SFalse -> \_ -> putStrLn "Cannot knock door!"
```

```
tryKnock :: SingI s => Door s -> IO ()
tryKnock = tryKnock' sing
```

```
ghci> tryKnock (initializeDoor SOpened)
Cannot knock door!
ghci> tryKnock (initializeDoor SClosed)
knock knock!
```

Vectors

```
$(singletons [d|  
  data N = Z | S N  
  |])
```

```
data Vec :: N -> Type -> Type where  
  VNil :: Vec Z a  
  (:*) :: a -> Vec n a -> Vec (S n) a
```

```
infixr 5 :*
```

The Types demand it

```
replicateV'  
  :: Sing n  
  -> a  
  -> Vec n a  
replicateV' = \case  
  SZ    -> \_ -> VNil  
  SS n -> \x -> x :* replicateV' n x
```


The Types demand it

```
replicateV'
  :: Sing n
  -> a
  -> Vec n a
replicateV' = \case
  SZ    -> \_ -> VNil
  SS n -> \x -> x :* replicateV' n x

replicateV
  :: SingI n
  => a
  -> Vec n a
replicateV = replicateV' sing
```

Thank you!

- ▶ Further confusion:

<https://blog.jle.im/entry/verified-instances-in-haskell.html>