Strategic deriving.

Will Jones

HABITO

What’s in a derivation?

```
newtype T = T Int deriving (Show)

ghci> T 42
```

What’s in a derivation?

```
newtype T = T Int deriving (Show)

ghci> T 42
T 42

(Stock code for generating Show instances)

instance Show T where
  show (T x) = “T” ++ show x
```

What's in a derivation?

```
newtype T = T Int deriving (Show)

ghci> T 42
42

{-# LANGUAGE GeneralizedNewtypeDeriving #-}

instance Show T where
  show (T x) = show x
```

What's in a derivation?

```
class TwoOf a where
  twoOf :: a -> a

  default twoOf :: Num a => a -> a
  twoOf x = 2 * x

newtype T = T Int deriving (Num, Show, TwoOf)

ghci> twoOf (T 42)
T 84

instance TwoOf T where {}
```

HABITO

Strategy is what you don’t do.

```
class TwoOf a where
  twoOf :: a -> a

  default twoOf :: Num a => a -> a
  twoOf x = 2 * x

newtype T
  = T Int deriving stock    (Show)
          deriving newtype  (Num)
          deriving anyclass (TwoOf)
```

HABITO

Truly, a broken record.

```
data Map k a

lookup :: k -> Map k a -> Maybe a

data Key
  = NameK
  | AgeK

data Value
  = NameV Name
  | AgeV Age

newtype Name = Name String
newtype Age  = Age Int
```

HABITO

Truly, a broken record.

```
data DMap k

lookup :: k a -> DMap k -> Maybe a

data Key :: Type -> Type where
    NameK :: Key Name
    AgeK  :: Key Age
```

```
newtype Name = Name String
newtype Age  = Age Int
```

HABITO

Truly, a broken record.

```
data DMap k

lookup :: k a -> Map k -> Maybe a

data Key :: Type -> Type where
    NameK :: Key Name
    AgeK  :: Key Age
```

```
newtype Name = Name String
newtype Age  = Age Int
```

HABITO

Enter the boilerplate.

```
data Key :: Type -> Type where
    NameK :: Key Name
    AgeK  :: Key Age

-- Eq doesn't work on GADTs; need geq :: k a -> k b -> Bool
instance GEq Key where
  geq NameK NameK = True
  geq AgeK  AgeK  = True
  geq _     _     = False

-- Serialisation would be nice
instance Field Key where
  fieldKey key = case key of
    NameK -> "Name"
    AgeK  -> "Age"
```

Just use types as keys.

```
data DMap k

lookup :: k a -> DMap k -> Maybe a

data Key :: Type -> Type where
    NameK :: Key Name
    AgeK  :: Key Age
```

```
newtype Name = Name String
newtype Age  = Age Int
```

HABITO

Just use types as keys.

```
data TMap

lookup :: forall a. TMap -> Maybe a
```

```
lookup @Name :: TMap -> Maybe Name
lookup @Age  :: TMap -> Maybe Age

newtype Name = Name String
newtype Age  = Age Int
```

HABITO

Some types appear more than once.

```
-- No they don't.

newtype Primary a   = Primary a
newtype Secondary a = Secondary a

lookup @(Primary Name)  :: ..
lookup @(Secondary Age) :: ..
```

Serialisation killer.

```
{
  “Name”: “Alice”,
  “Secondary/Age”: 42
}
```

```
class Field (s :: Symbol) (a :: Type) | a -> s

instance Field “Name” Name

fieldKey :: forall a s. Field s a => String


fieldKey @Name == (“Name” :: String)
```

HABITO

Serialisation killer.

```
{
  “Name”: “Alice”,
  “Secondary/Age”: 42
}
```

```
class Group (s :: Symbol) (f :: Type -> Type) | f -> s

instance (Group s1 f, Field s2 a)
       => Field (s1 ++ “/” ++ s2) (f a)

instance Group “Secondary” Secondary

fieldKey @(Secondary Age) == (“Secondary/Age” :: String)
```

Just tell me about the type.

```
newtype Age = Age Int
  deriving stock    (Show)
  deriving newtype  (FromJSON, ToJSON)
  deriving anyclass (Field “Age”)

newtype Primary a = Primary a
  deriving ..
  deriving anyclass (Group “Primary”)
```

Now we can write the library.

```
newtype TMap = TMap (M.Map String JSON)


insert :: forall a s. (ToJSON a, Field s a)
       => a -> TMap -> TMap

insert x (TMap m) =
  TMap (M.insert (keyField @s) x m)

lookup :: forall a s. (FromJSON a, Field s a)
       => TMap -> Maybe a

lookup (TMap m) = M.lookup (keyField @s) m
```

HABITO

Now we can write the library.

```
newtype TMap = TMap (M.Map String JSON)
  deriving newtype (Semigroup, Monoid)

insert :: forall a s. (ToJSON a, Field s a)
       => a -> TMap -> TMap

insert x (TMap m) =
  TMap (M.insert (keyField @s) x m)

lookup :: forall a s. (FromJSON a, Field s a)
       => TMap -> Maybe a

lookup (TMap m) = M.lookup (keyField @s) m
```

HABITO

Unmapped territory.

```
data Person
  = Person
      { _pName     :: Name
      , _pAge      :: Age
      }


personFromTMap :: TMap -> Maybe Person
```

Unmapped territory.

```
data Person
  = Person
      { _pName     :: Name
      , _pAge      :: Age
      }

  deriving stock (Generic)

personFromTMap :: TMap -> Maybe Person
personFromTMap = gfromTMap
```

Unmapped territory.

```
type Rep Person =
  C1 “Person”
    (   S1 “_pName” (Rec0 Name)
    :*: S1 “_pAge”  (Rec0 Age)
    )
```

```
data Person
  = Person
      { _pName     :: Name
      , _pAge      :: Age
      }

  deriving stock (Generic)

personFromTMap :: TMap -> Maybe Person
personFromTMap = gfromTMap
                       |
(\tm -> Person <$> lookup @Name tm <*> lookup @Age tm)
```

HABITO

Unmapped territory.

```
data Person
  = Person
      { _pName    :: Name
      , _pAge     :: Age
      , _pSavings :: Maybe Savings
      }
```

Unmapped territory.

```
data Person f
  = Person
      { _pName    :: Name
      , _pAge     :: Age
      , _pSavings ::   f   Savings
      }

type UnqualifiedPerson = Person Maybe
type QualifiedPerson   = Person Identity
```

Unmapped territory.

```
data Customer
  = Customer
      { _cName    :: Name
      , _cAge     :: Age
      , _cSavings :: Savings
      }
```

```
personToCustomer :: Person -> Savings -> Customer
personToCustomer
```

HABITO

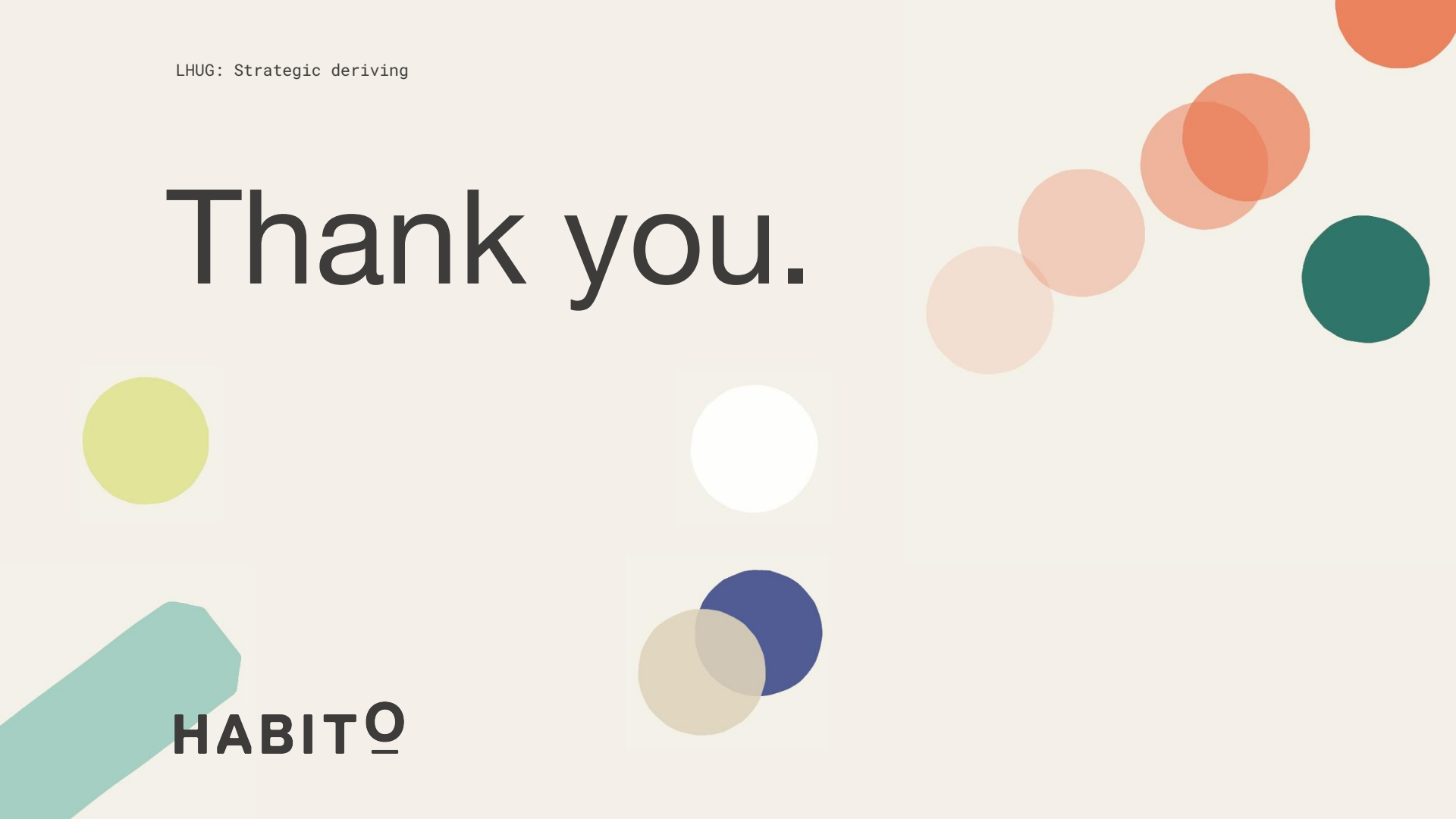
Unmapped territory.

```
data Customer
  = Customer
      { _cName    :: Name
      , _cAge     :: Age
      , _cSavings :: Savings
      }

  deriving stock (Generic)

personToCustomer :: Person -> Savings -> Customer
personToCustomer = gexpand
                       |
(\Person{..} savings -> Customer _pName _pAge savings)
```

```
type Rep Person =
  C1 “Person”
    (   S1 “_pName”    (Rec0 Name)
    :*: S1 “_pAge”     (Rec0 Age)
    )

type Rep Customer =
  C1 “Customer”
    (   S1 “_cName”    (Rec0 Name)
    :*: S1 “_cAge”     (Rec0 Age)
    :*: S1 “_cSavings” (Rec0 Savings)
    )
```

HABITO

Thank you.



HABITO