Deriving Via
or, How to Turn Hand-Written Instances into an Anti-Pattern

Baldur Blöndal
Well-Typed LLP

Andres Löh
Indiana University

Abstract
Haskell’s deriving construct is a cheap and cheerful way to quickly generate instances of type classes that follow common patterns. But at present, there are only a subset of such type class patterns that deriving supports, and if a particular class lies outside of this subset, then one cannot derive it at all, with no alternative except for laboriously declaring the instances by hand.

To overcome this deficit, we introduce Deriving Via, an extension to deriving that enables programmers to compose instances from named programming patterns, thereby turning deriving into a high-level domain-specific language for defining instances. Deriving Via leverages newtypes—an already familiar tool of the Haskell trade—to declare recurring patterns in a way that both feels natural and allows a high degree of abstraction.

ACM Reference Format:

1 Introduction
In Haskell, type classes capture common interfaces. When defining class instances, we often discover repeated patterns where different instances have the same definition. For example, the following instances appear in the base library of the Glasgow Haskell Compiler (GHC):

\[
\begin{align*}
\text{instance } \text{Monoid} &\ a \Rightarrow \text{Monoid} \ (\text{IO} \ a) \ where \\
& \quad \text{mempty} = \text{pure mempty} \\
& \quad \text{mappend} = \text{liftA2 mappend}
\end{align*}
\]

\[
\begin{align*}
\text{instance } \text{Monoid} &\ a \Rightarrow \text{Monoid} \ (\text{ST} \ s \ a) \ where \\
& \quad \text{mempty} = \text{pure mempty} \\
& \quad \text{mappend} = \text{liftA2 mappend}
\end{align*}
\]

These have completely identical instance bodies. The underlying pattern works not only for IO and ST s, but for any applicative functor f.

It is tempting to avoid this obvious repetition by defining an instance for all such types in one fell swoop:

\[
\begin{align*}
\text{instance } (\text{Applicative } f, \text{Monoid } a) &\Rightarrow \text{Monoid } (f \ a) \ where \\
& \quad \text{mempty} = \text{pure mempty} \\
& \quad \text{mappend} = \text{liftA2 mappend}
\end{align*}
\]

Unfortunately, this general instance is undesirable as it overlaps with all other \((f \ a)\)-instances. Instance resolution will match the instance head first before considering the context, whether \(f\) is applicative or not. Once GHC has committed to an instance, it will never backtrack. Consider:

\[
\begin{align*}
\text{newtype } \text{Endo} &\ a \ = \text{MkEndo} \ (a \rightarrow \ a) \ -- \text{Data.Monoid}
\end{align*}
\]

Here, Endo is not an applicative functor, but it still admits a perfectly valid Monoid instance that overlaps with the general instance above:

\[
\begin{align*}
\text{instance } \text{Monoid} &\ (\text{Endo } a) \ where \\
& \quad \text{mempty} = \text{MkEndo} \ \text{id} \\
& \quad \text{mappend} \ (\text{MkEndo } f) \ (\text{MkEndo } g) = \text{MkEndo} \ (f \ . \ g)
\end{align*}
\]

Moreover, even if we have an applicative functor \(f\) on our hands, there is no guarantee that this is the definition we want. Notably, lists are the free monoid (the most ‘fundamental’ monoid) but that instance does not coincide with the rule above and in particular, imposes no \((\text{Monoid } a)\) constraint:

\[
\begin{align*}
\text{instance } \text{Monoid} &\ [a] \ where \\
& \quad \text{mempty} = [] \\
& \quad \text{mappend} = (++)
\end{align*}
\]

In fact, the monoid instance for lists is captured by a different rule based on Alternative:

\[
\begin{align*}
\text{instance } \text{Alternative} &\ f \Rightarrow \text{Monoid } (f \ a) \ where \\
& \quad \text{mempty} = \text{empty} \\
& \quad \text{mappend} = (<|>)
\end{align*}
\]

Because instance resolution never backtracks, we can’t define these two distinct rules for Monoid \((f \ a)\) at the same time, even with overlapping instances.

The only viable workaround using the Haskell type class system is to write the instances for each data type by hand, each one with an identical definition (like the instances for IO a and ST s a), which is extremely unsatisfactory:
• It is not obvious that we are instantiating a general principle.
• Because the general principle is not written down in code with a name and documentation, it has to be communicated through folklore or in comments and is difficult to discover and search for. Our code has lost a connection to its origin.
• There are many such rules, some quite obvious, but others more surprising and easy to overlook.
• While the work required to define instances manually for Monoid—which only has two methods—is perhaps acceptable, it quickly becomes extremely tedious and error-prone for classes with many methods.

As an illustration of the final point, consider Num. There is a way to lift a Num instance through any applicative functor:\footnote{Similarly for Floating and Fractional, numeric type classes with a combined number of 25 methods (15 for a minimal definition).}

```haskell
instance (Applicative f, Num a) => Num (f a) where
    (+) = liftA2 (+)
    (-) = liftA2 (-)
    (*) = liftA2 (*)
    negate = liftA negate
    abs = liftA abs
    signum = liftA signum
    fromInteger = pure . fromInteger
```

Defining such boilerplate instances manually for concrete type constructors is so annoying that Conal Elliott introduced a preprocessor \cite{elliott} for this particular use case several years ago.

### 1.1 Deriving

Readers familiar with Haskell’s deriving mechanism may wonder why we cannot simply derive all the instances we just discussed. Unfortunately, our options are very limited.

To start, Monoid is not one of the few blessed type classes that GHC has built-in support to derive. It so happens that (\texttt{(10 a)}, \texttt{(ST s a)} and \texttt{(Endo a)} are all newtypes, so they are in principle eligible for \texttt{generalized newtype deriving} (GND), in which their instances could be derived by reusing the instances of their underlying types \cite{haskell}. However, this would give us the wrong definition in all three cases.

Our last hope is that the the Monoid type class has a suitable generic default implementation \cite{deriving}. If that were the case, we could use a deriving clause in conjunction with the DeriveAnyClass extension, and thereby get the compiler to generate an instance for us.

However, there is no generic default for Monoid, a standard class from the base library (which would be difficult to change). But even if a generic instance existed, it would still capture a single rule over all others, so we couldn’t ever use it to derive both the monoid instance for lists and that for ST s a.

We thus have no other choice but to write some instances by hand. This means that we have to provide explicit implementations of at least a minimal subset of the class methods.

There is no middle ground here, and the additional work required compared to deriving can be drastic—especially if the class has many methods—so the option of using deriving remains an appealing alternative.

### 1.2 Introducing Deriving Via

We are now going to address this unfortunate lack of abstraction and try to bridge the gap between manually defined instances and the few available deriving mechanisms we have at our disposal.

Our approach has two parts:

1. We capture general rules for defining new instances using newtypes.
2. We introduce Deriving Via, a new language construct that allows us to use such newtypes to explain to the compiler exactly how to construct the instance without having to write it by hand.

As a result, we are no longer limited to a fixed set of predefined ways to define particular class instances, but can instead teach the compiler new rules for deriving instances, selecting the one we want using a high-level description.

Let us look at examples. For the first part, we revisit the rule that explains how to lift a monoid instance through an applicative functor. We can turn the problematic generic and overlapping instance for Monoid \((f a)\) into an entirely unproblematic instance by defining a suitable adapter newtype \cite{adapter} and wrapping the instance head in it:

```haskell
newtype App f a = App (f a)

instance (Applicative f, Monoid a) => Monoid (App f a) where
    mempty = App (pure mempty)
    mappend (App f) (App g) = App (liftA2 mappend f g)
```

Since GHC 8.4, we also need a Semigroup instance, because it is now a superclass of Monoid\footnote{See Section 4.4 for a more detailed discussion of this aspect.}:

```haskell
instance (Applicative f, Semigroup a) => Semigroup (App f a) where
    App f <> App g = App (liftA2 (<>)) f g
```

The second part is to now use such a rule in our new form of deriving statement. We can do this when defining a new data type, such as in

```haskell
data Maybe a = Nothing | Just a

deriving Monoid via (App Maybe a)
```

This requires that we independently have an Applicative instance for Maybe, but then we obtain the desired Monoid instance nearly for free.

In the deriving clause, via is a new language construct that explains how GHC should derive the instance, namely...
Deriving Via

by reusing the Monoid instance already available for the via
type, App Maybe a. It should be easy to see why this works:
due to the use of a newtype, App Maybe a has the same internal
representation as Maybe a, and any instance available on
one type can be made to work on the other by suitably wrapping
or unwrapping a newtype. In more precise language, App Maybe a and Maybe a are representationally equal.

The Data.Monoid module defines many further adapters
that can readily be used with Deriving Via. For example, the
rule that obtains a Monoid instance from an Alternative in-
stance is already available through the Alt newtype:

define Alt f a = Alt (f a)

instance Alternative f => Monoid (Alt f a) where
  mempty = Alt empty
  mappend (Alt f) (Alt g) = Alt (f <> g)

instance Alternative f => Semigroup (Alt f a) where
  (<>') = mappend

Using adapters such as App and Alt, a vast amount of Monoid
instances that currently have to be defined by hand can instead be derived using the via construct.

1.3 Contributions and structure of the paper

The paper is structured as follows: In Section 2, we use the
QuickCheck library as a case study to explain in more de-
tail how Deriving Via can be used, and how it works. In Sec-
tion 3, we explain in detail how to typecheck and translate
Deriving Via clauses. In Section 4, we discuss several addi-
tional applications of Deriving Via. We discuss related ideas
in Section 5, describe the current status of our extension in
Section 6 and conclude in Section 7.

Our extension is fully implemented in a GHC branch3, and we are working on a proposal to incorporate it into GHC proper, so it will hopefully be available in a future release of GHC.

The idea of Deriving Via is surprisingly simple, yet it has
a number of powerful and equally surprising properties:

- It further generalizes the generalized newtype deriving
  extension. (Section 3.2.1).
- It additionally generalizes the concept of default signatures. (Section 4.2).
- It provides a possible solution to the problem of intro-
ducing additional boilerplate code when introducing
new superclasses (such as Applicative for Monad, Section 4.4).
- It allows for reusing instances not just between represent-
ationally equal types, but also between isomor-
phic or similarly related types (Section 4.3).

2 Case study: QuickCheck

QuickCheck [3] is a well-known Haskell library for random-
ized property-based testing. At the core of QuickCheck’s

3https://github.com/RyanGScott/ghc/tree/deriving-via
test-case generation functionality is the Arbitrary class. Its
primary method is arbitrary, which describes how to gen-
erate suitable random values of a given size and type. It also
has a method shrink that is used to try to shrink failing counter-
examples of test properties.

Many standard Haskell types, such as Int and lists, are
already instances of Arbitrary. This can be very convenient,
because many properties involving these types can be quick-
checked without any extra work.

On the other hand, there are often additional constraints
imposed on the actual values of a type that are not suffi-
ciently expressed in their types. Depending on the context
and the situation, we might want to guarantee that we gen-
erate positive integers, or non-empty lists, or even sorted lists.

The QuickCheck library provides a number of newtype-

base adapters (called modifiers in the library) for this pur-
pose. As an example, QuickCheck defines:

newtype NonNegative a =
  NonNegative (getNonNegative :: a)

which comes with a predefined instance of the form

instance (Num a, Ord a, Arbitrary a)
  => Arbitrary (NonNegative a)

that explains how to generate and shrink non-negative num-
ers. A user who wants a non-negative integer can now use
NonNegative Int rather than Int to make this obvious.

This approach, however, has a drastic disadvantage: we have to wrap each value in an extra constructor, and the newtype and constructor are QuickCheck-specific. An
implementation detail (the choice of testing library) leaks into
the data model of an application. While we might be willing
to use domain-specific newtypes for added type safety, such as

Age or Duration, we might not be eager to add QuickCheck
modifiers everywhere. And what if we need more than one modifier? And what if other libraries export their own set of modifiers as well? We certainly do not want to change the actual definition of our data types (and corresponding code)
whenever we start using a new library.

With Deriving Via, we have the option to reuse the exist-
ing infrastructure of modifiers without paying the price of cluttering up our data type definitions. We can choose an
actual domain-specific newtype such as

newtype Duration = Duration Int -- in seconds

and now specify exactly how the Arbitrary should be der-
ived for this:

deriving Arbitrary via (NonNegative Int)

This yields an Arbitrary instance which only generates non-
negative integers. Only the deriving clause changes, not the
data type itself. If we later decide we want only positive in-
tegers as durations, we replace NonNegative with Positive.
in the deriving clause. Again, the data type itself is unaffected. In particular, we do not have to change any constructor names anywhere in our code.

2.1 Composition

Multiple modifiers can be combined. For example, there is another modifier called Large that will scale up the size of integral values being produced by a generator. It is defined as

newtype Large a = Large (getLarge :: a)

with a corresponding Arbitrary instance:

instance (Integral a, Bounded a) => Arbitrary (Large a)

For our Duration type, we can easily write

deriving Arbitrary via (NonNegative (Large Int))

and derive an instance which only generates Duration values that are both non-negative and large. This works because Duration still shares the same runtime representation as NonNegative (Large Int) (namely, that of Int), so the latter’s Arbitrary instance can be reused.

2.2 Adding new modifiers

Of course, we can add our own modifiers if the set of predefined modifiers is not sufficient. For example, it is difficult to provide a completely generic Arbitrary instance that works for all data types, simply because there are too many assumptions about what makes good test data that need to be taken into account.

But for certain groups of data types, there are quite reasonable strategies of coming up with generic instances. For example, for enumeration types, one strategy is to desire a uniform distribution of the finite set of values. QuickCheck even offers such a generator, but it does not expose it as a newtype modifier:

arbitraryBoundedEnum :: (Bounded a, Enum a) => Gen a

But from this, we can easily define our own:

newtype BoundedEnum a = BoundedEnum a

instance (Bounded a, Enum a) => Arbitrary (BoundedEnum a) where
  arbitrary = BoundedEnum <$> arbitraryBoundedEnum

We can then use this functionality to derive Arbitrary for a new enumeration type:

data Weekday = Mo | Tu | We | Th | Fr | Sa | Su

deriving (Enum, Bounded)
  deriving Arbitrary via (BoundedEnum Weekday)

2.3 Parameterized modifiers

Sometimes, we might want to parameterize a generator with extra data. We can do so by defining a modifier that has extra arguments and using those extra arguments in the associated Arbitrary instance.

An extreme case that also makes use of type-level programming features in GHC is a modifier that allows us to specify a lower and an upper bound of a generated natural number.

newtype Between (l :: Nat) (u :: Nat) = Between Integer instance (KnownNat l, KnownNat u) => Arbitrary (Between 1 u) where
  arbitrary = Between <$> choose (natVal @l Proxy, natVal @u Proxy)

(Note that this instance makes use of visible type application [6] in natVal @l and natVal @u.)

We can then equip an application-specific type for years with a generator that lies within a plausible range:

newtype Year = Year Integer
  deriving Show
  deriving Arbitrary via (Between 1900 2100)

In general, we can use this technique of adding extra parameters to a newtype to support additional ways to configure the behavior of derived instances.

3 Typechecking and translation

Seeing enough examples of Deriving Via can give the impression that it is a somewhat magical feature. In this section, we aim to explain the magic underlying Deriving Via by giving a more precise description of:

- How Deriving Via clauses are typechecked.
- What code Deriving Via generates behind the scenes.
- How to determine the scoping of type variables in Deriving Via clauses.

To avoid clutter, we assume that all types have monomorphic kinds. However, it is easy to incorporate kind polymorphism [12], and our implementation of these ideas in GHC does so.

3.1 Well-typed uses of Deriving Via

Deriving Via grants the programmer the ability to put extra types in her programs, but the flip side to this is that it’s possible for her to accidentally put total nonsense into a Deriving Via clause, such as:

newtype S = S Char
  deriving Eq via Maybe

In this section, we describe a general algorithm for when a Deriving Via clause should typecheck, which will allow us to reject ill-formed examples like the one above.
3.1.1 Aligning kinds

Suppose we are deriving the following instance:

```
data D d₁ ... dₙ
  deriving (C c₁ ... cₙ) via (V v₁ ... vₚ)
```

In order for this declaration to typecheck, we must check the kinds of each type. In particular, the following conditions must hold:

1. The type `C c₁ ... cₙ` must be of kind `(k₁ -> ... -> kᵣ -> *) -> Constraint` for some kinds `k₁, ..., kᵣ`. The reason is that the instance we must generate,

   `instance C c₁ ... cₙ (D d₁ ... dₙ) where ...
   ...
   requires that we can apply `C c₁ ... cₙ` to another type `D d₁ ... dₙ` where `i ≤ m` (see Section 3.1.2). Therefore, it would be nonsense to try to derive an instance of `C c₁ ... cₙ` if it had kind, say, `Constraint`.

2. The kinds `V v₁ ... vₚ` and `D d₁ ... dₙ` and the kind of the argument to `C c₁ ... cₙ` must all unify. This check rules out the above example of `deriving Eq via Maybe`, as it does not even make sense to talk about reusing the `Eq` instance for `Maybe`—which is of kind `(* -> *)`—as `Eq` instances can only exist for types of kind `*`.

3.1.2 Eta-reducing the data type

Note that in the conditions above, we specify `D d₁ ... dₙ` (for some `i`), instead of `D d₁ ... dₙ`. That is because in general, the kind of the argument to `C c₁ ... cₙ` is allowed to be different from the kind of `D d₁ ... dₙ`! For instance, the following example is perfectly legitimate:

```
class Functor (f :: * -> *) where ...
data Foo a = Foo a a
  deriving Functor
```

despite the fact that `Foo` a has kind `*` and the argument to `Functor` has kind `(-> *)`. This is because the code that actually gets generated has the following shape:

```
instance Functor Foo where ...

To put it differently, we have eta-reduced away the `a in Foo a` before applying `Functor` to it. The power to eta-reduce variables from the data type is part of what makes deriving clauses so flexible.

To determine how many variables to eta-reduce, we must examine the kind of `C c₁ ... cₙ`, which by condition (1) is of the form `((k₁ -> ... -> kᵣ -> *) -> Constraint)` for some kinds `k₁, ..., kᵣ`. Then the number of variables to eta-reduce is simply `r`, so to compute the `i` in `D d₁ ... dₙ`, we take `i = m - r`.

This is better explained by example, so consider the following two scenarios, both of which typecheck:

```
newtype A a = A a deriving Eq via (Identity a)
newtype B b = B b deriving Functor via Identity
```

In the first example, we have the class `Eq`, which is of kind `* -> Constraint`. The argument to `Eq`, which is of kind `*`, does not require that we eta-reduce any variables. As a result, we check that `A a` is of kind `*`, which is the case.

In the second example, we have the class `Functor`, which is of kind `(* -> *) -> Constraint`. The argument to `Functor` is of kind `(* -> *)`, which requires that we eta-reduce one variable from `B b` to obtain `B`. We then check that `B` is of kind of `(* -> *)`, which is true.

3.2 Code generation

Once the typechecker has ascertained that a `via` type is fully compatible with the data type and the class for which an instance is being derived, GHC proceeds with generating the code for the instance itself. This generated code is then fed back into the typechecker, which acts as a final sanity check that GHC is doing the right thing under the hood.

3.2.1 Generalized newtype deriving (GND)

The process by which Deriving Via generates code is heavily based off of the approach that the GND takes, so it is informative to first explain how GND works. From there, Deriving Via is a straightforward generalization—so much so that Deriving Via could be thought of as “generalized GND”.

Our running example in this section will be the newtype `Age`, which is a simple wrapper around `Int` (which we will call the representation type):

```
newtype Age = MkAge Int
deriving Enum
```

A naive way to generate code would be to manually wrap and unwrap the `MkAge` constructor wherever necessary, such as in the code below:

```
instance Enum Age where
  toEnum i = MkAge (toEnum i)
  fromEnum (MkAge x) = fromEnum x
  enumFrom (MkAge x) = map MkAge (enumFrom x)
```

This works, but is somewhat unsatisfying. After all, a newtype is intended to be a zero-cost abstraction that acts identically to its representation type at runtime. Accordingly, any function that mentions a newtype in its type signature should be able to be converted to a new function with all occurrences of the newtype in the type signature replaced with the representation type, and moreover, that new function should behave identically to the old one at runtime.

Unfortunately, the implementation of `enumFrom` may not uphold this guarantee. While wrapping and unwrapping the `MkAge` constructor is certain to be a no-op, the `map` function is definitely not a no-op, as it must walk the length of a list. But the fact that we need to call `map` in the first place feels rather silly, as all we are doing is wrapping a newtype at each element.
Luckily, there is a convenient solution to this problem: the safe `coerce` function [1]:

```haskell
coerce :: Coercible a b => a -> b
```

Operationally, `coerce` can be thought of as behaving like its wily cousin, `unsafeCoerce`, which takes a value of one type as casts it to a value at another type. Unlike `unsafeCoerce`, which can break programs if used carelessly, `coerce` is completely type-safe due to its use of the `Coercible` constraint. We will explain `Coercible` in more detail later, but for now, it suffices to say that a `Coercible a b` constraint witnesses the fact that two types `a` and `b` have the same representation at runtime, and thus any value of type `a` can be safely cast to type `b`.

Armed with `coerce`, we can show what code GND would actually generate for the `Enum Age` instance above:

```haskell
instance Enum Age where
toEnum =
  coerce @(Int -> Int) @(Int -> Age) toEnum
fromEnum =
  coerce @(Int -> Int) @(Age -> Int) fromEnum
enumFrom =
  coerce @(Int -> [Int]) @(Age -> [Age]) enumFrom
```

Now we have a strong guarantee that the `Enum` instance for `Age` has exactly the same runtime characteristics as the `Enum` instance for `Int`. As an added benefit, the code ends up being simpler, as every method can be implemented as a straightforward application of `coerce`. The only interesting part is generating the two explicit type arguments [6] that are being used to specify the source type (using the representation type) and the target type (using the newtype of `coerce`).

### 3.2.2 The `Coercible` constraint

A `Coercible` constraint can be thought of as evidence that GHC can use to cast between two types. `Coercible` is not a type class, so it is impossible to write a `Coercible` instance by hand. Instead, GHC can generate and solve `Coercible` constraints automatically as part of its built-in constraint solver, much like it can solve equality constraints. (Indeed, `Coercible` can be thought of as a broader notion of equality among types.)

As mentioned in the previous section, a newtype can be safely cast to and from its representation type, so GHC treats them as `inter-Coercible`. Continuing our earlier example, this would mean that GHC would be able to conclude that:

```haskell
instance Coercible Age Int
instance Coercible Int Age
```

But this is not all that `Coercible` is capable of. A key property is that GHC’s constraint solver can look inside of other type constructors when determining if two types are `inter-Coercible`. For instance, both of these statements hold:

```haskell
instance Coercible (Age -> [Age]) (Int -> [Int])
instance Coercible (Int -> [Int]) (Age -> [Age])
```

This demonstrates the ability to cast through the function and list type constructors. This ability is important, as our derived `enumFrom` instance would not typecheck otherwise!

Another crucial fact about `Coercible` that we rely on is that it is transitive: if `Coercible a b` and `Coercible b c` hold, then `Coercible a c` also holds. This is perhaps unsurprising if one views `coerce` as an equivalence relation, but it is a fact that is worth highlighting, as the transitivity of `Coercible` is what allows us to cast `between newtypes`. For instance, if we have these two newtypes:

```haskell
newtype A a = A [a]
newtype B = B [Int]
```

then GHC is able to conclude that `Coercible (A Int) B` holds, because we have the following `Coercible` rules:

```haskell
instance Coercible (A Int) [Int]
instance Coercible [Int] B
```

as well as transitivity. As we will discuss momentarily, `Deriving Via in particular makes heavy use of the transitivity of `Coercible`.

### 3.2.3 From GND to Deriving Via

As we saw in Section 3.2.1, the code which GND generates relies on `coerce` to do the heavy lifting. In this section, we will generalize this technique slightly to give us a way to generate code for `Deriving Via`.

Recall the following GND-derived instance:

```haskell
newtype Age = MkAge Int deriving Enum
```

As stated above, it generates the following code for `enumFrom`:

```haskell
instance Enum Age where
  enumFrom =
    coerce @(Int -> [Int]) @(Age -> [Age]) enumFrom
```

Here, there are two crucially important types: the representation type, `Int`, and the original newtype itself, `Age`. The implementation of `enumFrom` simply sets up an invocation of `coerce enumFrom`, with explicit type arguments to indicate that we should reuse the existing `enumFrom` implementation for `Int` and reappropriate it for `Age`.

The only difference in the code that GND and `Deriving Via` generate is that in the former strategy, GHC always picks the representation type for you, but in `Deriving Via`, the `user` has the power to choose this type. For example, if a programmer had written this instead:

```haskell
newtype T = T Int
instance Enum T where ...
newtype Age = MkAge Int deriving Enum via T
```

then the following code would be generated:

```haskell
instance Coercible (Age -> [Age]) (Int -> [Int])
instance Coercible (Int -> [Int]) (Age -> [Age])
```
3.3 Type variable scoping

In the remainder of this section, we will present an overview of how type variables are bound in Deriving Via clauses, and over what types they scope. Deriving Via introduces a new place where types can go, and more importantly, it introduces a new place where type variables can be quantified, so it takes some amount of care to devise a consistent treatment for it.

3.3.1 Binding sites

Consider the following example:

data Foo a = ...
deriving (Baz a b c) via (Bar a b)

Where is each type variable quantified?

- a is bound by Foo itself in the declaration data Foo a.
- Such a variable scopes over both the derived class, Baz a b c, as well as the via type, Bar a b.
- b is bound by the via type, Bar a b. Note that b is bound here but a is not, as it was bound earlier by the data declaration. b scopes over the derived class type, Baz a b c, as well.
- c is bound by the derived class, Baz a b c, as it was not bound elsewhere. (a and b were bound earlier.)

In other words, the order of scoping starts at the data declaration, then the via type, and then the derived classes associated with that via type.

3.3.2 Establishing order

This scoping order may seem surprising, as one might expect the type variables bound by the derived classes to scope over the via type instead. However, this choice introduces additional complications that are tricky to resolve.

For instance, consider a scenario where one attempts to derive multiple classes at once with a single via type:

data D
  deriving (C1 a, C2 a) via (T a)

Suppose we first quantified the variables in the derived classes and made them scope over the via type. Because each derived class has its own type variable scope, the a in C1 a would be bound independently from the a in C2 a. In other words, we would have something like this (using a hypothetical forall syntax):

deriving (forall a . C1 a, forall a . C2 a) via (T a)

Now we are faced with a thorny question: which a is used in the via type, T a? There are multiple choices here, since the a variables in C1 a and C2 a are distinct! This is an important decision, since the kinds of C1 and C2 might differ, so the choice of a could affect whether T a kind-checks or not.

On the other hand, if one binds the a in T a first and has it scope over the derived classes, then this becomes a non-issue. We would instead have this:

deriving (C1 a, C2 a) via (forall a . T a)

Now, there is no ambiguity regarding a, as both a variables in the list of derived classes were bound in the same place.

It might feel strange visually to see a variable being used before its binding site (assuming one reads code from left to right). However, this is not unprecedented within Haskell, as this is also legal:

\[ f = g + h \text{ where } g = 1 ; h = 2 \]

In this example, we have another scenario where things are bound (g and h) after their use sites. In this sense, the via keyword is continuing a rich tradition pioneered by where clauses.

One alternative idea (which was briefly considered) was to put the via type before the derived classes so as to avoid this “zigzagging” scoping. However, this would introduce additional ambiguities. Imagine one were to take this example:

deriving Z via X Y

And convert it to a form in which the via type came first:

deriving via X Y Z

Should this be parsed as \((X \ Y) \ Z\), or \((X \ (Y \ Z))\)? It’s not clear visually, so this choice would force programmers to write additional parentheses.

4 More use cases

We have already seen in Section 2 how Deriving Via facilitates greater code reuse in the context of QuickCheck. This is far from the only domain where Deriving Via proves to be a natural fit, however. In fact, there are so many of these domains, there would be enough to fill pages upon pages!

Unfortunately, we do not have enough space to document all of these use cases, so in this section, we present a cross-section of scenarios in which Deriving Via can capture interesting patterns and allow programmers to abstract over them in a convenient way.
4.1 Asymptotic improvements with ease

A widely used feature of type classes is their ability to give default implementations for their methods if a programmer leaves them off. One example of this can be found in the `Applicative` class. The main workhorse of `Applicative` is the `(<*)` method, but on occasion, it is more convenient to use the `(*>)` or `(**>*)` methods, which sequence their actions but discard the result of one of their arguments:

```haskell
class Functor f => Applicative f where
  pure :: a -> f a
  (**) :: f (a -> b) -> f a -> f b
  (**>) :: f a -> f b -> f b
  (**>) = lift2 (\_ _ -> a)
  (**>) :: f a -> f b -> f b
  (**>) = lift2 (\_ _ -> b)
```

As shown here, `(**)` and `(**>)` have default implementations in terms of `lift2`. This works for any `Applicative`, but is not as efficient as it could be in some cases. For some instances of `Applicative`, we can actually implement these methods in $O(1)$ time instead of using `lift2`, which can often run in superlinear time. One such `Applicative` is the function type `(*)`:

```haskell
instance Applicative ((->) r) where
  pure = const
  (**>) f g x = f x (g x)
```

Note that we had to explicitly define `(**)` and `(**>)`, as the default implementations would not have been as efficient. But `((->) r)` is not the only type for which this trick works—it also works for any data type that is isomorphic to `((->) r)`. These function-like types are characterized by the `Representable` type class:

```haskell
class Functor f => Representable f where
  type Rep f
  index :: f a -> (Rep f -> a)
  tabulate :: (Rep f -> a) -> f a
```

This is a good deal more abstract than `((->) r)` itself, so it can be helpful to see how `Representable` works for `((->) r)` itself:

```haskell
instance Representable ((->) r) where
  type Rep ((->) r) = r
  index f = f
  tabulate f = f
```

With `Representable`, we can codify the `Applicative` shortcut for `(**)` and `(**>)` with a suitable newtype:

```haskell
newtype WrapRep f a = WrapRep (f a)
  deriving (Functor, Representable)

instance Representable f => Applicative (WrapRep f) where
  pure = tabulate . pure
```

Now, instead of having to manually override `(**)` and `(**>)` to get the desired performance, one can accomplish this in a more straightforward fashion by using `Deriving Via`:

```haskell
newtype IntConsumer a = IntConsumer (Int -> a)
  deriving (Functor, Representable)
  deriving Applicative via (WrapRep IntConsumer)
```

Not only does this save code in the long run, but it also gives a name to the optimization being used, which allows it to be documented, exported from a library, and thereby easier to spot "in the wild" for other programmers.

4.2 Making defaults more flexible

In the previous section, we saw an example of how relying too much on a type class’s default implementations can backfire. This is an unfortunately common trend with type classes in general: Many classes try to pick one-size-fits-all defaults that don’t work well in certain scenarios, but because Haskell allows specifying only one default per method, if the provided default doesn’t work for a programmer’s use case, then she is forced to implement her own implementations by hand.

In this section, we continue the trend of generalizing defaults by looking at another language extension that Deriving Via can substitute for: `default signatures`. Default signatures (a slight generalization of default implementations) can eliminate large classes of boilerplate, but they too are limited by the one-default-per-method restriction. Here, we demonstrate how one can scrap uses of default signatures in favor of Deriving Via, and show how Deriving Via can overcome the limitations of default signatures.

The typical use case for default signatures is when one has a type class method that has a frequently used default implementation at a constrained type. For instance, consider a `Pretty` class with a method `pPrint` for pretty-printing data:

```haskell
class Pretty a where
  pPrint :: a -> Doc
```

Coming up with `Pretty` instances for the vast majority of data types is repetitive and tedious, so a common pattern is to abstract away this tedium using generic programming libraries, such as those found in GHC.Generics [10] or generics-sop [4]. For example, using `GHC.Generics`, we can define

```haskell
genericPPrint ::
  (Generic a, GPretty (Rep a)) => a -> Doc
```

The details of how `Generic`, `GPretty`, and `Rep` work are not important to understanding the example. What is important is to note that we cannot just add

```haskell
pPrint = genericPPrint
```
as a conventional default implementation to the `Pretty` class, because it does not typecheck due to the extra constraints.

Before the advent of default signatures, one had to work around this by defining `pPrint` to be `genericPPrint` in every `Pretty` instance, as in the examples below:

```haskell
instance Pretty Bool where
  pPrint = genericPPrint

instance Pretty a => Pretty (Maybe a) where
  pPrint = genericPPrint
```

To avoid this repetition, default signatures allow one to provide a default implementation of a type class method using additional constraints on the method’s type. For example:

```haskell
class Pretty a where
  pPrint :: a -> Doc
  default pPrint :: (Generic a, GPretty (Rep a)) => a -> Doc
              pPrint = genericPPrint
```

Now, if any instances of `Pretty` are given without an explicit definition of `pPrint`, the default implementation is used. For this to typecheck, the data type `a` used in the instance must satisfy the constraints `(Generic a, GPretty (Rep a))`. Thus, we can reduce the instances above to just:

```haskell
instance Pretty Bool
instance Pretty a => Pretty (Maybe a)
```

Although default signatures remove the need for many occurrences of boilerplate code, it also imposes a significant limitation: every type class method can have at most one default implementation. As a result, default signatures effectively endorse one default implementation as the canonical one. But in many scenarios, there is far more than just one way to do something. Our `pPrint` example is no exception. Instead of `genericPPrint`, one might one to:

- Leverage a `Show`-based default implementation instead of a `Generic`-based one,
- Use a different generic programming library, such as generics-sop, instead of GHC.Generics, or
- Use a tweaked version of `genericPPrint` which displays extra debugging information.

All of these are perfectly reasonable choices a programmer might want to make, but alas, GHC only lets type classes bless each method with one default.

Fortunately, Deriving Via provides a convenient way of encoding default implementations with the ability to toggle between different choices: newtypes! For instance, we can codify two different approaches to implementing `pPrint` as follows:

```haskell
newtype GenericPPrint a = GenericPPrint a

instance ShowPPrint a => Pretty (GenericPPrint a) where
  pPrint (GenericPPrint x) = stringToDoc (show x)

With these newtypes in hand, choosing between them is as simple as changing a single type:

```haskell
deriving Pretty via (GenericPPrint Data_Type1)
deriving Pretty via (ShowPPrint Data_Type2)
```

We have seen how Deriving Via makes it quite simple to give names to particular defaults, and how toggling between defaults is a matter of choosing a name. In light of this, we believe that many current uses of default signatures ought to be removed entirely and replaced with the Deriving Via-based idiom presented in this section. This avoids the need to bless one particular default, and forces programmers to consider which default is best suited to their use case, instead of blindly trusting the type class’s blessed default to always do the right thing.

An additional advantage is that it allows decoupling the definition of such defaults from the site of the class definition. Hence, if a package author is hesitant to add a default because that might incur an unwanted additional dependency, nothing is lost, and the default can simply be added in a separate package.

4.3 Deriving via isomorphisms

All of the examples presented thus far in the paper rely on deriving through data types that have the same runtime representation as the original data type. In the following, however, we point out that—perhaps surprisingly—we can also derive through data types that are isomorphic, not just representationally equal. To accomplish this feat, we rely on techniques from generic programming.

Let us go back to QuickCheck (as in Section 2) once more and consider the data type:

```haskell
data Track = Track Title Duration
```

for which we would like to define an `Arbitrary` instance. Let us further assume that we already have `Arbitrary` instances for both `Title` and `Duration`.

The QuickCheck library defines an instance for pairs, so we could generate values of type `(Title, Duration)`, and in essence, this is exactly what we want. But unfortunately, the two types are not inter-Coercible, even though they are isomorphic\(^4\).

However, we can exploit the isomorphism and still get an instance for free, and the technique we apply is quite widely applicable in similar situations. As a first step, we declare a newtype to capture that one type is isomorphic to another:

```haskell
newtype SameRepAs a b = SameRepAs a
```

\(^4\)Isomorphic in the sense that we can define a function from `Track` to `(Title, Duration)` and vice versa. Depending on the class we want to derive, sometimes an even weaker relationship between the types is sufficient, but we will focus on the case of isomorphism here for reasons of space.
Note that the idea here is that \(a\) and \(b\) are isomorphic in some sense, but only \(a\) is used as the value of the type. So \(\text{SameRepAs}\ a\ b\ \text{is inter-Coercible with} \ a\).

We choose to witness an isomorphism between the two types via their generic representations: if two types have inter-Coercible generic representations, we can transform back and forth using the \text{from} and \text{to} methods of the \text{Generic} class from GHC.Generics [10]. We can use this to define a suitable \text{Arbitrary} instance for \text{SameRepAs}:

\[
\begin{align*}
\text{instance} & \quad (\text{Generic} \ a, \text{Generic} \ b, \text{Arbitrary} \ b) \\
& \quad \Rightarrow \text{Arbitrary} \ (a \ '\text{SameRepAs}' \ b) \text{ where} \\
& \quad \text{arbitrary} = \text{SameRepAs} \ . \ \text{coerceViaRep} <\$> \text{arbitrary} \\
& \quad \text{coerceViaRep} :: b \rightarrow a \\
& \quad \text{coerceViaRep} = \\
& \quad \text{to} \ . \ \text{(coerce :: Rep} \ b \ () \rightarrow \text{Rep} \ a \ () \) \ . \ \text{from}
\end{align*}
\]

Here, we first use \text{arbitrary} to give us a generator of type \text{Gen} \ b, then coerce this via the generic representations into an arbitrary value of type \text{Gen} \ a.

Finally, we can use the following \text{deriving} declarations for \text{Track} to obtain the desired \text{Arbitrary} instance:

\[
\begin{align*}
\text{deriving} & \quad \text{Generic} \\
\text{deriving} & \quad \text{Arbitrary} \\
& \quad \text{via} \ (\text{Track} '\text{SameRepAs}' \ (\text{String}, \text{Duration}))
\end{align*}
\]

With this technique, we can significantly expand the “equivalence classes” of data types that can be used when picking suitable types to derive through.

4.4 Retrofitting superclasses

On occasion, the need arises to retrofit an existing type class with a superclass, such as when \text{Monad} was changed to have \text{Applicative} as a superclass (which in turn has \text{Functor} as a superclass).

One disadvantage of such a change is that if the primary goal is to define the \text{Monad} instance for a type, one now has to write two additional instances, for \text{Functor} and \text{Applicative}, even though these instances are actually determined by the \text{Monad} instance.

With Deriving Via, we can capture this fact as a newtype, thereby making the process of defining such instances much less tedious:

\[
\begin{align*}
\text{newtype} & \quad \text{FromMonad} \ m \ a = \text{FromMonad} \ (m \ a) \\
& \quad \text{deriving} \ \text{Monad} \\
\text{instance} & \quad \text{Monad} \ m = \text{fromMonad} \ (\text{FromMonad} \ m) \ \text{where} \\
& \quad \text{fmap} = \text{liftM} \\
\text{instance} & \quad \text{Monad} \ m = \text{Applicative} \ (\text{FromMonad} \ m) \ \text{where} \\
& \quad \text{pure} = \text{return} \\
& \quad (\text{<*>}) = \text{ap}
\end{align*}
\]

Now, if we have a data type with a \text{Monad} instance, we can simply derive the corresponding \text{Functor} and \text{Applicative} instances by referring to \text{FromMonad}:

\[
\begin{align*}
\text{data} & \quad \text{Stream} \ a \ b = \text{Done} \ b \mid \text{Yield} \ a \ (\text{Stream} \ a \ b) \\
& \quad \text{deriving} \ (\text{Functor}, \ \text{Applicative}) \\
& \quad \text{via} \ (\text{FromMonad} \ (\text{Stream} \ a))
\end{align*}
\]

\[
\begin{align*}
\text{instance} & \quad \text{Monad} \ (\text{Stream} \ a) \ \text{where} \\
& \quad \text{return} = \text{Done} \\
& \quad \text{Yield} \ a \ k \Rightarrow f = \text{Yield} \ a \ (k \Rightarrow f) \\
& \quad \text{Done} \ b \Rightarrow f = f \ b
\end{align*}
\]

One potentially problematic aspect remains. Another proposal [11] has been put forth (but has not been implemented, as of now) to remove the \text{return} method from the \text{Monad} class and make it a synonym for \text{pure} from \text{Applicative}. The argument is that \text{return} is redundant, given that \text{pure} does the same thing with a more general type signature. All other prior discussion about the proposal aside, it should be noted that removing \text{return} from the \text{Monad} class would prevent \text{FromMonad} from working, as then \text{Monad} instances would not have any way to define \text{pure}. 5

4.5 Avoiding orphan instances

Not only can Deriving Via quickly procure type class instances, in some cases, it can eliminate the need for certain instances altogether. Haskell programmers often want to avoid \text{orphan instances}: instances defined in a separate module from both the type class and data types being used. Sometimes, however, it’s quite tempting to reach for orphan instances, as in the following example adapted from a blog post by Gonzalez [9]:

\[
\begin{align*}
\text{newtype} & \quad \text{Plugin} = \text{Plugin} \ (\text{IO} \ (\text{String} \rightarrow \text{IO} \ ()) \\
& \quad \text{deriving} \ \text{Semigroup}
\end{align*}
\]

In order for this derived \text{Semigroup} instance to typecheck, there must be a \text{Semigroup} instance for \text{IO} available. Suppose for a moment that there was no such instance for \text{IO}. How could one work around this issue?

- One could patch the base library to add the instance for \text{IO}. But given base’s slow release cycle, it would be a while before one could actually use this instance.
- Write an orphan instance for \text{IO}. This works, but is undesirable, as now anyone who uses \text{Plugin} must incur a possibly unwanted orphan instance.

Luckily, Deriving Via presents a more convenient third option: re-use a \text{Semigroup} instance from another data type. Recall the \text{App} data type from Section 1.2 that lets us define a \text{Semigroup} instance by lifting through an \text{Applicative} instance. As luck would have it, \text{IO} already has an \text{Applicative} instance.

\[
\begin{align*}
& \Rightarrow \text{pure} \ \text{liftM} \\
& \Rightarrow \text{ap}
\end{align*}
\]

5A similar, yet somewhat weaker, argument applies to suggested changes to relax the constraints of \text{liftM} and \text{ap} to merely \text{Applicative} and to change their definitions to be identical to \text{fmap} and \text{<*>}, respectively.
instance, so we can derive the desired Monoid instance for Plugin like so:

```haskell
newtype Plugin = Plugin (IO (String -> IO ()))
deriving Semigroup
  via (App IO (String -> App IO ()))
```

Note that we have to use App twice in the via type, corresponding to the two occurrences of IO in the Plugin type. This is ok, because App IO has the same representation as IO. As desired, we completely bypass the need for a Semigroup instance for IO.

5 Related Ideas

We have demonstrated in the previous section that Deriving Via is an extremely versatile technique, and can be used to tackle a wide variety of problems. Deriving Via also bears a resemblance to other distinct language features, such as ML functors and explicit dictionary passing, so in this section, we present an overview of their similarities and differences.

5.1 ML functors

Languages in the ML family, such as Standard ML or OCaml, provide functors, which are a feature of the module system that allows writing functions from modules of one signature to modules of another signature. In terms of functionality, functors somewhat closely resemble Deriving Via, as functors allow “lifting” of code into the module language much like Deriving Via allows lifting of code into GHC’s deriving construct.

5.2 Explicit dictionary passing

The power and flexibility of Deriving Via is largely due to GHC’s ability to take a class method of a particular type and massage it into a method of a different type. This process is almost completely abstracted away from the user, however. A user only needs to specify the types involved, and GHC will handle the rest behind the scenes.

An alternative approach, which would put more power into the hands of the programmer, is to permit the ability to explicitly construct and pass the normally implicit dictionary arguments corresponding to type class instances [5]. Unlike in Deriving Via, where going between class instances is a process that is carefully guided by the compiler, permitting explicit dictionary arguments would allow users to actually coerce concrete instance values and pass them around as first-class values. In this sense, explicit dictionary arguments could be thought of as a further generalization of the technique that Deriving Via uses.

However, explicit dictionary arguments are a considerable extension of the language and its type system, and we feel that to be too large a hammer for the nail we are trying to hit. Deriving Via works by means of a simple desugaring of code with some light typechecking on top, which makes it much simpler to describe and implement. Finally, the problem which explicit dictionaries aims to solve—resolving ambiguity in implicit arguments—almost never arises in Deriving Via, as the programmer must specify all the types involved in the process.

6 Current status

We have implemented Deriving Via within GHC. Our implementation also interacts well with other GHC features that were not covered in this paper, such as kind polymorphism [12], StandaloneDeriving, and type classes with associated type families [2]. However, there are still challenges remaining, which we will describe in this section.

6.1 Quality of error messages

The nice thing about deriving is that when it works, it tends to work extremely well. When it doesn’t work, however, it can be challenging to formulate an error message that adequately explains what went wrong. The fundamental issue is that error messages resulting from uses of deriving are usually rooted in generated code, and pointing to code that the user didn’t write in error messages can sometimes lead to a confusing debugging experience.

Fortunately, we have found in our experience that the quality of Deriving Via-related error messages is overall on the positive side. GHC has already invested significant effort into making type errors involving Coercible to be easily digestible by programmers, so Deriving Via benefits from this work. For instance, if one inadvertently tries to derive through a type that is not inter-Coercible with the original data type, such as in the following example:

```haskell
newtype UhOh = UhOh Char deriving Ord via Int
```

Then GHC will tell you exactly that, in plain language:

• Couldn’t match representation of type Char with that of Int arising from the coercion of the method compare
  from type 'Int -> Int -> Ordering'
  to type 'UhOh -> UhOh -> Ordering'

That is not to say that every error message is this straightforward. There are some scenarios that produce less-than-ideal errors, such as this:

```haskell
newtype Foo a = Foo (Maybe a) deriving Ord via a
```

• Occurs check: cannot construct the infinite type: a ~ Maybe a arising from the coercion of the method 'compare'
  from type 'a -> a -> Ordering'
  to type 'Foo a -> Foo a -> Ordering'

The real problem is that a and Maybe a do not have the same representation at runtime, but the error does not make this obvious. It is possible that one could add an ad hoc check for this class of programs, but there are likely many more tricky corner cases lurking around the corner, given that one can put anything after via.
We do not propose a solution to this problem here, but instead note that issues with Deriving Via error quality are ultimately issues with \texttt{coerce} error quality, given that the error messages are a result of \texttt{coerce} failing to typecheck. It is likely that investing more effort into making \texttt{coerce}'s error messages easier to understand would benefit Deriving Via as well.

### 6.2 Multi-Parameter Type Classes

GHC extends Haskell by permitting type classes with more than one parameter. Multi-parameter type classes are extremely common in modern Haskell, to the point where we assumed the existence of them in Section 3.1.1 without further mention. However, multi-parameter type classes pose an intriguing design question when combined with Deriving Via and StandaloneDeriving, another GHC feature which allows one to write \texttt{deriving} declarations independently of a data type.

For example, one can write the following instance using StandaloneDeriving:

```haskell
class Triple a b c where
tuple :: (a, b, c)
instance Triple () () () where
tuple = ((), (), ())
newtype A = A ()
newtype B = B ()
newtype C = C ()
deriving via () instance Triple A B C
```

However, the code it generates is somewhat surprising. Instead of reusing the \texttt{Triple} \((\cdot) \ (\cdot) \ (\cdot)\) instance in the derived instance, it will attempt to reuse an instance for \texttt{Triple A B} \((\cdot) \ (\cdot) \ (\cdot)\). This is because, by convention, StandaloneDeriving will only ever coerce through the \texttt{last} argument of a class. That is because the standalone instance above would be the same as if a user had written:

```haskell
newtype C = C () deriving (Triple A B) via ()
```

This consistency is perhaps a bit limiting in this context, where we have multiple arguments to \texttt{C} that one could “derive through”. But it is not clear how GHC would figure out which of these arguments to \texttt{C} should be derived through, as there seven different combinations to choose from! It is possible that another syntax would need to be devised to allow users to specify which arguments should be coerced to avoid this ambiguity.

### 7 Conclusions

In this paper, we have introduced the Deriving Via language extension, explained how it is implemented, and shown a wide variety of use cases. We believe that Deriving Via has the potential to dramatically change the way we write instances, as it encourages giving names to recurring patterns and reusing them where needed. It is our feeling that most instance declarations that occur in the wild can actually be derived by using a pattern that deserves to be known and named, and that instances defined manually should become an anti-pattern in all but some rare situations.

### Acknowledgements

We would like to thank Richard Eisenberg for his feedback on Section 3.3, as well as the first author’s former colleagues at Standard Chartered Bank for their feedback.

### References


