CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Deriving Via

Type-directed instances

Master's thesis in Algorithms, Languages and Logic

BALDUR BLÖNDAL

Computer Science and Engineering Department
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2020

Master's thesis 2020

Deriving Via

Type-directed instances

BALDUR BLÖNDAL

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Computer Science and Engineering Department
Algorithms, Languages and Logic
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2020

Deriving Via
Type-directed instances
Baldur Blöndal

Supervisor: Mary Sheeran, Computer Science and Engineering
Examiner: Mary Sheeran, Computer Science and Engineering

Master's Thesis 2020
Computer Science – algorithms, languages and logic
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2020

Deriving Via
Type-directed instances
Baldur Blöndal
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Type classes are at the heart of Haskell and constitute a language of type-directed behaviour. The programmer defines behaviour for each type by defining a class *instance* where the compiler transparently fills in the blanks with code. Types guide this process.
Each type has a single instance: what if there is more than one way to act? The established approach is to wrap such a type in a `newtype`, ensuring it has the same memory representation. `newtype`s require require laborious manual wrapping and unwrapping which have no effect at runtime.
Haskell's `deriving` construct allows easily generating instances that follow a common pattern by simply listing the classes you want derived. At present, GHC only supports `deriving` a few classes. The only alternative is to write it by hand.
This thesis offers an alternative: the language extension `-XDerivingVia` (appeared in *GHC 8.6*) and the *GHCi* command `:instances` (appeared in *GHC 8.10*) which lists instances of types. The Deriving Via introduces a new deriving strategy `via` which allows deriving classes from one or more '*via types*'. These types must be identical (at runtime) to the type we are deriving for. We instantiate behaviour at a *via type* and then coerce it to our type. The `:instances` command lists candidates that can be derived via a given type.
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.

Keywords: Thesis, deriving, Haskell, program synthesis, type classes, safe coercions.

Acknowledgements

I would like to thank Andres Löh, Ryan Scott for making the paper happen and Simon Peyton Jones for his encouragement and his review. I also thank anonymous reviewers for their suggestions. Thanks to Richard Eisenberg for his feedback on Section 4.3, as well as my former colleagues at Standard Chartered Bank for their feedback.

Baldur Blöndal, Gothenburg, May 2020

Contents

1
Introduction

Why repeat yourself when the compiler can write it?
The purely functional programming language Haskell has excellent facilities for abstracting repeated patterns. One such example instance derivation.
In Haskell, a set of operations is specified by a *type class* and a data type can be given a class *instance* that specifies the behaviour of that type under those operations.
These instances can be written by hand but are often so formulaic that the task of generating them can be relegated to the compiler, freeing the programmer from writing any code. This is an example of a `deriving` clause. With time the scope of `deriving` expanded, adding support for new built-in functionality to be derived as well as functionality defined by the programmer who is no longer dependent on compiler writers.
Despite these advancements there are still numerous limitations on what can be derived; there are many cases where line-for-line identical code can not be derived and should a programmer wish to parameterise existing `deriving` clauses with their own configuration values they must forego the niceties of `deriving` and write it by hand.
This thesis introduces *Deriving via*, a simple zero-cost language extension with far-reaching consequences and a *ghci* command `:instances` that lists instances that can derive via a given type. This thesis aims to answer the research questions

1. Can boilerplate in instance declarations be reduced to composable behaviours? This should be done leveraging the Haskell ecosystem.
2. Can we list the behaviours a given type has?

The first half introduces the extension and relevant machinery. The consequences will be explored in the latter half of the thesis.

1.1 Haskell and Type Classes

The type class system of Haskell is one of its most fundamental aspects. It uses open ad-hoc polymorphism[1] to capture common interfaces.
The operations of a type class are its *methods* and a class *instance* is an implementation of those methods for a data type, this specifies the semantics of the class. Openness means other users can independently define new types and instances for them.
To showcase typeclasses we introduce *semigroup*; an algebraic structure with an associative binary operator. This structure is captured by the type class `Semigroup` that lives in Data.Semigroup from base, a core library that ships with the Glasgow

Haskell Compiler (GHC). The first line requires `-XStandaloneKindSignatures` (see chapter 2) indicating that it is defining a type class constraint for a given type `a`.

```
type  Semigroup :: Type -> Constraint
class Semigroup a where
  (<>) :: a -> a -> a
```

The `Semigroup` class specifies the binary operator `<>` parameterised over a single type *a* which scopes over the class body. The associativity condition is not checked by the compiler, the programmer is expected to make sure the implementation adheres to the law:

```
a <> (b <> c) = (a <> b) <> c
```

This class declaration defines two things, a `Semigroup` constraint and a binary operator constrained by it. This is how they appear in the GHCi interactive interpreter:

```
>> :kind Semigroup
Semigroup :: Type -> Constraint
>> :type (<>)
(<>) :: Semigroup a => a -> a -> a
```

The left-hand side of the fat arrow `=>` constrains `a` to types with a semigroup instance. Having defined a type class we can define instances of types.

The central theme of this thesis is the type-directed form of programming that type classes enable and how types direct the behaviour of a program.

The Semigroup behaviour of lists (`[a]`) generalises the standard Haskell operator `++` for appending lists. Writing signatures of instances requires the `-XInstanceSigs` extension described in chapter 2.

```
-- >> "one" <> " " <> "two"
-- "one two"
instance Semigroup [a] where
  (<>) :: [a] -> [a] -> [a]
  (<>) = (++)
```

This is the base `Semigroup` instance for lists but it is not the only one we could have given. Lists, indeed any type, forms a semigroup where the operator `<>` returns its first argument (dropping its second) and vice versa.

1.1.1 Extending classes

Now that we have defined `Semigroup` it is time to extend it with more operations. We define a type class `Monoid` which extends the `Semigroup` class with a `mempty` method. We say that `Monoid` is a *subclass* of `Semigroup` indicated by a fat arrow (`Semigroup a => ..`). Conversely `Semigroup` is a *superclass* of `Monoid`:

```
type  Monoid :: Type -> Constraint
class Semigroup a => Monoid a where
  mempty :: a
```

This means that `Monoid` inherits all the methods of its superclass `Semigroup` and extends it with one of its own.

```
>> :kind Monoid
Monoid :: Type -> Constraint
```

```
>> :type mempty
mempty :: Monoid a => a
```

`mempty` serves as an identity element for `<>`. The associativity law of semigroup is additionally extended with the following laws which the programmer is expected to verify:

```
a <> mempty = a
mempty <> b = b
```

For lists this is the empty list:

```
instance Monoid [a] where
  mempty :: [a]
  mempty = []
```

The `:instances` command lists the instances of `[a]` for an arbitrary type, written with a partial type (_)

```
>> :set -XPartialTypeSignatures -Wno-partial-type-signatures
>> :instances [_]
..
instance Semigroup [_]
instance Monoid [_]
```

This command was proposed by the author to supplement the ideas presented in this thesis. It was implemented by Xavier Denis. This command is the positive answer to the second research question: "Can we list the behaviours a given type has?".

A subclass like `Monoid` implies its superclass `Semigroup`. It's because of this that `memptyMempty` does not need to list a `Semigroup` constraint:

```
memptyMempty :: Monoid a => a
memptyMempty = mempty <> mempty
```

1.1.2 Multiple Candidates

What follows are informal rules for constructing `Semigroup` instances that will be made concrete later on. The first two rules are applicable at any type, the third rule presupposes a `<>` definition:

'First' rule Return the first argument.

'Last' rule Return the second argument.

'Dual' rule Flip the arguments of an existing `<>` definition.

While there can only be a single `Semigroup [a]` instance the choice is clearly not unique, we could have used any of the three rules. Using the **First** rule `(<>)` is defined to be `const`, a binary function that returns its first argument. **Last** is a flipped `const` and and the third example flips the standard definition of `Semigroup [a]`. Interactions with GHCi are indicated by » in comments.

```
-- >> "one" <> " " <> "two"
-- "one"
instance Semigroup [a] where     -- First
  (<>) :: [a] -> [a] -> [a]
  (<>) = const
```

```
-- >> "one" <> " " <> "two"
-- "two"
instance Semigroup [a] where    -- Last
  (<>) :: [a] -> [a] -> [a]
  (<>) = flip const

-- >> "one" <> " " <> "two"
-- "two one"
instance Semigroup [a] where    -- Dual
  (<>) :: [a] -> [a] -> [a]
  (<>) = flip (++)
```

An observant reader may notice that **Last** is a combination of the other two rules. The rules for **First** and **Last** are valid for all types, so one might reasonaibly want to capture them with the mechanisms of type classes, as instances. Here is a "first" attempt to capture **First** as an instance:

```
instance Semigroup a where    -- First
  (<>) :: a -> a -> a
  (<>) = const
```

Unfortunately, this general instance is undesirable because it overlaps with all other instances. Standard instance resolution of a constraint like `Semigroup Bool` works by matching it against a single instance; a semigroup constraint for `Bool` does not match the list instance so there is no overlap:

```
instance Semigroup a where    -- First
  (<>) :: a -> a -> a
  (<>) = const

instance Semigroup [a] where
  (<>) :: [a] -> [a] -> [a]
  (<>) = (++)

-- yes = False
yes = False <> True
```

Constraints such as `Semigroup [Bool]` do however match more than one instance, as the compiler informs us:

```
--   • Overlapping instances for Semigroup [Bool] arising from a
--     use of '<>'
--   Matching instances:
--     instance Semigroup a
--     instance Semigroup [a]
no = [False] <> [True]
```

Additionally the **Last** rule cannot coexist with **First**:

```
--   Duplicate instance declarations:
--     instance Semigroup a
--     instance Semigroup a
instance Semigroup a where (<>) = const         -- First
instance Semigroup a where (<>) = flip const    -- Last
```

The same situation crops up with arithmetic and Boolean operations. There are many such associative operations, most notably addition, multiplication, conjunction and disjunction:

```
instance Semigroup Int where     -- Sum
  (<>) :: Int -> Int -> Int
  (<>) = (+)

instance Semigroup Int where     -- Product
  (<>) :: Int -> Int -> Int
  (<>) = (*)

instance Semigroup Bool where     -- All
  (<>) :: Bool -> Bool -> Bool
  (<>) = (&&)

instance Semigroup Bool where     -- Any
  (<>) :: Bool -> Bool -> Bool
  (<>) = (||)
```

We now turn to the solution.

1.2 `newtypes` — "datatype renamings"

When a type has many possible instances the standard approach is to introduce a new type that wraps the existing type. This new type can be given a distinct instance from its underlying type.

Haskell supports the definition of new types represented as other types in memory, using `newtype`. The underlying type is called the *representation type*.

`newtype`s are algebraic datatypes with a single constructor taking one argument exactly. A newtype declaration is *generative*, declaring a new type that is not equal to any existing type.

The key reason for using `newtype` instead of Haskell's `data` declaration is that `Sum a` and `Product a` are both guaranteed to have the same representation in memory as their representation type `a`. All three types are said to be *representationally equal* which will be described in more detail in section 1.3.

```
type    Sum :: Type -> Type
newtype Sum a = Sum a

type    Product :: Type -> Type
newtype Product a = Product a
```

There are in fact no `Semigroup` instances for numbers or `Bool` in GHC base. Instead of making one operation default the designers of GHC chose to capture addition and multiplication as instances of `newtype`s:

```
instance Num a => Semigroup (Sum a) where
  (<>) :: Sum a -> Sum a -> Sum a
  Sum a <> Sum a = Sum (a + b)
```

```haskell
instance Num a => Semigroup (Product a) where
  (<>) :: Product a -> Product a -> Product a
  Product a <> Product b = Product (a * b)
```

Same for disjunction and conjunction:

```haskell
type    All :: Type
type    Any :: Type
newtype All = All Bool
newtype Any = Any Bool

newtype Semigroup All where
  (<>) :: All -> All -> All
  All a <> All b = All (a && b)

instance Semigroup Any where
  (<>) :: Any -> Any -> Any
  Any a <> Any b = Any (a || b)
```

Notice the boilerplate that results from wrapping and unwrapping. Unlike previous definitions (`(<>) = (+)`) we must now unwrap the arguments and wrap the return value of these `newtype` instances. This drudgery is a longstanding complaint in the Haskell community.

The rules mentioned previously are made concrete as instances of `newtypes`, these all exist in GHC base:

```haskell
type    First :: Type -> Type
type    Last  :: Type -> Type
type    Dual  :: Type -> Type
newtype First a = First a
newtype Last  a = Last  a
newtype Dual  a = Dual  a
instance Semigroup (First a) where
  (<>) :: First a -> First a -> First a
  First a <> First _ = First a    -- (<>) = const

instance Semigroup (Last a) where
  (<>) :: Last a -> Last a -> Last a
  Last _ <> Last b = Last b    -- (<>) = flip const

instance Semigroup a => Semigroup (Dual a) where
  (<>) :: Dual a -> Dual a -> Dual a
  Dual a <> Dual b = Dual (b <> a)
```

The left-hand side of the double arrow `=>` is the context of the instance. It says that `Dual a` can only be a semigroup if `a` is a semigroup.

As previously mentioned these definitions can be composed. `First` and `Last` behave the same as `Dual` of the other. We can obtain the same behaviour (modulo `newtype` wrapping) by defining new versions as:

```haskell
type First' :: Type -> Type
type Last'  :: Type -> Type
```

```
type First' a = Dual  (Last a)
type Last'  a = Dual (First a)
```

We can view the instances of these (composite) types using the newly released *ghci* command `:instances` which lists instances of any type. It shows that `First' a` and `Last' a` are unconditionally semigroups, just like `First a` and `Last a`:

```
>> :instances First' _
instance Semigroup (Dual (Last _))
..
>> :instances Last' _
instance Semigroup (Dual (Last _))
..
```

This is a benefit of `:instances` over the command `:info` which only works for base types. Now we can query what nested behaviour we can derive from nested types. This composition of behaviour is as if we were using the definition:

```
instance Semigroup (First' a) where
  (<>) :: First' a -> First' a -> First' a
  Dual (Last a) <> Dual (Last _) = Dual (Last a)

instance Semigroup (Last' a) where
  (<>) :: Last' a -> Last' a -> Last' a
  Dual (First _) <> Dual (First b) = Dual (First b)
```

We invoke it by wrapping and unwrapping its argument and result:

```
>> Dual (Last 1) <> Dual (Last 2)
Dual (Last 1)
>> Dual (First 1) <> Dual (First 2)
Dual (First 2)
```

1.3 Representational equality

The usual notion of type equality in Haskell is *nominal equality* (equality at compile time) written with a tilde ~. It is extended by equality at runtime, *representational equality*, which holds when two types share the same run-time representation. It is written *Coercible* and indicates that two types may be safely coerced to one another. Although *Coercible* constructs a constraint it is not backed by an actual type class. Instead, it can be thought of as a kind of psuedo-type class (*faux-clas*?) with custom solving rules and no user-defined instances[2].

These special *Coercible* rules are determined by what `newtype` constructors are in scope so when types are representationally equal as `Bool`, `Any` and `All` are coercible to one another.

Any *Coercible* types can be `coerced`; the function `coerce` taps into the power of representational equality, a function that safely converts between coercible types:

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

`coerce` is a zero-cost operation, it changes the type but has no effect operationally.

```
coerce :: Bool         -> Any
coerce :: Any          -> All
coerce :: (Any, Bool) -> (Bool, All)
```

This slices through arbitrary layers of type constructors like a hot knife, at arbitrary nesting depth:

```
coerce :: Maybe [[(    [Int],  All)]]
       -> Maybe [[(Sum [Int], Bool)]]
```

This holds even for function types which can be coerced to remove all layers of a `newtype`. This uses the visibility override `@` enabled by the visible `-XTypeApplications` extension.

```
(<>)     @(Sum a) :: Num a =>     Sum a ->     Sum a ->     Sum a
(<>) @(Product a) :: Num a => Product a -> Product a -> Product a
```

The type variables are brought into scope by `forall` with the `-XScopedTypeVariables` extension.

```
add :: forall a. Num a => a -> a -> a
add = coerce do
  (<>) @(Sum a)

mul :: forall a. Num a => a -> a -> a
mul = coerce do
  (<>) @(Product a)
```

To reduce parentheses I use `-XBlockArguments` allowing functions to take `do`-block directly as an argument. All these extensions are discussed in chapter 2.

1.4 newtypes for behaviour

This is a powerful ability. With one hand we instantiate types, specifying potentially nested behaviour of arbitrary complexity. With the other we remove all trace of them.

It means we can define

```
first :: forall a. a -> a -> a
first = coerce do (<>)                   @(First a)
first = coerce do (<>)             @(Dual (Last  a))
first = coerce do (<>)       @(Dual (Dual (First a)))
first = coerce do (<>) @(Dual (Dual (Dual (Last  a))))

last :: forall a. a -> a -> a
last = coerce do (<>)                   @(Last  a)
last = coerce do (<>)             @(Dual (First a))
last = coerce do (<>)       @(Dual (Dual (Last  a)))
last = coerce do (<>) @(Dual (Dual (Dual (First a))))
```

This works equally with derived functions that use those methods. A function like `mconcat` which turns a list of monoidal values `[a,b,c,d]` into `a<>b<>c<>d<>mempty`. These definitions are equal ignoring bottom:

```
mconcat :: Monoid a => [a] -> a
mconcat = foldr (<>) mempty
sum :: forall a. Num a => [a] -> a
sum = coerce do mconcat                   @(Sum a)
sum = coerce do mconcat             @(Dual (Sum a))
```

```
sum = coerce do mconcat       @(Dual (Dual (Sum a)))
sum = coerce do mconcat @(Dual (Dual (Dual (Sum a))))

product :: forall a. [a] -> a
product = coerce do mconcat             @(Product a)
product = coerce do mconcat       @(Dual (Product a))
product = coerce do mconcat @(Dual (Dual (Product a)))
```

..

There is no sensible "first" or "last" element of an empty list. Correspondingly those `newtypes` have no lawful `Monoid` instance so they don't work for `mconcat`.

The fundamental data type `data Maybe a = Nothing | Just a` can be used to lift a `Semigroup` to a `Monoid` by adjoining `Nothing` to act as `mempty`:

```
type LiftSemigroup :: Type -> Type
type LiftSemigroup = Maybe

instance Semigroup a => Semigroup (Maybe a) where
  (<>) :: Maybe a -> Maybe a -> Maybe a
  Nothing <> as'     = as'
  as      <> Nothing = as
  Just a  <> Just a' = Just (a <> a')

instance Semigroup a => Monoid (Maybe a) where
  mempty :: Maybe a
  mempty = Nothing
```

Combining the behaviour of the `LiftSemigroup` with `First` and `Last` turns their `Semigroup` instances to `Monoid` instances with an ability to describe empty strutures.

```
type LiftFirst :: Type -> Type
type LiftList  :: Type -> Type
type LiftFirst a = LiftSemigroup (First a)
type LiftLast  a = LiftSemigroup (Last  a)
```

We now reap benefits from `:instances`. Unlike the `:info` command which lists information and instances of ground types our new command has the ability to enumerate instances of arbitrarily nested types.

```
>> :instances LiftFirst _
instance Monoid    (Maybe (First _))
instance Semigroup (Maybe (First _))
```

..

```
>> :instances LiftLast _
instance Semigroup (Maybe (Last _))
instance Monoid    (Maybe (Last _))
```

They now have a sensible answer for the first and last element of an empty list:

```
-- >> liftFirst []
-- Nothing
-- >> liftFirst [Just 'A', Nothing, Just 'Z']
-- Just 'A'
liftFirst :: forall a. [Maybe a] -> Maybe a
```

```haskell
liftFirst = coerce $ mconcat @(LiftFirst a)

-- >> liftLast []
-- Nothing
-- >> liftLast [Just 'A', Nothing, Just 'Z']
-- Just 'Z'
liftLast :: forall a. [Maybe a] -> Maybe a
liftLast = coerce $ mconcat @(LiftLast a)
```

The `First` and `Last` semigroups we have been using come from `Data.Semigroup`. Confusingly a different set of `newtypes` come from `Data.Monoid` with a `Monoid` instance from our `LiftFirst` and `LiftLast`.

1.5 Problem Description

We said that type classes capture interfaces but 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:

```haskell
instance Semigroup a => Semigroup (IO a) where
  (<>) = liftA2 (<>)
instance Semigroup a => Semigroup (ST s a) where
  (<>) = liftA2 (<>)

instance Monoid a => Monoid (IO a) where
  mempty = pure mempty
instance Monoid a => Monoid (ST s a) where
  mempty = pure mempty
```

Notice, these have completely identical instance bodies (method signatures are omitted to preserve this) 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.

One might think that that situation is different this time around. After all this is not a catch-all rule but is limited to applicatives applied to semigroups and monoids:

```haskell
instance (Applicative f, Semigroup a) => Semigroup (f a) where
  (<>) :: f a -> f a -> f a
  (<>) = liftA2 (<>)

instance (Applicative f, Monoid a) => Monoid (f a) where
  mempty :: f a
  mempty = pure mempty
```

Unfortunately, this general instance is undesirable. Because instance resolution matches the instance head first without considering the context this instance overlaps with all other instances of applied types `(f a)` whether they are headed by an applicative functor or not. Once GHC has commited to an instance, it will never

backtrack.[1]
Consider:

```
type    Endo :: Type -> Type
newtype Endo a = Endo (a -> a) -- Data.Monoid
```

Here, `Endo` is not an `Applicative` and not even a `Functor` (it is invariant since `a` appears in input and output position). `Endo a` would still match the above instance and promptly get rejected for lack of an `Applicative`.
It still admits a perfectly valid `Monoid` instance that overlaps with the above `f a` instance:

```
instance Semigroup (Endo a) where
  (<>) :: Endo a -> Endo a -> Endo a
  Endo f <> Endo g = Endo (f . g)

instance Monoid (Endo a) where
  mempty :: Endo a
  mempty = MkEndo id
```

This rule holds for any `Category` so the instance for `Endo a` can be captured by a different rule.

```
instance Category cat => Semigroup (cat a a) where
  (<>) :: cat a a -> cat a a -> cat a a
  (<>) = (.)

instance Category cat => Monoid (cat a a) where
  mempty :: cat a a
  mempty = id
```

`Endo a` does not have the right shape to match the instance `cat a a` as it expects a type constructor applied to two arguments. Even so the monoid instance for `Endo a` and `a -> a` are equal at runtime.
Since functions are categories the monoid instance for `Endo a` can be captured by a *different* rule. Even though `Endo a` doesn't have the right shape to match the monoid instance for `((->) a a)` they share the same representation in memory:
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) where `<>` is list concatenation whose identity is the empty list. These instances do not coincide with the rule above and in particular impose no constraint on `a`:

```
instance Semigroup [a] where
  (<>) :: [a] -> [a] -> [a]
  (<>) = (++)

instance Monoid [a] where
  mempty :: [a]
  mempty = []
```

This can be captured by yet another rule, this time headed by an `Alternative`:

[1] Backtracking is "fundamentally anti-modular." as discussed by Edward Kmett[3].

```
instance Alternative f => Semigroup (f a) where
  (<>) :: f a -> f a -> f a
  (<>) = (<|>)

instance Alternative f => Monoid (f a) where
  mempty :: f a
  mempty = empty
```

Because instance resolution doesn't backtrack, we can't define rules that are given by `Applicative` as well as `Alternative`. This is true even with overlapping instances, and there is no way for `Endo a` to even match the rule given by `Category`.

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 clear that we are instantiating a general principle.
- We do not express this general principle as code with a name and documentation. Its use must be communicated in another way; by comments or as folklore. This makes it difficult to search for and discover. Our code has lost a connection to its intention.
- There are many such rules, some quite obvious, but others more surprising and easy to overlook.
- Combining rules must be done manually.
- While the work required to define instances manually for `Semigroup`, `Monoid`—totalling only two methods—is perhaps acceptable, it quickly becomes unwieldy 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:[2]

```
class Num a where
  (+), (-), (*)       :: a -> a -> a
  negate, abs, sinum :: a -> a
  fromInteger         :: Integer -> a
instance (Applicative f, Num a) => Num (f a) where
  (+), (-), (*) :: f a -> f a -> f a
  (+) = liftA2 (+)
  (-) = liftA2 (-)
  (*) = liftA2 (*)

  negate, abs, signum :: f a -> f a
  negate = liftA negate
  abs    = liftA abs
  signum = liftA signum

  fromInteger :: Integer -> f a
  fromInteger = pure . fromInteger
```

Defining such boilerplate instances manually for concrete type constructors is so

[2]Similarly for `Floating` and `Fractional`, numeric type classes with a combined number of 25 methods (15 for a minimal definition).

annoying that Conal Elliott introduced a preprocessor [4] for this particular use case several years ago.

1.6 Deriving

Readers familiar with Haskell's deriving mechanism (discussed in section 2.3) may wonder why we cannot simply derive all the instances we just discussed. Unfortunately, our options are very limited.

To start, `Semigroup` and `Monoid` are not one of the few blessed type classes that GHC has built-in support to derive. It so happens that IO a, `ST s a` and `Endo a` are all newtypes, so they are in principle eligible for *generalized newtype deriving* (GND), in which their instances could be derived by reusing the instances of their underlying types [5]. 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 [6]. 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 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.

1.7 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. As before 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 semigroup and 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 [7] and wrapping the instance head in it:[3]

```
type    Ap :: (k -> Type) -> (k -> Type)
newtype Ap f a = Ap (f a)
```

The first line (`type Ap :: ...`) is a *standalone kind signature* enabled the an extension with the same name explained in chapter 2.

```
instance (Applicative f, Semigroup a) => Semigroup (Ap f a) where
  (<>) :: Ap f a -> Ap f a -> Ap f a
  Ap f <> Ap g = Ap (liftA2 (<>) f g)

instance (Applicative f, Monoid a) => Monoid (Ap f a) where
  mempty :: Ap f a
  mempty = Ap (pure mempty)
```

If we derive an applicative instance for `Ap f` with *generalized newtype deriving* we get rid of the wrapping and unwrapping of newtypes in the `Semigroup` and `Monoid` instances. The resulting instance bodies now look like the overlapping instances in the beginning:

```
newtype Ap f a = Ap (f a)
  deriving
  newtype (Functor, Applicative)

instance (Applicative f, Semigroup a) => Semigroup (Ap f a) where
  (<>) :: Ap f a -> Ap f a -> Ap f a
  (<>) = liftA2 (<>)

instance (Applicative f, Monoid a) => Monoid (Ap f a) where
  mempty :: Ap f a
  mempty = pure mempty
```

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

```
data Opt a = No | Yes a
  deriving (Semigroup, Monoid)
  via Ap Opt a

instance Functor     Opt ..
instance Applicative Opt ..
```

This requires that we independently have an `Applicative` instance for `Opt`, 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 by reusing the `Semigroup` and `Monoid` instances already available for the `via` type—`Ap Opt a`. It should be easy to see why this works: due to the use of a newtype, `Ap Opt a` has the same internal representation as `Opt a`, and any instance available on one type can be made to work on the other by suitably wrapping or unwrapping a newtype. `Ap Opt a` and `Opt a` are *representationally equal* and thus a binary function on one is representationally

[3] Has since been added to base as `Data.Monoid.Ap` in GHC 8.6.

equal to a binary function on the other.
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` instance is already available through the Data.Monoid.Alt newtype (now deriving `Alternative` with newtype deriving to avoid wrapping and unwrapping):

```
type    Alt :: (k -> Type) -> (k -> Type)
newtype Alt f a = Alt (f a)
  deriving
  newtype (Functor, Applicative, Alternative)

instance Alternative f => Semigroup (Alt f a) where
  (<>) :: Alt f a -> Alt f a -> Alt f a
  (<>) = (<|>)

instance Alternative f => Monoid (Alt f a) where
  mempty :: Alt f a
  mempty = empty
```

We can also define an adapter to capture the `Endo` instances

```
type    Join :: (k -> k -> Type) -> (k -> Type)
newtype Join cat a = Join (cat a a)

instance Category cat => Semigroup (Join cat a) where
  (<>) :: Join cat a -> Join cat a -> Join cat a
  Join f <> Join g = Join (f . g)

instance Category cat => Monoid (Join cat a) where
  mempty :: Join cat a
  mempty = Join id
```

Equipped with adapters such as `Ap`, `Alt` and `Join` many `Monoid` instances that had to be written by hand can be derived using the `via` construct.

1.8 Contributions and structure of the thesis

This thesis is based off an existing paper "Deriving Via: or, How to Turn Hand-Written Instances into an Anti-pattern"[8] written by Andres Löh, Ryan Scott and myself. I proposed a design for `:instances`[9] on GitLab (which was then known as GHC Trac). That design was eventually implemented by Xavier Denis[10].
The thesis is structured as follows: In Section 3, we use the QuickCheck library as a case study to explain in more detail how Deriving Via can be used, and how it works. In particular how rules can be combined. In Section 4, we explain in detail how to typecheck and translate Deriving Via clauses. In Section 5.1, we discuss several additional applications of Deriving Via. We discuss related ideas in Section 7.1, describe the current status of our extension in Section 7.2 and conclude in Section 7.3.

Deriving Via is fully implemented and landed in GHC 8.6 and `:instances` landed in GHC 8.10.
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 4.2.1).
- It additionally generalizes the concept of *default signatures*. (Section 5.1.4).
- It provides a possible solution to the problem of introducing additional boilerplate code when introducing new superclasses (such as `Applicative` for `Monad`, Section 5.1.6).
- It allows for reusing instances not just between representationtally equal types, but also between isomorphic or similarly related types (Section 5.1.5).

1.9 Statement of Contribution

I started thinking about this during my time at Standard Chartered Bank. The original motivator was making use of isomorphisms between datatypes to derive functionality for one from the other. It turned out to be cumbersome before realizing that deriving via isomorphisms is a special case of deriving via representationally-equal types.
Following that I attended ICFP 2017 where I was able to discuss the idea. Simon Peyton Jones encouraged me to turn it into a fully-fledged paper. I was uncertain I would finish it on my own so he got me in contact with Ryan Gl. Scott who was familiar with GHC's deriving mechanism. Andres Löh also expressed interest in the project at ICFP.
After ICFP, I together with Ryan and Andres would discuss and split up work on the paper for the next half-a-year with weekly video calls, finally submitting it to the Haskell Symposium 2018 where it was accepted.[8]
During the development of the extension I collected unusual or seemingly impossible use cases that guided the design of the extension.
Ryan implemented the idea in GHC. It was a simple and direct generalisation of `-XGeneralizedNewtypeDeriving` where a lot of the code was already in place.

2

Background

This thesis uses several GHC extensions that will be discussed here.
I write type signatures for methods when writing instances of type classes, this is enabled by `-XInstanceSigs`

```
instance Eq ABC where
  (==) :: Int -> Int -> Bool    -- enabled by -XInstanceSigs
  A == A = True
  B == B = True
  C == C = True
  _ == _ = False
```

Arguments of `->` are given explicitly by the programmer while type arguments on the other hand are solved by unification. We say that the quantifiee `a` (argument being quantifed over) of `a -> ...` is *visible* while the quantifiee of `forall a. ...` is *invisible*:

```
pure :: forall f a. Applicative f => a -> f a
```

The syntax `@` is used to override the visibility of type arguments and is enabled by the extension `-XTypeApplications`

```
pure @Maybe     :: forall a. a  -> Maybe a
pure @Maybe @() ::           () -> Maybe ()
```

Kinds signatures (especially of type classes) will be specified with the extension `-XStandaloneKindSignatures` which appeared in GHC 8.10:

```
type  Eq :: Type -> Constraint
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

2.1 Kinds

Just as types classify values, kinds classify types. As originally defined in the Haskell98 report[11], kinds were extremely minimal, following the simple grammar

```
Kind = Type | Kind -> Kind
```

`Type` is the kind of proper, inhabited types such as:

```
Int              :: Type
()               :: Type
Int->()->Double  :: Type
Maybe ()         :: Type
Maybe Int        :: Type
```

Type constructors like `Maybe` and `(->)` can appear without their arguments, in which case they are type level functions[12] with no computational content.

```
[]        :: Type -> Type
Maybe     :: Type -> Type
IO        :: Type -> Type
(->) Int  :: Type -> Type

Either :: Type -> Type -> Type
(->)   :: Type -> Type -> Type
```

Constructors can appear as arguments to datatypes as well like `AtIntBool` which contains a value of that constructor at `Int` and at `Bool`:

```
type AtIntBool :: (Type -> Type) -> Type
data AtIntBool f where
  Pair :: f Int -> f Bool -> AtIntBool f
```

An `AtIntBool []` would then contain a list of `Int`s and `Bool`s

```
Pair @[] :: [Int]  -> [Bool]  -> AtIntBool []
Pair @IO :: IO Int -> IO Bool -> AtIntBool IO
```

Just as we can have polymorphism at the term level with `forall.` we can also express

2.2 Type Classes

Extending the discussion in the introduction:

2.2.1 Default methods and Minimal pragmas

Haskell 98 supports specifying default implementations of methods, where the type of the default method matches the method type.
Using the type class for equality as an example, if the `/=` method is omitted then the definition `a /= b = not (a == b)` is assumed.
This is how Eq appears in base:

```
type  Eq :: Type -> Constraint
class Eq a where
  {-# Minimal (==) | (/=) #-}

  (==) :: a -> a -> Bool
  a == b = not (a /= b)

  (/=) :: a -> a -> Bool
  a /= b = not (a == b)
```

The default methods along with the `{-# Minimal .. #-}` pragma, specifies to the compiler that at least one of the two methods must be defined.
With the extension `-XDefaultSignatures` it is possible to supply methods with a more restrictive type. We can always compare `Enum`
for example knowing we are comparing `Enum`s for equality

```
type  Enum :: Type -> Constraint
class Enum a where
  toEnum   :: Int -> a
  fromEnum :: a -> Int
```

it would be possible to define a default methods for `Eq` that compares their `Int` values for equality:

```
type  Eq :: Type -> Constraint
class Eq a where
  (==) :: a -> a -> Bool
  default
    (==) :: Enum a => a -> a -> Bool
  a == b = fromEnum a == fromEnum b

  (/=) :: a -> a -> Bool
  a /= b = not (a == b)
```

Using the deriving strategy `anyclass` (which generates a type class without providing method bodies) we can derive `Eq` instances for any `Enum` type, in doing so we commit to this single path between the two.

It could alternatively be defined as a `newtype` instance

```
type    Enumy :: Type -> Type
newtype Enumy a = Enumy a
  deriving
  newtype Enum

instance Enum a => Eq (Enumy a) where
  (==) :: Enumy a -> Enumy a -> Bool
  a == b = fromEnum a == fromEnum b
```

and this can now be derived with Deriving Via for any type that has an `Enum` instance

```
data A = ..
  deriving
  stock Enum

  deriving Eq
  via Enumy A
```

2.2.2 Associated type families

The primary purpose of type classes is to achieve ad-hoc overloading by indexing functions (class methods) by types but they can be used to index data types and type synonyms as well. This thesis only considers the case of synonyms, known as *associated type families*.[13] They are functions on types which must be defined once for each instance.

An example where this is useful is representable functors. They are types that can be equally represented as functions from a particular *representing type* (not to be confused with *representation type* or *representable equality*). A polymorphic pair, or a 2-D vector is representable

```
type V2 :: Type -> Type
data V2 a = V2 a a
```

In this case `V2 a` is equivalent to the function type `Bool -> a` so we say that `Bool` is the *representing type* of `V2`.
This representing type can be made precise by defining an associated type family `Rep`:

```
type  Representable :: (Type -> Type) -> Constraint
class Functor f => Representable f where
  type Rep f :: Type

  index    :: f a -> (Rep f -> a)
  tabulate :: f a <- (Rep f -> a)
```

This really describes a natural isomorphism between the functors `f` and function types `(Rep f ->)`,

```
index    :: f ~> (Rep f ->)
tabulate :: f <~ (Rep f ->)
```

We can now formally define `V2` to be representable

```
instance Representable V2 where
  type Rep V2 = Bool

  index :: V2 a -> (Bool -> a)
  index (V2 fls tru) = \case
    False -> fls
    True  -> tru

  tabulate :: (Bool -> a) -> V2 a
  tabulate make = V2 (make False) (make True)
```

2.3 Deriving

Many datatypes have instances that follow directly from the structure of the type. Using `deriving` automatically generates these *derived instances*. At first these derived instances could only be generated for *built-in* classes, which became known as the ***`stock`*** *deriving strategy* as other strategies were added.
Currently three strategies exist

`stock` Deriving strategies hard coded into the compiler: The Haskell Report specifies `Eq`, `Ord`, `Enum`, `Bounded`, `Ix`, `Show` and `Read`[1].
Language extensions to GHC enable deriving

- `Functor` (enabled by `DeriveFunctor`)
- `Foldable` (`DeriveFoldable`)
- `Traversable` (`DeriveTraversable`)
- `Generic` and `Generic1` (`DeriveGeneric`)
- `Lift` (`DeriveLift`)
- `Data` (`DeriveDataTypeable`)

[1] Haskell 98

- and `Typeable`, althought `Typeable` is now derived automatically for all types

`newtype` This deriving strategy enabled by the `GeneralizedNewtypeDeriving` extension (GND) makes use of the unique property of `newtype` that they have the same memory representation as their representation type.
This means that instances of the representation type can be `coerce`d into an instance for the new type.

`anyclass` This deriving strategy enabled by `DeriveAnyClass` which, when valid, is the same as writing an empty instance declaration.
This is useful for generic default implementation for a particular instance.

All of those deriving strategies will only derive a type class in a single way. This thesis proposes a fourth strategy `via` that generalizes `newtype` and allows multiple ways of deriving a class.
Like other deriving strategies we specify the list of classes we want to derive for our type. In addition to that we specify a *via*-type that is equal to our type at runtime, this is where we derive our instances from. For multiple types of instances we can use more than one via type.

3

Case study: QuickCheck

QuickCheck [14] is a well-known Haskell library for randomized property-based testing. At the core of QuickCheck's test-case generation functionality is the `Arbitrary` class. Its primary method is `arbitrary`, which describes how to generate suitable random values of a given size and type. It also has a method `shrink` that is used to try to shrink failing counterexamples of test properties.

```
type  Arbitrary :: Type -> Constraint
class Arbitrary a where
  {-# MINIMAL arbitrary #-}
  arbitrary :: Gen a

  shrink :: a -> [a]
  shrink _ = []
```

`MINIMAL` is a pragma that specifies the minimal complete `Arbitrary` definition. As `shrink` has a default definition we only need to provide `arbitrary`.

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 sufficiently expressed in their types. Depending on the context and the situation, we might want to guarantee that we generate positive integers, or non-empty lists, or even sorted lists.

The QuickCheck library provides a number of newtype-based adapters (called *modifiers* in the library) for this purpose. As an example, QuickCheck defines:

```
type    NonNegative :: Type -> Type
newtype NonNegative a = MkNonNegative 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 numbers. 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 existing infrastructure of modifiers without paying the price of cluttering up our data type definitions. We can define a domain-specific newtype and specify how `Arbitrary` should be derived, the simplest option is to derive via `Int`

```
newtype Duration = MkDuration Int -- in seconds
  deriving Arbitrary
  via Int
```

This declaration has exactly the same effect as using the `GeneralizedNewtypeDeriving` extension to derive the instance: because `Int` and `Duration` have the same run-time representation. We can reuse the instance for `Int` which this allows negative durations.
If we want to restrict ourselves to non-negative durations, we replace this by

```
newtype Duration = MkDuration Int
  deriving Arbitrary
  via NonNegative Int
```

If `:instance` is integrated into the editor our programming environment could suggest it as a derivable instance:

```
>> :instances NonNegative Int
..
instance Arbitrary (NonNegative Int)
  -- Defined in ‘Test.QuickCheck.Modifiers’
```

This `Arbitrary` instance generates non-negative durations. The deriving clause changes but the data type itself does not. If we later want positive durations, we replace `NonNegative` with `Positive` in the deriving clause. In particular, we do not have to change any constructor names anywhere in our code.

3.1 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

```
type    Large :: Type -> Type
newtype Large a = MkLarge a
```

with a corresponding `Arbitrary` instance for large numbers:

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

For our `Duration` type, we can easily write

```
type    Duration :: Type
newtype Duration = MkDuration Int
  deriving Arbitrary
  via NonNegative (Large Int)
>> :instances NonNegative (Large Int)
..
instance Arbitrary (NonNegative (Large Int))
  -- Defined in ‘Test.QuickCheck.Modifiers’
```

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 we can use its `Arbitrary` instance.

3.2 Adding new modifiers

Of course, we can add 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 data types may have other reasonable generic behaviours. 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
```

We use this to define a new "rule" for `Arbitrary`

```
type    BoundedEnum :: Type -> Type
newtype BoundedEnum a = MkBoundedEnum a

instance (Bounded a, Enum a) => Arbitrary (BoundedEnum a) where
  arbitrary :: Gen (BoundedEnum a)
  arbitrary = MkBoundedEnum <$> 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
  stock (Enum, Bounded)

  deriving Arbitrary
  via BoundedEnum Weekday
```

3.3 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. This instance makes use of visible type applications `@low` and `@high` described in chapter 2.

```
type    Between :: Nat -> Nat -> Type
newtype Between low high = MkBetween Integer

instance (KnownNat low, KnownNat high)
      => Arbitrary (low `Between` high) where
```

```
  arbitrary :: Gen (low `Between` high)
  arbitrary = MkBetween <$> choose
   ( natVal @low  Proxy
   , natVal @high Proxy
   )
```

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

```
newtype Year = MkYear Integer
  deriving
  stock Show

  deriving Arbitrary
  via 1900 `Between` 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.

3.4 Conclusions

This showcases how Deriving Via can be used to naturally specify and derive. Types specify high-level behaviour where more complex types give composed behaviour, answering the first research question positively

1. Can boilerplate in instance declarations be reduced to composable behaviours? This should be done leveraging the Haskell ecosystem.

They are supported out of the box with the Haskell ecosystem that has a long history of assigning secondary behaviour to `newtype`s. We reap the benefits and treat types as a domain-specific language of behaviour. The type `Between` already shows Haskell's limited capability of passing values to `newtype`s. With the promise of dependent types in Haskell[12] we stand to see immediate benefits to Deriving Via where `newtype`s can be configured with arbitrary run-time values and functions.

4

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 [15], and our implementation of these ideas in GHC does so.

4.1 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 = MkS 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.

4.1.1 Aligning kinds

Suppose we are deriving the following instance:

```
data D d1 ... dm
  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 -> Type) -> Constraint` for some kinds `k1, ..., kr`. The reason is that the instance we must generate,
   ```
   instance C c1 ... cn (D d1 ... di) where ...
   ```
   requires that we can apply `C c1 ... cn` to another type `D d1 ... di` (where $i \leq m$, see Section 4.1.2). 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 ... di`, 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 `(Type -> Type)`—as `Eq` instances can only exist for types of kind `Type`.

4.1.2 Eta-reducing the data type

Note that in the conditions above, we specify `D d1 ... di` (for some `i`), instead of `D d1 ... dm`. 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 ... dm`! For instance, the following example is perfectly legitimate:

```
type Functor :: (Type -> Type) -> Constraint

class Functor f where ...

data Foo a = MkFoo a a
  deriving
  stock Functor
```

despite the fact that `Foo a` has kind `Type` and the argument to `Functor` has kind `(Type -> Type)`. 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*[1] 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 -> Type) -> 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 ... di`, we take $i = m - r$.

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

```
newtype A a = MkA a deriving Eq      via Identity a
newtype B b = MkB b deriving Functor via Identity
```

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

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

[1] The term *eta-reduction* is taken from the internals of GHC for reducing `forall a. Foo a` to `Foo`, where `a` is not free in `Foo`. It is inspired from *eta reduction* of functions where a function `a -> f a` can be replaced with the eta-reduced `f`.

4.2 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.

4.2.1 Generalized newtype deriving (GND)

The process by which Deriving Via generates code is heavily based on 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".
As we saw in Section 4.2.1, 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:

```
type    Age :: Type
newtype Age = MkAge { getAge :: Int }
  deriving
  newtype Enum
```

As stated above, GND basically generates the following code:

`enumFrom`:

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

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:

```
newtype T = MkT Int
instance Enum T where ...

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

then the following code would be generated:

```
  enumFrom = coerce (enumFrom @T)
```

This time, GHC coerces from an `enumFrom` implementation for `T` (the `via` type) to an implementation for `Age`.
Now we can see why the instances that Deriving Via can generate are a strict superset of those that GND can generate. Our earlier GND example could equivalently have been written using Deriving Via like so:

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

4.3 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*. It takes care to devise a consistent treatment for it.

4.3.1 Binding sites

Consider the following example:

```
type Foo :: Type -> Type
data Foo a = MkFoo..
  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.

4.3.2 Establishing order

This scoping order may seem somewhat 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:

```
type D :: 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 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.

4.3.3 Conclusions

This shows how to typecheck and generate code for Deriving Via. A limitation of the `deriving` mechanism is that it always derives an instance over its last parameter of a (multi-parameter) type class.

```
type  Sieve :: (Type -> Type -> Type) -> (Type -> Type) -> Constraint
class ..
   => Sieve pro f | pro -> f where
  sieve :: pro a b -> (a -> f b)

instance Sieve (->) Identity
```

We cannot derive `Sieve Fun..` from `newtype Fun a b = Fun (a->b)` because it is not the last parameter. This issue is discussed in section 6.1.
We can derive `Sieve .. Identity'` as long as we respect the functional dependency (`pro -> f`) and use something other than the function arrow.
The translation is in terms of `coerce`. Future developments described in chapter 6 give uniform translation in terms of Applying Via

```
instance Num Age where
  (+)         = (+)         @(via getAge)
  (*)         = (*)         @(via getAge)
  (-)         = (-)         @(via getAge)
  negate      = negate      @(via getAge)
  abs         = abs         @(via getAge)
  signum      = signum      @(via getAge)
  fromInteger = fromInteger @(via getAge)
```

or Instances Via

```
instance Num (Age via Int) where
  (+)         = (+)
  (*)         = (*)
  (-)         = (-)
  negate      = negate
  abs         = abs
  signum      = signum
  fromInteger = fromInteger
```

5
Use Cases

5.1 More Use Cases

We have already seen in Section 3 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.
Unfortunately, there is not enough space to document all use cases, so in this section, I 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.

5.1.1 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` are the `(<*>)` or `liftA2` methods, 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:

```
type Applicative :: (Type -> Type) -> Constraint

class Functor f => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a->b) -> f a -> f b

  (<*) :: f a -> f b -> f a
  (*>) :: f a -> f b -> f b
  (<*) = liftA2 \a _ -> a
  (*>) = liftA2 \_ b -> b

  liftA2 :: (a -> b -> c) -> (f a -> f b -> f c)
  liftA2 f as bs = f <$> as <*> bs
```

As shown here, `(<*)` and `(*>)` have default implementations in terms of `liftA2`. 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 `liftA2`, which can often run in superlinear time. One such `Applicative` is the function type `(->)`:

```
instance Applicative (a ->) where
  pure :: b -> (a -> b)
```

```
  pure = const

  (<*>) :: (a -> (b->b')) -> (a -> b) -> (a -> b')
  fs <*> bs a = fs a $ bs a

  (<*) :: (a -> b) -> (a -> b') -> (a -> b)
  (*>) :: (a -> b) -> (a -> b') -> (a -> b')
  bs <* _   = bs
  _  *> bs' = bs'
```

Note that we had to explicitly define `(<*)` and `(*>)`, as the default implementations would not have been as efficient. But `(->)` is not the only type for which this trick works—it also works for any data type that is isomorphic to `(a ->)` for some `a`. These 'function-like' types are characterized by the `Representable` type class we presented in subsection 2.2.2. This is a good deal more abstract than `(a ->)`, so it can be helpful to see how `Representable` works for `(a ->)` itself:

```
instance Representable (a ->) where
  type Rep (a ->) = a

  index    :: (a -> b) -> (a -> b)
  tabulate :: (a -> b) -> (a -> b)
  index    = id
  tabulate = id
```

With `Representable`, we can codify the `Applicative` shortcut for `(<*)` and `(*>)` with a suitable newtype:[1]

```
type    Co :: (k -> Type) -> (k -> Type)
newtype Co f a = Co (f a)
  deriving
  newtype (Functor, Representable)

instance Representable f => Applicative (Co f) where
  pure :: a -> Co f a
  pure = tabulate . pure

  (<*>) :: Co f (a -> a') -> Co f a -> Co f a'
  fs <*> as = tabulate (index fs <*> index as)

  (<*) :: Co f a -> Co f a' -> Co f a
  (*>) :: Co f a -> Co f a' -> Co f a'
  as <* _   = as
  _  *> as' = as'
```

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 to derive `Applicative` and `Monad` instances for `V2 a` using the `Representable V2` instance from subsection 2.2.2.

[1] Naming follows the *adjunctions* library `https://hackage.haskell.org/package/adjunctions-4.4/docs/Data-Functor-Rep.html`.

```
data V2 a = V2 a a
  deriving (Functor, Applicative, Monad, ..)
  via Co Pair
```

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.

5.1.2 Deriving with configuration

This lets us pass static values to instance deriving.

```
type Person :: Type
data Person = P
  { name :: String
  , age  :: Int
  , addr :: Maybe Address
  }
  deriving (Show, Read, ToJSON, FromJSON)
  via Person `EncodeAs` Config OmitNothing
```

Many of these newtypes existed a long time before `-XDerivingVia` did but can be used directly with it which is promising.

5.1.3 Equivalent Applicative definition

There is an equivalent, more symmetric definition of `Applicative` arising from category theory (characterizing Applicative as a strong lax monoidal functor)[16] that can be more convenient to define and work with[17][18]

```
type Monoidal :: (Type -> Type) -> Constraint
class
  Functor f => Monoidal f
 where
  unit  ::  f ()
  mult  ::  f a -> f b -> f (a, b)
```

To establish the equivalence between `Applicative` and `Monoidal` we define two modifiers: `WrapMonoidal` and `WrapApplicative` to derive one via the other.

```
instance Monoidal    f => Applicative (WrapMonoidal    f)
instance Applicative f => Monoidal    (WrapApplicative f)
```

This lists one direction of the isomorphism, which shows how to get `Applicative` via a `Monoidal` instance:

```
type    WrapMonoidal :: (k -> Type) -> (k -> Type)
newtype WrapMonoidal f a = WrapMonoidal (f a)
  deriving
  newtype (Functor, Monoidal)

instance Monoidal f => Applicative (WrapMonoidal f) where
  pure :: a -> WrapMonoidal f a
  pure a = a <$ unit
```

```
  (<*>) :: WrapMonoidal f (a -> a')
        -> (WrapMonoidal f a -> WrapMonoidal f a')
  fs <*> as = fmap (\(f, x) -> f x) (mult fs as)
```

Apart from codifying their relationship this gives freedom to choose the formulation that they deem most suitable.

5.1.4 Making Defaults more Flexible

As discussed in subsection 2.2.1 *default signatures* marry the type class with a particular default behaviour. This is an unfortunately common trend with type classes in general: many classes try to pick one-size-fits-all defaults that do not work well in certain scenarios, but because Haskell allows specifying only one default per method, if the provided default does not work for a programmer's use case, then she is forced to write 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 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:

```
type Pretty :: Type -> Constraint

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` [6] or generics-sop [19]. For example, using `GHC.Generics`, we can define

```
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

```
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:

```
instance Pretty Bool where
  pPrint :: Bool -> Doc
  pPrint = genericPPrint

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

```
  pPrint :: Maybe a -> Doc
  pPrint = genericPPrint
```

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

```
class Pretty a where
  default pPrint
          :: Generic a
          => GPretty (Rep a)
          => a -> Doc
  pPrint :: 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 `Generic a` and `GPretty (Rep a)` constraints. Thus, we can reduce the instances above to just

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

Although default signatures remove the need for many occurrences of boilerplate code, they also retains a significant limitation of Haskell default methods: every 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 want 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` that displays extra debugging information.

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

```
type GenericPPrint :: Type -> Type
type ShowPPrint    :: Type -> Type

newtype GenericPPrint a = MkGenericPPrint a
newtype ShowPPrint    a = MkShowPPrint    a

instance (Generic a, GPretty (Rep a)) => Pretty (GenericPPrint a) where
  pPrint :: GenericPPrint a -> Doc
  pPrint (MkGenericPPrint x) = genericPPrint x

instance Show a => Pretty (ShowPPrint a) where
  pPrint :: ShowPPrint a -> Doc
```

```
  pPrint (MkShowPPrint x) = stringToDoc (show x)
```

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

```
  deriving Pretty via GenericPPrint DataType1
  deriving Pretty via ShowPPrint    DataType2
```

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.

5.1.5 Deriving via Isomorphisms

All of the examples presented thus far in the thesis 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 3) once more and consider the data type

```
type Track :: Type
data Track = MkTrack 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[2]. 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:

```
type SameRepAs :: Type -> Type -> Type

newtype SameRepAs a b = MkSameRepAs a
```

We call this type `SameRepAs`, because it denotes that `a` and `b` have inter-`Coercible` generic representations, i.e., that

```
Rep a () `Coercible` Rep b ()
```

[2]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 focus on the case of isomorphism here for reasons of space.

holds. Furthermore, the type `SameRepAs a b` is representationally equal to `a`, which implies that `a` and `SameRepAs a b` are inter-`Coercible`.
We now witness the isomorphism between the two types via their generic representations: if they have inter-`Coercible` generic representations, we can transform back and forth between the two types using the `from` and `to` methods of the `Generic` class from `GHC.Generics` [6]. We can use this to define a suitable `Arbitrary` instance for `SameRepAs`:

```
instance ( Generic a, Generic b, Arbitrary b
         , Rep a () `Coercible` Rep b ()
         )
      => Arbitrary (a `SameRepAs` b) where

  arbitrary :: Gen (a `SameRepAs` b)
  arbitrary = MkSameRepAs . to . co . from <$> arbitrary @b where

    co :: Rep b () -> Rep a ()
    co = coerce
```

Here, we first use `arbitrary` to give us a generator of type `Gen b`, then coerce this via the generic representations into an `arbitrary` value of type `Gen a`.
Finally, we can use the following `deriving` declarations for `Track` to obtain the desired `Arbitrary` instance:

```
  deriving
  stock Generic

  deriving Arbitrary
  via Track `SameRepAs` (String, Duration)
```

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

5.1.6 Retrofitting Superclasses

On occasion, the need arises to retrofit an existing type class with a superclass, such as when `Monad` was changed to have `Applicative` as a superclass (which in turn has `Functor` as a superclass).
One disadvantage of such a change is that if the primary goal is to define the `Monad` instance for a type, one now has to write two additional instances, for `Functor` and `Applicative`, even though these instances are actually determined by the `Monad` instance.
With Deriving Via, we can capture this fact as a newtype, thereby making the process of defining such instances much less tedious:

```
type    FromMonad :: (k -> Type) -> (k -> Type)
newtype FromMonad m a = MkFromMonad (m a)
  deriving
  newtype Monad

instance Monad m => Functor (FromMonad m) where
```

```
  fmap = liftM

instance Monad m => Applicative (FromMonad m) where
  pure  = return
  (<*>) = ap
```

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

```
type Stream :: Type -> Type -> Type
data Stream a b = Done b | Yield a (Stream a b)
  deriving (Functor, Applicative)
  via FromMonad (Stream a)

instance Monad (Stream a) where
  return :: a -> Stream a
  return = Done

  (>>=) :: Stream a -> (a -> Stream b) -> Stream b
  Yield a k >>= f = Yield a (k >>= f)
  Done b    >>= f = f b
```

One potentially problematic aspect remains. Another proposal [20] has been put forth (but has not been implemented, as of now) to remove the `return` method from the `Monad` class and make it a synonym for `pure` from `Applicative`. The argument is that `return` is redundant, given that `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 `return` from the `Monad` class would prevent `FromMonad` from working, as then `Monad` instances would not have any way to define `pure`. [3]

5.1.7 Avoiding Orphan Instances

Not only can Deriving Via quickly procure class instances, in some cases, it can eliminate the need for certain instances altogether. Haskell programmers often want to avoid *orphan instances*: instances defined in a separate module from both the type class and data types being used. Sometimes, however, it is quite tempting to reach for orphan instances, as in the following example adapted from a blog post by Gonzalez [21]:

```
newtype Plugin = MkPlugin (IO (String -> IO ()))
  deriving
  newtype Semigroup
```

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

- One could patch the base library to add the instance for `IO`. But given base's slow release cycle, it would be a while before one could actually use this in-

[3] A similar, yet somewhat weaker, argument applies to suggested changes to relax the constraints of `liftM` and `ap` to merely `Applicative` and to change their definitions to be identical to `fmap` and `(<*>)`, respectively.

stance.

- Write an orphan instance for `IO`. This works, but is undesirable, as now anyone who uses `Plugin` must incur a possibly unwanted orphan instance.

Luckily, Deriving Via presents a more convenient third option: re-use a `Semigroup` instance from *another* data type. Recall the `Ap` data type from Section 1.7 that lets us define a `Semigroup` instance by lifting through an `Applicative` instance. As luck would have it, `IO` already has an `Applicative` instance, so we can derive the desired `Semigroup` instance for `Plugin` like so:

```
type    Plugin :: Type
newtype Plugin = MkPlugin (IO (String -> IO ()))
  deriving Semigroup
  via Ap IO (String -> Ap IO ())
```

Note that we have to use `Ap` twice in the `via` type, corresponding to the two occurences of `IO` in the `Plugin` type. This is possible because `Ap IO` has the same representation as `IO`, and it is also necessary if we want to completely bypass the need for a `Semigroup` instance for `IO`: Via the inner `Ap IO ()` and the existing instance

```
instance Semigroup b => Semigroup (a -> b)
```

we first obtain a `Semigroup` instance for `String -> IO ()`, which we then, via the outer `Ap IO` application, lift to `IO (String -> IO ())` and therefore the `Plugin` type.

5.2 Conclusions

This demonstrates the versatility of Deriving Via whose applications run the gamut from asymptotic improvement, type class backwards interoperability to serving a more flexible replacement for *default methods*, parameterizing deriving with configuration data and even deriving via isomorphisms.

Again Dependent Haskell[12] will greatly improve the ability to pass configuration data, and has the ability to explicitly pass an isomorphism through which to derive. *Default methods* can now be derived `via` one or more user defined types. This is a big improvement where default methods specify a single behaviour intimately tied to the class declaration.

6

Current Status

Given that the extension `-XDerivingVia` and `:instances` have been added to GHC there is more work that can be done in this space.

6.1 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 their existence in Section 4.1.1 without further mention. However, multi-parameter type classes pose an intriguing design question when combined with *DerivingVia* and *StandaloneDeriving*, another *GHC* feature that allows one to write `deriving` declarations independently of a data type.
For example, one can write the following instance using *StandaloneDeriving*:

```
instance Triple A B () where
  triple :: (A, B, ())
  triple = undefined
class Triple a b c where
  triple :: (a, b, c)

instance Triple () () () where
  triple :: ((), (), ())
  triple = ((), (), ())

newtype A = MkA ()
newtype B = MkB ()
newtype C = MkC ()

deriving via () instance Triple A B C
```

However, the code this generates is somewhat surprising. Instead of reusing the `Triple () () ()` instance in the derived instance, *GHC* will attempt to reuse an instance for the type `Triple A B ()`. The reason is that, by convention, *StandaloneDeriving* will only ever coerce through the *last* argument of a class. That is because the standalone instance above would be the same as if a user had written:

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

This consistency is perhaps a bit limiting in this context where there are multiple arguments to `C` that one could "derive through". But it is not clear how GHC would figure out which of these arguments to `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.

6.2 Applying Via

The idea of a *via*-type can be extended further.

6.2.1 `ala'`

Take Conor McBride's ala' as an example.[22] This magical combinator transforms a higher-order function that produces `newtypes`
`(a -> newty) -> (as -> newty')`
into a function operating on the underlying representation
`(a -> ty) -> (as -> ty')`
This third-order function takes an additional proxy argument: the value of this argument is discarded and its type information is used to determine the desired behaviour. The `Newtype` methods `pack` and `unpack` are used to wrap and unwrap the representation type. A new instance of `Newtype` must be written for each `newtype`.

```
ala' :: Newtype newty  ty
     => Newtype newty' ty'
     => (ty -> newty)
     -> ((a -> newty) -> (as -> newty'))
     -> ((a ->    ty) -> (as ->    ty'))
ala' _ = dimap (fmap pack) (fmap unpack)
```

The function `ala' Sum foldMap` uses the type information of `Sum` to guide the instantiation of `foldMap @_ @(Sum a)` for additive behaviour. Like Deriving Via this uses `newtypes` to direct behaviour.
A lot of standard definitions can be defined tersely with `ala'` in what McBride calls *adverbial programming*[23]. `ala'` is primarily used with `foldMap` and `traverse`:

```
sum, product :: Foldable f => Num a => f a -> a
sum     = ala' Sum     foldMap id
product = ala' Product foldMap id

and, or :: Foldable f => f Bool -> Bool
and = ala' All foldMap id
or  = ala' Any foldMap id

all, any :: Foldable t => (a -> Bool) -> (t a -> Bool)
all = ala' All foldMap
any = ala' Any foldMap

fmapDefault :: Traversable t => (a -> b) -> (t a -> t b)
fmapDefault = ala' Identity traverse
```

```
foldMapDefault :: Traversable t => Monoid m => (a -> m) -> (t a -> m)
foldMapDefault = ala' Const traverse
```

This even replaces some functions defined in the *async* package[24] where `Concurrently` is a newtype of `IO` with concurrent semantics:

```
mapConcurrently :: Traversable t => (a -> IO b) -> (t a -> IO (t b))
mapConcurrently = ala' Concurrently traverse

mapConcurrently_ :: Foldable t => (a -> IO b) -> (t a -> IO ())
mapConcurrently_ = ala' Concurrently traverse_
```

Despite these select terse definitions the honeymoon ends here: the definition of `ala'` only takes higher-order functions of a particular shape into account.

Definitions like `replicateConcurrently` and `replicateConcurrently_` can not be defined in terms of `ala'`:

```
replicateM  :: Applicative f => Int -> f a -> f [a]
replicateM_ :: Applicative f => Int -> f a -> f ()
```

6.2.2 `ala'` versus *via*

Enter Applying Via[1], a work-in-progress GHC extension that instantiates Haskell functions to *via* types and coerces them to the desired type. It generalises `ala'` to work with any shape, it doesn't require user-written `Newtype` instances and allows composing behaviour.

Instantiating `replicateM` to `IO` will sequentially execute its action n times and collect the results; the sequentiality comes from the `Applicative IO` instance.

To spawn a n threads, concurrent behaviour is used: `Applicative Concurrently`.

```
replicateM :: Int -> Concurrently a -> Concurrently [a]
```

The core idea is the same as Deriving Via: running it `Concurrently` it is representationally equal to `replicateConcurrently`. The original function is retrieved by coercing `Concurrently` to IO-actions.

```
replicateConcurrently :: forall a. Int -> IO a -> IO [a]
replicateConcurrently = coerce (replicateM @Concurrently @a)

replicateConcurrently_ :: forall a. Int -> IO a -> IO ()
replicateConcurrently_ = coerce (replicateM_ @Concurrently @a)
```

The idea is to take the promise of `ala'` and apply it to any term. The proposed syntax for Applying Via is twofold. One is explicit `@(<ty> via <viaty>)` where two types are specified, the type we want to end up with (`IO`) and the *via* type (`Concurrently`) that gets the desired behaviour:

```
replicateConcurrently  = replicateM  @(IO via Concurrently)
replicateConcurrently_ = replicateM_ @(IO via Concurrently)
```

The implicit syntax takes a page out of `ala'`'s book by taking a proxy function. The type of this proxy will then be used to determine the *via* type and end type. For example the constructor function `Concurrently :: Concurrently a -> IO`

[1] Applying Via GHC Proposal: `https://github.com/ghc-proposals/ghc-proposals/pull/218`

`a` contains enough information to redefine all the concurrent variants that exist in *async*:

```
replicateConcurrently  = replicateM  @(via Concurrently)
replicateConcurrently_ = replicateM_ @(via Concurrently)
```

6.3 Instances Via

The same idea can be extended to instance declarations. Instances over `newtype`s still require a lot of manual wrapping which can be avoided using the same bag of tricks.

```
newtype Semigroup All where
  (<>) :: All -> All -> All
  All a <> All b = All (a && b)

instance Semigroup Any where
  (<>) :: Any -> Any -> Any
  Any a <> Any b = Any (a || b)
```

The meaning of these instances is more clearly expressed without `newtype` clutter. The syntax of Instances Via and is left as future work.

```
newtype Semigroup (Bool via All) where
  (<>) :: Bool -> Bool -> Bool
  (<>) = (&&)

instance Semigroup (Bool via Any) where
  (<>) :: Bool -> Bool -> Bool
  (<>) = (||)
```

6.4 Reactions

It has been some time since Deriving Via was released as part of GHC. A patch has been submitted to add Deriving Via to the *PureScript* programming language[2] and is being used in production to reduce boilerplate[3][4][5]. New libraries are being build

[2] https://github.com/purescript/purescript/pull/3824
[3] https://archive.is/cmxb0
[4] https://archive.is/cmxb0
[5] https://github.com/takoeight0821/malgo/commit/06a2cb6ffed3e2d9795b0f9bf790d4fee05cdd61

with Deriving Via in mind[6][7][8][9][10][11][12][13][14][15][16].
It has been presented[17] and written about[18][19][20][21].
In an informal poll, *61.5%* of respondants said it was okay to rely on Deriving Via when making a new Haskell library.[22]
The response has been very positive and is encouraging for future work.

> Deriving Via is imo one of the most significant changes to ever hit GHC. The power to weight ratio is incredible. It will literally change how idiomatic Haskell is written forever. It is a generative change.
>
> — *@RainH* (`https://archive.is/y17PO`)
>
> Wow, Deriving Via is another HUGE patch that not only offers tons of power, but is much simpler than many predecessors. Amazing times.
>
> — *@KirinDave* (`https://archive.is/4Vqjk`)

[6] `https://archive.is/1T9BM`
[7] `https://github.com/fumieval/deriving-aeson`
[8] `https://github.com/trailofbits/indurative`
[9] `https://hackage.haskell.org/package/woe-0.1.0.3/docs/Data-WOE.html`
[10] `https://hackage.haskell.org/package/enum-text-0.5.1.0/docs/Text-Enum-Text.html`
[11] `https://hackage.haskell.org/package/capability`
[12] `https://hackage.haskell.org/package/semigroups-0.19.1/docs/Data-Semigroup-Generic.html`
[13] `https://hackage.haskell.org/package/one-liner-instances-0.1.2.1/docs/Data-Ord-OneLiner.html`
[14] `https://hackage.haskell.org/package/primitive-0.7.0.0/docs/Data-Primitive-Types.html`
[15] `https://hackage.haskell.org/package/generic-deriving-1.12.3/docs/Generics-Deriving-Default.html`
[16] `https://hackage.haskell.org/package/text-show-3.8.3/docs/TextShow-Generic.html`
[17] `https://www.reddit.com/r/haskell_jp/comments/98btal/%E6%9C%AC%E5%BD%93%E3%81%AF%E3%81%99%E3%81%94%E3%81%84_newtype/`
[18] `https://www.tweag.io/posts/2018-10-04-capability.html`
[19] `https://archive.is/kc5tb`
[20] `https://archive.is/KS4Te`
[21] `https://archive.is/Ce1ix`
[22] `https://archive.is/ZGlML`

7
Conclusions

7.1 Related Ideas

The idea behind *DerivingVia* is extremely versatile technique and can be used to tackle a wide variety of problems. *DerivingVia* also bears a resemblance to other distinct language features which address similar issues, so in this section, we present an overview of their similarities and differences.

7.1.1 ML functors

(I'm not sure I will discuss them, but this is what we took out of the thesis)
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 *DerivingVia*, as functors allow "lifting" of code into the module language much like *DerivingVia* allows lifting of code into *GHC*'s deriving construct.

7.1.2 Code Reuse in Dependent Type Theory

Diehl *et al.* present a dependent type theory which permits zero-cost conversions between indexed and non-indexed variants of data types [25], much in the same vein as `Coercible`. However, these conversions must be explicitly constructed with combinators, whereas `Coercible`-based casts are built automatically by GHC's constraint solver. Therefore, while Diehl *et al.* allow conversions between more data types than *DerivingVia* does, it also introduces some amount of boilerplate that *DerivingVia* avoids.

7.1.3 Explicit Dictionary Passing

The power and flexibility of *DerivingVia* 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.
A similar approach is to permit the ability to explicitly construct and pass the normally implicit dictionary arguments corresponding to type class instances [26]. Unlike in *DerivingVia*, 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.
This is not just a superficial difference as the *dictinary-directed* approach means passing an explicit dictionary for *each* constraint. This stands in stark contrast to the *type-directed* approach of this paper where a type is a *family of behaviours*. A single type constructor [] will discharge an extensible menagerie of unnamed instances `Functor`, `Applicative`, `Monad`, .. while the *dictionary-directed* approach would have to pass them individually.
Explicit dictionary arguments are a considerable extension of the language and its type system, too large a hammer for the nail Deriving Via aims 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 that explicit dictionaries aim to solve—resolving ambiguity in implicit arguments—almost never arises in *DerivingVia*, as the programmer must specify all the types involved in the process.

7.2 Current Status

We have implemented *DerivingVia* within *GHC*. Our implementation also interacts well with other *GHC* features that were not covered in this thesis, such as kind polymorphism [15], *StandaloneDeriving*, and type classes with associated type families [13]. However, there are still challenges remaining, which we describe in this section.

7.2.1 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 did not write in error messages can lead to a confusing debugging experience.
Fortunately, we have found in our experience that the quality of *DerivingVia*-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 *DerivingVia* 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:

```
newtype UhOh = MkUhOh 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’
• When deriving the instance for (Ord UhOh)
```

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:

```
newtype Foo a = MkFoo (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’
• When deriving the instance for (Ord (Foo a))
```

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 *DerivingVia* error quality are ultimately issues with `coerce` error quality, given that the error messages are a result of `coerce` failing to typecheck. It is likely that investing more effort into making `coerce`'s error messages easier to understand would benefit *DerivingVia* as well.

7.3 Conclusions

In this thesis I laid out the *DerivingVia* language extension; explained how it is implemented, and shown a wide variety of use cases. I believe that *DerivingVia* has the potential to dramatically change the way we write instances, as it encourages giving names to recurring patterns and reusing them where needed. Many 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.

The Deriving Via feature arose from my frustration with instances that carried no new information. The development took many twists and turns: this was first separate from my master's thesis and it was not until after I published it at ICFP that my advisor suggested it as thesis material.

Bibliography

Bibliography

[1] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76. ACM, 1989.

[2] Stephanie Weirich, Pritam Choudhury, Antoine Voizard, and Richard A Eisenberg. A role for dependent types in haskell. *Proceedings of the ACM on Programming Languages*, 3(ICFP):101, 2019.

[3] Edward Kmett. A comment on 'the constraint trick for instances'. `https://www.reddit.com/r/haskell/comments/3afi3t/the_constraint_trick_for_instances/cscblgz/`, 2015. [Online; accessed 19-December-2019].

[4] Conal Elliott. applicative-numbers: Applicative-based numeric instances. `https://hackage.haskell.org/package/applicative-numbers`, 2009.

[5] Joachim Breitner, Richard A. Eisenberg, Simon Peyton Jones, and Stephanie Weirich. Safe zero-cost coercions for haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 189–202, New York, NY, USA, 2014. ACM.

[6] José Pedro Magalhães, Atze Dijkstra, Johan Jeuring, and Andres Löh. A generic deriving mechanism for haskell. In *Proceedings of the Third ACM Haskell Symposium on Haskell*, Haskell '10, pages 37–48, New York, NY, USA, 2010. ACM.

[7] Jeremy Gibbons and Bruno c. d. s. Oliveira. The essence of the iterator pattern. *J. Funct. Program.*, 19(3-4):377–402, July 2009.

[8] Baldur Blöndal, Andres Löh, and Ryan Scott. Deriving via. In *Proceedings of the 11th ACM Haskell Symposium (Haskell'18)*, 2018.

[9] Baldur Blöndal. Ghci command to list instances a (possibly compound) type belongs to. `https://gitlab.haskell.org/ghc/ghc/issues/15610`, 2018. [Online; accessed 19-December-2019].

[10] Xavier Denis. List instances for a type in ghci. `https://github.com/xldenis/ghc-proposals/blob/master/proposals/0000-ghci-instances.rst`, 2018. [Online; accessed 19-December-2019].

[11] Simon Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[12] Richard A Eisenberg. Dependent types in haskell: Theory and practice. *arXiv preprint arXiv:1610.07978*, 2016.

[13] Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. In *Proceedings of the Tenth ACM SIGPLAN*

International Conference on Functional Programming, ICFP '05, pages 241–253, New York, NY, USA, 2005. ACM.

[14] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM.

[15] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12, pages 53–66, New York, NY, USA, 2012. ACM.

[16] Exequiel Rivas and Mauro Jaskelioff. Notions of computation as monoids. `http://arxiv.org/abs/1406.4823`, 2014.

[17] Conor Mcbride and Ross Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, January 2008.

[18] Ross Paterson. Constructing applicative functors. In *Proceedings of the 11th International Conference on Mathematics of Program Construction*, MPC'12, pages 300–323, Berlin, Heidelberg, 2012. Springer-Verlag.

[19] Edsko de Vries and Andres Löh. True sums of products. In *Proceedings of the 10th ACM SIGPLAN Workshop on Generic Programming*, WGP '14, pages 83–94, New York, NY, USA, 2014. ACM.

[20] Herbert V. Riedel and David Luposchainsky. Monad of no `return` proposal (mrp): Moving `return` out of `Monad`. `https://mail.haskell.org/pipermail/libraries/2015-September/026121.html`, Sep 2015.

[21] Gabriel Gonzalez. Equational reasoning at scale, Jul 2014.

[22] Edwin Brady, James Chapman, Pierre-Évariste Dagand, Adam Gundry, Conor McBride, Peter Morris, Ulf Norell, and Nicolas Oury. An epigram implementation. *URL http://www. e-pig. org*, 2011.

[23] Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of functional programming*, 18(1):1–13, 2008.

[24] Simon Marlow. `async`: Run io operations asynchronously and wait for their results. `https://hackage.haskell.org/package/async`, 2012. [Online; accessed 2-March-2020].

[25] Larry Diehl, Denis Firsov, and Aaron Stump. Generic zero-cost reuse for dependent types. *Proc. ACM Program. Lang.*, 2(ICFP):104:1–104:30, July 2018.

[26] Atze Dijkstra and S. Doaitse Swierstra. Making implicit parameters explicit. Technical Report UU-CS-2005-032, Department of Information and Computing Sciences, Utrecht University, 2005.