Improving Typeclass Relations by Being Open

(extended version)

Guido Martínez
CIFASIS-CONICET
Rosario, Argentina
martinez@cifasis-conicet.gov.ar

Mauro Jaskelioff
CIFASIS-CONICET
Rosario, Argentina
jaskelioff@cifasis-conicet.gov.ar

Guido De Luca
Universidad Nacional de Rosario
Rosario, Argentina
gdeluca@dcc.fceia.unr.edu.ar

Abstract

Mathematical concepts such as monads, functors, monoids, and semigroups are expressed in Haskell as typeclasses. Therefore, in order to exploit relations such as "every monad is a functor", and "every monoid is a semigroup", we need to be able to also express relations between typeclasses.

Currently, the only way to do so is using *superclasses*. However, superclasses can be problematic due to their closed nature. Adding a superclass implies modifying the subclass' definition, which is either impossible if one does not own such code, or painful as it requires cascading changes and the introduction of boilerplate throughout the codebase.

In this article, we introduce *class morphisms*, a way to relate classes in an open fashion, without changing class definitions. We show how class morphisms improve the expressivity, conciseness, and maintainability of code. Further, we show how to implement them while maintaining canonicity and coherence, two key properties of the Haskell type system. Extending a typechecker with class morphisms amounts to adding an elaboration phase and is an unintrusive change. We back this claim with a prototype extension of GHC.

1 Introduction

Typeclasses infuse Haskell with a mathematical flavour. Concepts such as monads, functors, and total and partial orders can all be modelled in programs along with specific instances of them. Usually, some classes within a given program are related. For instance, every monad is a functor, and every total order is also a partial order. Expressing these relations within the type system allows programmers to reuse code written for general concepts (functors, partial orders) over the more specific ones (monads, total orders).

The way of expressing these relations in Haskell is via *superclasses*. Suppose we have classes C and S and we want to express that every instance of C is also an instance of S. We can, at the point of definition of a class C, declare S as a superclass. This means that every C-instance declaration must have a corresponding S-instance. In return, functions that expect a type in S can be used over any type in C, allowing for more code reuse.

However, using superclasses to represent these implication relations between classes has several drawbacks (see §2) due to their closed nature:

- Adding a superclass to an existing class implies modifying the latter's definition. However, it might not be practical or even possible to change it (for instance, if it belongs to a third-party library or the language's standard library).
- When the programmer does modify the class definition, existing code may (and often does) break because of missing instances of the new superclass.
- These problems are recurrent, as programmers wish to model more relations. However, anticipating all needed superclasses is hardly ever possible.
- When using superclasses for this purpose, the instances of the superclass are *generically definable from instances of the subclass*. (For example, a functor instance is easily defined from just the bind and return methods of a monad, independently of the particular monad involved.) Writing these generic instances by hand is then clearly boilerplate.

These limitations are not hypothetical: the need to relate classes is bound to appear in any long-living project. Recently, GHC [The Glasgow Haskell Team 2018] has been such an example. Seeking to model the mathematical relation between monads, applicative functors, and functors, the *Functor-Applicative-Monad proposal* [Haskell Wiki 2014] was put forward. Following this proposal, the class system was modified to make Functor a superclass of Applicative, and Applicative a superclass of Monad. However, all the problems mentioned above surfaced:

- The proposal required to change the standard prelude, thus deviating from the language definition.
- A significant amount of code stopped compiling because of missing instances.
- Other classes, such as pointed functors, were left out of the hierarchy.
- Most programmers writing a Monad instance use boilerplate definitions for Applicative and Functor.

In this paper, motivated by such issues, we describe *class morphisms*, a way of introducing class relations in an open fashion, independently of class definitions (§3). We show how the problems stated above are avoided by the use of morphisms, and how they allow for a smoother evolution of software. By internalising these relations, class morphisms bring the typeclass system closer to our mathematical view of it: relations between typeclasses can be incrementally added without changing what each typeclass *is*.

We have formalised class morphisms by extending Jones's [1995] theory of qualified types and proved that any program with class morphisms can be elaborated to a well-typed program without them (§4). This shows that class morphisms preserve *canonicity*: the key property that there is at most one instance for each type and class. Further, class morphisms also preserve *coherence*, which is essentially the property that a well-typed program's behaviour is determined by its text, and not by how it was typechecked.

We have also developed a prototype implementation of class morphisms in GHC, which we describe in §5. It can be downloaded from http://github.com/cifasis/ghc-cm.

The brittleness of typeclass hierarchies is a well-known problem, and there have been several proposed solutions. We compare them with ours in §6.

2 The Problem with Superclasses

Superclasses, present ever since the origins of typeclasses, organise classes into a hierarchy that allows for reuse of instances and member function names [Wadler and Blott 1989]. For example, a class for groups may have Monoid as a superclass and only declare an operation for inverses. In this way, the names for the associative operation mappend and the unit mempty from the Monoid class are reused. Additionally, when declaring a group instance there is no need to declare the unit and associative operations for types with an existing monoid instance.

Another use of superclasses is to increase polymorphism, as they allow the typechecker to conclude some implications between constraints. For example, given the following standard classes:

```
class Eq a where
  (≡) :: a → a → Bool
data Ordering = LT | EQ | GT
class Eq ⇒ Ord a where
  compare :: a → a → Ordering
```

we can conclude Eq τ from Ord τ, regardless of the shape of τ or the instances in scope. Operationally, this amounts to including a dictionary for Eq τ inside the dictionaries for Ord τ, which can then be projected and used. To ensure such dictionaries exist, an Ord τ instance can only be accepted if there is also an Eq τ instance, or a more general one.

While adding new instances and classes works extremely well, adding a superclass to an *existing* class is often a breaking change. Let us explore the reasons behind this.

Let C be an existing class, and suppose we want to add S as a superclass of C. To start with, we need to modify C's definition, which might be impractical if C is part of the language standard or belongs to a third-party library. However, let us imagine that we indeed do so. Now, C instances are only valid if they have a corresponding S instance; a new requirement. Thus, existing C instances will be rejected unless they happen to have a matching S instance. To fix this, the programmer must add S instances across the codebase—a task of considerable effort. If C is part of a library, then fixing code written by users is simply impossible for the library developer, and hence backwards-compatibility is lost.

This problem is sometimes unavoidable, since one cannot in general expect to get instances of the superclass for free. For instance, take Num, the class of types with basic numeric operations. If we decide we want Eq as a superclass of Num, we really must provide an equality for every Num instance. Here, where our decision was somewhat arbitrary, the Eq superclass behaves as a *prerequisite* of Num. However, for Eq and Ord, there should not be any extra requirements for defining an ordering, since (≡) can always be implemented via compare. The class Eq is a *consequence* of Ord, not a prerequisite! By the superclass, we are simply trying to make the typeclass system "aware" of this, but doing so results in generalised breakage. In practice, programmers will often find all offending Ord τ instances and add:

```
instance Eq τ where
  x ≡ y = case compare x y of
            EQ → True
            _  → False
```

which, exploiting mutual recursion, fixes the problem and does not require any insight in defining (≡). In a case like this, where the superclass is definable from the methods of the subclass, we call the superclass *degenerate*.

Summarising, for degenerate superclasses, programmers are not only required to revisit their existing code, but also forced to write instances that have no *logical* raison d'être and are simply boilerplate. Even worse, this process might be repeated as programmers wish to model more relations, since one can hardly predict all future needs in advance. For example, if the programmer later wishes to model partial orders, a similar chaos ensues.

This situation is very unsatisfactory, and it goes against our intuitive understanding of relations. In mathematics, we learn new relations without the need to revisit definitions. After all, learning that all total orders are also partial orders does not change the definition of what a total order is, nor requires us to revisit any previously-known total orders.

3 Class Morphisms

The main contribution of this paper is the notion of *class morphism*, which express "is a" relations within the typeclass system. As such, they bear resemblance to superclasses, with two key differences. Firstly, class morphisms are *open*: they can be added without modifying existing classes. Secondly, class morphisms include a *generic definition* of a class in terms of another, beyond merely stating their connection.

The basic idea is that, when there is such a relation, an instance declaration for a class one *induces* an instance declaration for the other. Class morphisms allow the programmer to state this connection within the type system itself, along with *how* the new instance is constructed.

Syntactically, class morphisms are rather simple:

class morphism C → D **where**
  $m_1 = e_1$
  $m_2 = e_2$
  …

Here, C and D are class names, called the *antecedent* and *consequent* respectively; and $m_1, m_2, \ldots$ are the methods of D. The expressions $e_1, e_2, \ldots$ must provide *generic* definitions for the methods of D. By "generic", we mean these definitions must be polymorphic: they must define a D a instance where a (a fresh type variable) can be assumed to be an instance of C, but nothing else. The interpretation for such a morphism, on a logical level, is evident: all C-types are D-types and here is the proof. Operationally, any C τ instance *induces* a D τ instance via this morphism, where the instance context is taken from the C τ instance and the methods from the class morphism. That is, given the previous morphism and the instance declaration

instance $\mathsf{C}_1\ \tau_1, \ldots, \mathsf{C}_n\ \tau_n \Rightarrow \mathsf{C}\ \tau$ **where** …

One can *apply* the morphism to the instance and obtain:

instance $\mathsf{C}_1\ \tau_1, \ldots, \mathsf{C}_n\ \tau_n \Rightarrow \mathsf{D}\ \tau$ **where**
  $m_1 = e_1$
  $m_2 = e_2$
  …

which, with the C instance in-scope, is well-typed.

The semantics of class morphisms is given by an *elaboration* into morphism-free code. Essentially, this elaboration consists of (1) expanding qualified contexts (2) generating new instances by applying morphisms and (3) trimming some derived instances. Step 1 expands contexts with additional constraints in order to defer the choice of some dictionaries, as they cannot be (canonically) solved on the spot. Step 2 essentially saturates a module by generating all derivable instances. These generated instances are not generic, but concrete instances based on the concrete instances in scope. The saturated set may be "too large", and contain overlap which must be removed; step 3 takes care of eliminating it, asking the programmer for help when needed.

This elaboration is performed for source Haskell *modules*, and not whole programs, which is crucial for separate compilation. Once a module is elaborated, typechecking proceeds as usual, and morphisms have no further impact in the compilation of the module (besides being exported). Importantly, Haskell's semantics and constraint resolution process are completely unaffected, and derived instances have the same status as source ones (in particular, they are exported).

3.1 An Example Class Morphism

A well-known Haskell typeclass is Enum, for types which can be put in correspondence with (a subset of) the integers. Its definition is essentially the following:

class Enum a **where**
  toEnum :: Int → a
  fromEnum :: a → Int

It is clear that any Enum-type can be tested for equality: one can simply map to the integers and do the comparison there. However Enum is wholly unrelated to Eq, and hence the following definitions for f_1 and f_2 rightly fail:

module A **where**
  data ABC = A | B | C
  instance Enum ABC **where** …

  f_1 :: ABC → Bool
  $\mathsf{f}_1\ x = x \equiv x$ -- No instance for (Eq ABC)

  f_2 :: Enum $a \Rightarrow a \rightarrow a \rightarrow a \rightarrow$ Bool
  $\mathsf{f}_2\ x\ y\ z = x \equiv y \vee y \equiv z$ -- No instance for (Eq a)

  t :: Bool
  t = f_2 A B C

While fixing f_1 is possible by declaring a (boilerplate) Eq ABC instance, fixing f_2 requires either adding a superclass to Enum, possibly wreaking havoc in many modules; or manually adding Eq a to f_2's context, which needs to be propagated through the call graph and quickly becomes cumbersome. One can instead add a class morphism:

class morphism Enum → Eq **where**
  $x \equiv y$ = fromEnum $x \equiv$ fromEnum y

With this definition, the missing Eq ABC instance is generated and spliced into the program, causing f_1 to succeed. Further, the typechecker will expand the context of f_2 to (Enum a, Eq a), turning it valid. The module is then elaborated to:

data ABC = A | B | C
instance Enum ABC **where** …

f_1 :: ABC → Bool
$\mathsf{f}_1\ x = x \equiv x$

f_2 :: (Enum a, Eq a) $\Rightarrow a \rightarrow a \rightarrow a \rightarrow$ Bool
$\mathsf{f}_2\ x\ y\ z = x \equiv y \vee y \equiv z$

t :: Bool
t = f_2 A B C

class morphism Enum → Eq **where**
  $x \equiv y$ = fromEnum $x \equiv$ fromEnum y

instance Eq ABC **where**
  $x \equiv y$ = fromEnum $x \equiv$ fromEnum y

which, ignoring the morphism itself, is a valid vanilla Haskell program. Note that the body of t must now discharge an extra constraint, namely Eq ABC, which is given by the new instance. This expansion is safe since the new constraints are always dischargeable (see proof in §4).

Going one step further, one can obtain Ord instances from Enum, via the following class morphism:

$$\textbf{class morphism}\ \mathsf{Enum} \rightarrow \mathsf{Ord}\ \textbf{where}$$
$$x\ \text{`compare`}\ y = \mathsf{fromEnum}\ x\ \text{`compare`}\ \mathsf{fromEnum}\ y$$

In cases like this, where the consequent (Ord) has superclasses (Eq), the morphism can only be allowed if there is a generic way to build dictionaries for each of the superclasses. It is therefore required that there be a *morphism path* from the antecedent, or one of its transitive superclasses, into each of the consequent's superclasses. This ensures that instances generated by the morphism are valid w.r.t. their superclasses. Given the previous Enum → Eq morphism, this one is accepted.

3.2 Finding Middle Ground

Suppose that in a development one becomes interested in using partial orders. A class for them can be defined as:

$$\textbf{class}\ \mathsf{POrd}\ a\ \textbf{where}$$
$$\mathsf{pcompare} :: a \rightarrow a \rightarrow \mathsf{Maybe\ Ordering}$$

where instances are expected to satisfy the proper laws, i.e. to be reflexive, transitive, and anti-symmetric. Of course, every partial order determines a notion of equality and every total order is a partial order, but this is not at all reflected in the type system; i.e. POrd bears no relation to Eq and Ord.

In Haskell, Eq is already a superclass of Ord, as can be seen see in the definition in §2, and POrd should be in between the two. Making Eq a superclass of POrd is readily done—no POrd instances yet exist so no breakage ensues from it. On the other hand, expressing that "every total order is a partial order" entails making POrd a superclass of Ord, which results in breaking *every* existing Ord instance since, again, no POrd instances yet exist.

Instead of superclasses, one can state the relation between these three classes as class morphisms. One may also avoid making Eq a (degenerate) superclass of POrd, since a POrd → Eq morphism provides the same logical behaviour.

$$\begin{array}{l}\textbf{class morphism}\ \mathsf{POrd} \rightarrow \mathsf{Eq}\ \textbf{where}\\ \quad x \equiv y = \textbf{case}\ \mathsf{pcompare}\ x\ y\ \textbf{of}\\ \qquad\qquad \mathsf{Just\ EQ} \rightarrow \mathsf{True}\\ \qquad\qquad \_ \qquad\quad\ \rightarrow \mathsf{False}\end{array}$$

$$\begin{array}{l}\textbf{class morphism}\ \mathsf{Ord} \rightarrow \mathsf{POrd}\ \textbf{where}\\ \quad \mathsf{pcompare}\ x\ y = \mathsf{Just}\ (\mathsf{compare}\ x\ y)\end{array}$$

The second morphism has effectively the same effect as adding POrd as superclass of Ord, even though POrd is defined in a different module. The definition of Ord does not need to be modified. Given these morphisms, the type-checker will enforce that every Ord-type is a POrd-type, and that every POrd-type is an Eq-type. Assuming the following instances for Int,

$$\begin{array}{l}\textbf{instance}\ \mathsf{Eq\ Int}\ \textbf{where}\\ \quad (\equiv) = \mathsf{primEqInt}\end{array}$$

$$\begin{array}{l}\textbf{instance}\ \mathsf{Ord\ Int}\ \textbf{where}\\ \quad \mathsf{compare}\ x\ y = \textbf{if}\ x = y\ \textbf{then}\ \mathsf{EQ}\ \textbf{else}\\ \qquad\qquad\qquad\qquad \textbf{if}\ \mathsf{primLtInt}\ x\ y\ \textbf{then}\ \mathsf{LT}\ \textbf{else}\ \mathsf{GT}\end{array}$$

the morphisms above generate the following new instance:

$$\begin{array}{l}\textbf{instance}\ \mathsf{POrd\ Int}\ \textbf{where}\\ \quad \mathsf{pcompare}\ x\ y = \mathsf{Just}\ (\mathsf{compare}\ x\ y)\end{array}$$

An Eq Int instance will also be generated by the morphism, but "trimmed" away as there is a user-written one already (and having both would cause an overlap error). At this point, pcompare can be used over Ints, without any boilerplate.

In the case of a polymorphic function such as:

$$\begin{array}{l}\mathsf{related} :: \mathsf{Ord}\ a \Rightarrow a \rightarrow a \rightarrow \mathsf{Bool}\\ \mathsf{related}\ x\ y = \mathsf{isJust}\ (\mathsf{pcompare}\ x\ y)\end{array}$$

the context is expanded using the morphisms to obtain

$$\begin{array}{l}\mathsf{related} :: (\mathsf{Ord}\ a, \mathsf{POrd}\ a) \Rightarrow a \rightarrow a \rightarrow \mathsf{Bool}\\ \mathsf{related}\ x\ y = \mathsf{isJust}\ (\mathsf{pcompare}\ x\ y)\end{array}$$

making the function valid. (There is no need to add Eq a, since it is a superclass of Ord a.) Later, when compiling, the Ord a constraint can be discarded as it is in fact unneeded, decreasing the amount of dictionaries at runtime. Intuitively, deferring the choice works since, eventually, a caller will provide a dictionary for Ord t from an instance declaration, and there will be a corresponding POrd t instance.

The reader might wonder, since there is both Ord → POrd and POrd → Eq, can one obtain an instance for Eq τ by simply defining an Ord τ instance? Yes; as shown next, morphisms may be declared from a class to one of its (transitive) superclasses, as long as the morphisms meet a few restrictions.

3.3 Dealing with Degenerate Superclasses

In recent versions of GHC, functors, applicatives, and monads have the following superclass relation:

$$\begin{array}{l}\textbf{class}\ \mathsf{Functor}\ f\ \textbf{where}\\ \quad \mathsf{fmap}\ :: (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b\end{array}$$

$$\begin{array}{l}\textbf{class}\ \mathsf{Functor}\ f \Rightarrow \mathsf{Applicative}\ f\ \textbf{where}\\ \quad \mathsf{pure}\ :: a \rightarrow f\ a\\ \quad (\circledast)\ :: f\ (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b\end{array}$$

$$\begin{array}{l}\textbf{class}\ \mathsf{Applicative}\ m \Rightarrow \mathsf{Monad}\ m\ \textbf{where}\\ \quad \mathsf{return} :: a \rightarrow m\ a\\ \quad (\ggg)\ :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b\end{array}$$

Hence, when defining a monad instance, say for a type constructor M, not only must a programmer define return and ($\ggg$), but she must also define an Applicative instance and a Functor instance. Here, in fact, the extensional behaviour of such instances is determined by the monadic operations. The only real choice for these definitions is how to implement them, i.e. their intensional behaviour. In most cases, programmers are happy enough with default definitions:

```
instance Functor M where
  fmap f x = pure f ⊛ x
instance Applicative M where
  pure        = return
  mf ⊛ mx = mf >>= λf → mx >>= λx → return (f x)
```

We can avoid this boilerplate by declaring morphisms which provide these default definitions once and for all:

```
class morphism Applicative → Functor where
  fmap f x = pure f ⊛ x
class morphism Monad → Applicative where
  pure        = return
  mf ⊛ mx = mf >>= λf → mx >>= λx → return (f x)
```

Now, a monad instance can be given just by implementing return and (>>=), the rest is generated automatically.

Note the difference with the previous example: this morphism is from a class into one of its superclasses. In such cases, we call the morphism *upwards*. Upwards morphisms do not increase polymorphism; their only purpose is to avoid boilerplate instances for degenerate superclasses.

For upwards morphisms, the superclass check is more restrictive. We can accept the Monad → Applicative morphism only because there is a way to generically build its Functor superclass from the Monad antecedent, namely composing both morphisms. On its own, it must be rejected, since the compiler cannot generate valid instances. Therefore, it is required that for an upwards morphism C → D, there are morphism paths from C into the superclasses of D, without considering C's superclasses.

3.4 Overriding Instances

Let us put the morphisms defined in the previous subsection to use. Consider the writer monad, which consists of the pairing of a monoid type (with unit mempty and multiplication mappend) and a value.

```
data Writer m a = Wr m a
instance Monoid m ⇒ Monad (Writer m) where
  return x        = Wr mempty x
  (Wr m x) >>= f = let Wr m′ x′ = f x in
                   Wr (mappend m m′) x′
```

Thanks to the two morphisms of the previous subsection, there is no need for boilerplate instances, and this definition is accepted. However, the derived instances are not ideal.

```
instance Monoid m ⇒ Applicative (Writer m)
instance Monoid m ⇒ Functor (Writer m)
```

The Monoid m assumption is in fact not needed for making Writer m a Functor, but the typechecker has no way of realising that. To avoid this loss of generality, one may *override* the generated instance with a more general one:

```
instance Functor (Writer m) where
  fmap f (Wr m a) = Wr m (f a)
```

Then, the derived Functor instance will be discarded during trimming, as a strictly more general one exists. In general, the programmer may override any derived instance by an equivalent or more general one. She might do so to choose a different behaviour, to use a more efficient implementation, or, as above, to relax constraints. In any case, any programmer-written instance is the canonical, unique one.

Note, however, that derived instances are exported and cannot be overridden from other modules. This is usually not a problem: by following the accepted good practice of declaring instances in the module where the datatype or class is defined (i.e. avoiding orphans), the user-declared instance always prevails.

3.5 Using Different Presentations

Imagine a functional programmer gets stranded on a desert island on their way to ICFP'04. There, after much thought, she comes up with a brilliant idea for representing certain computational effects and defines the following class:

```
class Functor f ⇒ Monoidal f where
  point  :: a → f a
  merge :: f a → f b → ((a, b) → c) → f c
```

Upon returning to civilisation, she finds out an equivalent class, Applicative, has been formulated and many libraries developed for it. Wanting to be able to use these libraries from her own code, she adds two morphisms:

```
class morphism Monoidal → Applicative where
  pure      = point
  ff ⊛ xx = merge ff xx (λ(f, x) → f x)
class morphism Applicative → Monoidal where
  point = pure
  merge f g h = pure (curry h) ⊛ f ⊛ x
    where curry f x y = f (x, y)
```

Now she may use Applicative instances with functions expecting Monoidal, and use her Monoidal instances with functions expecting Applicatives. She can even use such morphisms to seamlessly bridge between separate libraries, e.g. one dealing with Applicatives and one with Monoidal, without modifying either one.

In general, class morphisms can transparently convert two different class presentations of the same concept.

3.6 Solving Conflicts

With class morphisms, situations may arise where it is not clear which instance should be generated. For example, if one adds the following morphism to the ones in §3.3,

```
class morphism Monad → Functor where
  fmap f x = x >>= (return ∘ f)
```

then two different Functor M instances are possible.

An option here is to simply fail, and ask the programmer to disambiguate the situation by providing her own instance. However, this quickly becomes tiring: every Monad instance would need to disambiguate its Functor instance, and hence morphisms would fail to avoid boilerplate.

Instead, instances may be disambiguated via some automated (and possibly arbitrary) *policy*. For instance, one might favour short morphism paths over long ones, thus preferring the new morphism to the composition of the other two. By virtue of elaboration, *any* such policy will preserve canonicity, as instances are chosen only once (even if differently at different types). To preserve coherence, however, the policy must be definable at the source code level—a policy such as "take any instance" would be canonical, but incoherent, as the meaning of "any" is not well-defined.

There are many coherent policies, with slightly different guarantees about what happens when instances or morphisms are added. From here on, our policy will be simply to choose the shortest path to generate instances, and fail if there is more than one path of minimum length. Then, for the example at the beginning of this subsection, one can define a Monad, Applicative, or Functor instance for any type constructor and obtain all of its consequences without conflict. However, declaring both Monad T and Applicative T will cause an ambiguity error for the Functor T consequence, requiring disambiguation.

Other interesting options are possible, by adding some more structure. For instance, each morphism could be decorated with a "weight" used to infer a cost for derived instances. Then, a programmer can avoid more of these errors by careful choice of the costs: if the Monad → Functor and Applicative → Functor morphisms had different weights, the previous example would be unambiguous.

In any case, the policy is best-effort. When it fails, programmers must manually disambiguate their programs. We only consider disambiguating via declaring instances here, but other methods are certainly possible. For instance, and similarly to "Deriving Via" [Blöndal et al. 2018] the programmer could manually specify that a given instance is to be found via some morphism, instead of spelling out its methods explicitly.

3.7 Morphisms and Modules

Class morphisms interact smoothly with modular programming and separate compilation. Consider the following example program:

```
module A where
  class C a
  f :: C a ⇒ ...

module B where
  import A
  class D a
  class morphism C → D
  g :: C a ⇒ ...
```

When expanding contexts, the morphism is in-scope for g but not for f, which means g's context will be expanded to (C a, D a), and f's will be unchanged. Therefore, callers of f only need to solve a C a constraint, as expected. Whether the morphism is in-scope at f's call-sites is irrelevant. For g, its call-sites need to solve both C a and D a, instead of the C a it advertises. However, if g is in-scope, the morphism must be in-scope as well, and thus the calling function will either have its own context expanded, or the D a constraint will be solved via the generated instances. The same argument extends to any layout of morphisms across modules.

In general, contexts are only expanded with the morphisms in scope. Previously checked modules need no modification when morphisms are added in modules importing them. This means separate compilation, an essential feature in large projects, is not affected. This is in contrast to superclasses where, even if there was some automatic generation of instances, the shape of the dictionaries for C would change, and modules compiled with the superclass could not interoperate with those that were compiled without it.

For a detailed example, consider the following modules. All modules import Prelude, and we omit methods since they are irrelevant here.

```
module Prelude where
  class morphism Monad → Applicative
  class morphism Applicative → Functor
  class morphism Pointed → Functor
  data T a

module ModA where
  instance Monad T

module ModB where
  import ModA
  instance Pointed T

module ModC where
  import ModA
  instance Applicative T
```

module ModD **where**
  import ModA
  class morphism Monad → Functor
  data R a
  instance Monad R

module ModE **where**
  instance Functor T

The Prelude module contains no instances.

While checking ModA, instances for Applicative T and Functor T are generated and added to the module. They are also exported, alongside the Monad T one.

While checking ModB, since ModA's Functor T instance is in-scope, the derived Functor T is trimmed (otherwise, ModB would raise an overlap error).

For ModC, there is a Functor T candidate, which is trimmed for the same reason as in ModB. However, the Applicative instance is rejected since it overlaps with ModA's Applicative T instance. While trimming ModC's instance would succeed (canonically and coherently!), we believe the compiler should honour the source instance. The sensible choice is to fail, and have the programmer amend the situation.

In ModD, the shortest path between Monad and Functor has now changed, but this does not affect existing instances in any way. No conflict arises for T since no new instances are generated, even if ModB, ModC, or both, are imported. For R, a Functor R instance is generated via the new morphism, and an Applicative R one from the one in ModA. Since R and T are different types, canonicity holds.

Module ModE (which generates no new instances) overlaps with ModA, but neither module imports each other, so this cannot be detected here. This could be detected later if the compiler performed overlap checking for imports; but (currently) GHC does not perform this check and would accept this example.

Note how the errors and (global) losses of canonicity arise from "orphan" instances and "orphan" morphisms. Moving T to ModA, and the Monad → Functor morphism to Prelude, prevents these situations altogether.

3.8 A Limitation: Higher-rank Polymorphism

Consider the following code using higher-rank polymorphism, available in GHC via the `-XRankNTypes` option:

module A **where**
  $f :: (\forall a.\mathsf{Enum}\ a \Rightarrow a \to a \to \mathsf{Bool}) \to \mathsf{Bool}$
  $f\ c = c\ 1\ 2$ -- a is instantiated to Int

module B **where**
  class morphism Enum → Eq **where** ...
  $g :: \mathsf{Enum}\ a \Rightarrow a \to a \to \mathsf{Bool}$
  $g\ x\ y = x \equiv y$
  $h = f\ g$

Terms			
E, F	::=	x	*variables*
	$\mid$	EF	*application*
	$\mid$	$\lambda x.E$	*abstraction*
	$\mid$	let $x : \sigma := E$ in F	*local definition*
Types			
τ	::=	t	*type variables*
	$\mid$	$\tau \to \tau$	*function types*
ρ	::=	$P \Rightarrow \tau$	*qualified types*
σ	::=	$\forall T.\rho$	*type schemes*
Logical components			
$C, D, \ldots$			*class names*
π	::=	$C\ \tau$	*constraints*
P	::=	$\pi_1, \pi_2, \ldots$	*contexts*
c	::=	Class $P \Rightarrow \pi$ where $m_i : t_i$	*class declarations*
i	::=	Inst $P \Rightarrow \pi$ where $m_i = E_i$	*instance declarations*
m	::=	Morph $C \to D$ where $m_i = E_i$	*morphism declarations*
Γ	::=	$(\bar{c}, \bar{i}, \overline{m})$	*program contexts*

Figure 1. Syntax for DML

In module A, at the time of the definition of f, the morphism was not in scope and so the annotated type of f is not changed: it expects an Enum-polymorphic function. The body of f can then call its argument c providing only a dictionary for Enum, as it does, providing the one for Enum Int. Now in module B, the g function uses $(\equiv)$ over a type for which only Enum was assumed. This, of course, succeeds only due to the morphism present in B. We have previously argued (and we shall prove in §4) that, for Hindley-Milner polymorphism, this expansion is safe.

However, typechecking h fails, as f expects a function with only an Enum constraint, for which it can provide evidence. It was not compiled to provide a Eq a dictionary and it is even possible that no Eq Int instance is in-scope at A.

Trying to coerce g into the (real) type Enum $a \Rightarrow a \to a \to \mathsf{Bool}$ is also not possible. While, due to saturation, there *must* be a matching Eq instance for the Enum a constraint, it cannot generically be computed at this point without losing canonicity, as a is unknown.

Some possible solutions to this problem could be to allow opting-out of context expansion, or to allow explicitly constructing the derived dictionary via a morphism, at the cost of losing canonicity.

4 Class Morphisms, Formally

In this section we provide a formal description of class morphisms for a a core calculus with typeclasses, dubbed DML (for "deductive" ML). The formalisation is heavily based on Jones's [1995] theory of qualified types. Similarly to the language used there (OML), our language is Hindley-Milner polymorphic and contains global (unscoped) declarations of

$$\frac{}{P \Vdash P}\textsc{id} \qquad \frac{}{P \Vdash \emptyset}\textsc{term} \qquad \frac{}{P, Q \Vdash P}\textsc{fst} \qquad \frac{}{P, Q \Vdash Q}\textsc{snd}$$

$$\frac{P \Vdash Q \qquad P \Vdash R}{P \Vdash Q, R}\textsc{univ} \qquad \frac{P \Vdash Q \qquad Q \Vdash R}{P \Vdash R}\textsc{trans} \qquad \frac{P \Vdash Q}{SP \Vdash SQ}\textsc{close}$$

$$\frac{(\text{Class } P \Rightarrow \pi) \in \Gamma}{\pi \Vdash P}\textsc{super} \qquad \frac{(\text{Inst } P \Rightarrow \pi) \in \Gamma}{P \Vdash \pi}\textsc{inst} \qquad \frac{(C \rightarrow D) \in \Gamma}{Ca \Vdash Da}\textsc{morph}$$

Figure 2. DML predicate entailment

$$\frac{P \mid A \vdash E : \tau' \rightarrow \tau \qquad P \mid A \vdash F : \tau'}{P \mid A \vdash EF : \tau}\rightarrow_E \qquad \frac{P \mid A, x : \tau' \vdash E : \tau}{P \mid A \vdash \lambda x.E : \tau' \rightarrow \tau}\rightarrow_I$$

$$\frac{P \mid A \vdash E : \pi \Rightarrow \rho \qquad P \Vdash \pi}{P \mid A \vdash E : \rho}\Rightarrow_E \qquad \frac{P, \pi \mid A \vdash E : \rho}{P \mid A \vdash E : \pi \Rightarrow \rho}\Rightarrow_I$$

$$\frac{P \mid A \vdash E : \forall \alpha.\sigma}{P \mid A \vdash E : \sigma[\tau/\alpha]}\forall_E \qquad \frac{P \mid A \vdash E : \sigma \qquad \alpha \notin TV(A) \cup TV(P)}{P \mid A \vdash E : \forall \alpha.\sigma}\forall_I$$

$$\frac{(x : \sigma) \in A}{P \mid A \vdash x : \sigma}\textsc{var} \qquad \frac{P \mid A \vdash E : \sigma \qquad Q \mid A, x : \sigma \vdash F : \tau}{P, Q \mid A \vdash (\mathsf{let}\ x : \sigma := E\ \mathsf{in}\ F) : \tau}\textsc{let}$$

Figure 3. Typing rules for DML

typeclasses and instances. We also take OML as the target of the elaboration. Types in DML are stratified in order to restrict where type quantification and logical contexts are allowed; types such as $(\forall a.a \rightarrow b) \rightarrow b$ or $a \rightarrow \pi \Rightarrow b$ are not part of the language.

In DML, class morphisms are part of the program environment and taken into account for the entailment relation, but the calculus is otherwise identical to OML. We give its syntax in Figure 1 and its entailment rules in Figure 2. Entailment in DML differs from that of OML only in that rule MORPH is added. Its typing judgement (described in Figure 3) is of the form $P \mid A \vdash E : \sigma$, where P is a logical context, A is a standard typing environment and σ is a qualified type.

Program contexts Γ contain finite sets of classes, instances, and morphisms. Throughout this section, we ignore method definitions in these constructs. Although they need to be typechecked, this is orthogonal to both resolution and elaboration, which is our current interest.

When instantiating an overloaded term of type $P \Rightarrow \tau$, the constraints in P must be proven. This is formalised via the *entailment* relation, described in Figure 2. Entailment makes use of superclasses, instance declarations, and morphisms to prove constraints. We distinguish two sub-relations: $\Vdash_o$ is the subset that does not use the MORPH rule, and $\Vdash_v$ the one which does not use MORPH nor INST. The first subset coincides with the entailment relation in the target (OML). The second provides a way to compare constraints without depending on the set of morphisms or instances, and is used to compare instances by generality.

As previously described, our translation is based on three program transformations:

- *Close*: Contexts in functions and instance declarations are transformed to their logical closure.
- *Saturate*: Instance declarations are automatically generated to obtain a “cover” of the morphisms.
- *Trim*: Overlap of derived instances is removed, by comparing generality and by conflict policy.

The result after these transformations is an OML program, without any class morphisms. Since contexts were expanded, more constraints may need to be solved to typecheck it. We prove that these extra constraints can always be discharged, i.e. that the translation does not introduce errors. Trimming can fail, however, for two reasons: either a proper set of instances does not exist, or there might be several of them with no clear way to disambiguate. In either case, the programmer can fix the situation by providing extra instances.

4.1 Preliminaries

From here onwards, we fix a program context Γ with classes C, source instances $\mathcal{I}_0$ and morphisms $\mathcal{M}$.

When D is a superclass of C, we note it as D $\ll$ C (with Γ implicit). We say a constraint π_2 is a *consequence* of π_1, and note it as $\pi_1 \rightarrow \pi_2$, when π_2 can be concluded from π_1 in one step via a morphism or superclass assumption. Formally, the consequence relation is defined by the following rules:

$$\frac{D \ll C}{C\ \tau \rightarrow D\ \tau} \qquad \frac{(m : C \rightarrow D) \in \mathcal{M}}{C\ \tau \rightarrow D\ \tau}$$

We note the reflexive-transitive closure of $(\rightarrow)$ as $(\rightarrow^*)$.

Given a constraint $C\ \tau$, its *deductive closure* is obtained by collecting its transitive consequences; for a set of constraints, it is the union of the closures of its members.

$$\begin{aligned} \overline{\pi} &= \{\pi' \mid \pi \rightarrow^* \pi'\} \\ \overline{\{\pi_1, \pi_2, \ldots, \pi_n\}} &= \overline{\pi_1} \cup \overline{\pi_2} \cup \ldots \cup \overline{\pi_n} \end{aligned}$$

The deductive closure of a constraint $C\ \tau$ must be finite, as it is bound by the set $\{D\ \tau \mid D \in C\}$, and C is finite. Further, the deductive closure of a finite set is finite. The deductive closure is also *monotone*, that is, $\pi_1 \subseteq \pi_2 \implies \overline{\pi_1} \subseteq \overline{\pi_2}$. The fact that the deductive closure is monotone is crucial for modular compilation, where not all modules are in-scope at once. However, we do not formalise the modular aspect here.

We use the same notation on types and instance heads to denote the recursive transformation of every qualified context in them. That is, $\overline{P \Rightarrow \sigma} = \overline{P} \Rightarrow \overline{\sigma}$ and $\overline{\text{Inst } P \Rightarrow \pi} = \text{Inst } \overline{P} \Rightarrow \pi$ (note π is left unchanged).

We call an instance $(i : \text{Inst } P \Rightarrow \pi)$ *more general* than $(i' : \text{Inst } P' \Rightarrow \pi')$, and denote it by $i' \leq i$, when there is a substitution S such that $S\pi = \pi'$ and $P' \Vdash_v SP$. The reason $\Vdash_v$ is used is that it does not depend on the set of instances. The intuition is that i can replace i' since, after some instantiation, it requires the same set of hypotheses, or a weaker one. When $i \leq i'$ but $i' \not\leq i$, we say i' is *strictly* more general and note it as $i \prec i'$. Also, when $i \leq i'$ and $i' \leq i$, we call the instances *equivalent*.

Due to superclasses, not every program context is valid. When S is a superclass of C, an instance declaration for P $\Rightarrow$ C τ can only be accepted if there is a way to solve S τ from P. We model this by requiring that, for every instance $(i : \text{Inst } P \Rightarrow C\ \tau) \in \mathcal{I}_0$, there is an $i' \in \mathcal{I}_0$ such that $(\text{Inst } P \Rightarrow S\ \tau) \leq i'$. If this is the case for all instances in $\mathcal{I}_0$, we say that $\mathcal{I}_0$ *satisfies the classes in* C, and note it as $\mathcal{I}_0 \models C$.

Similarly, morphisms must also respect superclasses. We say a set of morphisms $\mathcal{M}$ satisfies a set of classes C (noted $\mathcal{M} \models C$) when for every morphism $(m : C \longrightarrow D)$ in $\mathcal{M}$ and $S \ll D$, there is a morphism path from C to S. Note that, in this formalisation, we take the more stringent approach required for 'upwards' morphisms in all cases. This simplifies the formalisation, and is inconsequential since one can always write these "direct" morphisms anyway.

We say two instances $(i : \text{Inst } P \Rightarrow C\ \tau)$ and $(i' : \text{Inst } P' \Rightarrow C\ \tau')$ *overlap* when τ unifies with τ'. In order to guarantee canonicity, overlaps must be forbidden, so we require that instances in $\mathcal{I}_0$ are non-overlapping from here onwards. We also assume that $\mathcal{I}_0 \models C$ and $\mathcal{M} \models C$. We now describe the elaboration steps.

4.2 Closing Contexts

For every instance declaration in Γ, its qualified contexts are expanded to its deductive closure (w.r.t. all morphisms in-scope). That is, we transform them like so:

$$\overline{\text{Inst } P \Rightarrow \pi} = \text{Inst } \overline{P} \Rightarrow \pi$$

This transformation is applied each instance in $\mathcal{I}_0$, generating a new set $\overline{\mathcal{I}_0}$, which we abbreviate as $\mathcal{I}_1$. Class definitions and morphisms are unaffected by this step.

The *types* of every term in the program also have their contexts expanded. For DML, the only place where qualified types appear in term syntax is in the 'let' construct. Therefore, we transform:

$$\begin{aligned}
\overline{x} &= x \\
\overline{EF} &= \overline{E}\ \overline{F} \\
\overline{\lambda x.E} &= \lambda x.\overline{E} \\
\overline{\text{let } x : \sigma := E \text{ in } F} &= \text{let } x : \overline{\sigma} := \overline{E} \text{ in } \overline{F}
\end{aligned}$$

In practice, the contexts of *every* type annotation (such as signatures) must be expanded.

The reason for classes to be unaffected by this step is that morphisms can be circular, but cycles in the class hierarchy can make typechecking undecidable. Even if decidability could be guaranteed, it is not clear one would accept that the declaration of a morphism may introduce superclasses. The lack of this expansion is only noticeable in default class methods. However, since they are usually defined in terms of each other, this does not seem like a significant limitation.

4.3 Saturating the Set of Instances

After expanding contexts, the translation proceeds to generate derived instances.

A morphism m may be *applied* to an instance declaration i, obtaining a derived instance $m\langle i \rangle$. More formally, application is defined in the following manner:

$$\begin{aligned}
m &= \text{Morph } C \longrightarrow D \text{ where } m_i = E_i \\
i &= \text{Inst } P \Rightarrow C\ \tau \text{ where } m_j = E'_j \\
m\langle i \rangle &= \text{Inst } P \Rightarrow D\ \tau \text{ where } m_i = E_i
\end{aligned}$$

Somewhat surprisingly, i's methods are completely ignored. The explanation is that definitions in the morphism (each E_i) are overloaded themselves, and i (or a more general instance) will be in the final set of instances. Therefore, the overloading will be *resolved* to the proper definitions by the typeclass system of OML.

Given the sets $\mathcal{I}_1$ and $\mathcal{M}$, we can consider the (possibly infinite) set of all instances that can be built from them by morphism application. We call this set the *saturation* of $\mathcal{I}_1$ (w.r.t. $\mathcal{M}$), and note it as $\mathcal{S}(\mathcal{I}_1)$ (with $\mathcal{M}$ implicit).

As an example, if $\mathcal{I}_1$ and $\mathcal{M}$ are given as per the following code (omitting methods):

```
     newtype Pair a = P { unP :: (a, a) }
i0 : instance Monad [ ]
i1 : instance Applicative Pair
m0 : class morphism Monad → Applicative
m1 : class morphism Applicative → Functor
```

then $\mathcal{S}(\mathcal{I}_1)$ is $\mathcal{I}_1$ with the following extra instances:

```
i2 = m0⟨i0⟩ : instance Applicative [ ]
i3 = m1⟨i2⟩ : instance Functor [ ]
i4 = m0⟨i1⟩ : instance Functor Pair
```

while if the two morphisms from §3.5 are added,

```
m2 : class morphism Applicative → Monoidal
m3 : class morphism Monoidal    → Applicative
```

then $\mathcal{S}(\mathcal{I}_1)$ contains an infinite number of Monoidal [ ] and Applicative [ ] instances by cycling through m_2 and m_3.

```
i5 = m2⟨i2⟩ : instance Monoidal [ ]
i6 = m3⟨i5⟩ : instance Applicative [ ]
i7 = m2⟨i6⟩ : instance Monoidal [ ]
...
```

We abbreviate $\mathcal{S}(I_1)$ as I_2. Given that $\mathcal{I}_0$ is valid, i.e. satisfies its classes, $\mathcal{I}_2$ is valid as well, since the required instances are generated by morphism application. A proof of this fact is given in the appendix. Furthermore, the saturation of any instance set satisfies the morphisms ($\mathcal{S}(\mathcal{I}) \models \mathcal{M}$) in the following sense: for any instance $(i : \text{Inst } P \Rightarrow C\ \tau) \in \mathcal{S}(\mathcal{I})$ and $(m : C \rightarrow D) \in \mathcal{M}$, there is an $i' \in \mathcal{S}(\mathcal{I})$ such that $(\text{Inst } P \Rightarrow D\ \tau) \preceq i'$. (Note the similarity to satisfying superclasses.) This is trivially attained by picking $i' = m\langle i \rangle$.

4.4 Trimming

After saturation, the resulting set of instances $\mathcal{I}_2$ satisfies its classes and its morphisms. This implies that the elaborated program can be typechecked correctly by resolving all the extra arising constraints, as shown in §4.5. However, there are two potential problems with this set: (1) it may be infinite and (2) it may contain overlapping instances. In order to be able to typecheck efficiently and canonically, a finite, non-overlapping set of instances with the same logical power is needed. This motivates the following definition:

Definition 4.1. We say that a set of instances I *covers* (or *is a cover of*) a set of instances J, when for every instance $j \in J$, there is some $i \in I$ such that $j \preceq i$. We denote this by $I \sqsubseteq J$. This relation is reflexive and transitive; i.e. a preorder.

Our goal is then to find a finite subset of $\mathcal{I}_2$ that is a non-overlapping cover of it. A first step is to remove the less-general instances from it:

Lemma 4.2. *If for i, i' in a set of instances $\mathcal{I}$ we have $i \preceq i'$, then $\mathcal{I} \setminus \{i\} \sqsubseteq \mathcal{I}$. That is, i can be removed without affecting the logical power of $\mathcal{I}$.*

We can iteratively apply this lemma in order to remove redundant instances from the generated context, but doing so unrestrictedly leads to ambiguity: if i and i' are equivalent (and no other instance is more general than them), which one should be kept? We can however safely remove instances that are *strictly* less general than others, without risking ambiguity. In that case, the process of removing instances is confluent and normalising, and thus one can automatically find an instance set where every instance is *maximal*, and which covers $\mathcal{I}_2$. It is here where $\Vdash_v$ proves valuable to compare instances, as it does not depend on the instances, which vary during trimming.

After this first cut, some overlap of maximal instances might still exist. Firstly, there might be an arbitrary number of equivalent instances (as in the example above with Monoidal and Applicative). In this case, it suffices for the policy or the programmer to pick a single representative of the equivalence class; by the previous lemma, this does not change the expressive power. Once done, the set of instances forcibly becomes finite.

However, that is not enough, as overlaps might exist between non-equivalent instances. Consider the definitions:

j_0 : **instance** A (Int, b)
j_1 : **instance** B (a, Bool)
n_0 : **class morphism** A $\rightarrow$ C
n_1 : **class morphism** B $\rightarrow$ C

Its saturation contains two extra instances, namely:

$n_0\langle j_0 \rangle$: **instance** C (Int, b)
$n_1\langle j_1 \rangle$: **instance** C (a, Bool)

which overlap, but neither of which is more general than the other, and removing either of which hinders coverage. This same issue can also be manifest via contexts, such as:

j_0 : **instance** P a $\Rightarrow$ A [a]
j_1 : **instance** Q a $\Rightarrow$ B [a]
n_0 : **class morphism** A $\rightarrow$ C
n_1 : **class morphism** B $\rightarrow$ C

whose saturation has the same dilemma as above:

$n_0\langle j_0 \rangle$: **instance** P a $\Rightarrow$ C [a]
$n_1\langle j_1 \rangle$: **instance** Q a $\Rightarrow$ C [a]

This kind of overlap, which we call *logical overlap*, cannot be automatically resolved by the typechecker. Therefore, programs containing logical overlap are rejected. The user can fix such an overlap by removing some of the offending instances or morphisms, or by declaring an instance which subsumes the conflicting ones (e.g. **instance** C a).

At the end of trimming, if successful, we are left with a finite non-overlapping set of instances $\mathcal{I}_3$ which covers $\mathcal{I}_2$, and which becomes the set of instances of the elaborated OML program. Class morphisms are discarded at this stage and the elaborated program context is defined as:

$$\overline{\Gamma} = (C, \mathcal{I}_3)$$

4.5 Correctness of Elaboration

At this stage, we have a program context $\overline{\Gamma}$ that covers the logical closure of Γ. Consider our initial program p, well-typed in Γ at type σ in DML. In OML, due to the fact that functions and instances now have extended contexts, more constraints need to be solved in order to typecheck $\overline{p}$. We prove that it is always the case that $\overline{p}$ is typeable in the target, at type $\overline{\sigma}$. This fact is mostly a consequence of the following lemma, whose proof we provide in the appendix.

Lemma 4.3 (closure entailment)**.**

$$\frac{\Gamma \mid P \Vdash P'}{\overline{\Gamma} \mid \overline{P} \Vdash_o \overline{P'}}$$

Using the previous lemma, it can be shown that typing is preserved by the closure translation:

Lemma 4.4 (correctness of elaboration, open environments)**.**

$$\frac{P \mid A \vdash e : \sigma}{\overline{P} \mid A \vdash_o \overline{e} : \overline{\sigma}}$$

As a corollary, by taking A and P to be empty environments, it follows that the transformation preserves well-typing of whole programs.

Theorem 4.5 (correctness of elaboration)**.**

$$\frac{\cdot \mid \cdot \vdash e : \sigma}{\cdot \mid \cdot \vdash_o \overline{e} : \overline{\sigma}}$$

5 Class Morphisms, in GHC

We describe the main parts of our prototype implementation: typechecking morphisms, calculating a cover, and generating instances efficiently. Additional technical details, can be found in the `README.md` file in the repository at http://github.com/cifasis/ghc-cm.

While we have presented the semantics of class morphisms as an elaboration *previous* to the elaboration of typeclasses, our implementation in fact performs both at once. This enables an optimisation: derived instances can be constructed by composing dictionary functions, instead of by generating source code. While the bodies of source instances and morphisms must be typechecked, there is no such need for derived instances, cutting down on compilation time. Interleaving morphism elaboration into the existing typechecking pipeline is not trivial, hence this section.

For context expansion, we also require that all relevant morphisms are in-scope when computing deductive closures, which is non-trivial given that Haskell code is unordered. We ensure this by tweaking GHC's dependency analysis to ensure every class C is typechecked in the same recursive "group" as all morphisms with antecedent C, which transitively ensures that every time a closure $\overline{\pi}$ is computed, it is not missing any constraints.

Our current prototype is not optimised, and we have not yet performed significant benchmarks. Also, we have not yet implemented error messages over source code. Errors are currently reported over elaborated terms.

The implementation is hardly invasive: our net change is around 1000 lines of code. About 300 lines are bureaucratic changes such as writing morphisms to `.hi` files and threading them through environments. The rest is almost completely accounted for by code to expand contexts and compute the cover. Existing typeclass code required virtually no modifications.

5.1 Typechecking Morphism Heads

GHC typechecks instances in two phases—first the heads, then the bodies. We take the same approach for morphisms. In this first phase, we simply check that each morphism is well-formed w.r.t. the superclass hierarchy and allocate a "dictionary function" for it. This dictionary function will later be used to build the derived instances.

Concretely, when typechecking a morphism such as:

```
class morphism Enum → Ord where
  compare x y = compare (fromEnum x) (fromEnum y)
```

the first step is to allocate a fresh name for the dictionary function. Its definition will be provided later, once the methods have been typechecked. The type of the dictionary function is morally $\forall a.\mathsf{EnumDict}\ a \to \mathsf{OrdDict}\ a$.

However, this function must also build dictionaries for the superclasses of the consequent, namely an $\mathsf{EqDict}\ a$, and this cannot (canonically) be done with a unknown. While the superclass check ensures there *must* be one such dictionary, the correct choice varies with the type variable a, which is unknown. Thus, these dictionaries are simply deferred by taking more arguments, to be filled-in later when they can be correctly resolved. In this example, the type for the dictionary function is:

$$m_d :: \forall a.\mathsf{EnumDict}\ a \to \mathsf{EqDict}\ a \to \mathsf{OrdDict}\ a$$

Having generated this identifier, we simply store it in the environment along with an internal representation of the morphism and its (not yet typechecked) set of methods.

5.2 Computing the Trimmed Cover

After having typechecked all instance and morphism heads in a module, we proceed to saturate and trim them, considering as well all imported instances and morphisms. As saturated sets can be infinite, attempting to first saturate and them trim could diverge. We avoid divergence by performing both steps at once.

Our implementations makes a choice tied to our particular policy of choosing shortest paths. Basically, instances are generated in a breadth-first fashion, ensuring that derived instances "closer" to a source instance are generated first. The immediate successors of an instance are the applications of it with all compatible morphisms, as expected. Thus, if at any point an instance i is generated, and an equally (or more) general i' was already generated (necessarily with a smaller distance), then *i and all its successors* will be "shadowed" by i' and its successors. Hence, it is safe to ignore i and cut the tree at this point.

Since the infiniteness of the saturated set can only stem from equivalence classes of instances, the procedure is guaranteed to terminate. After obtaining a finite cover, the implementation proceeds to trim it by removing non-maximal instances, as described in §4, but considering source instances as maximal. If any overlap (of equivalent derived instances, or a logical overlap) remains after this stage, the compiler rejects the program as ambiguous.

In reality, throughout this step, real instances are not yet generated: only "markers" with information on *how* they should be built are. Since many of them will be trimmed, it would be wasteful to generate them fully. More importantly, because of superclasses, their internal representations cannot be built independently, and this must be deferred until the instance set is fully determined.

5.3 Generating Instances

Internally, instances are also essentially dictionary-building functions. To build the dictionary function corresponding to a morphism application $m\langle i\rangle$, we essentially *compose* the dictionary functions of m and i. Take for example an instance declaration:

$\textbf{instance}\ \mathsf{Enum}\ a \Rightarrow \mathsf{Enum}\ (\mathsf{T}\ a)$

Its dictionary function will be of type:

$i_d : \forall a.\mathsf{EnumDict}\ a \rightarrow \mathsf{EnumDict}\ (\mathsf{T}\ a)$

Composing it with the previous m_d needs to account for both the the hypotheses and superclasses. For $m\langle i\rangle$, we generate the following dictionary function:

$j_d :: \forall a.\mathsf{EnumDict}\ a \rightarrow \mathsf{OrdDict}\ (\mathsf{T}\ a)$
$j_d\ enumDa = m_d\ (i_d\ enumDa)\ \_$

where the superclass constraint $\mathsf{EqDict}\ (\mathsf{T}\ a)$ marked with an underscore is filled by resolution with the proper instance, which is uniquely determined by now.

No typechecking of $m\langle i\rangle$'s methods is needed. In fact, the body of m need not even be known for this procedure. When m is imported from a module, its body is indeed hidden—only the name of its dictionary function is known.

5.4 Typechecking Morphism Bodies

Morphisms themselves must, of course, be checked, and their dictionary functions given a definition. To do this, we simply typecheck them *as if* they were instances, adding the superclasses of the consequent to their context. For instance, the morphism in §5.1 is elaborated into the following instance declaration:

$\textbf{instance}\ (\mathsf{Enum}\ a, \mathsf{Eq}\ a) \Rightarrow \mathsf{Ord}\ a\ \textbf{where}$
$\quad \mathsf{compare}\ x\ y = \mathsf{compare}\ (\mathsf{fromEnum}\ x)\ (\mathsf{fromEnum}\ y)$

Then, this instance is typechecked by GHC's existing instance typechecking procedures, without any modification. The result of checking the instance is a definition for m_d at the proper type, completing the program.

While a morphism is typechecked as if it were an instance, this "virtual" instance is simply a typechecking artifact: it does not participate in resolution at all.

6 Related work

Elimination of Boilerplate Instances There have been several proposals for extending GHC which tackle typeclass refactoring and boilerplate instances. In *default superclasses* [McBride 2011] and *intrinsic superclasses* [McBride 2014] one can have default definitions for superclasses. The *instance template* proposal [Eisenberg 2014] is closer to ours in the sense that from one class one can obtain instances for other classes which are not necessarily superclasses. However, as opposed to class morphisms, all these proposals require modifying class definitions, prohibiting the addition of relations between classes in imported code.

Extensible Superclasses Extensible superclasses extend the typeclass system with a kind of constraint handling rules to specify superclasses openly [Sulzmann and Wang 2006]. However, these rules do not provide a definition of the superclass in term of the subclass, and hence do not remove the need for new instances. A notable strength of extensible superclasses is that they seamlessly support higher-rank polymorphism, while class morphisms do not in general. On the other hand, while class morphisms only require an elaboration, implementing extensible superclasses entails a deep modification of the language semantics: typing information needs to be present during execution, which in turn causes a non-trivial overhead.

Deriving Via Deriving Via is an extension to Haskell's generalised newtype deriving mechanism [Blöndal et al. 2018]. It allows the programmer to obtain an instance from that of a representationally equivalent type. While it alleviates the definition of boilerplate instances, its does not solve (nor attempts to solve) the problem of refactoring the class hierarchy. It would be interesting to extend it to allow instance derivation via (named) class morphisms.

Instance Chains Instance chains [Morris and Jones 2010] is a mechanism for defining instances via closed, backtracking, user-defined pattern-matching on types. It allows programmers to go beyond the usual power of instances, while maintaining the canonicity and coherence of the typeclass system. While some default generic definitions can be given via instance chains, they must either be closed or manually triggered, two characteristics which bring maintainability issues. On the other hand, their ability to control overlapping could be well-appreciated in the generation of a cover, as it would allow ordering some overlaps and thus accepting more programs.

Typeclasses Based on Implicit Search Another kind of typeclass systems, mostly used in proof assistants [de Moura et al. 2015; Devriese and Piessens 2011; Sozeau and Oury 2008], is based on implicit arguments and proof search. These systems do not guarantee canonicity, but using dependent types, the property can sometimes be encoded in dictionaries. In these systems, defining a morphism from a class C to a class D can amount to simply defining a function $\mathsf{C} \rightarrow \mathsf{D}$ on dictionaries and marking it as an instance. Our work can be seen as bringing this expressivity into Haskell, safely.

7 Conclusions and Future Work

We have presented class morphisms, a new feature for introducing and exploiting relations between classes. A key aspect of class morphisms is that they are open: anyone can add a class morphism even without access to modifying the class definition. This goes against the grain for systems of qualified types, in which the need for preserving canonicity favours closed global definitions. Therefore, although the idea of class morphisms is quite intuitive, its semantics needed care in order not to lose canonicity.

One possible generalisation of class morphisms is to allow instance-like shapes such as

$$\textbf{class morphism}\ (\mathsf{X}\ a, \mathsf{Y}\ [a]) \rightarrow \mathsf{Z}\ a$$

where the consequent must be "smaller" than every antecedent if deductive closures and covers are to be finite. However, the usefulness of this generalisation is yet unclear.

A limitation of class morphisms is that higher-order polymorphism (as available in GHC via extensions) is not seamlessly handled. Expanding the context of functions can be problematic in the presence of higher-order polymorphism, mainly due to the contravariance of left-nested contexts.

As illustrated by the examples, class morphisms allow the expression of class relations as first-class language constructs, thus making several painful situations easy on the programmer. Their semantics is given by a simple elaboration and, importantly, do not require Haskell's resolution and dynamic semantics to be affected, making their implementation in a real compiler straightforward.

Acknowledgments

We thank the anonymous reviewers for their helpful feedback. This work has been funded by Agencia Nacional de Promoción Científica y Tecnológica PICT 2016-0464.

References

Baldur Blöndal, Andres Löh, and Ryan Scott. 2018. Deriving Via or, How to Turn Hand-Written Instances into an Anti-Pattern. In *Proceedings of the ACM SIGPLAN Haskell Symposium 2018*.

Leonardo Mendonça de Moura, Jeremy Avigad, Soonho Kong, and Cody Roux. 2015. Elaboration in Dependent Type Theory. *CoRR* abs/1505.04324 (2015). arXiv:1505.04324 http://arxiv.org/abs/1505.04324

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

Richard Eisenberg. 2014. Instance Template Proposal. https://ghc.haskell.org/trac/ghc/wiki/InstanceTemplates

Haskell Wiki. 2014. Functor-Applicative-Monad Proposal. https://wiki.haskell.org/Functor-Applicative-Monad_Proposal

Mark P. Jones. 1995. *Qualified Types: Theory and Practice*. Cambridge University Press, New York, NY, USA.

Conor McBride. 2011. Default Superclasses Proposal. https://ghc.haskell.org/trac/ghc/wiki/DefaultSuperclassInstances

Conor McBride. 2014. Intrinsic Superclasses Proposal. https://ghc.haskell.org/trac/ghc/wiki/IntrinsicSuperclasses

J. Garrett Morris and Mark P. Jones. 2010. Instance Chains: Type Class Programming Without Overlapping Instances. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP '10)*. ACM, New York, NY, USA, 375–386. https://doi.org/10.1145/1863543.1863596

Matthieu Sozeau and Nicolas Oury. 2008. First-Class Type Classes. In *Theorem Proving in Higher Order Logics*, Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 278–293.

Martin Sulzmann and Meng Wang. 2006. Modular Generic Programming with Extensible Superclasses. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Generic Programming (WGP '06)*. ACM, New York, NY, USA, 55–65. https://doi.org/10.1145/1159861.1159869

The Glasgow Haskell Team. 1989–2018. GHC: The Glasgow Haskell Compiler. https://www.haskell.org/ghc/.

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

A Proofs

Firstly, we show that the elaboration of a valid context Γ is valid at the target; i.e., that it satisfies its superclasses (Lemma A.4). We start with some auxiliary lemmas.

Lemma A.1. *If $P \Vdash_v P'$, then $\overline{P} \Vdash_v \overline{P'}$*

Proof. By induction on the shape of the hypothesis. All applicable cases, except for SUPER, are trivial by the inductive hypothesis and since the closure is monotonic over sets (trivially by its definition). For SUPER, we need to prove $\overline{\pi} \Vdash_v \overline{P}$, but $\overline{P}$ is contained in $\overline{\pi}$, since $\pi \longrightarrow \pi'$ for each $\pi' \in P$. Therefore we conclude through FST and ID. □

Lemma A.2. *For instances i and i', if $i \le i'$ then $\overline{i} \le \overline{i'}$.*

Proof. Let $i = \text{Inst } P \Rightarrow C\ \tau$ and $i' = \text{Inst } P' \Rightarrow C\ \tau'$. By hypothesis, we have that there is a substitution S such that $S\tau' = \tau$ and $P \Vdash_v SP'$. Since the elaborated instances differ only in their contexts, all that needs to be proven is that $\overline{P} \Vdash_v \overline{SP'}$. This follows from Lemma A.1. □

The following lemmas use the names $\mathcal{C}$, $\mathcal{M}$, and $\mathcal{I}_i$ as in §4, with the same assumptions. For an instance head Inst $P \Rightarrow C\ \tau$, we write $i[D]$ for the instance head created by replacing its class with D, i.e. Inst $P \Rightarrow D\ \tau$.

Lemma A.3 (validity of saturation)**.** $\mathcal{I}_2 \models \mathcal{C}$.

Proof. We pick an instance $(i : \text{Inst } P \Rightarrow C\ \tau)$ in Γ', where $S \ll C$. We need to show that an instance as least as general as (Inst $P \Rightarrow S\ \tau$) exists in $\mathcal{I}_2$. We proceed by case analysis.

- Say $i = \overline{i_0}$ for $i_0 \in \mathcal{I}_0$. Since $\mathcal{I}_0 \models \mathcal{C}$, there must be an i' in $\mathcal{I}_0$ at least as general as $i_0[S]$. Then $\overline{i'} \in \mathcal{I}_2$, and by Lemma A.2, it is more general than $\overline{i_0[S]} = i[S]$.
- Say $i = m\langle i' \rangle$, for some i' in $\mathcal{I}_2$. Since $\mathcal{M} \models \mathcal{C}$, we have that there must be morphisms $m_1, \ldots, m_n$ such that $m_1\langle \ldots \langle m_n \langle i' \rangle \rangle \rangle = i[D]$. Clearly, this instance also belongs to $\mathcal{I}_2$.

□

Lemma A.4 (validity of trimmed set)**.** $\mathcal{I}_3 \models \mathcal{C}$.

Proof. This follows trivially from Lemma A.3 and $\mathcal{I}_3$ covering $\mathcal{I}_2$. □

Lemma A.5. *If $(i : \text{Inst } P \Rightarrow C\ \tau) \in \mathcal{I}_3$ and $C\ \tau \longrightarrow D\ \tau$, then there is an instance $i' \in \mathcal{I}_3$ such that $i[D] \le i'$.*

Proof. By case analysis on $C\ \tau \longrightarrow D\ \tau$. If the relation holds by a superclass assumption, then the fact that $\mathcal{I}_3 \models \mathcal{C}$ (by Lemma A.4) guarantees exactly that the required instance exists. Suppose instead is it via a morphism $m : C \longrightarrow D$. Therefore, $m\langle i \rangle$ must be in $\mathcal{I}_2$, as it is saturated set and $i \in \mathcal{I}_2$. The instance $m\langle i \rangle$ is equivalent to $i[D]$, and since $\mathcal{I}_3$ is a cover of $\mathcal{I}_2$, it must contain a more general instance, hence we conclude. □

Lemma A.6. *If $(i : \text{Inst } P \Rightarrow C\ \tau) \in \mathcal{I}_3$ and $C\ \tau \longrightarrow^* D\ \tau$, then there is an instance $i' \in \mathcal{I}_3$ such that $i[D] \le i'$.*

Proof. We proceed by induction on the $\pi \longrightarrow^* \pi'$ hypothesis. If the path is of length zero, take $i' = i$. Suppose instead that $C\ \tau \longrightarrow E\ \tau \longrightarrow^* D\ \tau$. By the previous lemma, there is an instance $(j : \text{Inst } P_j \Rightarrow E\ \tau_j)$ such that $i[E] \le j$. That is, there is a substitution S_1 such that $S_1\tau_j = \tau$ and $P \Vdash_v S_1 P_j$. Since the $\longrightarrow$ relation does not depend on concrete types, we also have $E\ \tau_j \longrightarrow^* D\ \tau_j$. By the induction hypothesis, then, there exists an instance $(k : \text{Inst } P_k \Rightarrow D\ \tau_k)$ such that $j[D] \le k$. That is, there is a substitution S_2 such that $S_2\tau_k = \tau_k$ and $P_j \Vdash_v S_2 P_k$. We take $i' = k$. To see that it is more general than $i[D]$, take the substitution $S_1 \cdot S_2$. Firstly, $(S_1 \cdot S_2)\tau_k = S_1(S_2\tau_k) = S_1\tau_j = \tau$, as needed. Then, by the SUBST rule and a hypothesis, we also obtain that $S_1 P_j \Vdash_v S_1(S_2 P_k) = (S_1 \cdot S_2) P_k$. By applying TRANS, and another hypothesis, we obtain that $P \Vdash_v (S_1 \cdot S_2) P_k$, as needed. □

Lemma A.7. *If $(i : \text{Inst } P \Rightarrow C\ \tau) \in \mathcal{I}_3$, then $P \Vdash_o \overline{C\ \tau}$.*

Proof. For each $D\ \tau \in \overline{C\ \tau}$, there is, from the previous lemma, an instance that allows to conclude $P \Vdash_v D\ \tau$. We conclude by repeated application of UNIV. □

Now we are ready to prove our main lemma, stating that entailment is preserved by taking closures.

Proof for Lemma 4.3. We proceed by induction over the derivation of $P \Vdash P'$ in DML. The cases for the first six rules of entailment are trivial. The case for rule CLOSE follows from the fact that substitutions commute with the deductive closure, which is easily established. The interesting cases are rules SUPER, MORPH, and INST. For the SUPER, suppose we concluded P from $C\ a$ via SUPER. By the premise of SUPER, we have that P is the set of superclasses of $C\ a$. By the definition of the deductive closure, it is immediate that $\overline{P} \subseteq \overline{C\ a}$. We conclude by monotonicity of entailment. The case for MORPH is analogous to SUPER, as they have the same effects regarding entailment. For INST, suppose $C\ \tau$ was concluded from an instance declaration Inst $P \Rightarrow C\ \tau$ in $\mathcal{I}_0$. Then, an instance at least as general as Inst $\overline{P} \Rightarrow C\ \tau$ exists in $\mathcal{I}_3$. We can conclude by applying Lemma A.7 to this instance, obtaining $\overline{P} \Vdash_o \overline{C\ \tau}$. □

Using the previous lemma, the proof for Lemma 4.4 is immediate, since the only significant difference between DML and OML is the notion of entailment. Theorem 4.5 is simply a corollary of Lemma 4.4.