The `verified-classes` Library: Big Proofs, Little Tedium

Ryan Scott
Ryan Newton

Indiana University

Midwest PL Summit 2019

```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
```

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

```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
```

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

```haskell
instan    Ord a => VOrd (T a) where
  le      ive t t' t'' Refl Refl
    c      ', t'') of
      (     kT1 y, MkT1 z)
        |      leqTransit          Refl Refl
        = R
      (MkT2 x          M
        | Refl                 x y z Refl Refl
        = Refl
      (MkT3 x, MkT          z)
        | Refl <           e x y z Refl Refl
        = Refl
      (MkT1              _)
        = R
      (M          MkT3 _)

             _, MkT3 _)
            fl
```

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

instance VOrd a => VOrd (T a) where
  leqTransitive = defaultLeqTransitive
```

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

instance VOrd a => VOrd (T a) where
  leqTransitive = defaultLeqTransitive
```

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

Ryan G. Scott
Indiana University
United States
rgscott@indiana.edu

Ryan R. Newton
Indiana University
United States
rrnewton@indiana.edu

Abstract

Dependently typed languages allow programmers to state and prove type class laws by simply encoding the laws as class methods. But writing implementations of these methods frequently give way to large amounts of routine, boilerplate code, and depending on the law involved, the size of these proofs can grow superlinearly with the size of the datatypes involved.

We present a technique for automating away large swaths of this boilerplate by leveraging datatype-generic programming. We observe that any algebraic data type has an equivalent *representation type* that is composed of simpler, smaller types that are simpler to prove theorems over. By constructing an isomorphism between a datatype and its represen-

(Haskell '19), August 22–23, 2019, Berlin, Germany. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3331545.3342591

1 Introduction

Various programming languages support combining type classes [34], or similar features, with dependent type systems, including Agda [11], Clean [29], Coq [25], F* [19], Idris [7], Isabelle [14], and Lean [5]. Even Haskell, the language which inspired the development of type classes, is moving towards adding full-spectrum dependent types [12, 35], and determined Haskell users can already write many dependently typed programs using the *singletons* encoding [13].

Type classes and dependent types together make an appealing combination since many classes come equipped with

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

Ryan G. Scott
Indiana University
United States
rgscott@indiana.edu

Ryan R. Newton
Indiana University
United States
rrnewton@indiana.edu

Abstract

Dependently typed languages allow programmers to state and prove type class laws by simply encoding the laws as class methods. But writing implementations of these methods frequently give way to large amounts of routine, boilerplate code, and depending on the law involved, the size of these proofs can grow superlinearly with the size of the datatypes involved.

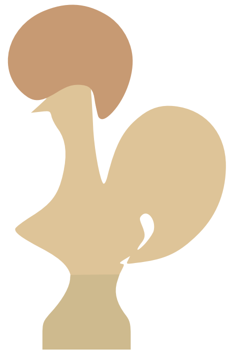
We present a technique for automating away large swaths of this boilerplate by leveraging datatype-generic programming. We observe that any algebraic data type has an equivalent *representation type* that is composed of simpler, smaller types that are simpler to prove theorems over. By constructing an isomorphism between a datatype and its represen-

(Haskell '19), August 22–23, 2019, Berlin, Germany. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3331545.3342591

1 Introduction

Various programming languages support combining type classes [34], or similar features, with dependent type systems, including Agda [11], Clean [29], Coq [25], F* [19], Idris [7], Isabelle [14], and Lean [5]. Even Haskell, the language which inspired the development of type classes, is moving towards adding full-spectrum dependent types [12, 35], and determined Haskell users can already write many dependently typed programs using the *singletons* encoding [13].

Type classes and dependent types together make an appealing combination since many classes come equipped with



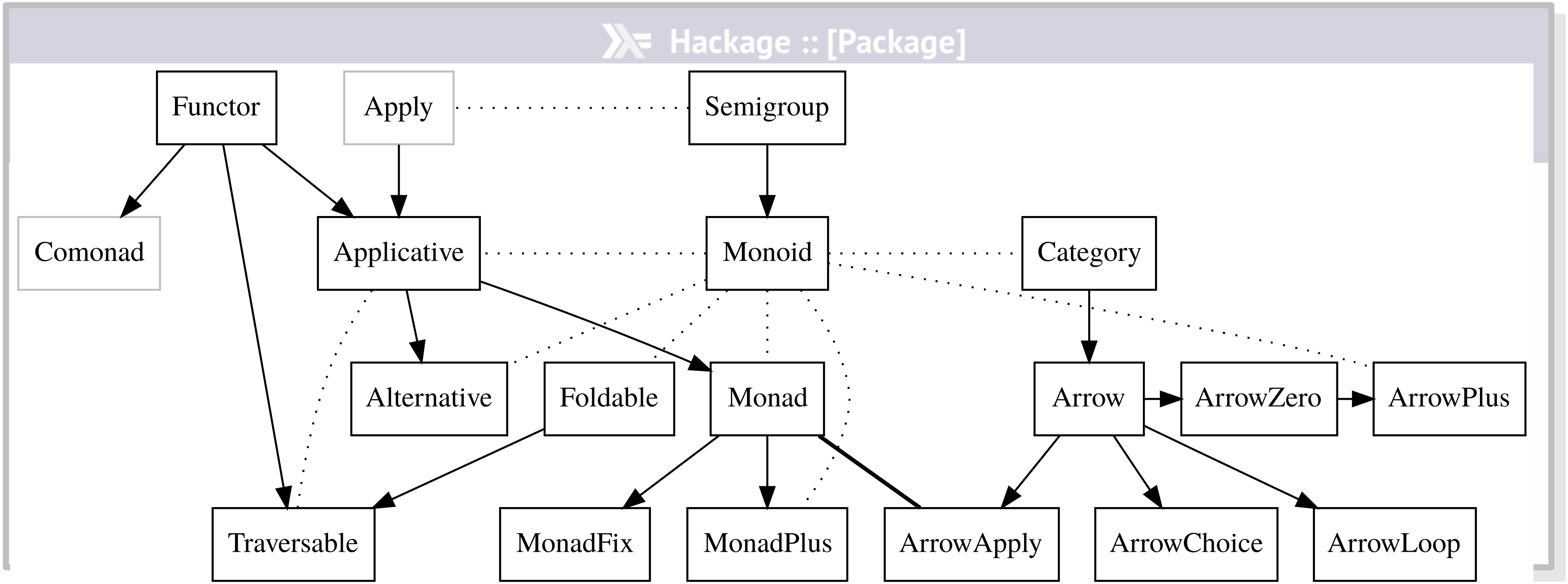
Hackage :: [Package]

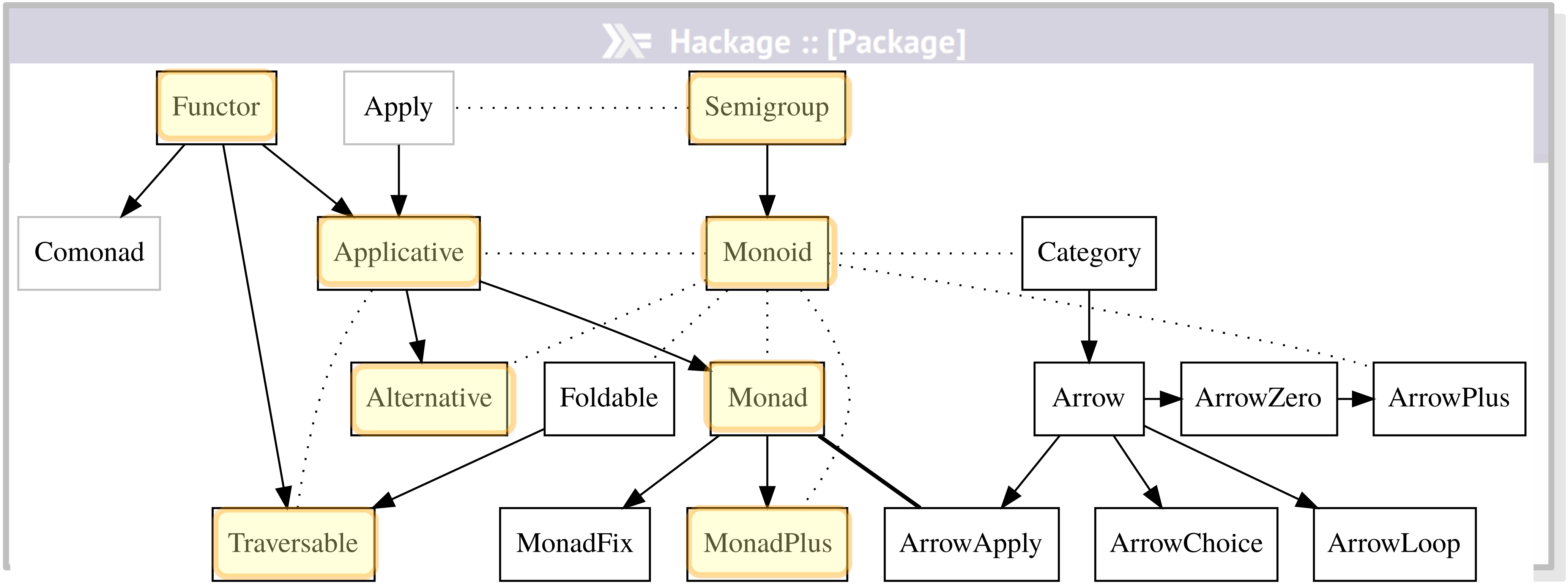
Search · Browse · What's new · Upload · User accounts

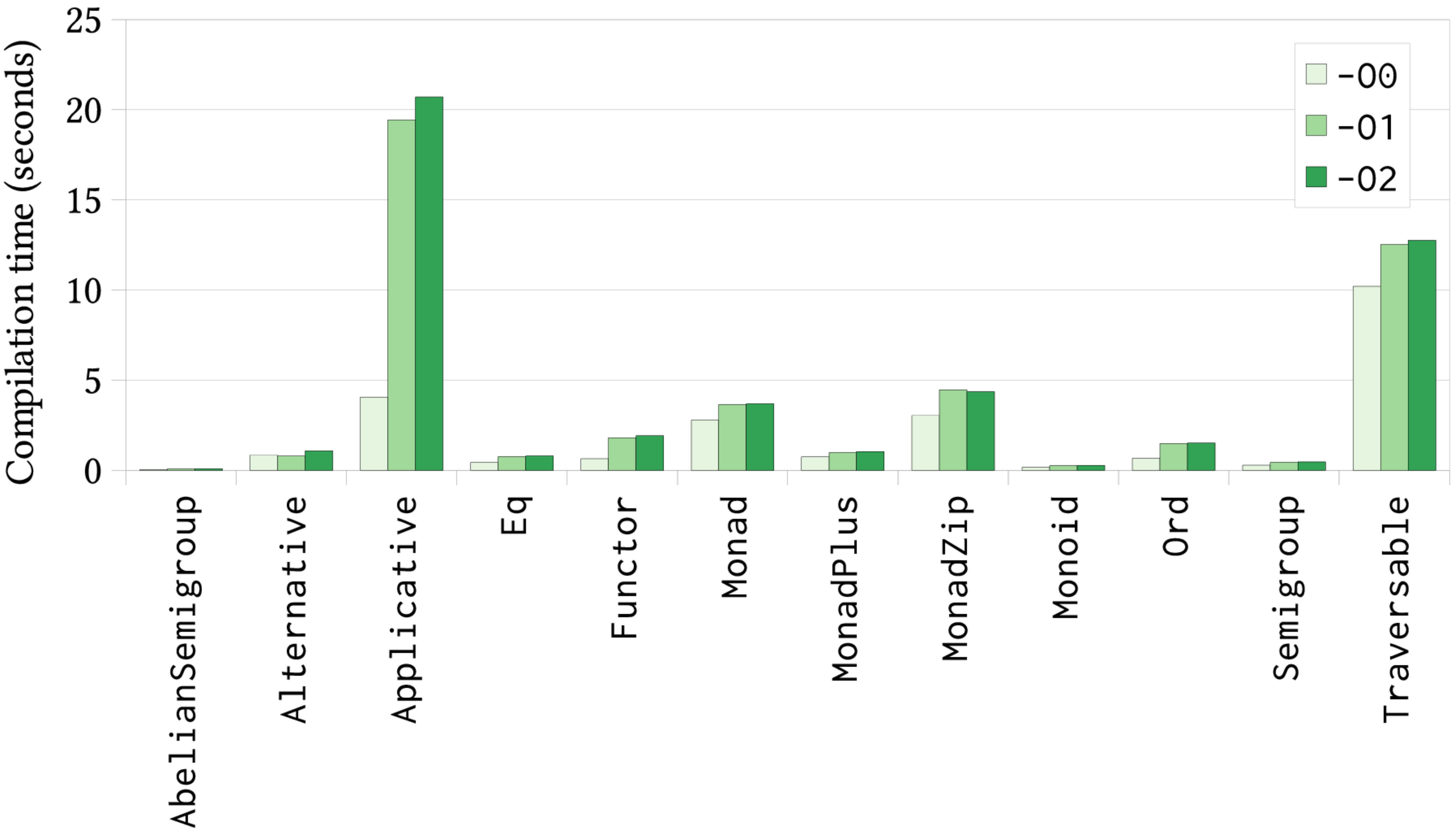
base: Basic libraries

[bsd3, library, prelude] [Propose Tags]

This package contains the Standard Haskell Prelude and its support libraries, and a large collection of useful libraries ranging from data structures to parsing combinators and debugging utilities.


Hackage :: [Package]
Functor
Apply
Semigroup
Comonad
Applicative
Monoid
Category
Alternative
Foldable
Monad
Arrow
ArrowZero
ArrowPlus
Traversable
MonadFix
MonadPlus
ArrowApply
ArrowChoice
ArrowLoop


Hackage :: [Package]
Functor
Apply
Semigroup
Comonad
Applicative
Monoid
Category
Alternative
Foldable
Monad
Arrow
ArrowZero
ArrowPlus
Traversable
MonadFix
MonadPlus
ArrowApply
ArrowChoice
ArrowLoop



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

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