Generic and Flexible Defaults for Verified,Law-Abiding Type-Class Instances

Ryan Scott Ryan Newton

Indiana University

Haskell Symposium 2019
Berlin, Germany

How can we make verifying type class laws more pleasant?

```haskell
class Ord a where
  (<=) :: a -> a -> Bool
```

```haskell
-- Transitivity:
--   if x <= y && y <= z,
--   then x <= z
class Ord a where
  (<=) :: a -> a -> Bool
```

```haskell
class Ord a => VOrd a where
  leqTransitive
    :: Π (x, y, z :: a)
    -> (x <= y) :~: True
    -> (y <= z) :~: True
    -> (x <= z) :~: True
```

```
class Ord a => VOrd a where
  leqTransitive
    :: Π (x, y, z :: a)
    -> (x <= y) :~: True
                    True
                    True
```

```
data a :~: b where
  Refl :: a :~: a
```

```
class Ord a => VOrd a where
  leqTransitive
    :: Π (x, y, z :: a)
    -> (x <= y) :~: True
    -> (y <= z) :~: True
    -> (x <= z) :~: True
```

Writing out proofs can be tiresome

```
data T a = MkT1 a
```

```haskell
data T a = MkT1 a

instance Ord a => Ord (T a) where
  (MkT1 x) <= (MkT1 y) = (x <= y)
```

```haskell
data T a = MkT1 a

instance Ord a => Ord (T a) where
  (MkT1 x) <= (MkT1 y) = (x <= y)

instance VOrd a => VOrd (T a) where
  leqTransitive (MkT1 x) (MkT1 y) (MkT1 z) Refl Refl
    | Refl <- leqTransitive x y z Refl Refl
    = Refl
```

```haskell
data T a = MkT1 a

instance Ord a => Ord (T a) where
  (MkT1 x) <= (MkT1 y) = (x <= y)

instance VOrd a => VOrd (T a) where
  leqTransitive (MkT1 x) (MkT1 y) (MkT1 z) Refl Refl
    | Refl <- leqTransitive x y z Refl Refl
    = Refl
```

```haskell
data T a = MkT1 a | MkT2 a

instance Ord a => Ord (T a) where
  (MkT1 x) <= (MkT1 y) = (x <= y)

instance VOrd a => VOrd (T a) where
  leqTransitive (MkT1 x) (MkT1 y) (MkT1 z) Refl Refl
    | Refl <- leqTransitive x y z Refl Refl
    = Refl
```

```haskell
data T a = MkT1 a | MkT2 a

instance Ord a => Ord (T a) where
  (MkT1 x) <= (MkT1 y) = (x <= y)
  (MkT2 x) <= (MkT2 y) = (x <= y)
  (MkT1 _) <= (MkT2 _) = True
  (MkT2 _) <= (MkT1 _) = False
```

```haskell
data T a = MkT1 a | MkT2 a

instance VOrd a => VOrd (T a) where
  leqTransitive t t' t'' Refl Refl =
    case (t, t', t'') of
      (MkT1 x, MkT1 y, MkT1 z)
        | Refl <- leqTransitive x y z Refl Refl
        = Refl
      (MkT2 x, MkT2 y, MkT2 z)
        | Refl <- leqTransitive x y z Refl Refl
        = Refl
      (MkT1 _, _, MkT2 _)
        = Refl
```

```
data T a = MkT1 a | MkT2 a

instance VOrd a => VOrd (T a) where
  leqTransitive t t' t'' Refl Refl =
    case (t, t', t'') of
      (MkT1 x, MkT1 y, MkT1 z)
        | Refl <- leqTransitive x y z Refl Refl
        = Refl
      (MkT2 x, MkT2 y, MkT2 z)
        | Refl <- leqTransitive x y z Refl Refl
        = Refl
      (MkT1 _, _, MkT2 _)
        = Refl
```

```haskell
data T a = MkT1 a | MkT2 a | MkT3 a
  instance VOrd a => VOrd (T a) where
    leqTransitive t t' t'' Refl Refl =
      case (t, t', t'') of
        (MkT1 x, MkT1 y, MkT1 z)
          | Refl <- leqTransitive x y z Refl Refl
          = Refl
        (MkT2 x, MkT2 y, MkT2 z)
          | Refl <- leqTransitive x y z Refl Refl
          = Refl
        (MkT1 _, _, MkT2 _)
          = Refl
```

```haskell
data T a = MkT1 a | MkT2 a | MkT3 a
```

```haskell
instance VOrd a => VOrd (T a) where
  leqTransitive t t' t'' Refl Refl =
    case (t, t', t'') of
      (MkT1 x, MkT1 y, MkT1 z)
        | Refl <- leqTransitive x y z Refl Refl
        = Refl
      (MkT2 x, MkT2 y, MkT2 z)
        | Refl <- leqTransitive x y z Refl Refl
        = Refl
      (MkT3 x, MkT3 y, MkT3 z)
        | Refl <- leqTransitive x y z Refl Refl
        = Refl
      (MkT1 _, _, MkT2 _)
        = Refl
      (MkT1 _, _, MkT3 _)
        = Refl
      (MkT2 _, _, MkT3 _)
        = Refl
```

```
data T a = MkT1 a | MkT2 a | MkT3 a
```

```
instance VOrd a => VOrd (T a) where
  leqTransitive t t' t'' Refl Refl =
    case (t, t', t'') of
      (MkT1 x, MkT1 y, MkT1 z)
        | Refl <- leqTransitive x y z Refl Refl
        = Refl
      (MkT2 x, MkT2 y, MkT2 z)
        | Refl <- leqTransitive x y z Refl Refl
        = Refl
      (MkT3 x, MkT3 y, MkT3 z)
        | Refl <- leqTransitive x y z Refl Refl
        = Refl
      (MkT1 _, _, MkT2 _)
        = Refl
      (MkT1 _, _, MkT3 _)
        = Refl
      (MkT2 _, _, MkT3 _)
        = Refl
```

```
data T a = MkT1 a | ... | MkTn a

  instance VOrd a => VOrd (T a) where
    leqTransitive t t' t'' Refl Refl =
      case (t, t', t'') of
        (MkT1 x, MkT1 y, MkT1 z)
          | Refl <- leqTransitive x y z Refl Refl
          = Refl
        (MkT2 x, MkT2 y, MkT2 z)
          | Refl <- leqTransitive x y z Refl Refl
          = Refl
        (MkT3 x, MkT3 y, MkT3 z)
          | Refl <- leqTransitive x y z Refl Refl
          = Refl
        (MkT1 _, _, MkT2 _)
          = Refl
        (MkT1 _, _, MkT3 _)
          = Refl
        (MkT2 _, _, MkT3 _)
          = Refl
```

```
data T a = MkT1 a | ... | MkTn a
```

```
instance VOrd a => VOrd (T a) where
  leqTransitive t t' t'' Refl Refl =
    case (t, t', t'') of
      (MkT1 x, MkT1 y, MkT1 z)
        | Refl <- leqTransitive x y z Refl Refl
        = Refl
      (MkT2 x, MkT2 y, MkT2 z)
        | Refl <- leqTransitive x y z Refl Refl
        = Refl
      (MkT3 x, MkT3 y, MkT3 z)
        | Refl <- leqTransitive x y z Refl Refl
        = Refl
      (MkT1 _, _, MkT2 _)
        = Refl
      (MkT1 _, _, MkT3 _)
        = Refl
      (MkT2 _, _, MkT3 _)
        = Refl
```

$\frac{n^2+n}{2}$ **cases!**

Our solution: datatype-generic proofs

Our solution: datatype-generic proofs (à la `GHC.Generics`)

How to deal with proof boilerplate

How to deal with proof boilerplate

Tactics

How to deal with proof boilerplate

Tactics

How to deal with proof boilerplate

Tactics

How to deal with proof boilerplate

Tactics

???

```
class Ord a => VOrd a where
  leqTransitive
    :: Π (x, y, z :: a)
    -> (x <= y) :~: True
    -> (y <= z) :~: True
    -> (x <= z) :~: True
```

```
class Ord a => VOrd a where
  leqTransitive
    :: Π (x, y, z :: a)
    -> (x <= y) :~: True
    -> (y <= z) :~: True
    -> (x <= z) :~: True
```

```haskell
class Ord a => VOrd a where
  leqTransitive
    :: Sing (x :: a)
    -> Sing (y :: a)
    -> Sing (z :: a)
    -> (x <= y) :~: True
    -> (y <= z) :~: True
    -> (x <= z) :~: True
```

```haskell
class Ord a => VOrd a where
  leqTransitive
    :: Sing (x :: a)
    -> Sing (y :: a)
    -> Sing (z :: a)
    -> (x <= y) :~: True
    -> (y <= z) :~: True
    -> (x <= z) :~: True
```

Uses the *singletons* encoding to emulate dependent types

```haskell
class Ord a => VOrd a where
  leqTransitive
    :: Sing (x :: a)
    -> Sing (y :: a)
    -> Sing (z :: a)
    -> (x <= y) :~: True
    -> (y <= z) :~: True
    -> (x <= z) :~: True
```

Uses the *singletons* encoding to emulate dependent types

```haskell
class Ord a => VOrd a where
  leqTransitive
    :: Sing (x :: a)
    -> Sing (y :: a)
    -> Sing (z :: a)
    -> (x <= y) :~: True
    -> (y <= z) :~: True
    -> (x <= z) :~: True
```

Uses the *singletons* encoding to emulate dependent types

```
class Ord a => VOrd a where
  leqTransitive
    :: Sing (x :: a)
    -> Sing (y :: a)
    -> Sing (z ::
    -> (x <= y) :
    -> (y <= z) :~: True
    -> (x <= z) :~: True
```

Uses the *singletons* encoding to emulate dependent types

```
type family x <= y
```

```haskell
class Ord a => VOrd a where
  leqTransitive
    :: Π (x, y, z :: a)
    -> (x <= y) :~: True
    -> (y <= z) :~: True
    -> (x <= z) :~: True
```

Our solution: datatype-generic proofs (à la `GHC.Generics`)

Our solution: datatype-generic proofs (à la `GHC.Generics`)

```haskell
instance Ord a => Ord (T a) where
  (MkT1 x) <= (MkT1 y) = (x <= y)
  (MkT2 x) <= (MkT2 y) = (x <= y)
  (MkT1 _) <= (MkT2 _) = True
  (MkT2 _) <= (MkT1 _) = False
```

```haskell
instance Ord a => Ord (T a) where
  (<=) = defaultLeq
```

```haskell
instance Ord a => Ord (T a) where
  (<=) = defaultLeq
instance Ord Bool where
  (<=) = defaultLeq
```

```haskell
instance Ord a => Ord (T a) where
  (<=) = defaultLeq
instance Ord Bool where
  (<=) = defaultLeq
instance Ord a => Ord (Maybe a) where
  (<=) = defaultLeq
```

```haskell
instance Ord a => Ord (T a) where
  (<=) = defaultLeq
instance Ord Bool where
  (<=) = defaultLeq
instance Ord a => Ord (Maybe a) where
  (<=) = defaultLeq
instance Ord a => Ord [a] where
  (<=) = defaultLeq
```

```haskell
defaultLeq :: (Generic a, Ord (Rep a))
           => a -> a -> Bool
defaultLeq x y = (from x <= from y)
```

```haskell
defaultLeq :: (Generic a, Ord (Rep a))
           => a -> a -> Bool
defaultLeq x y = (from x <= from y)
```

```haskell
defaultLeq :: (Generic a, Ord (Rep a))
           => a -> a -> Bool
defaultLeq x y = (from x <= from y)
```

```haskell
defaultLeq :: (Generic a, Ord (Rep a))
           => a -> a -> Bool
defaultLeq x y = (from x <= from y)
```

```haskell
class Generic a where
  type Rep a
  from :: a -> Rep a
  to   :: Rep a -> a
```

```haskell
defaultLeq :: (Generic a, Ord (Rep a))
           => a -> a -> Bool
defaultLeq x y = (from x <= from y)
```

```haskell
class Generic a where
  type Rep a
  from :: a -> Rep a
  to   :: Rep a -> a
```

1. **A canonical representation type (Rep)**

```haskell
defaultLeq :: (Generic a, Ord (Rep a))
           => a -> a -> Bool
defaultLeq x y = (from x <= from y)
```

```haskell
class Generic a where
  type Rep a
  from :: a -> Rep a
  to   :: Rep a -> a
```

1. A canonical representation type (Rep)
2. An isomorphism between a and Rep a

```haskell
defaultLeq :: (Generic a, Ord (Rep a))
           => a -> a -> Bool
defaultLeq x y = (from x <= from y)
```

```haskell
class Generic a where
  type Rep a
  from :: a -> Rep a
  to   :: Rep a -> a
```

1. A canonical representation type (Rep)
2. An isomorphism between a and Rep a
3. Generic instances can be derived automatically.

4. Rep is minimalist

Unit	`data U1 = MkU1`
Constants	`newtype K1 c = MkK1 c`
Products	`data a :*: b = a :**: b`
Sums	`data a :+: b = L1 a | R1 b`

4. Rep is minimalist

Unit	`data U1        = MkU1`
Constants	`newtype K1 c = MkK1 c`
Products	`data a :*: b = a :**: b`
Sums	`data a :+: b = L1 a | R1 b`

Every instance of Rep is some combination of these four types.

4. Rep is minimalist

Unit	`instance Ord U1`
Constants	`instance Ord c => Ord (K1 c)`
Products	`instance (Ord a, Ord b) => Ord (a :*: b)`
Sums	`instance (Ord a, Ord b) => Ord (a :+: b)`

Every instance of Rep is some combination of these four types.

Our solution: **datatype-generic** proofs (à la `GHC.Generics`)

Our solution:
datatype-generic proofs
(à la `GHC.Generics`)

```haskell
instance Ord a => Ord (T a) where
  (<=) = defaultLeq
instance Ord Bool where
  (<=) = defaultLeq
instance Ord a => Ord (Maybe a) where
  (<=) = defaultLeq
instance Ord a => Ord [a] where
  (<=) = defaultLeq
```

```haskell
instance VOrd a => VOrd (T a) where
  leqTransitive = defaultLeqTransitive
instance VOrd Bool where
  leqTransitive = defaultLeqTransitive
instance VOrd a => VOrd (Maybe a) where
  leqTransitive = defaultLeqTransitive
instance VOrd a => VOrd [a] where
  leqTransitive = defaultLeqTransitive
```

```
defaultLeqTransitive
  :: ???
  => Π (x, y, z :: a)  -> (x <= y) :~: True
  -> (y <= z) :~: True -> (x <= z) :~: True
defaultLeqTransitive = ???
```

```
defaultLeqTransitive
  :: (Generic a, VOrd (Rep a))
  => Π (x, y, z :: a)  -> (x <= y) :~: True
  -> (y <= z) :~: True -> (x <= z) :~: True
defaultLeqTransitive = ???
```

Unit	`instance VOrd U1`
Constants	`instance VOrd c => VOrd (K1 c)`
Products	`instance (VOrd a, VOrd b) => VOrd (a :*: b)`
Sums	`instance (VOrd a, VOrd b) => VOrd (a :+: b)`

Unit	`instance VOrd U1`
Constants	`instance VOrd c => VOrd (K1 c)`
Products	`instance (VOrd a, VOrd b) => VOrd (a :*: b)`
Sums	`instance (VOrd a, VOrd b) => VOrd (a :+: b)`

```
Unit         instance VOrd U1
Constants    instance VOrd c => VOrd (K1 c)
Products     instance (VOrd a, VOrd b) => VOrd (a :*: b)
Sums         instance (VOrd a, VOrd b) => VOrd (a :+: b)
                 where
               leqTransitive s s’ s’’ Refl Refl =
                 case (s, s’, s’’) of
                   (L1 x, L1 y, L1 z)
                     | Refl <- leqTransitive x y z
                                 Refl Refl = Refl
                   (R1 x, R1 y, R1 z)
                     | Refl <- leqTransitive x y z
                                 Refl Refl = Refl
                   (L1 _, _, R1 _) -> Refl
```

```haskell
defaultLeqTransitive
  :: (Generic a, VOrd (Rep a))
  => Π (x, y, z :: a)  -> (x <= y) :~: True
  -> (y <= z) :~: True -> (x <= z) :~: True
defaultLeqTransitive = ???
```

```haskell
defaultLeqTransitive
  :: (Generic a, VOrd (Rep a))
  => Π (x, y, z :: a)  -> (x <= y) :~: True
  -> (y <= z) :~: True -> (x <= z) :~: True
defaultLeqTransitive x y z xLeqY yLeqZ =
  leqTransitive (from x) (from y) (from z)
                xLeqY yLeqZ
```

```haskell
defaultLeqTransitive
  :: (Generic a, VOrd (Rep a))
  => Π (x, y, z :: a)  -> (x <= y) :~: True
  -> (y <= z) :~: True -> (x <= z) :~: True
defaultLeqTransitive x y z xLeqY yLeqZ =
  leqTransitive (from x) (from y) (from z)
                xLeqY yLeqZ
```

```haskell
defaultLeqTransitive
  :: (Generic a, VOrd (Rep a))
  => Π (x, y, z :: a)  -> (x <= y) :~: True
  -> (y <= z) :~: True -> (x <= z) :~: True
defaultLeqTransitive x y z xLeqY yLeqZ =
  leqTransitive (from x) (from y) (from z)
                xLeqY yLeqZ
```

```
Expected type: (from x <= from y) :~: True
Actual type:   (x <= y) :~: True
```

```
defaultLeqTransitive
  :: (Generic a, VOrd (Rep a))
  => Π (x, y, z :: a)  -> (x <= y) :~: True
  -> (y <= z) :~: True -> (x <= z) :~: True
defaultLeqTransitive x y z xLeqY yLeqZ =
  leqTransitive (from x) (from y) (from z)
                    xLeqY yLeqZ
```

```
Expected type: (from x <= from y) :~: True
Actual type:   (x <= y) :~: True
```

```
class (Generic a, Ord a, Ord (Rep a))
 => GOrd a where
  genericLeqC
    :: Π (x, y :: a)
    -> (x <= y) :~: (defaultLeq x y)
```

```haskell
class (Generic a, Ord a, Ord (Rep a))
  => GOrd a where
   genericLeqC
     :: Π (x, y :: a)
     -> (x <= y) :~: (defaultLeq x y)
```

```haskell
defaultLeq :: (Generic a, Ord (Rep a))
           => a -> a -> Bool
defaultLeq x y = (from x <= from y)
```

```
defaultLeqTransitive
  :: (Generic a, VOrd (Rep a))
  => Π (x, y, z :: a)  -> (x <= y) :~: True
  -> (y <= z) :~: True -> (x <= z) :~: True
defaultLeqTransitive x y z xLeqY yLeqZ =
  leqTransitive (from x) (from y) (from z)
                xLeqY yLeqZ
```

```haskell
defaultLeqTransitive
  :: (Generic a, VOrd (Rep a), GOrd a)
  => Π (x, y, z :: a)  -> (x <= y) :~: True
  -> (y <= z) :~: True -> (x <= z) :~: True
defaultLeqTransitive x y z xLeqY yLeqZ =
  leqTransitive (from x) (from y) (from z)
                xLeqY yLeqZ
```

```haskell
defaultLeqTransitive
  :: (Generic a, VOrd (Rep a), GOrd a)
  => Π (x, y, z :: a)  -> (x <= y) :~: True
  -> (y <= z) :~: True -> (x <= z) :~: True
defaultLeqTransitive x y z xLeqY yLeqZ
  | Refl <- genericLeqC x y
  , Refl <- genericLeqC y z
  , Refl <- genericLeqC x z
  = leqTransitive (from x) (from y) (from z)
                  xLeqY yLeqZ
```

```
defaultLeqTransitive
```

```
Expected type: (from x <= from y) :~: True
Actual type:    (x <= y) :~: True
```

```
  -> (y <= z) :~: True -> (x <= z) :~: True
defaultLeqTransitive x y z xLeqY yLeqZ
  | Refl <- genericLeqC x y
  , Refl <- genericLeqC y z
  , Refl <- genericLeqC x z
  = leqTransitive (from x) (from y) (from z)
                  xLeqY yLeqZ
```

```haskell
defaultLeqTransitive
  :: (Generic a, VOrd (Rep a), GOrd a)
  => Π (x, y, z :: a)  -> (x <= y) :~: True
  -> (y <= z) :~: True -> (x <= z) :~: True
defaultLeqTransitive x y z xLeqY yLeqZ
  | Refl <- genericLeqC x y
  , Refl <- genericLeqC y z
  , Refl <- genericLeqC x z
  = leqTransitive (from x) (from y) (from z)
                  xLeqY yLeqZ
```

```haskell
defaultLeqTransitive
  :: (Generic a, VOrd (Rep a), GOrd a)
  => Π (x, y, z :: a)  -> (x <= y) :~: True
  -> (y <= z) :~: True -> (x <= z) :~: True
defaultLeqTransitive x y z xLeqY yLeqZ
  | Refl <- genericLeqC x y
  , Refl <- genericLeqC y z
  , Refl <- genericLeqC x z
  = leqTransitive (from x) (from y) (from z)
                  xLeqY yLeqZ
```

```
defaultLeqTransitive
  :: (Generic a, VOrd (Rep a), GOrd a)
  => Π (x, y, z :: a)  -> (x <= y) :~: True
  -> (y <= z) :~: True -> (x <= z) :~: True
defaultLeqTransitive x y z xLeqY yLeqZ
  | Refl <- genericLeqC x y
  , Refl <- genericLeqC y z
  , Refl <- genericLeqC x z
  = leqTransitive (from x) (from y) (from z)
                  xLeqY yLeqZ
    :: (from x <= from z) :~: True
```

```
data T a = MkT1 a | ... | MkTn a
```

```
data T a = MkT1 a | ... | MkTn a
  deriving Generic
```

```haskell
data T a = MkT1 a | ... | MkTn a
  deriving Generic
instance Ord a => Ord (T a) where
  (<=) = defaultLeq
```

```haskell
data T a = MkT1 a | ... | MkTn a
  deriving Generic
instance Ord a => Ord (T a) where
  (<=) = defaultLeq
instance GOrd a => GOrd (T a) where
  genericLeqC _ _ = Refl
```

```haskell
data T a = MkT1 a | ... | MkTn a
  deriving Generic
instance Ord a => Ord (T a) where
  (<=) = defaultLeq
instance GOrd a => GOrd (T a) where
  genericLeqC _ _ = Refl
instance VOrd a => VOrd (T a) where
  leqTransitive = defaultLeqTransitive
```

```haskell
data T a = MkT1 a | ... | MkTn a
  deriving Generic
instance Ord a => Ord (T a) where ?
  (<=) = handwrittenLeqImpl
instance GOrd a => GOrd (T a) where
  genericLeqC _ _ = Refl
instance VOrd a => VOrd (T a) where
  leqTransitive = defaultLeqTransitive
```

```haskell
data T a = MkT1 a | ... | MkTn a
  deriving Generic
instance Ord a => Ord (T a) where
  (<=) = handwrittenLeqImpl
instance GOrd a => GOrd (T a) where
  genericLeqC = handwrittenAndDefaultCoincide
instance VOrd a => VOrd (T a) where
  leqTransitive = defaultLeqTransitive
```

?

More! (In the paper)

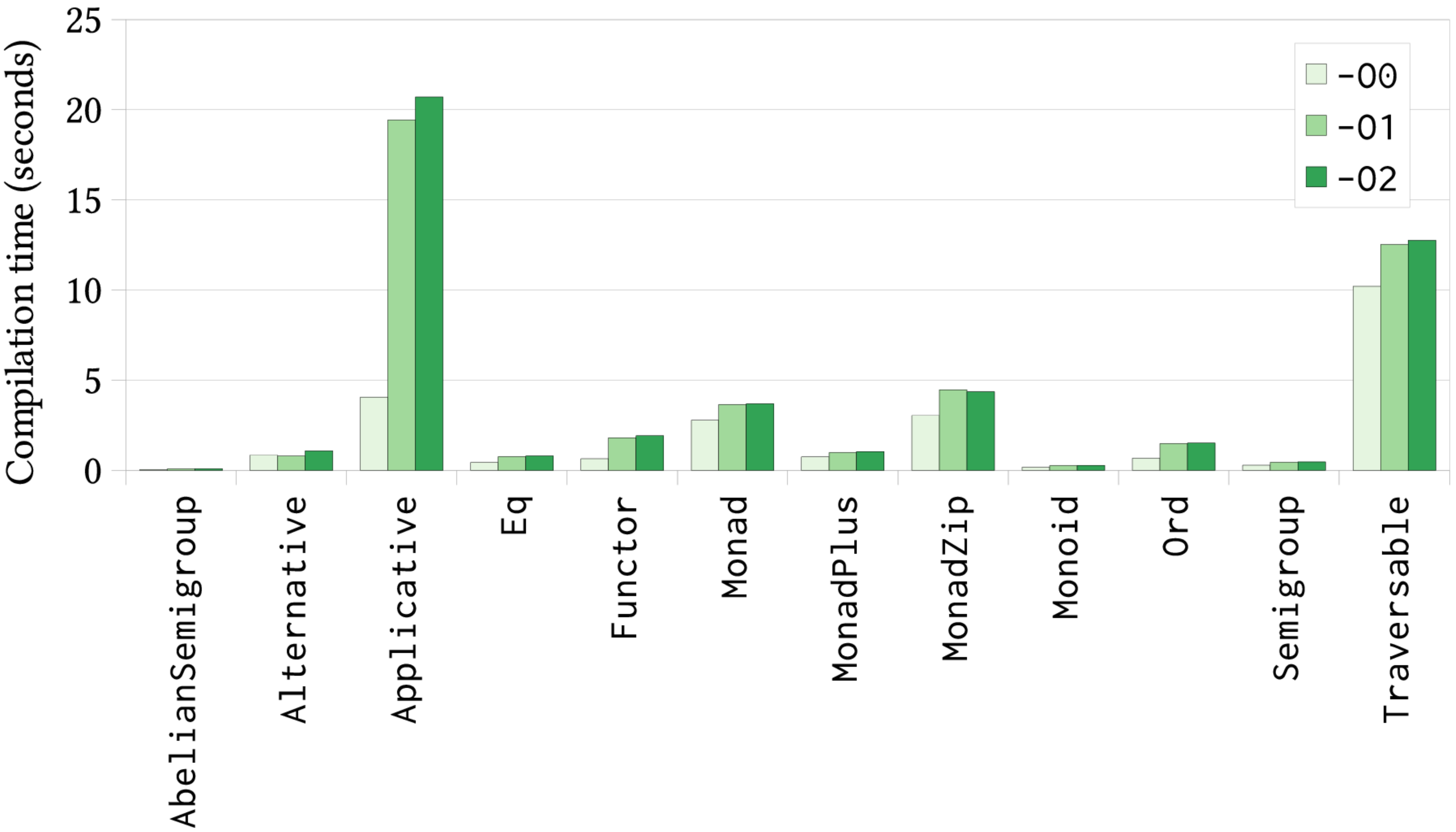
- Generically verifiying more laws from classes in base (`Eq`, `Functor`, `Monad`, `Traversable`, etc.)
- Porting over ideas to other dependently typed languages
  - Coq (successful!)
  - LiquidHaskell (not yet successful…)
- Compile times
- Comparison to other generic programming techniques in dependent types (e.g., univalent transport from HoTT)

verified-classes

- Scrap your type class proof boilerplate as easily as any other type class boilerplate
- Flexible enough to deal with existing code
- Implemented in GHC, but ideas can be ported to other dependently typed languages

`https://gitlab.com/RyanGlScott/verified-classes`

```
Theorem defaultLeqReflexive :
  forall {a : Type} `{VGeneric a}
         `{VOrd (Rep a)} `{Ord a}
         `{! GOrd a}
         (x : a), leq x x = True.
Proof.
  intros. rewrite genericLeqC.
  unfold genericLeq.
  apply leqReflexive.
Qed.
```



Compilation time (seconds)
25
20
15
10
5
0
-O0
-O1
-O2
AbelianSemigroup
Alternative
Applicative
Eq
Functor
Monad
MonadPlus
MonadZip
Monoid
Ord
Semigroup
Traversable