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 representation type, we derive proofs for the original datatype by reusing the corresponding proof over the representation type. Our work is designed to be general-purpose and does not require advanced automation techniques such as tactic systems. As evidence for this claim, we implement these ideas in a Haskell library that defines generic, canonical implementations of the methods and proof obligations for classes in the standard `base` library.

CCS Concepts • **Software and its engineering → Functional languages**; *Data types and structures.*

Keywords Type classes, generic programming, dependent types, reuse

ACM Reference Format:
Ryan G. Scott and Ryan R. Newton. 2019. Generic and Flexible Defaults for Verified, Law-Abiding Type-Class Instances. In *Proceedings of the 12th ACM SIGPLAN International Haskell Symposium (Haskell '19), August 22–23, 2019, Berlin, Germany.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3331545.3342591

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Haskell '19, August 22–23, 2019, Berlin, Germany
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6813-1/19/08…$15.00
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 *algebraic laws* that every class instance must obey. In a simply-typed setting, checking whether a given instance satisfies these laws is not straightforward, so these laws usually are presented as comments that the user must keep in mind, informally, when writing code. For example, the following is an abridged version of Haskell's `Ord` class along with one of its laws:

```
-- Transitivity: if x ≤ y && y ≤ z = True,
--               then x ≤ z = True
class Ord a where
  (≤) :: a → a → Bool
```

In an ordinary Haskell setting, this important transitivity law languishes as an unenforceable comment, and unless programmers are diligent, they may accidentally produce unlawful `Ord` instances. With dependent types, however, the statement of this law can be encoded in the type signature of a class method, and anyone who defines an instance of the class must write a proof for it, ensuring that unlawful instances will be rejected from the get-go. For instance, one could envision a version of `Ord` that bundles along the proof of `(≤)`'s transitivity:

```
class Ord a where
  (≤) :: a → a → Bool
  leqTransitive :: Π (x, y, z :: a)
                → (x ≤ y) :~: True
                → (y ≤ z) :~: True
                → (x ≤ z) :~: True
```

Unfortunately, writing proofs in class instances can be laborious, as many of these proofs require excessive amounts

of boilerplate code. Even worse, the size of some proofs can grow super-linearly in the size of the datatypes used, as the length of the proofs can grow extremely quickly due to the sheer number of cases one has to exhaust.

Many dependently typed languages automate the generation of boilerplate code by providing tactics, as in Coq or Lean. While heavy use of tactics can make swift work of many type-class proofs, they can sometimes make it tricky to update existing code, as basic changes to the code being verified can cause theorems that rely on the code's implementation details to suddenly fail. Moreover, not every dependently typed language is equipped with an advanced tactic system. Porting tactic-rich proofs in one language to a different language that lacks them can be nontrivial, especially if the tactics automate away key steps in the proofs behind the scenes.

Our work aims to introduce a general technique for eliminating boilerplate code for type class laws that does not rely on tactics as the primary vehicle for proof automation. Instead, we adopt techniques from the field of datatype-generic programming—in particular, a dialect of generic programming that makes use of *pattern functors* [17, 20, 36]. We leverage pattern functors to decompose proofs into the smallest algebraic components possible, compose larger proofs out of these verified building blocks, and then define instances for full datatypes by *reusing* the proofs over the pattern functor types.

In fact, we can package up this style of code reuse into generic default implementations of type class laws. These defaults are flexible enough to work over any data type that has an equivalent *representation type* built out of pattern functors. In general, there may be many ways to define an instance such that it abides by the class's laws (see Section 3.3), but generic defaults *à la* pattern functors allow programmers to abstract out canonical, "off-the-shelf" implementations that work over a wide variety of data types. These canonical defaults are highly useful—for instance, the *n*-body simulation in Vazou et al. [31] uses a canonical instance of the `Monoid` class—so they receive the most attention in our work. In particular, the primary contributions of this paper are:

- We show how to derive implementations of type-class methods *and* laws with the pattern functor approach to datatype-generic programming (Section 3).
- We provide the first datatype-generic approach to verifying class laws that accommodates *existing* instances (not written in any special style), while still providing substantial proof automation (Section 4).
- We present a prototype implementation of these ideas and evaluate the lines of code required and impact on compile time (Section 5).

Our prototype implementation is a Haskell library, which we call `verified-classes`. The `verified-classes` library provides generic implementations of laws for several type classes in Haskell's `base` library (previously asserted as unverified comments). We additionally consider what an implementation of these ideas would look like in other languages in Section 6.

2 Background

Dependent types offer a vehicle for verifying type class laws by simply defining additional class methods with proof obligations. In this section, we flesh out a complete example of a verified type class and demonstrate how one can define instances for it using progressively larger datatypes. As the size of the datatypes increases, it will become apparent that there is a problem of scale, since the number of cases required to complete the corresponding proofs increases quadratically.

2.1 A Tour of Verified Classes

The goal of our work is to present a technique that can be ported to any dependently typed programming language that supports type classes (a topic that we will revisit in Section 6). In pursuit of this goal, in this paper we adopt the convention of using a minimalistic language that resembles Haskell. We say "resembles Haskell" rather than "is Haskell" since, in practice, our evaluation uses the singletons technique [13] to encode dependent types in Haskell. This process is quite straightforward, and we invite the interested reader to Appendix A.1 [21] for the full details of how this works. However, the singletons encoding introduces a certain degree of syntactic noise, so for the sake of presentation clarity, we use a syntax that is closer to most other dependently typed languages. (One can imagine that we are writing code in a hypothetical Dependent Haskell [12].)

2.1.1 Classes

As a running example, we use a minimal version of the `Ord` type class, which describes datatypes that support boolean-returning comparison operations:

```
class Eq a ⇒ Ord a where
  (≤) :: a → a → Bool
```

Where `Eq` and `Bool` are defined to be:

```
class Eq a where
  (==) :: a → a → Bool

data Bool = False | True
```

In order for a datatype with an `Ord` instance to be considered verified, we require that the implementation of $(\leq)$ must behave like a total order. That is to say, it must satisfy the following four laws:

$$\begin{array}{rl} \textit{Reflexivity} & \forall x.\ x \leq x \\ \textit{Transitivity} & \forall x, y, z.\ x \leq y \leq z \Rightarrow x \leq z \\ \textit{Antisymmetry} & \forall x, y.\ x \leq y \wedge y \leq x \Rightarrow x = y \\ \textit{Totality} & \forall x, y.\ x \leq y \vee y \leq x \end{array}$$

Using dependent types, we encode these laws as proof methods of a type class. We define a new class, `VOrd` (short for

"verified Ord"), to represent the group of Ord instances that are known to be lawful total orders:[1]

```
class Ord a ⇒ VOrd a where
  leqReflexive :: Π (x :: a) → (x ≤ x) :~: True
  leqTransitive
    :: Π (x, y, z :: a)
    → (x ≤ y) :~: True → (y ≤ z) :~: True
    → (x ≤ z) :~: True
  leqAntisymmetric
    :: Π (x, y :: a)
    → (x ≤ y) :~: True → (y ≤ x) :~: True
    → (x == y) :~: True
  leqTotal
    :: Π (x, y :: a)
    → Either ((x ≤ y) :~: True)
             ((y ≤ x) :~: True, (x == y) :~: False)
```

The methods of VOrd translate the laws above into type signatures, making use of the propositional equality type (:~:), which reflects the judgment that two types are equal into a type of its own:

```
data (:~:) :: ∀ k. k → k → Type where
  Refl :: a :~: a
```

For the sake of making these type signatures a little more brief, we introduce an auxiliary definition that reflects True-valued boolean judgments into types:

```
data So :: Bool → Type where
  Oh :: So True
```

Using So, we can tighten up the presentation of VOrd somewhat:

```
class Ord a ⇒ VOrd a where
  leqReflexive :: Π (x :: a) → So (x ≤ x)
  leqTransitive
    :: Π (x, y, z :: a)
    → So (x ≤ y) → So (y ≤ z) → So (x ≤ z)
  leqAntisymmetric
    :: Π (x, y :: a)
    → So (x ≤ y) → So (y ≤ x) → So (x == y)
  leqTotal
    :: Π (x, y :: a)
    → Either (So (x ≤ y))
             (So (y ≤ x), So (not (x == y)))
```

2.1.2 Examples of Instances

To demonstrate VOrd in action, we first define an extremely simple datatype, along with accompanying instances of Eq and Ord:

```
data T a = MkT1 a
instance Eq a ⇒ Eq (T a) where
  (MkT1 x) == (MkT1 y) = (x == y)
instance Ord a ⇒ Ord (T a) where
  (MkT1 x) ≤ (MkT1 y) = x ≤ y
```

Unsurprisingly, this implementation of (≤) comprises a total order. We verify this claim by implementing a VOrd instance for T like so:

```
instance VOrd a ⇒ VOrd (T a) where
  leqReflexive (MkT1 x)
    | Oh ← leqReflexive x = Oh
  leqTransitive (MkT1 x) (MkT1 y) (MkT1 z) Oh Oh
    | Oh ← leqTransitive x y z Oh Oh = Oh
  leqAntisymmetric (MkT1 x) (MkT1 y) Oh Oh
    | Oh ← leqAntisymmetric x y Oh Oh = Oh
  leqTotal (MkT1 x) (MkT1 y) =
    case leqTotal x y of
      Left Oh        → Left Oh
      Right (Oh, Oh) → Right (Oh, Oh)
```

The implementations of these proofs outline the basic approach to verifying algebraic properties. For each property we wish to prove, we proceed by cases on T (encoded using pattern matching), appeal to our induction hypothesis (encoded using recursion), and then complete the proof.

This instance was relatively painless to write as there was only one case to consider, the MkT1 constructor of T. However, the size of the proofs grows noticeably when we increase the size of T. For instance, consider what happens when we add another constructor:

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

First, we must adjust our Eq and Ord instances accordingly. We pick the convention that MkT1 is always less than MkT2:

```
instance Eq a ⇒ Eq (T a) where
  (MkT1 x) == (MkT1 y) = (x == y)
  (MkT2 x) == (MkT2 y) = (x == y)
  (MkT1 _) == (MkT2 _) = False
  (MkT2 _) == (MkT1 _) = False
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
```

Next, now that the code that we're verifying has changed, we must synchronize the accompanying proofs in the VOrd instance. We start with the proof of reflexivity:

```
instance VOrd a ⇒ VOrd (T a) where
  leqReflexive (MkT1 x)
    | Oh ← leqReflexive x = Oh
  leqReflexive (MkT2 x)
    | Oh ← leqReflexive x = Oh
  ...
```

[1] Alternatively, we could inline the methods of VOrd directly into Ord. We choose not to do so here since many languages, such as Haskell, make use of certain unlawful instances, such as the Ord instances for Floats and Doubles.

Updating `leqReflexive` ended up not requiring that much effort, as we only needed to have one extra case for the additional constructor `MkT2`. Updating `leqTransitive`, however, requires more effort:

```
instance VOrd a ⇒ VOrd (T a) where
  ...
  leqTransitive t t' t'' Oh Oh =
    case (t, t', t'') of
      (MkT1 x, MkT1 y, MkT1 z)
        | Oh ← leqTransitive x y z Oh Oh = Oh
      (MkT2 x, MkT2 y, MkT2 z)
        | Oh ← leqTransitive x y z Oh Oh = Oh
      (MkT1 _, _, MkT2 _) = Oh
  ...
```

In addition to requiring an extra case for the combination of arguments where all of them are `MkT2`, we now require an additional case to cover the two combinations of arguments where the first argument is `MkT1` and the third argument is `MkT2`. The four other possible combinations of arguments were ruled out by dependent pattern matching, as the type system concluded that they were impossible-to-reach cases (for example, if `t = MkT2 _` and `t'' = MkT1 _`, then it cannot be the case that `t ≤ t''` is `True`). In Haskell, we do not need to write these cases out at all, although some other languages may require explicit "absurd" cases (e.g., using the `()` pattern in Agda).

Having to add one more case might not seem that burdensome, but the number of cases one has to supply for `leqTransitive` grows quickly as we add more and more constructors. For example, if we added a third constructor `MkT3` to `T`, then after updating the `Eq` and `Ord` instances (using the convention that `MkT1` is always less than `MkT2`, and `MkT2` is always less than `MkT3`), this is what the proof of transitivity would become:

```
instance VOrd a ⇒ VOrd (T a) where
  ...
  leqTransitive t t' t'' Oh Oh =
    case (t, t', t'') of
      (MkT1 x, MkT1 y, MkT1 z)
        | Oh ← leqTransitive x y z Oh Oh = Oh
      (MkT2 x, MkT2 y, MkT2 z)
        | Oh ← leqTransitive x y z Oh Oh = Oh
      (MkT3 x, MkT3 y, MkT3 z)
        | Oh ← leqTransitive x y z Oh Oh = Oh
      (MkT1 _, _, MkT2 _) = Oh
      (MkT1 _, _, MkT3 _) = Oh
      (MkT2 _, _, MkT3 _) = Oh
  ...
```

This time, we have *six* cases! If we add a fourth constructor, we would have 10 cases, and if we add a fifth constructor, we would have fifteen cases. In general, if we had n constructors, then `leqTransitive` would require $n + \binom{n}{2} = \frac{1}{2}(n^2 + n)$ cases, which grows quadratically! This has the potential to make scaling up proofs extremely tiresome.

```
class Generic a where
  type Rep a :: Type
  from :: a → Rep a
  to   :: Rep a → a

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

Figure 1. A slightly simplified presentation of the definitions from Haskell's `GHC.Generics` library, on which we base our approach to datatype-generic programming.

Perhaps even more troublesome than the size of these proofs themselves is the fact that most of these cases are sheer boilerplate. For instance, `leqTransitive` follows a very predictable pattern. For combinations of arguments where all the constructors are the same, recurse, and for combinations where there are different constructors, it must be the case that the first argument is less than or equal to the third argument, so immediately return `Oh`. This is routine code that is begging to be automated with a proof-reuse technique.

3 Verified Instances, Generically

Having seen the tedium of manually constructing certain proofs, we present a solution. Notably, our solution does not require a tremendous amount of support from the language itself (in terms of tactics or metaprogramming). Instead, we rely on techniques that could be ported to any dependently typed programming language that, at a minimum, supports type classes.[2]

We adapt an approach from the field of datatype-generic programming where we take an algebraic datatype and construct a *representation type* which is isomorphic to it. The representation type itself is a composition of smaller *pattern functor* datatypes that encode primitive "building blocks" of other datatypes, such as products, sums, and individual fields. We also establish a type class for witnessing the isomorphism between a datatype and its representation type [17, 20, 36].

By leveraging these tools, we shift the burden of proof from the original datatype (which may be arbitrarily complex) to the handful of simple pattern functor types that constitute its representation type. This way, we are able to prove properties for a wide range of possible datatypes by simply proving the same properties for a finite number of "building block" types.

3.1 A Primer on Datatype-Generic Programming

To build up representation types, we build upon the techniques from Magalhães et al. [17], which influenced the design of Haskell's popular `GHC.Generics` library. Figure 1

[2] It is possible to further reduce the amount of code needed by generating boilerplate code with metaprogramming techniques (e.g., Template Haskell [24]), but support for metaprogramming is not a strict requirement.

presents the central definitions used in this style of datatype-generic programming.

The `Generic` type class is the focal point of the library. A datatype with a `Generic` instance comes equipped with a `Rep`, the representation type, and functions `from` and `to` which witness the isomorphism between the representation type and the original datatype. The `Rep` type is always some combination of the following pattern functor types[3]:

1. The `U1` type represents constructors with no fields.
2. The `K1` type represents a single field in a constructor.
3. The `(:*:)` type (pronounced "product") represents two consecutive fields in a constructor.
4. The `(:+:)` type (pronounced "sum") represents the choice between two consecutive constructors.

This is best explained with an example, so recall this version of the `T` data type from Section 2.1.2:

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

We define its canonical `Generic` instance like so:

```
instance Generic (T a) where
  type Rep (T a) = K1 a :+: K1 a
  from (MkT1 x) = L1 (MkK1 x)
  from (MkT2 x) = R1 (MkK1 x)
  to (L1 (MkK1 x)) = MkT1 x
  to (R1 (MkK1 x)) = MkT2 x
```

Here, we see that because `T` has two constructors, `MkT1` and `MkT2`, the `(:+:)` type is used once to represent the choice between them. The field of type `a` in each constructor is, in turn, represented with a `K1` type. The same sort of pattern would follow if we added more constructors to `T`. For example, here is how the `Rep` instance would appear if there were three constructors [4]:

```
data T a = MkT1 a | MkT2 a | MkT3 Bool
instance Generic (T a) where
  type Rep (T a) = K1 a :+: (K1 a :+: K1 Bool)
  ...
```

Each constructor corresponds to another use of `(:+:)` to denote another choice of constructor. Despite the size of the `T` type increasing, the number of distinct datatypes in its `Rep` has *not* changed, which is an important property.

The implementations of `from` and `to` are are entirely mechanical to implement and constitute one of the few sources of boilerplate in this style of datatype-generic programming. Languages like Haskell offer a metaprogramming facility (**`deriving Generic`**) for generating this boilerplate, although it can just as well be written by hand.

[3] There is another pattern functor type, `M1`, which is used to attach metadata such as constructor names, fixities, etc. For the sake of simplicity, we leave it out of the discussion in this section, but we revisit it in 5.1.

[4] Although `(:+:)` is right-associative, we have added explicit parentheses in this `Rep` instance for clarity. In general, a canonical `Generic` instance balances nested uses of `(:+:)` and `(:*:)` so that it is always possible to go from the root of the representation type to a leaf in logarithmic (rather than linear) time.

3.2 Verifying `Generic`

While `Generic` is convenient for quickly coming up with representation types, it alone isn't enough for our needs, as we need to be able to prove that `from` and `to` form an isomorphism. In pursuit of that goal, we define a subclass of `Generic` with two proof methods that state that `from` and `to` are mutual inverses:

```
class Generic a ⇒ VGeneric a where
  to∘f :: Π (z :: a) → to (from z) :~: z
  fo∘t :: Π (r :: Rep a) → from (to r) :~: r
```

Like `Generic`, instances of `VGeneric` are predictably straightforward. Here is an example of instance for `T` (with two constructors):

```
instance VGeneric (T a) where
  to∘f (MkT1 _) = Refl
  to∘f (MkT2 _) = Refl
  fo∘t (L1 (MkK1 _)) = Refl
  fo∘t (R1 (MkK1 _)) = Refl
```

The implementations of `to∘f` and `fo∘t` are the other sources of boilerplate besides `from` and `to`. For this reason, we expose Template Haskell [24] functionality in the `verified-classes` library to automatically generate `VGeneric` instances.

The class `VGeneric` plays double duty in the style of proofs we write. One of its roles is to serve as a tool for "cancelling out" compositions of `from` and `to`, as the need often arises to simplify `to (from z)` into `z` or `from (to r)` into `r` when reasoning about generic implementations of class methods. It also serves the role of ensuring that the `Generic` instance we are using to go between a datatype and its `Rep` is a legitimate isomorphism. Even if the `Generic` instance we are using is generated behind the scenes (say, with **`deriving Generic`**), we can use `VGeneric` as an additional sanity check to ensure that the `Generic` automation is functioning properly.

3.3 Orderings on Representation Types

We have now identified the four basic datatypes in Figure 1 that can be composed in various ways to form representation types in `Generic` instances, as well as a mechanism to verify that the choice of representation type truly forms an isomorphism. The next step is to utilize these tools to write verified `Ord` instances, and by doing so, demonstrate how to obtain a valid total ordering for any algebraic data type using this technique.

First, we define a generic version of $(\leq)$ by defining `Ord` instances for the pattern functor types. The instances for `U1` and `K1` are quite straightforward:

```
instance Ord U1 where
  MkU1 ≤ MkU1 = True
instance Ord c ⇒ Ord (K1 c) where
 (MkK1 x) ≤ (MkK1 y) = (x ≤ y)
```

It is worthwhile to take the time to reflect on what these instances mean from a datatype-generic programming perspective. The instance for `U1` abstracts the idea that a constructor with no fields is always less than or equal to itself, and the instance for `K1` abstracts the idea that for constructors with fields, the comparing them amounts to comparing the constituent fields. The idea of "comparing the constituent fields" becomes more precise when we define the `Ord` instance for `(:*:)`:

```
instance (Ord a, Ord b) ⇒ Ord (a :*: b) where
  (x1 :**: y1) ≤ (x2 :**: y2) =
    if (x1 == x2) then (y1 ≤ y2) else (x1 ≤ x2)
```

This instance abstracts the idea that we use a lexicographic ordering on products. That is, we check the first fields of each constructor, and if they are the same, skip them and proceed to the next fields. If they are not the same, return the result of comparing them. Importantly, this approach scales to any number of fields, as this instance iterates over nested uses of `(:*:)` to process the remaining fields.

Finally, we encode how to compare multiple constructors with `(:+:)`:

```
instance (Ord a, Ord b) ⇒ Ord (a :+: b) where
  (L1 x) ≤ (L1 y) = (x ≤ y)
  (R1 x) ≤ (R1 y) = (x ≤ y)
  (L1 _) ≤ (R1 _) = True
  (R1 _) ≤ (L1 _) = False
```

This instance abstracts the convention that constructors defined earlier in a datatype's definition are always less than constructors defined later. Moreover, comparing two of the same constructor amounts to comparing their respective fields.

Defining these four `Ord` instances for the pattern functor types means that any datatype equipped with a `Generic` instance can derive an `Ord` instance cheaply. This is because it is possible to define an implementation of `(≤)`, which we call `genericLeq`, that works for any instance of `Generic`:

```
genericLeq :: (Generic a, Ord (Rep a))
           ⇒ a → a → Bool
genericLeq x y = (from x ≤ from y)

instance Ord a ⇒ Ord (T a) where
  (≤) = genericLeq
```

This works because we can define an order over a datatype in terms of the ordering on its representation type, to which it is isomorphic. Previously, we would have had to implement `Ord` once per datatype, with each implementation possibly requiring several cases. With datatype-generic programming, we can reduce the implementation burden to defining `(≤)` for each pattern functor (a one-time cost), defining a `Generic` instance per datatype (which are simple and can be automated), and defining an `Ord` instance per datatype using `genericLeq` (which only requires one line per type).

It is worth clarifying that `genericLeq` is "generic" in the sense that it provides a *canonical* implementation of a total ordering on datatypes. It is canonical in the sense that it is law-abiding and works for a wide variety of datatypes, even if they have different numbers of constructors or fields. That is not to say that this is the *only* valid total order in existence. For instance, we could choose a reverse lexicographic ordering that treats rightmost constructors as always being less than leftmost constructors. We could certainly accommodate datatypes with this ordering by inventing a `ReverseLexicoOrd` class and defining appropriate instances of it for the pattern functor types, but in general, there might be arbitrarily many law-abiding implementations of a class's methods.

In this work, we are primarily concerned with canonical implementations of class methods, as they reflect "off-the-shelf" solutions that programmers reach for when they want to define instances for their data types in a cheap-and-cheerful manner. Therefore, we restrict our focus to one generic default per class method, even though many more legitimate defaults may exist.

3.4 Proofs over Representation Types

Just as we defined a generic version of `(≤)` over pattern functors, so too can we define generic versions of the total order laws by defining `VOrd` instances. Here is a subset of the `VOrd` instance for sums:

```
instance (VOrd a, VOrd b) ⇒ VOrd (a :+: b) where
  ...
  leqTransitive s s' s'' Oh Oh =
    case (s, s', s'') of
      (L1 x, L1 y, L1 z)
        | Oh ← leqTransitive x y z Oh Oh = Oh
      (R1 x, R1 y, R1 z)
        | Oh ← leqTransitive x y z Oh Oh = Oh
      (L1 _, _, R1 _) = Oh
  ...
```

Astute readers will notice that this is the exact same as a `VOrd` instance that we defined in Section 2.1.2. However, note that this is the *only* instance of `VOrd` that we need to provide for a sum type. Moreover, `(:+:)` is in some ways the "simplest" possible sum type, so this allows us to manage their complexity much more than if we were directly writing a `VOrd` instance for a datatype with many constructors.

Another interesting `VOrd` instance is that for `K1`, as these are found at the leaves of a tree of pattern functor types:

```
instance VOrd c ⇒ VOrd (K1 c) where
  ...
  leqTransitive k k' k'' Oh Oh =
    case (k, k', k'') of
      (MkK1 x, MkK1 y, MkK1 z)
        | Oh ← leqTransitive x y z Oh Oh = Oh
  ...
```

In terms of code, this instance is quite simple, as it simply uses the underlying `VOrd c` instance to complete the proof of transitivity. What may be less obvious is that the `c` type no longer contains any pattern functor types, as this is point where we switch to the fields of the datatype's constructors themselves. Put another way, the generic computation "bottoms out" at occurrences of `K1`—or, if a constructor has no fields, it bottoms out at `U1`.

In addition to `(:+:)` and `K1`, we prove the total order laws for the remaining two pattern functor types. Once this is done, we obtain for free proofs that any `Rep` in a `Generic` instance is a valid total ordering, as we can compose these four instances as desired to obtain a valid `VOrd` instance. This is crucial, as it ensures that this technique scales up to whatever size of datatype that we might envision.

3.5 Carrying Over Proofs

Given a `VOrd` instance for a representation type, how can we relate it back to the original datatype? This is where the `VGeneric` class becomes important. `VGeneric` gives us precisely enough power to take a `VOrd` proof for one type and reuse it for the other type, in either direction.

As a first example, we demonstrate how to define a generic implementation of `leqTransitive` for any type that implements a `VGeneric` instance. Intuitively, we want to show that a total order on `a` is transitive by appealing to the fact that the total order on `Rep a` (to which `a` is isomorphic) is transitive. Computationally, this amounts to taking each of the three arguments of type `a`, converting them to their `Rep` counterparts with `from`, and invoking `leqTransitive` on them. The general structure of this function looks like this:

```
defaultLeqTransitive
  :: Π (x, y, z :: a)
  → (VGeneric a, VOrd (Rep a))
  ⇒ So (x ≤ y) → So (y ≤ z) → So (x ≤ z)
defaultLeqTransitive x y z xLeqY yLeqZ =
  leqTransitive (from x) (from y) (from z)
                xLeqY yLeqZ
```

This would *almost* typecheck save for one issue: the final two arguments we are attempting to pass to `leqTransitive` are of types `So (x ≤ y)` and `So (y ≤ z)`, but since the first three arguments involve `from`, this requires that the last two arguments should *actually* have the types `So (from x ≤ from y)` and `So (from y ≤ from z)`, respectively.

What goes wrong in this example? The problem is that while we attempt to appeal to the transitivity of representation types, the type signature does not reflect this. A proof that `x ≤ y`, for instance, does not help much when the proof that we actually care about involves `from x` and `from y`. In other words, we need to find some way to relate the behavior of `(≤)` over type `a` to the behavior of `(≤)` over type `Rep a`.

There is a crude trick available that seemingly gets the job done: we can assume that the implementations of `(≤)` for `a` and `Rep a` coincide exactly. That is to say, instead of using `(≤)` in the type signature of `defaultLeqTransitive`, which is not precise enough, we instead use `genericLeq` from Section 3.3. Recall that `genericLeq x y = (from x ≤ from y)`, which is exactly what we need here. Therefore, we need only change the type of `defaultLeqTransitive` slightly:

```
defaultLeqTransitive
  :: Π (x, y, z :: a)
  → (VGeneric a, VOrd (Rep a))
  ⇒ So (genericLeq x y) → So (genericLeq y z)
  → So (genericLeq x z)
```

This suffices to make `defaultLeqTransitive` typecheck, but it comes with a somewhat steep price, as it requires that `(≤)` be implemented exactly the same way as `genericLeq`. We will revisit this limitation in Section 4, and show how to overcome it. For the remainder of this section, it suffices to assume that `(≤)` and `genericLeq` are definitionally equal.

We use this trick to define default implementations of the other laws in `VOrd` as well. Using these defaults, we can derive the total order laws much more easily than we could before. For example, showing that `T` has a lawful total order now amount to very little code:

```
instance VOrd a ⇒ VOrd (T a) where
  ...
  leqTransitive = defaultLeqTransitive
  ...
```

The generic defaults for `VOrd` ended up not making use of any specific laws for `VGeneric`, but this is not always the case for every class. A more complicated example is found in Figure 2, which defines the `Semigroup` class for datatypes supporting binary, associative operations. In order to define a generic proof that this operation is associative, we must make use of the fact that `from ∘ to = id`:

```
defaultAssociative
  :: Π (x, y, z :: a)
  → (VGeneric a, VSemigroup (Rep a))
  ⇒ (genericAppend x (genericAppend y z)) :~:
    (genericAppend (genericAppend x y) z)
defaultAssociative x y z
  | Refl ← fot (from x <> from y)
  , Refl ← fot (from y <> from z)
  = associative (from x) (from y) (from z)
```

Using `fot` as a lemma guides the typechecker to realize that the conclusion reduces to `(to (from x <> (from y <> from z))) :~: (to ((from x <> from y) <> from z))`. From there, appealing to the associativity of `(<>)` causes both sides of the equation to become equal. This example demonstrates that the `VGeneric` laws not only give us peace of mind that the conversions to and from a representation type are structure-preserving, but that they are useful computational tools for deriving generic proofs.

```
class Semigroup a where
  (<>) :: a → a → a
class Semigroup a ⇒ VSemigroup a where
  associative :: Π (x, y, z :: a)
              → (x <> (y <> z)) :~: ((x <> y) <> z)
```

```
genericAppend
  :: (Generic a, Semigroup (Rep a))
  ⇒ a → a → a
genericAppend x y =
  to (from x <> from y)
```

Figure 2. The `Semigroup` class, its associativity law (in `VSemigroup`), and a generic implementation of `(<>)`.

4 More Permissive Generic Defaults

In Section 3, we demonstrated how datatype-generic programming could derive verified proofs of type class laws in addition to the implementations of the class methods themselves. But the utility of these proofs is still somewhat limited. When defining generic defaults for laws, we assumed (in Section 3.4) that the implementation of the class methods involved were *exactly* the same as the datatype-generic versions. While this worked out well enough for the one `Ord`/`VOrd` example we used in Section 3.5, this approach has a serious flaw: it cannot work if the implementation of `(≤)` is anything other than `genericLeq`. As a consequence, there are large swaths of `Ord` instances in the wild that cannot benefit from generic proofs, since their `Ord` instances weren't designed to accommodate it.

With a little modification, however, we can make the generic proofs more inclusive. The key insight is that the implementation of `(≤)` doesn't need to be the exact same as `leqGeneric`—it only needs to behave the same. In other words, all we need to be able to show that `x ≤ y` is equivalent to `leqGeneric x y` for all `x` and `y`. We factor out this task into its own type class, `GOrd` (short for "generic `Ord`"):

```
class (Generic a, Ord a, Ord (Rep a)) ⇒ GOrd a where
  genericLeqC :: Π (x, y :: a)
              → (x ≤ y) :~: (genericLeq x y)

defaultLeqTransitive
  :: Π (x, y, z :: a)
  → (VGeneric a, VOrd (Rep a), GOrd a)
  ⇒ So (x ≤ y) → So (y ≤ z) → So (x ≤ z)
defaultLeqTransitive x y z xLeqY yLeqZ
  | Refl ← genericLeqC x y
    -- To conclude that
    -- (So (x ≤ y)) equals (So (genericLeq x y))
  , Refl ← genericLeqC y z
    -- To conclude that
    -- (So (y ≤ z)) equals (So (genericLeq y z))
  , Refl ← genericLeqC x z
    -- To conclude that
    -- (So (x ≤ z)) equals (So (genericLeq x z))
  = leqTransitive (from x) (from y) (from z)
                  xLeqY yLeqZ
```

In order to use `defaultLeqTransitive`, one must now provide an appropriate instance of `GOrd`. For datatypes like `T` where `(≤) = genericLeq`, defining a `GOrd` instance is a trivial task, since `genericLeqC` can be implemented simply as `λ_ _ → Refl`. For other datatypes, more work is required, although the amount of code one has to write to implement `GOrd` generally pales in comparison to the amount of code saved by using the generic proofs in the first place. As one example, here is how one could use generic proofs for a `Bool` datatype whose `Ord` instance is defined in a non–datatype-generic fashion:

```
instance Ord Bool where
  True  ≤ True  = True
  False ≤ False = True
  False ≤ True  = True
  True  ≤ False = False
instance Generic Bool where
  type Rep Bool = ...
  from          = ...
  to            = ...
instance GOrd Bool where
  genericLeqC True  True  = Refl
  genericLeqC False False = Refl
  genericLeqC False True  = Refl
  genericLeqC True  False = Refl
instance VOrd Bool where
  ...
  leqTransitive = defaultLeqTransitive
  ...
```

There are a number of situations where this more permissive form of generic defaults could be desirable. We consider examples of such situations in the remainder of this section.

4.1 Proofs for Instances Defined Elsewhere

It is often the case that one wishes to prove properties of datatypes for which the implementation of their instances is unchangeable. For instance, a datatype instance might be defined in a library over which the proof author has no control, such as Haskell's `base` library. An example of this is the `Ord` instance for `Bool`, which `base` defines in much the same way as it is presented in Section 4, without using datatype-generic programming. [5] It is unlikely that the maintainers of

[5] Strictly speaking, this `Ord Bool` instance is defined using the `deriving` mechanism, although the generated code would be similar in structure.

base will change the implementation of this instance just to make verifying it easier, but thankfully, this is not an issue in practice. Having permissive defaults for VOrd enables users to write generic proofs for it without having to patch the source code of base.

The verified-classes library uses this technique in various places to help write proofs for data types in base, including proofs of the laws for the standard list type's Eq instance and the laws for the Compose type's Applicative instance, where Compose is defined as follows:

```
newtype Compose f g a = MkCompose (f (g a))
```

In the particular case of Compose, being able to use generic defaults is a big win in terms of code size. Writing an instance of GApplicative for Compose (which gives rise to a VApplicative instance that verifies the Applicative laws) only requires 109 lines of code. Writing the full VApplicative proofs out by hand, however, would require 280 lines of code.

4.2 Performance-Critical Code

There are some situations in which compilers are unable to optimize away the runtime overhead that datatype-generic programming can introduce [16, 18]. For these performance-sensitive situations, this technique allows one to define class methods in a performant way, without datatype-generic programming, while still benefitting from the proof automation that it provides. Even if the proofs themselves are defined using datatype-generic programming, as long as they are used in a runtime-irrelevant fashion, they do not risk introducing extra runtime costs.

4.3 Working with Standards

Sometimes, the implementations of class methods are expected to adhere to a common standard. For instance, consider the Traversable class from the Haskell base library:

```
class Traversable t where
  traverse :: Applicative f
           ⇒ (a → f b) → t a → f (t b)
```

The documentation for Traversable[6] requires that implementations of traverse must satisfy traverse MkIdentity = MkIdentity, where Identity must be defined as the following datatype with this Applicative instance:

```
newtype Identity a = MkIdentity a
instance Applicative Identity where
  pure x = MkIdentity x
  (MkIdentity f) <*> (MkIdentity x) =
    MkIdentity (f x)
```

While this Applicative Identity instance is convenient for specifying the Traversable laws, it enforces very particular implementations of pure and (<*>) that do not datatype-generic programming techniques. Moreover, Identity is a very commonly used datatype in the Haskell ecosystem in its own right, so it is conceivable that one may wish to verify its Applicative instance. It would be a shame if we could not verify this instance generically simply because its documentation required that it be implemented in a certain way.

Fortunately, we do not have to sacrifice convenience in the name of standardization. Instead, we simply define generic defaults for VApplicative by leveraging a GApplicative class. This way, deriving proofs of the Applicative laws for Identity is tantamount to defining a GApplicative Identity instance, which is not challenging. Indeed, this is the approach that the verified-classes library adopts.

5 Evaluation

We demonstrate the effectiveness of the techniques in the paper by implementing them in a library, verified-classes. This library demonstrates the practical utility of these ideas by finding as many type classes with unchecked proof obligations as possible from Haskell's standard base package and defining generic versions of each law. It also offers insight into the compile-time performance of writing proofs in this style.

5.1 The verified-classes Library

Figure 3 categorizes every type class from base for which verified-classes implements generic versions of its associated laws. [7] The classes in this figure are divided into two groups: classes whose argument is of kind Type, and classes whose argument is of kind Type → Type. The former groups of classes are handled with the Generic machinery introduced in Section 3.1, whereas the latter group of classes are handled by a slight variation of Generic, called Generic1. Generic1 features mostly cosmetic changes from Generic to support datatypes of kind Type → Type:

```
class Generic1 (f :: Type → Type) where
  type Rep1 f :: Type → Type
  from1 :: f a → Rep1 f a
  to1   :: Rep1 f a → f a
```

Just as Generic has a verified counterpart in VGeneric, so too does there exist a VGeneric1 class to ensure that a Generic1 instance's implementations of from1 and to1 form a valid isomorphism:

```
class Generic1 f ⇒ VGeneric1 f where
  to∘f1 :: ∀ a. Π (z :: f a)
        → to1 (from1 z) :~: z
  f∘t1 :: ∀ a. Π (r :: Rep1 f a)
        → from1 (to1 r) :~: r
```

[6] https://hackage.haskell.org/package/base-4.12.0.0/docs/Prelude.html#t:Traversable

[7] Note that there is no AbelianSemigroup class directly in base, but since several Semigroup instances end up being commutative (or Abelian) in practice, we opted to include AbelianSemigroup as its own class for the sake of completeness.

Class name	Argument kind	Laws	Pattern functors	SLoC
AbelianSemigroup	Type	1	V1, U1, K1, M1, (:*:)	50
Alternative	Type → Type	3	U1, M1, (:*:), Rec1, (:.:)	173
Applicative	Type → Type	5	U1, K1, M1, (:*:), Rec1, (:.:)	504
Eq	Type	3	V1, U1, K1, M1, (:*:), (:+:)	154
Functor	Type → Type	2	V1, U1, K1, M1, (:*:), (:+:), Par1, Rec1, (:.:)	167
Monad	Type → Type	3	U1, M1, (:*:), Par1, Rec1	350
MonadPlus	Type → Type	2	U1, M1, (:*:), Rec1	138
MonadZip	Type → Type	2	U1, M1, (:*:), Par1, Rec1	343
Monoid	Type	2	U1, K1, M1, (:*:)	94
Ord	Type	4	V1, U1, K1, M1, (:*:), (:+:)	232
Semigroup	Type	1	V1, U1, K1, M1, (:*:)	105
Traversable	Type → Type	3	V1, U1, K1, M1, (:*:), (:+:), Par1, Rec1, (:.:)	591

Figure 3. Every class from Haskell's base library for which verified-classes offers generic defaults. (The pattern functors V1, M1, Par1, Rec1, and (:.:) are explained in Section 5.1.)

To convey an approximate sense of how much effort the generic implementations of each class's proofs requires, Figure 3 includes three metrics. The numbers of laws that each class has, as well as the source lines of code (SLoC) involved in the generic implementation, give a rough idea of how "complex" the proofs are for each class. For instance, verifiying Applicative's five laws takes 454 more SLoC than AbelianSemigroup's one law, which supports the idea that Applicative requires more work than AbelianSemigroup to verify.

The SLoC totals for some of these classes appear to be surprisingly high, and that is simply because the proofs for these classes can become surprisingly long-winded. Knowing this can instill an appreciation for how many SLoC one *saves* by using these generic defaults. As one example, the VTraversable instance for (:*:) alone takes about 110 SLoC. If one were to write a VTraversable instance by hand for a datatype with $n + 1$ fields, then it would take approximately $110n$ SLoC just to write the parts of the proofs to handle the fields alone!

Also included in Figure 3 are the pattern functors that the generic implementations of each class implements. Besides being a rough metric for how complex a class's proofs are, seeing which pattern functors each class supports gives a sense of how widely applicable each generic default is. For instance, it is not clear how one would generically write a Semigroup instance for (:+:), since one would have to arbitrarily choose between biasing towards L1 or R1. As a result, verified-classes's generic Semigroup implementation does not support sum types. Other defaults are more obvious. For instance, the generic implementations of Eq, Functor, Ord, and Traversable mirror a similar strategy that GHC uses to generate code when these classes are placed in **deriving** clauses.

Note that Figure 3 lists some pattern functors that were not included in the simplified presentation in 1. Briefly, they are:

- V1: represents datatypes with no constructors.
- M1: only exists to bundle metadata, such as constructor names and fixities.
- Par1: represents occurrences of the last type parameter (Generic1 only).
- Rec1: represents occurrences of a type constructor applied to the last type parameter (Generic1 only).
- (:.:): represents occurrences of the composition of multiple type constructors applied to the last type parameter (Generic1 only).

5.2 Compilation Time Results

While deriving proofs with datatype-generic programming saves authors from writing many lines of code, one may wonder if there is a tradeoff between the size of the source code and the time it takes to compile a program. The proofs in this paper can be deceptively small, as they are actually doing quite a lot of work behind the scenes. In this section, we attempt to quantify the amount of work being done by measuring how long it takes to compile the verified-classes library as well as some test programs that are built on top of verified-classes. Our measurements were performed using version 8.6.3 of the Glasgow Haskell Compiler (GHC) on a 4-core i5-4670 (3.4GHz, 8GB) machine, collecting the times reported by GHC's -ddump-timings flag.

5.2.1 Library Timing Results

Figure 4 contains timing results for compiling each module in verified-classes that defines a type class with proof obligations. Each module was compiled with GHC's three different optimization levels, -O0 (no optimization), -O1 (moderate optimization), and -O2 (max optimization), to give a sense of

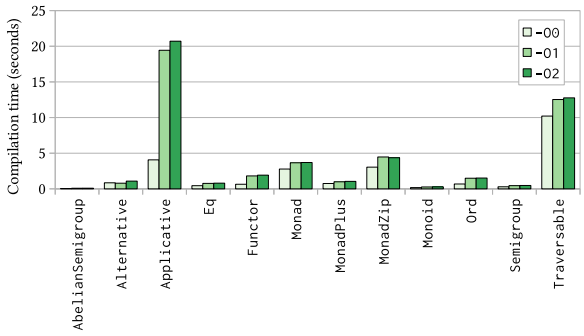

Figure 4. The time it takes to compile each module in the `verified-classes` that defines a verified type class along with generic default implementations of each proof, where `-O0` through `-O2` are the different levels of optimization.

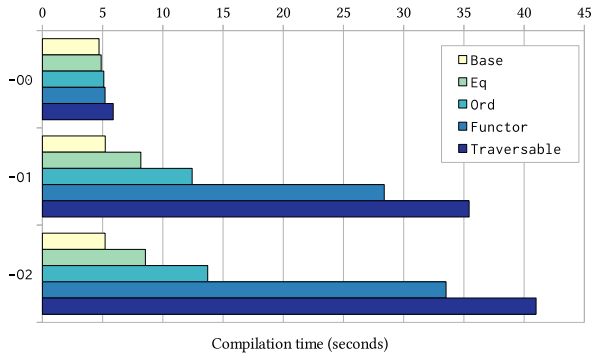

Figure 5. The time it takes to compile `SumEx`, a representative sum type, as progressively more verified class instances are defined using generic defaults. `-O0` through `-O2` are the different levels of optimization.

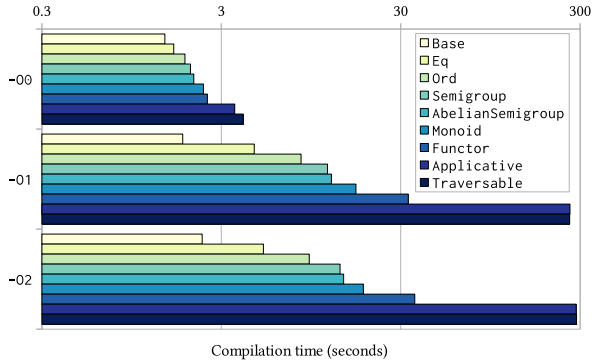

Figure 6. The time it takes to compile `ProductEx`, a representative product type, as progressively more verified class instances are defined using generic defaults. `-O0` through `-O2` are the different levels of optimization. Note that the x-axis in this figure uses a logarithmic scale, whereas a linear scale is used in Figure 5.

how different settings contribute to the overall compilation times.

Note that there are some module dependencies in this figure. For instance, the `Monoid` class and its verified counterpart, `VMonoid`, have `Semigroup` and `VSemigroup` as superclasses, respectively. Therefore, the figures for `Monoid` reflect only the laws that `VMonoid` adds on top of the existing laws for `VSemigroup`, not a cumulative total of all the laws in both `VSemigroup` and `VMonoid`.

The bulk of the classes in Figure 4 have fairly reasonable compilation times, most clocking in at under 5 seconds. `Applicative` and `Traversable`, both of which contain over 500 SLoC apiece in their respective modules, take noticeably longer. Even without optimization, the `Traversable` module takes about 10 seconds to compile. `Applicative`, which takes about 4 seconds with `-O0`, balloons to about 19 seconds with `-O1`. We will say more about this phenomenon in Section 5.2.3.

5.2.2 Example Program Timing Results

In addition to measuring the time it took to compile the `verified-classes` library itself, we also measured the times that it took to compile representative sum and product types with several generically-verified instances. The results in Figures 5 and 6 chart the times for `SumEx`, the representative sum type, and `ProductEx`, the representative product type, respectively. Here are the definitions of `SumEx` and `ProductEx`:

```
data SumEx a
  = MkSumEx1   | MkSumEx2 Unit
  | MkSumEx3 a | MkSumEx4 (Pair Unit a)
data ProductEx a =
  MkProductEx Unit a (Pair Unit a)
```

We say that `SumEx` and `ProductEx` are "representative" since their generic representation types feature most of the pattern functors discussed earlier. Since many of the classes in `verified-classes` do not support sum types, we use separate sum and product types so that the product type can demonstrate examples of instances that the sum type could not have. The `Unit` and `Pair` datatypes are convenient choices for types of fields as they support instances of most of the classes in `verified-classes`.

Each figure was measured by starting out with the barebones definition of its corresponding datatype along with `Generic`, `Generic1`, `VGeneric`, and `VGeneric1` instances; this data point is labeled as `Base`. After this is measured, the program is recompiled after adding generic `Eq` and `VEq` instances; this data point is labeled as `Eq`. This process is repeated for each label on the legend of each chart, and the bars for each data point are stacked so as to reveal how much additional time it took to compile every new additional set of classes.

5.2.3 Timing Results and Optimization

Figures 4, 5, and 6 all exhibit a curious spike in compilation time when certain classes are compiled with optimization. This is especially noticeable in the timing results for the `Traversable` and `Applicative` classes. Surprisingly, most of the CPU time that GHC spends during compilation is not in the typechecking phase, where one would typically expect lots of type-level computation to occur, but instead during the *simplifier* phase.

In order to understand why GHC spends so much time simplifying these programs, it is helpful to know, in a broad sense, what happens when GHC compiles programs with dependent types. GHC uses a well typed intermediate language called Core that features explicit type-equality coercions [27]. While coercions are erased before running a program, they do have an impact on compilation time, as optimization passes over Core often have to manipulate coercions to type of the Core involved. In general, the more coercions a piece of Core has, the longer it takes to optimize. GHC even features a dedicated optimization pass that attempts to reduce the size of coercions so as to mitigate the effect that they can have in later stages of compilation [33].

In picking a part of GHC to blame for the long compilation times observed in this evaluation, the simplifier's effect on coercions is the likeliest culprit. Haskell programs with lots of type-level computations typically correlate to Core with lots of coercions, especially in combination with the singletons encoding. As one extreme example, compiling `ProductEx` with `-O2` and all the verified classes from Figure 6 enabled initially produces Core with about 143,000 coercions directly after the typechecking phase is completed. This is already a sizable number as the overall Core has about 226,000 nodes total, making over 60% of the AST consist of coercions. Once the simplifier begins, the number of coercions skyrockets even further, at one point reaching a maximum of over 4.5 million!

While the measurements in these figures may seem bleak, we have reason to be hopeful. One silver lining is that GHC is leaving a lot of compilation performance on the table. GHC typically builds up coercions by combining many smaller coercions, and given that the design of coercions mirrors inference rules of equational logic, there is a lot of opportunity for groups of coercions to become quite bloated. One further potential optimization is to "zap" a group of coercions down into a single placeholder coercion. These placeholder coercions, while less useful for debugging purposes, are much more efficient to compile. An experimental GHC patch[8] that zaps coercions after every step of normalization in type-level computation has shown to reduce the compile times of a large program from 148.11 seconds to 0.65 seconds. We are optimistic that coercion zapping could have similar benefits for our work.

[8] https://gitlab.haskell.org/ghc/ghc/merge_requests/611

6 Generically Verifying Other Languages

The evaluation in Section 5 uses Haskell-plus-singletons (as described in Appendix A.1 [21]) as the language to implement the ideas in this paper, but they are by no means limited to this one language. In this section, we briefly consider implementations in two other languages: Coq, which natively supports dependent types, as well as Liquid Haskell, a verification tool for Haskell.

6.1 Coq

Coq is an automated theorem prover based on the Calculus of Inductive Constructions. Coq also supports type classes [25], which makes it a suitable target for datatype-generic proofs. To test this claim, we have ported some of the functionality from `verified-classes` to Coq. For example, here a generic proof of the fact that `(≤)` is reflexive: [9]

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

Happily, this approach meshes well with Coq. As shown in the `Proof` block, one can even use Coq's tactic system to fill in the implementation of the function if one desires.

There is one conceptual difference between Coq and Haskell to be aware of when writing this style of proofs, however. Unlike Haskell, which prevents defining multiple instances for the same type, Coq has no such restriction. By default, when a type signature contains both `Ord a` and `GOrd a` constraints, Coq will *not* assume that the `Ord a` superclass of `GOrd a` is the same as the `Ord a` listed in the type signature. This motivates the use of the `!` character before `GOrd a` in the type signature above, which instructs Coq to share the `Ord a` superclass with the one explicitly written out in the type signature. This is important, since the proof will not go through if the two classes are not connected in this manner.

6.2 Liquid Haskell

Liquid Haskell [30] is a tool that verifies properties of Haskell code using an SMT solver. More recently, Liquid Haskell has implemented *refinement reflection* [32], which allows specifying arbitrary properties about functions and retrofits Haskell into a theorem prover. Refinement reflection closes the gap between Liquid Haskell and languages that support dependent types, and one could envision a presentation of the techniques in this paper using Liquid Haskell instead of singletons.

[9] We rename `(<=)` to `leq` to avoid clashing which Coq's existing `(<=)` notation.

We originally explored implementing `verified-classes` using Liquid Haskell, but there exist technical issues which prevent reflecting class methods in the same way that other top-level functions can. [10] We were able to work around these issues by crudely replacing all type classes with reified dictionary datatypes, [11] although we felt this to be somewhat unsatisfactory. We hope that a future version of Liquid Haskell will make this style of programming more natural.

7 Related Work

7.1 Univalent Transport in Homotopy Type Theory

Fundamentally, our work generates proofs for datatypes by reusing proofs from the composition of several smaller datatypes. This technique bears a strong resemblance to the idea of *transport* in Homotopy Type Theory (HoTT), which aims to carry over definitions and proofs from one datatype to another datatype that is equivalent (i.e., there exists an isomorphism between the two types). Indeed, recent work into this area has demonstrated how to completely automate this transport in Coq [28].

One potential drawback to using the techniques by Tabareau et al. [28] is that some proofs must rely on the univalence axiom, which asserts that syntactic equivalence is equivalent to equality. Because univalence is not computable, any proof that uses univalence will itself be uncomputable. Tabareau et al. [28] identify a subset of the Calculus of Inductive Constructions in which transport is guaranteed to be free from uses of this axiom, but imposes some restrictions on the types involved (e.g., all indexed inductive families involved must have decidable equality).

In contrast, our approach does not rely on univalence, which means that the vast majority of our generated proofs are computable. Some of the particular proofs in `verified-classes` do rely on the function extensionality axiom (e.g., the proofs of the `Functor` laws, which involve statements about higher-order functions), but this is a property that we share in common with Tabareau et al. [28], which also uses function extensionality to prove a theorem about transporting dependent products.

7.2 Other Datatype-Generic Programming Styles

The techniques presented in this paper rely on the pattern functor style of generic programming. This is only one way to write datatype-generic programs, and other work approaches datatype-generic programming in a dependently typed setting differently.

Benke et al. [6], Altenkirch et al. [4], and Chapman et al. [9] demonstrate how to achieve datatype-generic programming by first writing codes in a universe of inductively defined sets and then generically programming over the interpretations of those codes. Benke et al. [6] in particular demonstrates that this technique can be used to derive proofs of reflexivity and substitutivity in a generic equality test. Al-Sibahi [2] implements a generic programming library in Idris largely based on these ideas, and uses it to derive instances of `Eq`, `Ord`, `Functor`, `Applicative`, and `Traversable`, as well as proofs of their algebraic properties [3].

The "universe" style of dependently typed generic programming as presented in Al-Sibahi [2] is perhaps the most closely related existing relative to the ideas in this paper, as both styles involve constructing descriptions of datatypes (similar to representation types), converting to and from the descriptions, and generically writing proofs over a minimal universe datatype (similar to pattern functors). However, these proofs assume particular generic implementations of type class methods, whereas our system is more flexible in what it allows, thanks to the techniques in Section 4.

8 Future Directions

8.1 Verifying Other Derived Instances

Haskell features a **`deriving`** mechanism that supports automatic generation of instances for commonly used type classes, such as `Eq`, `Ord`, and `Functor`. While it is generally believed that **`deriving`** generates code that is lawful, this is not a claim that has been formally verified. Indeed, the volume of open GHC tickets containing the keyword "`deriving`" —36 at the time of writing [12]— suggests that verification could help increase user confidence that derived instances are lawful.

The work in this paper addresses this concern to a limited extent, as the `VGeneric` class can verify that derived `Generic` instances behave correctly. It would be interesting to extend this treatment to every other derivable type class in Haskell. This would not be entirely straightforward to do, however, as some derived instances use low-level tricks for performance reasons. For example, derived `Ord` instances sometimes use the primitive `dataToTag#` function to more efficiently compare enumeration types as machine integers. [13] We are unaware of a way to use `dataToTag#` at the type level, so it remains to be seen how low-level derived code could be verified as-is.

8.2 More Sophisticated Generic Programming Styles

Our work leverages the pattern functor style of datatype-generic programming, while other work uses inductively

[10] See https://github.com/ucsd-progsys/liquidhaskell/issues/1196.

[11] See https://github.com/iu-parfunc/verified-instances/tree/1110df2516bd059336fc37c018d7292a340918a8.

[12] According to https://gitlab.haskell.org/ghc/ghc/issues?scope=all&utf8=%E2%9C%93&state=opened&label_name[]=deriving.

[13] See also https://gitlab.haskell.org/ghc/ghc/issues/15696 for an example of a bug caused by a bad interaction between `deriving Ord` and `dataToTag#`.

defined universes, generic combinators, and univalence, as discussed in Section 7.2. It would be interesting to extend these ideas to other forms of datatype-generic programming, such as the *sums of products* approach that has seen popularity in recent years. [10]

Our work uses representation types from GHC's own `GHC.Generics` module, which is essentially a faithful reproduction of the work in Magalhães et al. [17]. While these representation types are nice to work with because of their relative simplicity, they are somewhat limited in their ability to represent advanced language features, such as polymorphic kinds or GADTs. We anticipate that the same techniques from this paper could also be applied to more sophisticated datatype-generic programming libraries that do support these advanced features, such as the libraries presented in Serrano and Miraldo [22, 23].

8.3 External Verification of Code

We have explored a solution for defining and verifying code within the same language, and in the particular case of `verified-classes`, the language is Haskell. Some previous efforts to verify Haskell code take a different approach and verify Haskell code using an *external* tool, such as SMT solvers [30], Coq [8, 26], Alfa [15], or Agda [1]. An external verification tool could serve as the basis for deriving proofs, as the implementations of the generic class methods could live in Haskell, while the generic proofs could be constructed in the external tool separately. This could also be used to "bootstrap" languages that lack dependent types.

9 Conclusion

We have demonstrated how datatype-generic programming *à la* pattern functors significantly streamline many class-based proofs that would otherwise require excessive amounts of boilerplate to implement. The idea of reducing the implementation down to minimal pattern functor types greatly reduces the surface area that a library author has to cover, as verifying the laws for other datatypes becomes a much more manageable task of picking a suitable generic default. Our techniques are general, as they can be ported to any dependently typed programming language with the facilities to reason about type classes. Moreover, our design is flexible enough to adapt to existing class instances that were not themselves defined using generic programming, allowing this design to coexist with other code "in the wild". The implementation of these ideas in `verified-classes` offers a blueprint for this can work for several notable classes with proof obligations.

Acknowledgments

We thank the anonymous reviewers for their feedback. Special thanks goes to Vikraman Choudhury, Niki Vazou, and Ranjit Jhala for their contributions on a previous version of this paper. This material is based upon work supported by the National Science Foundation under Grant No. 1453508.

References

[1] Andreas Abel, Marcin Benke, Ana Bove, John Hughes, and Ulf Norell. 2005. Verifying Haskell Programs Using Constructive Type Theory. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell (Haskell '05)*. ACM, New York, NY, USA, 62–73. https://doi.org/10.1145/1088348.1088355

[2] Ahmad Salim Al-Sibahi. 2014. *The Practical Guide to Levitation.* Master's thesis. IT University of Copenhagen, Copenhagen, Denmark. http://itu.dk/people/asal/pubs/msc-thesis-report.pdf

[3] Ahmad Salim Al-Sibahi. 2019. Desc'n crunch. https://github.com/ahmadsalim/desc-n-crunch/tree/6da2675bb4e2f5386c9f6a264ffd64846d11baa9. Accessed: 2019-01-23.

[4] Thorsten Altenkirch, Conor McBride, and Peter Morris. 2007. Generic Programming with Dependent Types. In *Proceedings of the 2006 International Conference on Datatype-generic Programming (SSDGP'06)*. Springer-Verlag, Berlin, Heidelberg, 209–257. http://dl.acm.org/citation.cfm?id=1782894.1782898

[5] Jeremy Avigad, Leonardo de Moura, and Soonho Kong. 2017. Type Classes. https://leanprover.github.io/theorem_proving_in_lean/type_classes.html. Accessed: 2019-01-23.

[6] Marcin Benke, Peter Dybjer, and Patrik Jansson. 2003. Universes for Generic Programs and Proofs in Dependent Type Theory. *Nordic J. of Computing* 10, 4 (Dec. 2003), 265–289. http://dl.acm.org/citation.cfm?id=985799.985801

[7] Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 5 (2013), 552–593.

[8] Joachim Breitner, Antal Spector-Zabusky, Yao Li, Christine Rizkallah, John Wiegley, and Stephanie Weirich. 2018. Ready, Set, Verify! Applying Hs-to-coq to Real-world Haskell Code (Experience Report). *Proc. ACM Program. Lang.* 2, ICFP, Article 89 (July 2018), 16 pages. https://doi.org/10.1145/3236784

[9] James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. 2010. The Gentle Art of Levitation. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP '10)*. ACM, New York, NY, USA, 3–14. https://doi.org/10.1145/1863543.1863547

[10] Edsko de Vries and Andres Löh. 2014. True Sums of Products. In *Proceedings of the 10th ACM SIGPLAN Workshop on Generic Programming (WGP '14)*. ACM, New York, NY, USA, 83–94. https://doi.org/10.1145/2633628.2633634

[11] Dominique Devriese and Frank Piessens. 2011. On the Bright Side of Type Classes: Instance Arguments in Agda. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP '11)*. ACM, New York, NY, USA, 143–155. https://doi.org/10.1145/2034773.2034796

[12] Richard A. Eisenberg. 2016. Dependent Types in Haskell: Theory and Practice. arXiv:cs.PL/1610.07978

[13] Richard A. Eisenberg and Stephanie Weirich. 2012. Dependently Typed Programming with Singletons. In *Proceedings of the 2012 Haskell Symposium (Haskell '12)*. ACM, New York, NY, USA, 117–130. https://doi.org/10.1145/2364506.2364522

[14] Florian Haftmann and Makarius Wenzel. 2007. Constructive Type Classes in Isabelle. In *Proceedings of the 2006 International Conference on Types for Proofs and Programs (TYPES'06)*. Springer-Verlag, Berlin, Heidelberg, 160–174. http://dl.acm.org/citation.cfm?id=1789277.1789288

[15] Thomas Hallgren, James Hook, Mark P Jones, and Richard B Kieburtz. 2004. An overview of the programatica toolset. In *High Confidence Software and Systems Conference, HCSS04, http://www. cse. ogi. edu/~ hallgren/Programatica/HCSS04*. Citeseer.

[16] José Pedro Magalhães. 2013. Optimisation of Generic Programs Through Inlining. 104–121. https://doi.org/10.1007/978-3-642-41582-1_7

[17] José Pedro Magalhães, Atze Dijkstra, Johan Jeuring, and Andres Löh. 2010. A Generic Deriving Mechanism for Haskell. In *Proceedings of the Third ACM Haskell Symposium on Haskell (Haskell '10)*. ACM, New York, NY, USA, 37–48. https://doi.org/10.1145/1863523.1863529

[18] José Pedro Magalhães, Stefan Holdermans, Johan Jeuring, and Andres Löh. 2010. Optimizing Generics is Easy!. In *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '10)*. ACM, New York, NY, USA, 33–42. https://doi.org/10.1145/1706356.1706366

[19] Guido Martínez. 2018. Typeclasses (via meta arguments). https://github.com/FStarLang/FStar/wiki/Typeclasses-(via-meta-arguments)/c5719ae49bc8a00fa231057a94a8edee1db2a704. Accessed: 2019-01-23.

[20] Thomas van Noort, Alexey Rodriguez, Stefan Holdermans, Johan Jeuring, and Bastiaan Heeren. 2008. A Lightweight Approach to Datatype-generic Rewriting. In *Proceedings of the ACM SIGPLAN Workshop on Generic Programming (WGP '08)*. ACM, New York, NY, USA, 13–24. https://doi.org/10.1145/1411318.1411321

[21] Ryan G. Scott and Ryan R. Newton. 2019. Generic and Flexible Defaults for Verified, Law-Abiding Type-Class Instances (Appendix).

[22] Alejandro Serrano and Victor Cacciari Miraldo. 2018. Generic Programming of All Kinds. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell (Haskell 2018)*. ACM, New York, NY, USA, 41–54. https://doi.org/10.1145/3242744.3242745

[23] Alejandro Serrano and Victor Cacciari Miraldo. 2019. Classes of Arbitrary Kind. In *International Symposium on Practical Aspects of Declarative Languages*. Springer, 150–168.

[24] Tim Sheard and Simon Peyton Jones. 2002. Template Meta-programming for Haskell. *SIGPLAN Not.* 37, 12 (Dec. 2002), 60–75. https://doi.org/10.1145/636517.636528

[25] Matthieu Sozeau and Nicolas Oury. 2008. First-Class Type Classes. In *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs '08)*. Springer-Verlag, Berlin, Heidelberg, 278–293. https://doi.org/10.1007/978-3-540-71067-7_23

[26] Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. 2018. Total Haskell is Reasonable Coq. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2018)*. ACM, New York, NY, USA, 14–27. https://doi.org/10.1145/3167092

[27] Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. 2007. System F with Type Equality Coercions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI '07)*. ACM, New York, NY, USA, 53–66. https://doi.org/10.1145/1190315.1190324

[28] Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. 2018. Equivalences for Free: Univalent Parametricity for Effective Transport. *Proc. ACM Program. Lang.* 2, ICFP, Article 92 (July 2018), 29 pages. https://doi.org/10.1145/3236787

[29] Ron Van Kesteren, Marko Van Eekelen, and Maarten De Mol. 2006. Proof support for general type classes. *Trends in Functional Programming* 5 (2006), 1–16.

[30] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement Types for Haskell. *SIGPLAN Not.* 49, 9 (Aug. 2014), 269–282. https://doi.org/10.1145/2692915.2628161

[31] Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. 2017. Extended Version: Refinement Reflection: Complete Verification with SMT. https://nikivazou.github.io/static/popl18/extended-refinement-reflection.pdf

[32] Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. 2017. Refinement Reflection: Complete Verification with SMT. *Proc. ACM Program. Lang.* 2, POPL, Article 53 (Dec. 2017), 31 pages. https://doi.org/10.1145/3158141

[33] Dimitrios Vytiniotis and Simon Peyton Jones. 2013. Evidence normalization in system FC. *Leibniz International Proceedings in Informatics, LIPIcs* 21 (01 2013), 20–38. https://doi.org/10.4230/LIPIcs.RTA.2013.20

[34] P. Wadler and S. Blott. 1989. How to Make Ad-hoc Polymorphism Less Ad Hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '89)*. ACM, New York, NY, USA, 60–76. https://doi.org/10.1145/75277.75283

[35] Stephanie Weirich, Antoine Voizard, Pedro Henrique Azevedo de Amorim, and Richard A. Eisenberg. 2017. A Specification for Dependent Types in Haskell. *Proc. ACM Program. Lang.* 1, ICFP, Article 31 (Aug. 2017), 29 pages. https://doi.org/10.1145/3110275

[36] Alexey Rodriguez Yakushev, Stefan Holdermans, Andres Löh, and Johan Jeuring. 2009. Generic Programming with Fixed Points for Mutually Recursive Datatypes. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09)*. ACM, New York, NY, USA, 233–244. https://doi.org/10.1145/1596550.1596585