Type Families with Class, Type Classes with Family

Alejandro Serrano  Jurriaan Hage  
Department of Information and Computing Sciences  
Utrecht University  
{A.SerranoMena, J.Hage}@uu.nl

Patrick Bahr  
Department of Computer Science  
University of Copenhagen  
paba@di.ku.dk

Abstract
Type classes and type families are key ingredients in Haskell programming. Type classes were introduced to deal with ad-hoc polymorphism, although with the introduction of functional dependencies, their use expanded to type-level programming. Type families also allow encoding type-level functions, but more directly in the form of rewrite rules.

In this paper we show that type families are powerful enough to simulate type classes (without overlapping instances), and we provide a formal proof of the soundness and completeness of this simulation. Encoding instance constraints as type families eases the path to proposed extensions to type classes, like closed sets of instances, instance chains, and control over the search procedure.

The only feature which type families cannot simulate is elaboration, that is, generating code from the derivation of a rewriting. We look at ways to solve this problem in current Haskell, and propose an extension to allow elaboration during the rewriting phase.

Categories and Subject Descriptors D.3.2 [Programming Languages]: Language Classifications – Functional Languages; F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs – Type Structure

Keywords Type classes; Type families; Haskell; Elaboration; Functional dependencies; Directives

1. Introduction
Type classes are one of the distinguishing features of Haskell. They are widely used and extensively studied [14]. Their original purpose was to support ad-hoc polymorphism [21]: a type class gives a name to a set of operations along with their types; subsequently, a type may become an instance of such a class by giving the code for such operations. Furthermore, an instance for a type may depend on other instances (its context). The following is a classic example of the Show type class and its instance for lists, which illustrates these features:

```haskell
class Show a where
    show :: a -> String
```

```haskell
instance Show a ⇒ Show [a] where
    show lst = "[" ++ intersperse "," (map show lst) ++ "]"
```

The show function is said to be overloaded: the same name refers to several possible implementations. In order to choose the correct one in each position, the compiler needs to perform resolution over the set of available instances, and build the resulting code. This procedure is called elaboration.

Type classes have been extended to support multiple parameters: a unary type class describes a subset of types supporting an operation, whereas a multi-parameter type class describes a relation over types. For example, we can declare a Convertible class that describes those pairs of types for which the first can be safely converted into the second:

```haskell
class Convertible a b where
    convert :: a → b
```

In many cases, though, parameters in such a class cannot be given freely. For example, if we define a Collection class which relates types of collections and the type of their elements, it does not make sense to have more than one instance per collection type. Such constraints can be expressed using functional dependencies [10], a concept borrowed from database theory:

```haskell
instance Collection c e | c → e where
    empty :: c
    add :: e → c → c

    instance Collection [a] a where
    empty = []
    add = (:
```

If we try to add a new instance for [a], the compiler does not allow it, since for each type of collection c, we can only have one type e that satisfies the constraint Collection c e.

Functional dependencies determine a functional relation over types, and thus can be used to define functions at the level of types. It is now common folklore [15] how to do this: to encode a type level function of n parameters, we define a type class with an extra parameter (the result) and include a functional dependency of it on the remaining n parameters (the arguments). Each instance declaration will then act a rule for the function definition. Here is the archetypical addition function over unary numbers defined as a type class AddC:

```haskell
instance AddC m n r | m n → r
instance AddC n n n
instance AddC m n r ⇒ AddC (Succ m) n (Succ r)
```

1 This example works only in GHC with UndecidableInstances extension.
At the moment of writing, multi-parameter type classes and functional dependencies are not yet part of the Haskell Report, but are arguably one of the most widely-used extensions to the Haskell standard. Major implementations such as the Glasgow Haskell Compiler (GHC) and the Utrecht Haskell Compiler (UHC) support these features.

Type families were introduced as a more direct way to define type functions in Haskell. Each family is introduced by a declaration of its arguments and, optionally, kind annotations (for the arguments and the result). The rules for the function are stated in a series of type instance declarations. For example, addition can be defined as follows:

```haskell
type family AddF m n
type instance AddF Zero n = n
type instance AddF (Succ m) n = Succ (AddF m n)
```

Type families have one important feature in common with type classes: they are open. This means that in any other module, a new rule can be added to the family, given that it does not overlap with previously defined ones. Note that whereas in the case of type classes overlapping instances hurt maintainability of the code, in the case of type families an overlap induces unsoundness in the type system as a whole.

However, when thinking in terms of functions, we are not used to wear our open-world hat. In a case like `AddF`, we would want to define a complete function, with a restricted domain. Eisenberg et al. introduced closed type families to bridge this gap. The rules of closed type family definitions are matched in order: each rule is only tried when the previous one is assured never to match. Thus, overlapping is not a problem. On the other hand, these families cannot be extended in a later declaration. In GHC, closed type families are introduced using the following syntax:

```haskell
type family AddF' m n where
  AddF' Zero n = n
  AddF' (Succ m) n = Succ (AddF' m n)
```

Closed type families allow non-linear pattern matching, that is, making rules apply depending on whether several arguments are equal or not. This allows us to define an equality predicate:

```haskell
type family Equal x y where
  Equal x x = True
  Equal x y = False
```

In addition, families can be associated with a type class. This means that whenever we give an instance of such a class, we also need to provide an equation for the family. The `Collection` class is a good candidate to be given an associated type instead of the second type argument with a functional dependency:

```haskell
class Collection2 c where
  type Element c
  empty2 :: c
  add2 :: Element c -> c -> c

instance Collection2 [a] where
  type Element [a] = a
  empty2 = []
  add2 = (\)
```

Currently, type families are only available in GHC: open and associated type families are available since version 6.8, and closed type families since 7.8.

As we have seen above, type classes with functional dependencies can simulate type families. This translation works well in most situations, with the notable exception of certain data type definitions. For example, take the following family-dependent data type:

```haskell
type family Family a
data Example a = Example (Family a)
```

and its corresponding type class translation:

```haskell
class FunDep a b | a -> b
data Example' a where
  Example :: FunDep a b => b -> Example' a
```

where `b` is implicitly existentially quantified. The compiler can type check the following definition:

```haskell
f :: Example a -> Family a
f (Example x) = x
```

but not the one with functional dependencies:

```haskell
f :: FunDep a b => Example' a -> b
f (Example' x) = x
```

since the compiler does not know whether the type `b`, wrapped by the GADT constructor, is the same as in the signature.

Thus, at the moment, type class with functional dependencies do not cover all use cases of type families. Our main technical contribution in this paper is the converse direction: using type families to express type classes including functional dependencies. Our translation of type classes to type families is discussed in Section 3, and formally proven sound and complete with respect to the Haskell typing semantics in Section 4. In this paper, we consider type classes without support for overlapping instances. However, as we argue in Section 3, most common uses of overlapping instances can be expressed in a more controlled way using instance chains, which our translation does support.

Looking closer at our translation, we discover that use cases that are difficult to express using type classes, or which have been proposed as extensions to the Haskell language, can be more easily encoded using type families. In particular, we discuss type class directives, preconditions, instance chains and error messages in Section 4.

Our translation works perfectly well from the typing perspective, but a key ingredient is missing to make families as featureful as classes, namely elaboration. We discuss this issue in Section 5 in which we first review ways in which elaboration can be simulated for type families using classes. Then, we propose a new extension to the Haskell language to allow elaboration while rewriting.

2. Data Type Promotion and Kind Polymorphism

Throughout this paper we use data type promotion, an extension to Haskell implemented in GHC since version 7.4. In short, data type promotion allows us to reuse the constructors at the term level as types at the type level, and similarly lifts types into kinds.

We have already seen one example of this feature in the previous definition of the `Equal` type family, which uses the promoted data type of Booleans:

```haskell
data Bool = False | True
```

Data type promotion produces the two types `False` and `True` of kind `Bool`. Using type families we can define functions which operate on elements of a specific kind:

```haskell
type family And (x :: Bool) (y :: Bool) :: Bool where
  And True True = True
  And True False = False
  And False True = False
  And False False = False
```

In some cases, GHC needs a quote sign in front of promoted data types to distinguish them from the constructors and types they come from.
Data type promotion also enriches greatly the kind world. Instead of simple combinations of \( \ast \) and \( \rightarrow \), we can now have kind-level constructors coming from the promotion of a parametrized type. For example, the definition:

\[
\text{data} \quad \text{Maybe} \ a = \text{Nothing} \mid \text{Just} \ a
\]

promotes into types \( \text{Nothing} \) and \( \text{Just} \) whose kind is parametric:

\[
\text{Nothing} :: \forall k. \text{Maybe} \ k \\
\text{Just} :: k \rightarrow \text{Maybe} \ k
\]

**Kind polymorphism** is also reflected in type families. For example, if we write the following type-level version of the \( \text{isJust} \) function:

\[
\text{type family} \quad \text{IsJust} \ x \ where \\
\text{IsJust} \ (\text{Just} \ x) = \text{True} \\
\text{IsJust} \ \text{Nothing} = \text{False}
\]

Then the compiler will infer a polymorphic kind for that family:

\[
> :\text{kind!} \ \text{IsJust} \\
\text{IsJust} :: \forall k. \text{Maybe} \ k \rightarrow \text{Bool} \\
= \forall \ (k :: \text{BOX}). \text{IsJust}
\]

Not that in GHC BOX is the name given to the sort of kinds.

### 3. Simulating Type Classes Using Type Families

This section forms the core of the paper: we discuss how to simulate the typing part of type classes by means of type families. Elaboration, though, is a very different beast, and we defer discussion of this aspect until Section 5. Moreover, we keep the presentation in this section simple and somewhat informal. We revisit the translation with a focus on formal correctness in Section 6.

The essential idea is to represent a type class by the **characteristic function** of the relation that is given by the type class. That is, instead of an instance constraint \( \text{Show String} \) we write \( \text{IsShow String} \sim \text{Yes} \), where \( \sim \) is the notation for type equality. We follow the convention that a type class \( \text{D} \) gives rise to a corresponding type family \( \text{IsD} \). Let us look at all the components of this construction via an example.

In principle, we could reuse the promoted data type \( \text{Bool} \) as result kind of these characteristic functions. Instead, we define a fresh kind \( \text{Defined} \), given as follows:

\[
\text{data} \quad \text{Defined} = \text{Yes} \mid \text{No}
\]

There are two main reasons for defining a new kind instead of merely using \( \text{Bool} \). The first reason is that we distinguish the type families arising from translated type classes on one side, and the type families that happen to work on kind \( \text{Bool} \) on the other side. This distinction – maintained by the kind system – is important to obtain a sound and complete translation. The second reason is that throughout the paper we shall enlarge \( \text{Defined} \) to include more information and defining a separate kind gives us this freedom.

Every type class declaration of the form \( \text{class} \ C \ t_1 \ldots t_m \) is translated to a corresponding type family \( \text{IsC} \):

\[
\text{type family} \quad \text{IsC} \ t_1 \ldots t_m :: \text{Defined}
\]

For example, consider the definition of the \( \text{Eq} \) type class:

\[
\text{type class} \quad \text{Eq} \ a \ where \\
\quad (\equiv), (\neq) :: a \rightarrow a \rightarrow \text{Bool}
\]

\( \text{Eq} \) is translated to the following type family:

\[
\text{type family} \quad \text{IsEq} \ (t :: \ast :: \ast) :: \text{Defined}
\]

Note that we have included a kind signature \( \ast \) for the argument \( t \) because the definition of the \( \text{Eq} \) type class restricts its instances to that kind. This is inferred by the compiler from the signatures of \( (\equiv) \) and \( (\neq) \), but cannot be done automatically for \( \text{IsEq} \).

The next step is to change each function signature that uses an instance constraint into using this new type family instead. Being a member of class \( C \) is represented by the type constraint \( \text{IsC} \ t \sim \text{Yes} \). For example, say we want to declare an identity function whose domain is restricted to only those types that have an \( \text{Eq} \) instance:

\[
\text{eqIdentity} :: \text{Eq} \ t \Rightarrow t \rightarrow t \\
\text{eqIdentity} = \text{id}
\]

In the translation of type classes to type families, the type signature of \( \text{eqIdentity} \) is changed to the following:

\[
\text{eqIdentity} :: \text{IsEq} \ t \sim \text{Yes} \Rightarrow t \rightarrow t
\]

The whole point of declaring a type class is to populate it with instances. The simplest cases, such as \( \text{Char} \), are dealt with simply by defining a **type instance** which rewrites to \( \text{Yes} \):

\[
\begin{align*}
\text{type instance} & \quad \text{IsEq} \ \text{Char} = \text{Yes} \\
\text{type instance} & \quad \text{IsEq} \ \text{Int} = \text{Yes} \\
\text{type instance} & \quad \text{IsEq} \ \text{Bool} = \text{Yes}
\end{align*}
\]

Those cases whose definition depend on a context, such as \( \text{Eq} \) on lists, can call \( \text{IsC} \) on a smaller argument to defer the choice:

\[
\text{type instance} \quad \text{IsEq} \ [a] = \text{IsEq} \ a
\]

In the case of a more complex context, such as \( \text{Eq} \) on products, which needs to check both of its type variables, we introduce a type family \( \text{And} \) which checks for definedness of all its arguments:

\[
\begin{align*}
\text{type family} & \quad \text{And} \ (a :: \text{Defined}) \ (b :: \text{Defined}) \ where \\
& \quad \text{And} \ \text{Yes} \ \text{Yes} = \text{Yes} \\
& \quad \text{And} \ a \ b = \text{No}
\end{align*}
\]

As with type classes, we are not constrained to types (of kind \( \ast \)) in our type families; we can also use type constructors (of higher kind). For example, the \( \text{Functor} \) type class along with some instances is defined as follows:

\[
\begin{align*}
\text{type family} & \quad \text{IsFunctor} \ (t :: \ast \rightarrow \ast) :: \text{Defined} \\
\text{type instance} & \quad \text{IsFunctor} \ [\text{]} = \text{Yes} \\
\text{type instance} & \quad \text{IsFunctor} \ \text{Maybe} = \text{Yes}
\end{align*}
\]

Once again, we write a kind signature to prevent GHC from defaulting the kind of the \( t \) parameter to \( \text{IsFunctor} \) to \( \ast \), which would disallow writing the required instances. Having said that, in most of the cases where the declaration and instances of a type family are written together, the compiler is able to infer kinds correctly.

Finally, we are able to encode multi-parameter type classes in the same way as the \( \text{Collection} \) class in the introduction:

\[
\begin{align*}
\text{type family} & \quad \text{IsCollection} \ t \ e :: \text{Defined} \\
\text{type instance} & \quad \text{IsCollection} \ [e] = \text{Yes} \\
\text{type instance} & \quad \text{IsCollection} \ (\text{Set} \ e) = \text{Yes}
\end{align*}
\]

As in the case of one-parameter type classes, our \( \text{IsCollection} \) type family encodes the set of instances via its characteristic function. As a side remark, note that we are using non-linear patterns in the definition of this family instances.

#### 3.1 Overlapping Instances

We remark at this point that we consider type classes without support for overlapping instances.

Overlapping instances can be used to override an instance declaration in a more specific scenario. The

---

\[\text{Support for overlapping instances is available in GHC, from version 6.4 on, via the OverlappingInstances extension.}\]
best example is `Show` for strings, which are represented in Haskell as `[Char]`, and for which we want a different way to print them:

   instance Show [Char] where
   show str = ... -- show between quotes

Overlapping instances make reasoning about programs more difficult, since the resolution of instances may be changed by later overlapping declarations. In some cases, overlapping instances are crucial for a piece of code, so our lack of support is clearly a limitation of our approach.

However, the most common usage patterns of overlapping instances can be expressed using a more controlled mechanism of resolution, such as instance chains. As we shall see in Section 4, those mechanisms can be simulated using type families. Thus, we see the aforementioned limitation as a mild one: we cannot deal with all uses of overlapping instances, but we can with the most common ones.

### 3.2 Functional Dependencies and Injectivity

Thus far, our translation does not take into account functional dependencies in the definition of a type class. Functional dependencies [10] restrict the set of allowed instances of a type class. A functional dependency is a pair of sets, of the form $\{D_1, \ldots, D_n\} \to \{r\}$, where $D_i$ and $r$ are types. A functional dependency has the form $D_1 \to r$, expressing that type $r$ is uniquely determined by the types $D_1$. Examples of functional dependencies are given in the `Collection` and `AddC` type classes in the introduction.

In general, the left-hand side of a functional dependency declaration may include any type from the type class $D_1 \ldots D_n$, not only those occurring to the left of $r$. In some cases, overlapping instances are difficult, since the resolution of instances may be changed by later declarations. In other cases, overlapping instances are crucial for a piece of code, so our lack of support is clearly a limitation of our approach.

However, the most common usage patterns of overlapping instances can be expressed using a more controlled mechanism of resolution, such as instance chains. As we shall see in Section 4, those mechanisms can be simulated using type families. Thus, we see the aforementioned limitation as a mild one: we cannot deal with all uses of overlapping instances, but we can with the most common ones.

Functional dependencies influence the type checking, adding extra information, which is used by the compiler. In particular, two new kinds of steps are available when a type class $D_1 \ldots D_n \to r$ with a functional dependency $D_1 \to r$ is encountered:

- Suppose two different sequences of types are instances of $D_1$, that is, we have instances $D_1 t_1 \ldots t_m$ and $D_1 t_1' \ldots t_m'$. That means, whenever we have the equalities $t_i \sim t_i'$ for all $i = 1, \ldots, m$, then we also have the equality $t_1 \sim t_1'$. Intuitively, this comes from the requirement of $D_1$ defining a function to $r$: given equal arguments, the result must be the same.

- If we have enough information such that we know that only one instance declaration for $D$ matches $t_1' \ldots t_m'$, then we can obtain the corresponding value for $r'$. This is called instance improvement.

For example, take an instance constraint of the form `Collection [Int] x` with a yet-unknown `x`. By using improvement with the dependency of the second argument over the first, we can deduce that $x \sim $ `Int` from the instance declaration `Collection [e] e`.

A first approach to encode functional dependencies is to extract the function “inside” a functional dependency as a separate type family. We can always do so because of the definition of functional dependency. For example, the type class `Collection` gives rise to a family:

   type family `CollectionElement c`  
   type instance `CollectionElement [e] = e`

This technique is not new: the associated type family in the `Collection2` example is obtained by this method. We can also see that the `AddF` type family in the introduction is the extraction of the functional dependency of `AddC` as a family.

This approach is not completely satisfactory, though, because the link between the `Collection` type class and its functional dependency is lost if posed as an external function. First of all, it is not ensured that every time the `IsCollection` type family is instantiated, a new rule is also added to `CollectionElement` and that they are compatible, although it is possible to modify the compiler to check this. The second problem is that every time you use the `IsCollection` type family, you would have to mention the `CollectionElement` too, in order to ensure that the dependency is satisfied.

A better solution comes from the introduction of injectivity annotations on type families. At the moment of writing, no Haskell compiler supports these annotations, even though a draft of its design is available for GHC. Syntactically, injectivity annotations are similar to functional dependencies:

   type family $F t_1 \ldots t_m \to r s_1 \ldots s_k :: (\text{result} :: \kappa)$  
   | $\text{result} t_1 \ldots t_m \to r$

Their intuitive meaning is that given the result of the function and types $t_1 \to t_m$, we can obtain a single value of $r$. In the simplest case of an annotation $r \to r$, the description coincides exactly on the function $F$ being injective on the parameter $r$.

Injectivity annotations are exactly what we need for a faithful translation of functional dependencies. For each dependency $t_1 \ldots t_m \to r$ in a type class $C$, we add an annotation $\text{result} t_1 \ldots t_m \to r$ in the translated $\text{IsC}$. In the translation the only possible value for the result of the type family $\text{IsC}$ is `Yes`, and thus the addition of $\text{result}$ in the injectivity annotation does not add any further information.

The `Collection` type class introduced earlier has a functional dependency in its definition. Using the proposed translation, the declaration of the corresponding type family reads as follows:

   type family $\text{IsCollection}\ t\ e :: (\text{result} :: \text{Defined})$  
   | $\text{result} t \to e$

The injectivity constraint acknowledges the fact that when $\text{result} \sim \text{Yes}$ and we know the value of the $t$ parameter, we can infer $e$.

### 3.3 Superclasses

The last missing feature in our simulation is support for superclasses. A general type class definition (omitting functional dependencies) has the form:

   class $C_1 \ s_1, \ldots, C_k \ s_k \Rightarrow D \ t_1 \ldots t_n$

This declaration imposes a restriction over the set of instances of $D$: the types involved in such instances must be instances of $C_1, \ldots, C_k$, too. Then, the type checker can use a constraint $D \ t_1 \ldots t_n$ to deduce any of these superclass constraints.

Note that, in contrast to contexts in instance declarations, superclasses constraints only impose one direction of the implication, not equivalence. For example, the Haskell Prelude includes the following:

   class $\text{Eq}\ a \Rightarrow \text{Ord}\ a$  
   | $\text{instance}\ \text{Eq}\ a \Rightarrow \text{Eq}\ [a]$  

In the first case, knowing that $\text{Ord}\ t$ we can deduce $\text{Eq}\ t$. But from the fact that $\text{Eq}\ t$, we know nothing about its relation with the type

---

4. [https://ghc.haskell.org/trac/ghc/wiki/InjectiveTypeFamilies](https://ghc.haskell.org/trac/ghc/wiki/InjectiveTypeFamilies)

5. At the moment there are only plans to implement injectivity annotations of the form $\text{result} t_1 \ldots t_m r s_1 \ldots s_k$, i.e. the result determines all arguments. The implementation of the more general form is deferred until a compelling use case for it emerges. Our encoding provides such a use case.

6. Note, however, that some of the extensions that we implement in Section 4 do introduce type families that rewrite to `No`.  

class Ord. The second definition is different: from Eq [a] we know that Eq [a] and also the converse, given Eq a, we can construct Eq [a]. This second fact underlies the idea of encoding instances using type families, which relates equivalent types.

Type families in Haskell do not support implication, though, so we need a solution other than type family rewriting. We can derive an appropriate encoding from the observation that, under the common Haskell semantics for type classes, we have that:

\[
D \; t_1 \ldots \; t_m \leftrightarrow (D \; t_1 \ldots \; t_m, C_1 \; t_1 \ldots \; t_m, \ldots, C_k \; t_1 \ldots \; t_m)
\]

When applied to our Ord example, it means that being instance of Ord is equivalent to being instance of both Ord and Eq.

Now, every time we find D t in a context, instead of plainly translating it to IsD t ~ Yes, we also need to add translations of all superclasses. For example,

\[
(\rightarrow) :: \text{Ord} \; a \Rightarrow a \rightarrow a \rightarrow \text{Bool}
\]

is translated to

\[
(\rightarrow) :\ (\text{IsOrd} \; a \sim \text{Yes}, \text{IsEq} \; a \sim \text{Yes}) \Rightarrow a \rightarrow a \rightarrow \text{Bool}
\]

This results in very cumbersome contexts, though. We would like to find a way to automate this addition of superclasses without such a syntactic overhead.

We can achieve this goal by means of the ConstraintKinds extension in the GHC compiler. This extension enables us to make use of a type class or type equality constraint as a type itself, which is assigned the special kind Constraint. For example, we have:

\[
\begin{align*}
\text{Eq} \; a &:: \text{Constraint} \\
\text{Eq} \; a \rightarrow \text{Constraint} \\
\text{IsEq} \; a \sim \text{Yes} &:: \text{Constraint}
\end{align*}
\]

Considering constraints as types means that we can use all the facilities that are available to types when dealing with constraints. In particular, we can introduce type synonyms, like:

```
type Serializable t = (Show t, Read t)
```

The previous synonym can be used in any context that expects a constraint, and expands to the conjunction of being instance of both Show and Read.

The trick is to define a type synonym per type class that encodes both membership to the class itself and to all of its superclasses. In the case of Ord, it reads:

```
type IsOrd t a = (IsOrd a ~ Yes, IsEq a)
```

The IsOrd a ~ Yes constraint is the one taking care of being an instance of Ord itself. Then, for each superclass (in this case, only Eq) we ensure that a is also an instance of those, by adding the corresponding constraints. Note that in the case of IsEq a, this will in turn call to any superclass of that type class, until all direct and indirect superclasses are resolved.

The addition of superclasses forces us to reconsider the translation of instance constraints appearing in type signatures or data type contexts. Whereas before a signature such as

\[
(\rightarrow) :: \text{Ord} \; a \Rightarrow a \rightarrow a \rightarrow \text{Bool}
\]

would be translated to a call to IsOrd:

\[
(\rightarrow) :: \text{IsOrd} \; a \sim \text{Yes} \Rightarrow a \rightarrow a \rightarrow \text{Bool}
\]

now we use the type synonym we have just defined:

\[
(\rightarrow) :: \text{IsOrd} \; a \sim a \rightarrow a \rightarrow \text{Bool}
\]

Note that syntactically this last signature looks very similar to a "real" instance constraint.

---

4. Extending Type Classes Using Type Families

Our discussion up to this point shows that type classes can be simulated in a sound way via a characteristic function on the type-level. This encoding opens the door to simulating some extensions that have been proposed to Haskell type classes to describe more sharply the set of types that are instances of a type class, with the aim of producing better error messages for programmers.

Note that in all these cases, implementations of these extensions using only type classes are also available. Our goal is to present alternative definitions that capture the viewpoint of programming type families as representing type classes. Furthermore, by using our encoding, expressing these extensions require only compiler support for type families.

4.1 Type Class Directives

By the name of type class directives we refer to different techniques that refine the Haskell ad-hoc type polymorphism system by stating additional constraints on the possible instances of a type class, which typically results in better error messages. Both Heeren and Hage [8] and Stuckey and Sulzmann [16] provide examples of such directives. We shall use the syntax of the former throughout this section.

**Non-membership.** The first of these directives is never: as its name suggests, a declaration of the form never Eq (a → b) forbids any instance of Eq for a function type. Given that we translate Eq t to IsEq t ~ Yes (since it has no superclasses), we only need to ensure that IsEq (a → b) does not rewrite to Yes. We can do that easily with the following declaration:

```
type instance IsEq (a → b) = No
```

If we try to use Eq over a function, the compiler will complain:

```
Couldn't match type 'No' with 'Yes'
```

**Expected type:** 'Yes'

**Actual type:** IsEq (t → t)

Furthermore, since rules for a type family may not overlap, this definition also disallows anybody to write an instance for any instantiation of a → b, just as we wanted.

An implementation of never using only type classes was given by Kiselyov et al. [12]. Note however that their implementation relied on not having any instance of a Fail type class: adding one orphan instance would break the invariant. Our implementation does not rely on any invariant imposed over Defined. Alas, in order for the compiler to know that an instance is impossible, the module defining the IsC equation needs to be imported.

**Closed set of instances.** The second directive is close [8] [15], which limits the set of instances for a type class to those that have been defined until that point. In other words, the type class has a restricted number of instances, to which no new ones can be added. In this case, we only need to define a closed type family that rewrites to No for any forbidden instance.

An example of such a scenario is an Integral type class whose only instances are expected to be Int and Integer. Using this formulation, the corresponding type family IsIntegral is defined as follows:

```
type family IsIntegral t where
  IsIntegral Int = Yes
  IsIntegral Integer = Yes
  IsIntegral t = No
```

The closed nature of the type family ensures that no more instances can be added. The last equation in the definition indicates that any type not matching Int or Integer is not part of Integral.

The main difference with the close directive is that we need to define all instances in one place, whereas the directive defines a
point after which no more instances can be added. It is possible to define a source-to-source processor which would rewrite an open type family into a closed one with a fallback default case, which would behave similarly to `close` if applied to those families which simulate type classes.

**Disjointness.** Another directive given by Heeren and Hage [8] is disjoint `C D`, which constrains any instance of `C` not to be instance of `D`, and vice versa. For example, we could forbid a type to be at the same instance of both `Integral` and `Rational`. A naïve encoding of this directive for `Integral` is achieved as follows:

```haskell
type family IsIntegral t where
  IsIntegral t = IsICheckR t (IsRational t)
:: Defined where
  IsICheckR t Yes = No
  IsICheckR t No = IsIntegral t

type family IsRational t where
  IsRational t = IsICheckI t (IsIntegral t)

type family IsICheckR t (isRational :: Defined) :: Defined where
  IsICheckR t Yes = No
  IsICheckR t No = IsRational t

type family IsICheckI t (isIntegral :: Defined) :: Defined where
  IsICheckI t Yes = No
  IsICheckI t No = IsIntegral t

type family IsIntegralOrRational t :: IntegralOrRational
```

The idea is that `IsIntegral`, by calling `IsICheckR`, checks whether a `Rational` instance is present. If not, then it checks whether we have an explicit `Integral` instance, represented by `IsIntegral'`. Thus, for adding new instances, the latter needs to be extended.

```haskell
type instance IsIntegral' Int = Yes

Unfortunately, this naïve encoding does not work, the compiler loops when trying to resolve an instance. For example, `IsIntegral Int` gives rise to the infinite sequence:

```
IsIntegral Int ~ IsCheckR (IsRational Int)
   ~ IsCheckR (IsCheckl (IsIntegral Int))
   ~ ...
```

If instead of type families, we had defined `IsIntegral` and `IsRational` as type synonyms:

```haskell
type IsIntegral t = IsICheckR t (IsRational t)
type IsRational t = IsICheckl t (IsIntegral t)
```

the compiler itself would have detected this cycle in the definition and informed us with an error message similar to:

```
Cycle in type synonym declarations
```

The solution is to define both `IsIntegral` and `IsRational at once`. First of all, we introduce a new promoted data type which shall tell us to which of the classes it belongs to, if any:

```haskell
data IntegralOrRational = Integral | Rational | None
```

This data type is used in the definition of the `IsIntegralOrRational` type family below. In essence, it is like any other `IsC` family, but instead of merely `Yes` or `No`, gives some extra information about the actual instance the type satisfies.

```haskell
type family IsIntegralOrRational t :: IntegralOrRational
```

Examples of instances are:

```haskell
type instance IsIntegralOrRational Int = Integral
type instance IsIntegralOrRational Integer = Integral
type instance IsIntegralOrRational Double = Rational
```

By using this finite kind, types are forced to choose only one option from the set of type classes.

The final step is reworking `IsIntegral` and `IsRational` so that they look at the output of the joint type family. Here we only give the definition for `IsIntegral`; the definition `IsRational` is analogous:

```haskell
type IsIntegral t = IsIntegral' (IsIntegralOrRational t)
type family IsIntegral' what :: Defined where
  IsIntegral' Integral = Yes
  IsIntegral' what = No
```

Note that we have kept the same external interface, so that function signatures still use `IsIntegral t ~ Yes or IsRational t ~ Yes`.

We can go a step further when defining the type synonym to be used in contexts. For the case of `Integral`, this direct translation is:

```haskell
type IsIntegral' t = IsIntegralOrRational t ~ Integral
```

However, we know that `IsIntegral` only rewrites to `Yes` when `IsIntegralOrRational t` rewrites to `Integral`. Thus, we can save one rewriting step by taking:

```haskell
type IsIntegral' t = IsIntegralOrRational t ~ Integral
```

4.2 Intermezzo: Open-Closed Families

An interesting pattern with type families is the combination of open and closed type families to create a type-level function whose domain can be extended, but where some extra magic happens at each specific type. As a running example, let us construct a type family to obtain the spiciness of certain type-level dishes:

```haskell
data Water

data Nacho

data TikkaMasala

data Vindaloo

data SpicinessR = Mild | BitSpicy | VerySpicy

type family Spiciness f :: SpicinessR
```

The family instances for the dishes are straightforward to write:

```haskell
type instance Spiciness Water = Mild

type instance Spiciness TikkaMasala = Mild

type instance Spiciness Nacho = BitSpicy

set instance Spiciness Vindaloo = VerySpicy
```

However, when we have lists of a certain food, we want to behave in a more sophisticated way. In particular, if one is taking a list of dishes that are a bit spicy, the final result would definitely be very spicy. To express this special case, we define the `Spiciness` of a list in terms of an auxiliary type family `SpicinessL`:

```haskell
type instance SpicinessL [a] = SpicinessL (Spiciness a)
type family SpicinessL lst where
  SpicinessL BitSpicy = VerySpicy
  SpicinessL a = a
```

This trick has been used for more mundane purposes, such as creating lenses at the type level [9]. The key point is that the non-overlapping rules for open type families allow us to add new instances for those types for which no one is defined yet. Then, by calling a closed type family at a type instance rule, the behaviour of a particular instance can be refined.

4.3 Instance Chains

Instance chains were introduced by Morris and Jones [13] as an extension to type classes in which to encode certain patterns that would otherwise require overlapping instances. The new features are
**alternation**, that is, allowing different branches in an instance declaration, and explicit failure, which means that negative information about instances can be stated.

One case where overlapping instances are needed in Haskell programming is the definition of the `Show` instance for lists: in this case, a special instance is used for strings, which are represented by the type `Char`.

```
instance Show a ⇒ Show [a] where
  show = ... -- Common case

instance Show [Char] where
  show = ... -- Special case for strings
```

Using instance chains, the exception is handled as part of the instance declaration:

```
instance Show [Char] where
  show = ... -- Special case for strings
else instance Show [a] if Show a where
  show = ... -- Common case
```

When matching on a constraint of the form `Show [a]`, the chain will be checked in order. Thus, if we find out that `a ∼ Char`, then the first case is chosen.

Another feature of instance chains is explicit failure. Let us continue with `Show` as our guiding example. In general, we cannot make an instance for functions `a → b`. However, if the domain of the function supports the `Enum` class, we can give an instance which traverses the entire set of input values. In any other case, we want the system to know that no instance is possible:

```
instance Show (a → b) if (Enum a, Show a, Show b) where
  show = ...
else instance Show (a → b) fails
```

As in the previous case, when matching `Show (a → b)`, the compiler follows the chain in the same order. If the first case does not handle our type, then fails explicitly states that the `Show` instance does not exist.

As we did for type class directives, we can encode these cases using our type family translation as follows, where `And3` is a ternary variant of `And`:

```
type instance IsShow [a] = IsShowList a

type family IsShowList a where
  IsShowList Char = Yes
  IsShowList a = IsShow a


type instance IsShow (a → b) = IsShowFn a b

type family IsShowFn a b
  = And3 (IsEnum a) (IsShow a) (IsShow b)
```

The first thing we notice is that the `Show` instance chain follows the pattern of the open-closed type families: we allow adding new rules for those types not already covered by other rules. The fact that `IsShowList` and `IsShowFn` are closed type families enforces the equations there to be tried in order, as done in instance chains. Those instances without a guard simply resolve to `Yes`, and those failing to `No`. Those instances with guards are translated as any type class instance with context.

The family works nicely given some initial `IsShow` rules for atomic types:

```
type instance IsShow Bool = Yes

type instance IsShow Char = Yes
```

It is interesting to notice what happens if we ask for the information of a type for which we have not explicitly declared an instance, such as `Int`:

```
*Main> :kind! IsShow (Int -> [Char])
IsShow (Int -> [Char]) :: Defined
  = IsShowFn (IsEnum Int) (IsShow Int) 'Yes
```

The rewriting is stuck in the phase of rewriting `IsEnum Int` and `IsShow Int`. Intuitively, we may want the system to instead continue to the next branch, and return `No` as result. However, this poses a threat to the soundness of the system: since the type inference engine is not complete in the presence of type families, it may well be that `IsEnum Int ∼ Yes`, but the proof could not be found. If we decided to continue, and that proof finally exists, then the inference step we made is not correct. For this reason, we forbid taking the next branch until rewriting contradicts the expected results. A similar reasoning holds for the use of apartness to continue with the next branch in closed type families [6].

Essentially, what we do by rewriting instance chains into type families is making explicit the backtracking needed in these cases. In principle, Haskell does not backtrack on type class instances, but by rewriting across several steps, we simulate it. Note that backtracking search can also be simulated using type classes only [11]. Rewriting it as a type family gives a more operational point of view.

### 4.4 Better Error Messages

Until now, the only possibilities for a type family corresponding to a type class were to return `Yes` or `No`, or to get stuck. But this is very uninformative, especially in the case of a negative answer: we know that there is no instance of a certain class, but why is this the case? The solution is to add a field to the `Defined` type to keep failure information.

```
data Defined e = Yes | No e
```

We have decided to keep the error type open, so each type class could have its own way to report errors. A similar approach is taken by Kiselyov et al. [12], whose `Fail` type class is parametrized by an error type which documents the failure.

In the case of a closed type class, it makes sense to have a specific data type as a way to report errors. But in open scenarios, like `IsShow`, we need something more extensible. A good match is the `Symbol` kind, which is the type-level equivalent of strings, and which has special support in GHC for writing type-level literals. Thus, the `IsShow` type family is changed to:

```
import GHC.TypeLits -- defines Symbol

type family IsShow t :: Defined Symbol
```

An example like the function types could benefit from reporting different errors depending on the constraint that failed:

```
type instance IsShow (a → b)
  = IsShowFn (IsEnum a) (IsShow a) (IsShow b)

type family IsShowFn (isEnum :: Defined Symbol)
  (isShowA :: Defined Symbol)
  (isShowB :: Defined Symbol) where
  IsShowFn Yes Yes Yes = Yes
  IsShowFn (No e) a b
  = No "Function with non-enumerable domain"
  IsShowFn e (No a) b
  = No "Source type must be showable"
```

---

3 The kind signatures are needed for these examples to work in GHC. Had we not included them, the compiler would default to the kind `Defined ∗` for the arguments of `IsShowFn`, which is not correct.
We then define the closed type family Result. When the compiler resolves a specific instance of a type class, it checks that the typing is correct, and also generates the corresponding code for the operations in the type class. This second process is called elaboration, and is a key reason for the usefulness of type classes. Type families, on the other hand, only introduce type equalities. Any witnesses of these equalities at the term level are erased.

### 5.1 Elaboration via Type Classes

If we step back for a moment, consider the full Haskell language with type classes and type families, there is a way to elaborate terms depending on family rewriting. This solution has already been pointed out in several places, e.g. by Bahr [1], who uses it to implement a subtyping operator for compositional data types. Let us illustrate this idea with an example: we want to define a function mkConst that creates a constant function with a variable number of arguments. For instance, given the type \( a \rightarrow b \rightarrow \text{Bool} \), we want a function \( \text{mkConst} :: \text{Bool} \rightarrow (a \rightarrow b \rightarrow \text{Bool}) \). To start, we need a type-level function that computes the result type of a curried function type of arbitrary arity:

\[
\text{type family Result f where}
\begin{align*}
\text{Result} (s \rightarrow r) &= \text{Result} r \\
\text{Result} r &= r
\end{align*}
\]

This is the point where, if we could elaborate a function during rewriting, implementing \( \text{mkConst} \) would be quite easy. Instead, we have to define an auxiliary type family that computes the witness of the rewriting of \( \text{Result} \). The first step is to define a promoted data type to encode such witness on the type level.

\[
\text{data ResultWitness} = \text{End} \mid \text{Step } \text{ResultWitness}
\]

We then define the closed type family \( \text{Result'} \), which computes the witness. Note the use of a kind signature to restrict its result to the types defined by promotion of the above data type.

\[
\text{type family Result'} f :: \text{ResultWitness} where
\begin{align*}
\text{Result'} (s \rightarrow m) &= \text{Step} (\text{Result'} m) \\
\text{Result'} r &= \text{End}
\end{align*}
\]

Here comes the trick: we use a type class to elaborate the desired function in terms of the witness. The witness will be supplied via a nullary data constructor \( \text{Proxy} \), which serves the purpose of recording the witness information:

\[
\text{data Proxy a = Proxy}
\]

\[
\begin{align*}
\text{class Result f r where} \\
\text{mkConst :: Proxy w } \rightarrow r \rightarrow f
\end{align*}
\]

Each instance of \( \text{ResultE} \) will correspond to a way in which \( \text{ResultWitness} \) could have been constructed. Note that in the recursive cases, we need to provide a specific type argument using \( \text{Proxy} \):

\[
\begin{align*}
\text{instance ResultE r r End where} \\
\text{mkConst} - r = r
\end{align*}
\]

\[
\begin{align*}
\text{instance ResultE m r l } \Rightarrow \\
\text{ResultE } (s \rightarrow m) r \text{ (Step l) where} \\
\text{mkConst} - r = \lambda s \rightarrow \text{mkConstE } \text{(Proxy} : \text{Proxy} l) \text{ r}
\end{align*}
\]

However, we do not want the user to provide the value of \( \text{Proxy} \) in each case, because we can construct it via the \( \text{Result} \) family. The final touch is thus to create the \( \text{mkConst} \) function, which uses \( \text{mkConstE} \) and provides it with the correct \( \text{Proxy} \):

\[
\begin{align*}
\text{mkConst} :\forall f w .(r \sim \text{Result} f, w \sim \text{Result'} f, \\
\text{ResultE } f r w \Rightarrow r \rightarrow f) \\
\text{mkConst} x = \text{mkConstE } \text{(Proxy} : \text{Proxy} w) x
\end{align*}
\]

The main idea of this trick is to get hold of a witness for the type family rewriting. This is usually implemented by Haskell compilers as a coercion, but the user does not have direct access to it. By reifying it and promoting its constructors to the type-level, we become able to use the normal type class machinery to define elaborated operations.

### 5.2 Elaboration without Type Classes

The encoding in Section 3 is sound from a typing perspective, but does not generate any code. In the previous discussion, we fell back to type classes to perform the elaboration. But if we want to get rid of type classes altogether, we cannot use this trick.

One option is to extend the witnesses approach. This would mean that each type family representing a type class returns a trace of the steps taken by means of a data type. However, this does not work for two reasons:

1. In our translation, we mandate all instances to return the same \text{Yes} result. If that was not the case, we could not declare a constraint such as \text{IsEq} t \sim \text{Yes} that does not depend on the type itself.
2. Support for open type classes would require a notion of open data types, which is not present in Haskell.

For those reasons, we propose the concept of elaboration at rewriting. The idea is that in each rewriting step, the compiler generates a dictionary of values (similar to the one for type classes), which may depend on values from other inner rewritings. The generation of coercions for type families rewriting in GHC can be viewed as an instance of this mechanism, with only a single data type.

The shape of dictionaries must be the same across all type instances of a family. Thus, as with type classes, it makes sense to declare the signature of such a dictionary in the same place within a type family. Without any special preference, we shall use the \text{dictionary} keyword to introduce it. For example, the following declaration adds an \text{eq} function to the \text{IsEq} type family:

\[9\text{We would have preferred the where keyword in consonance with type classes, but this syntax is already used for closed type families.}\]
type family IsEq (t :: *) :: Defined
dictionary eq :: t -> t -> Bool

A type instance declaration should now define a value for each element in the dictionary, as shown below:

type instance IsEq Int = Yes
dictionary eq = primEqInt -- the primitive Int comparison

In the case of calling other type families on its right-hand side, a given instance can access its arguments’ dictionaries to build its own. As concrete syntax, we propose to give a name to a dictionary in the rule itself, or to refer to an element of the dictionary in the construction of the larger one. This idea can be seen in action in the declaration of IsEq for lists:

type instance IsEq [a] = e@(IsEq a)
dictionary eq [] [[]] = True
eq (x : xs) (y : ys) = e@eq x y ∧ eq xs ys
eq _ _ = False

The same syntax can be used to access the dictionary in a function that has an equality constraint. One example of this syntax is the definition of non-equality in terms of the eq operation in the IsEq family:

\[
\text{notEq :: e@((IsEq a) \sim a \Rightarrow a \rightarrow a \rightarrow \text{Bool})}
\]
\[
\text{notEq x y = \neg (e@eq x y)}
\]

We use \textit{e@} prefixes to make clear which dictionary we are using, but it is possible to drop the prefix when there is only one available possibility. Another option is making \textit{eq} a globally visible name, as type classes do.

As we have seen, elaboration at rewriting opens new possibilities for type families. It is also the only piece missing that we cannot directly encode in type families. Section 5.3 shows, though, that for the typing perspective our simulation can be encoded in \textit{current GHC}, with the sole addition of injectivity constraints to deal with functional dependencies.

5.3 Application: Compositional Data Types

Swierstra’s data types à la carte [18] demonstrate an elegant solution to the expression problem, that is, giving easy ways to extend both functions and data types inside of a programming language. Haskell comes with good support for defining new functions, the missing piece is the definition of extensible data types.

The key points of Swierstra’s solution are the definition of a type combinator \texttt{:+:}, which combines constructors from different types, and a relation \texttt{<:: g}, which specifies that the constructors in \texttt{f} are a subset of those in \texttt{g}. The relation \texttt{<::} is defined as a type class, which provides a method to inject one type into the other:

\[
inj :: f \rightarrow g a
\]

This definition is not perfect, though, because it does not handle well combinations of the form \((f :: g) :: h\). In order to solve this problem, Bahr [11] proposes an implementation using closed type classes and a type class for elaboration, as shown in Section 5.1. We find this a perfect scenario for using elaboration over a type family instead of a type class: the code for the \texttt{<::} relation is given in Figure [1]

This subtyping is an example of a relation for which a custom search procedure is useful, instead of the normal instance search. Closed type families have in many cases the power to define them at type level. Elaboration at rewriting allows us to maintain the code to be generated close to the search procedure.

5.4 Local Instances

One key decision in the design space of elaboration for type families is whether programmers may introduce them only in global scope, or also in local scopes. As a running example, let us consider the following data type declaration, in the form of a generalized algebraic data type:

\[
data ShowEverything t where
UsingInst :: IsShow t \sim \text{Yes} \Rightarrow t \rightarrow \text{ShowEverything} t
NoInst :: \rightarrow \text{ShowEverything} t
\]

The idea is that for this data type we can define a \textit{show} function for whatever \textit{t} is given as index, falling back to the actual \textit{Show} instance (defined via a type family) if they support one:

\[
\text{showE :: ShowEverything} t \rightarrow \text{ShowEverything} t
\]
\[
\text{showE} (\text{UsingInst} x) = \text{show} x
\]
\[
\text{showE} (\text{NoInst} x) = \text{""}
\]

Another way to do this is by introducing a new type family instance in the second case:

\[
\text{showE :: ShowEverything} t \rightarrow \text{ShowEverything} t
\]
\[
\text{showE} (\text{UsingInst} x) = \text{UsingInst} x
\]
\[
\text{showE} (\text{NoInst} x) = \text{let type instance IsShow} t = \text{Yes}
dictionary show = \text{""}
in \text{UsingInst} x
\]

Now, when \textit{UsingInstance} is unwrapped, the new instance is readily available for use.

But introducing this kind of local type family instances also introduces problems, especially on the principality of the typing and the elaboration of dictionaries. Many of those problems are discussed by Dijkstra et al. [5]. For those reasons, we prefer an approach similar to type classes, where new dictionaries can be introduced only in the global scope.

6. Formalization

In this paper we have build step by step a translation from type classes into type families. This section specifies the complete algorithm for such translation, and present its most important properties. The reader is referred to Appendix B for the proofs.

6.1 Formal Translation

In this section we look at the formal translation from type classes to type families. There are three constructs to translate: type class declarations, instance declarations and contexts in a type.

The general form of a \textit{type class declaration} declares its name \textit{D} and parameters \(t_1, \ldots, t_m\), along with its superclasses \(C_1, \ldots, C_k\) and a set of functional dependencies:

\[
\text{class} \quad C_1 \, \exists_1, \ldots, C_k \, \exists_k \Rightarrow D \, t_1 \ldots t_n \quad | \quad \exists_1 \rightarrow \forall_1, \ldots, \exists_n \rightarrow \forall_n
\]

where each \(\exists_i, \forall_1, \ldots, \forall_n\) is a sequence of type variables drawn from \(t_1 \ldots t_m\). Each of these class declarations gives rise to a new type family encoded as:

\[
\text{type family IsD} \, t_1 \ldots t_m :: \text{Defined}
\]

Here, \textit{Defined} is the kind which represents whether a type class instance is available. In addition, types \(t_1 \ldots t_m\) may include kind annotations inferred from their use in the elaborated methods.

This type family only represents the type class itself, missing the other two components of the declaration. We take care of \textit{functional dependencies} first, which translate to injectivity declarations for the type family. Note that injective type families are not implemented as the moment of writing in any Haskell compiler, is only a draft of its design. In our case we need the kind of injectivity in which the right-hand side of the equation plus some part of left-hand side
arguments determine some other left-hand side. Using the syntax proposed in the aforementioned draft, the type family declaration needs to be changed to:

```
| type family IsD t₁ ... tₘ = (r :: Defined) |
|    | r t₁ → τ₁,..., r τₙ → τₙ |
```

In short, we have kept the dependencies almost as-is, only with the addition of the extra result parameter.

The way in which we enforce `superclasses` is by defining a type synonym for the conjunction of all those prerequisites along with the instance we are actually looking for. In general, this means defining a synonym. Note that in GHC, we need to enable the `ConstraintKinds` extension to allow this definition:

```
| type IsD⁺ t₁ ... tₘ = (IsD t₁ ... tₘ ~ Yes,
|    | IsC₁ t₁ ~ Yes,..., IsCₙ tₙ ~ Yes) |
```

This type synonym is the one used when translating type class constraints in contexts such as function signatures or data type declarations. For example, a function with specification

```
| f :: D t₁ ... tₘ ⇒ r |
```

is translated using the synonym as

```
| f :: IsD⁺ t₁ ... tₘ ⇒ r |
```

Constraints in the context of instance declarations are translated slightly differently. Each type class instance declaration has a number of type class constraints (to the left of ⇒) and a list of types to which the instance declaration applies (to the right of ⇒):

```
| instance (Q₁,..., Qₙ) ⇒ D t₁ ... tₘ |
```

Each Qᵢ is of the form E s₁ ... sₖ for some type class E. Each of the above type class instance declarations is translated into a type family instance of the following form:

```
| type instance IsD t₁ ... tₘ = Andₙ Q₁' ... Qₙ' |
```

where Qᵢ' = IsE s₁ ... sₖ, given that Qᵢ = E s₁ ... sₖ. Moreover, for each number n of context declarations, we have a corresponding closed type family `Andₙ`, which checks that all its arguments are `Yes`. More explicitly:

```
| type family And₀ :: Defined |
| And₀ = Yes |
```

```
| type family And₁ d :: Defined |
| And₁ x = x |
```

Figure 1. Subtyping for compositional data types

---

6.2 Soundness and Completeness

In order to prove that our translation respects the semantics of type classes, we first need a formalization of Haskell’s type system. We
build upon the OUTSIDEIN(X) framework \[20\], which underlies the GHC compiler from version 7, and which we describe thoroughly in Appendix \[4\]. Note that type class resolution and type family rewriting have been described within this framework, but without support for functional dependencies or injectivity. Appendix \[13\] also includes our formalization of these concepts within OUTSIDEIN(X), based on the description as Constraint Handling Rules by Sulzmann et al. \[7\].

In OUTSIDEIN(X), typing depends on a concrete entailment judgment \( Q \vdash Q \), which symbolizes that under axioms \( Q \), the constraint \( Q \) is satisfiable. The shape of constraints depends on the specific domain \( X \): when \( X \) is the domain of Haskell type classes and type families, those constraints are either type equality \( \tau_1 \sim \tau_2 \) or instances \( D \tau_1 \ldots \tau_n \). Our theorems are true with respect to that concrete entailment.

**Theorem 1 (Soundness).** Let \( T \) be the derivation tree of \( Q^\text{trans} \vdash \tau_1 \sim \tau_2 \). Suppose that for each application of injectivity over a type family \( IsD \tau \) there exists a subtree that proves either \( IsD \tau \sim \text{Yes} \) or \( IsD \tau \sim \text{No} \). Then:

- If \( \tau_1 \) and \( \tau_2 \) are not of kind Defined, then \( Q \vdash \tau_1 \sim \tau_2 \).
- If \( \tau_1 \equiv IsD \tau \) and \( \tau_2 \equiv \text{Yes or vice versa} \), then \( Q \vdash D \tau \).

\[ \Box \]

Proof. See Appendix \[13\].

This soundness result states that whenever we can derive \( IsD \tau \sim \text{Yes} \) in our translation, then we can derive \( D \tau \) in the original system based on type classes. Additionally, the soundness result also guarantees that the translation does not introduce any additional type equalities for types outside the Defined kind.

Note, however, that in the presence of functional dependencies (which are translated to injectivity declarations), this soundness result is subject to a side condition. Informally speaking, this side condition means that injectivity is only used when there is positive evidence in the form of an equality \( IsD \tau \sim \text{Yes} \). That means, the above soundness result does not cover the case where we combine functional dependencies of type classes with the extensions described in Section \[4\], since the latter do introduce equations containing \( \text{No} \).

**Theorem 2 (Completeness).**

- If \( Q \vdash \tau_1 \sim \tau_2 \), then \( Q^\text{trans} \vdash \tau_1 \sim \tau_2 \).
- If \( Q \vdash D \tau \), then \( Q^\text{trans} \vdash IsD \tau \sim \text{Yes} \).

\[ \Box \]

Proof. See Appendix \[13\].

The completeness result is straightforward: our translation preserves all original type equalities, and any instance \( D \tau \) that is derivable in the original system, is derivable as \( IsD \tau \sim \text{Yes} \) in the translation.

### 6.3 Termination

An important issue to consider is whether the termination characteristics of class instances are also carried over to the translated families. The Haskell Report defines strict conditions that guarantee termination. However, GHC imposes more lenient ones known as the Paterson conditions \[14\], which will serve as basis to prove termination in our setting. The Paterson conditions state that for each constraint \( Q s_1 \ldots s_j \) in the instance context:

1. No type variable has more occurrences in the constraint than in the instance head.

2. The constraint has fewer constructors and variables (taken together, and counting repetitions) than the head.

GHC imposes similar termination conditions for type families \( F \tau_1 \ldots \tau_m = \tau \). In this case, the conditions require that for each type family application \( G \tau_1 \ldots \tau_n \) appearing in \( \tau \), we have:

1. None of the arguments \( \tau_1 \ldots \tau_n \) contains any other type family applications.
2. The total number of data type constructors and variables in \( \tau_1 \ldots \tau_n \) is strictly smaller than in \( \tau_1 \ldots \tau_m \).
3. Each variable occurs in \( \tau_1 \ldots \tau_n \) at most as often as in \( \tau_1 \ldots \tau_m \).

The translation of a class instance declaration that satisfies the Paterson conditions into a type family instance declaration

\[
\text{type instance } IsD t_1 \ldots t_m = \text{And}, Q'_1 \ldots Q'_n
\]

satisfies the terminations conditions (2) and (3) of type families. However, condition (1) is not satisfied, because each \( Q'_i \) is a type family application. Note that these are the only nested applications generated by the translation.

The key point in establishing termination in this setting is observing that each application of \( \text{And} \) adds just one extra rewriting step. If type families fulfill their termination conditions (2) and (3), \( \text{And} \) just adds a number of steps bounded by the size of the derivation tree. Thus, termination is still guaranteed.

### 7. Comparison

#### 7.1 Type Families as Functional Dependencies

In this paper we have looked at type families as an integrating framework for both families and classes. In contrast, previous literature \[15\] has considered type classes with functional dependencies as the integrating glue: why is our choice any better?

The answer lies in the use of instance improvement by functional dependencies, as discussed in \[32\]. This type of improvement makes type inference brittle: it depends on the compiler proving that only one instance is available for some case, which can be influenced by the addition of another, not related, instance for a class.

Other different problems with functional dependencies have been discussed in \[3, 15\], usually concluding that type-level functions are a better option. In this paper we agree with that statement, and we show that families could replace even more features of type classes by using other Haskell extensions such as data type promotion and closed type functions.

#### 7.2 Implicit Arguments

In essence, in Section \[25\] we are describing a new way to deal with type-level programming which needs to decide whether a certain proposition holds while elaborating some piece of code. This comes close to the instance arguments feature found in Agda \[3\], which was also proposed to simulate type classes. Any argument marked as such in a function with double braces, like:

\[
\text{myFunction} : \{ A : \text{Set} \} \rightarrow \{ \{ p : \text{Show} \ A \} \} \rightarrow A \rightarrow \text{String}
\]

will be replaced by any value of the corresponding type in the environment in which it was called. Thus, if \( \text{Show} \) is thought of as a class, an instance can be provided by constructing such a value:

\[
\text{showInt} : \text{Show} \ \text{Int}
\]

\[
\text{showInt} \ x = \ldots \quad -- \text{code for printing an integer}
\]

Since these values are constructed at the term level, we can use any construct available for defining functions. In that sense, it is close to our use of type families, with the exception that in Haskell type-level and term-level programming are completely separated. A difference between both systems is that Agda does not do any proof

---

\[13\] Unless the user turns on the UndecidableInstances extension, which turns off any termination checking.
search when looking for instance arguments, whereas our solution can simulate search with backtracking.

7.3 Tactics

The dependently type language Idris [2] generalizes the idea of Agda’s instance arguments allowing the programmer to customize the search strategy for implicit arguments. Similarly to Coq, Idris has a tactic language to customize proof search. Unlike Coq, however, Idris allows the programmer to use the same machinery to customize the search for implicit arguments [19].

For example we can write a function of the following type, where \( t \) is a tactic script that is used for searching the implicit argument of type \( \text{Show}\ a \):

\[
 f : \{ \text{default tactics } \{ t \} \ p : \text{Show} \ a \} \to a \to \text{String}
\]

The tactic \( t \) itself is typically written using reflection such that it can inspect the goal type -- in this case \( \text{Show} \ a \) -- and perform the search accordingly:

\[
 f : \{ \text{default tactics } \{ \text{applyTactic findShow; solve} \} \ p : \text{Show} \ a \} \to a \to \text{String}
\]

The search strategy is defined by \( \text{findShow} \), which is an Idris function that takes the goal type and the context as arguments and produces a tactic to construct a term of the goal type.

This setup is similar to closed type families with elaboration as presented in this paper. However, \( \text{findShow} \) has to operate on terms of Idris core type theory \( TT \), which is quite cumbersome. Moreover, there is no corresponding setup for open type families.

8. Conclusion

The relationship between type classes and type families is similar to that between subsets and functions. On the one hand, functions can be represented as subsets satisfying certain conditions: this is the point of view we take when describing families using functional dependencies. On the other hand, we can represent subsets via their characteristic function: this is the point of view we advocate.

By creating type families which simulate classes, we are able to incorporate algebraic types such as type class directives and instance chains. In general, we gain control over the search procedure.

Programmers can readily incorporate elaboration into type family rewriting by using witnesses, as discussed in this paper. This can help to bridge the gap when a custom search procedure is needed to define instances of a type class.

Acknowledgments

We want to thank Gabor Greif, the attendants to IFL 2014 in Boston and the Haskell Symposium reviewers for their helpful comments. This work was supported by the Netherlands Organisation for Scientific Research (NWO) project “DOMain Specific Type Error Diagnosis (DOMSTED)” (612.001.213) and by the Danish Council for Independent Research, Grant 12-132365, “Efficient Programming Language Development and Evolution through Modularity”.

References

A. OUTSIDEIN(X)

The current reference for type inference for Haskell, including type classes, type families and other extensions such as generalized algebraic data types, is described by Vytiniotis et al. [20]. The authors describe the inference process in terms of a general framework, called OUTSIDEIN(X), which is parametrized by a constraint system X. Each such constraint system defines a concrete entailment judgement $Q \vdash Q$, which gives semantics to a constraint $Q$ under the axioms in the set $Q$. Axioms are declarations such as class and family instances.

In particular, we are interested in the case where $X$ = “type classes and type families”, given in Figure 2 and extensively discussed by Vytiniotis et al. [20]. The first rules deal with reflexivity, symmetry and transitivity of the type equality relation.

Rules COMP, FCOMP and DICTEQ deal with the substitution of equal types under type constructors, type families and type classes, respectively. In addition, when we have two types headed by the same type constructor, we can split the equality into equality of their arguments using rule DECOMP. This rule holds because type constructors are injective.

The last rule describes the application of axioms: if we can prove the preconditions of an axiom for a specific substitution $[\tau_1 \mapsto \tau_2]$, then we can conclude the postcondition in the axiom. Note that in the case of type family instances, $Q_1$ is always empty, so the rule in that case reads:

$$\forall \exists F. p \sim q \in Q \\
F [\tau_1 \mapsto \tau_2] p \sim [\tau_1 \mapsto \tau_2] q$$

The rules in the original paper [20] did not take into account functional dependencies or improvement for type families, nor the use of superclasses. The latter problem can be solved by adding new axioms of the form $D \Rightarrow C_1, ..., C_i$ for each class declaration $class C_1, ..., C_i \Rightarrow D$.

In order to account for functional dependencies in type classes, we add two new rules, given in Figure 3. Those rules correspond to what it is known in the literature as functional dependency and instance improvement, respectively. Type family injectivity has a similar set of rules, which deal with the generalization of the case in which $F a \sim F b \implies a \sim b$, and the case in which we can make improvement because only one rule matches. We suppose that the correctness of instance respect of functional dependencies and injectivity annotation has been checked previously by the compiler.

B. Proofs

Now that we have a formal specification of the type system, we can prove the soundness and completeness of our translation from type classes to type families. In the following, let $Q^{trans}$ be obtained by applying the translation from type classes to type families to a set of axioms $Q$. Furthermore, assume that $Q$ does not contain any prior type families operating on types of kind Defined apart from the And$_a$ type families, nor any type constructor taking arguments of kind Defined.

In order to prove the main results, we make use of two auxiliary lemmas. The first lemma ensures that when we have a derivation of a conjunction of instance constraints translated using And$_a$, we also have a derivation for each of the translated instance constraints.

**Lemma 1.** $Q^{trans} \models \bigwedge D_i \tau_i \sim Yes$ if and only if $Q^{trans} \models \text{And}_a D_i \tau_i \sim Yes$.

**Proof.** By case analysis of the definition of And$_a$. ☐

The second lemma ensures that any of our type families IsD ultimately rewrites to Yes or gets stuck in some family application.

**Lemma 2.** If $Q^{trans} \models IsD \tau \sim \kappa$ with types of kind Defined, and where type $\kappa$ involves no type family application, then $\kappa \equiv Yes$.

**Proof.** The definitions of all IsD and And$_a$ type families only call other type families of kind Defined or end up in the type Yes. ☐

Note that this formal translation only applies to Haskell type classes without any extensions such as directives or instance chains, in which case a third option of rewriting to No might be available.

Using these two lemmas, we can prove that the translation preserves the semantics of type classes as given by OUTSIDEIN’s concrete entailment $\models$.

**Theorem 1 (Soundness).** Let $T$ be the derivation tree of $Q^{trans} \models \tau_1 \sim \tau_2$. Suppose that for every application of the rule REMIND over type families IsD $\tau$ there exists a subtree which proves either IsD $\tau \sim Yes$ or IsD $\tau \sim \tau$. Then:

- If $\tau_1$ and $\tau_2$ are not of kind Defined, then $Q \models \tau_1 \sim \tau_2$.
- If $\tau_1 \equiv IsD \tau$ and $\tau_2 \equiv Yes$ or vice versa, then $Q \models \tau \sim \tau$.

**Proof.** Let $r$ be the number of rule applications in $T$ and $n$ the number of occurrences of types of the form And$_a \xi$ and IsE $\pi$ to the right of $\models$ in $T$ (not counting nested occurrences). We proceed by induction on $r + n$, and consider the two possibilities in the theorem.

**Case $\tau_1$, $\tau_2$ not of kind Defined**

By inversion on the last rule applied, we need to consider the following cases:

**Case REFPL.** The same tree proves the equality in $Q$.

**Case SYM.** Apply the induction hypothesis on the antecedent, which has one rule application less, to get a derivation tree $T'$ for $Q \models \tau_2 \sim \tau_1$. Then, apply rule SYM to $T'$ to get a proof of $\models \tau_1 \sim \tau_2$, as desired.

**Case TRANS.** Given a derivation tree of the shape

$$Q^{trans} \models \tau_1 \sim \alpha \quad Q^{trans} \models \alpha \sim \tau_2$$

by kind checking we know that $\alpha$ is not of kind Defined. Thus, we can apply the induction hypothesis on each of the antecedents, whose number of rule application is fewer, to get derivation trees $T_1$ and $T_2$ over $Q$. Then, apply rule TRANS to $T_1$ and $T_2$.

**Cases DECOMP and COMP.** These cases deal with decomposing and taking together arguments of a type constructor. For example, if we have $[a] \sim [b]$, then we can derive $a \sim b$ and vice versa.

We further assumed that type constructors coming from $Q$ must not have arguments of kind Defined. The translation adds no further type constructors, so it must be the case that the arguments of $\tau_1$ and $\tau_2$ are not of kind Defined. Therefore, we can apply the induction hypothesis on the antecedents to get new derivation trees, and then apply the rule in the scenario in which $Q$ is the set of axioms in place.

**Case FCOMP.** Similar to the case COMP above. Note that we have assumed that any type family different from And$_a$ operates on types of kind different from Defined, which allows us to apply the induction hypothesis.
Case **AXIOM.** Given a derivation tree of the shape

$$Q \vdash \tau \leadsto \tau$$

we know that the axiom $\forall \exists. \sigma_1 \sim \sigma_2 \in Q^{trans}$ always involve types of kind Defined, either via a family of the form IsD or an instance of the And, family. Therefore, we can keep the derivation unaltered, and it also applies when replacing $Q^{trans}$ with $Q$.

Case **FAMINJ.** There are two cases, depending on the kind of the type family involved in the injectivity:

- **The kind is different from Defined:** in this case the declaration of the type family comes unaltered from the set of axioms $Q$. Thus, we can obtain new derivation trees for each of the antecedents using the induction hypothesis, and then apply the FAMINJ rule back again.

- **The kind is Defined:** a first attempt could be to try to apply the induction hypothesis somehow and then use the FD rule. However, it is not always guaranteed that we can convert $IsD \tau \leadsto IsD \tau'$ to a form corresponding to an actual type class derivation. For that reason, we need the extra hypothesis in our theorem, which might have seemed a bit odd at first sight. By taking this hypothesis into account, we know that there exist subtrees $T_1$ and $T_2$ of $T$ which prove that $IsD \tau \leadsto Yes$ and $IsD \tau' \leadsto Yes$, or its mirror type equalities. Given that they are subtrees, we know that the number of rule applications is smaller, and we can readily apply the induction hypothesis to get trees $T_1$ and $T_2$ which prove that $Q \vdash D \tau$ and $Q \vdash D \tau'$. Finally, we can apply the rule FD, since we know that the injectivity condition on $IsD$ comes from a functional dependency on $D$.

Case **FAMIMPROV.** It must be split as in the previous case:

- **The kind is different from Defined:** in this case the declaration of the type family comes unaltered from the set of axioms $Q$. Thus, we can obtain new derivation trees for each of the antecedents using the induction hypothesis, and then apply the FAMIMPROV rule back again.

- **The kind is Defined:** the rule FAMIMPROV demands that the right-hand side of the chosen rule be family-free. By Lemma[2] we know that this implies that $\kappa' \equiv Yes$. Thus, we can apply the induction hypothesis to get a derivation tree of $Q \vdash D \tau$. Now, apply the rule INSTIMPROV using this tree and the corresponding rule from $Q$ as antecedents.
Case $Q^{trans} \vdash IsD \top \sim Yes$ or $Q^{trans} \vdash Yes \sim IsD \top$

By inversion on the last rule applied, we need to consider the following cases:

Case SYM. Apply the induction hypothesis to the antecedent of the rule, which has one fewer rule application and also has one fewer occurrence of $IsD \top$.

Case AXIOM. In this case, we know that the axiom being applied must be of the form $\forall x.IsD \top \sim Yes$ or $\forall x.Yes \sim IsD \top$. The second possibility is ruled out because $Q^{trans}$ does not contain such axioms. There are still two other possibilities:

- The axiom from $Q^{trans}$ corresponds to an axiom $\forall x.D \top \in Q$ coming from an instance declaration. Then, we can build a derivation tree for $Q \vdash D \top$ by using this axiom.

- The axiom comes from the expansion of a synonym $IsC \top \top$ into its supporting family and its superclasses:

  $IsC \top \top \equiv (IsC \top \top \equiv Yes, \ldots, IsD \top \top \sim Yes, \ldots)$

In this case, by recreating the synonym expansion, we can create a linear derivation tree with AXIOM rules corresponding to superclass axioms.

Case TRANS. Given a derivation tree of the shape

\[
\frac{Q^{trans} \vdash IsD \top \sim \alpha \quad Q^{trans} \vdash \alpha \sim Yes}{Q^{trans} \vdash IsD \top \sim Yes}
\]

we know that $\alpha$ must be of kind Defined. Given our restrictions on the type constructors and families in $Q$, there are only a few cases to consider:

- $\alpha \equiv Yes$: in this case, we have as left argument a derivation tree for $Q^{trans} \vdash IsD \top \sim Yes$. This tree has three occurrences fewer of the type family $IsD$, and thus we can apply induction to get a proof of $Q \vdash D \top$.

- $\alpha \equiv No$: this case is not possible. By the soundness of $\vdash$, there is no way to obtain $Q^{trans} \vdash No \sim Yes$.

- $\alpha \equiv IsC \top \top$: this case branches in four different cases, depending on the shape of the left tree.

The first possibility is depicted at the top of Figure [4]. Note that in this case $IsC \top \top \equiv IsD$. By the construction of the $IsD$ family, we know that the kinds of $\gamma$ and $\rho$ involved in the tree must have a kind different from Defined. Thus, we can apply the induction hypothesis to get derivation trees $T_i'$ that work over the original $Q$.

The right argument to the TRANS rule has three occurrences fewer of the type family $IsD$. Thus, we can apply the induction hypothesis to get a new tree $T_i$ which proves $Q \vdash D \top$. Finally, the derivation tree that we are looking for is given at the bottom of Figure [4].

If the left argument uses the REFLEX rule, that means that on the right argument we already have a proof of $Q^{trans} \vdash IsD \top \sim Yes$, with one fewer type family application $IsD \top$. Thus, we can apply the induction hypothesis to get a proof of $Q \vdash D \top$.

The third case involves the use of another TRANS rule in the left argument, as given at the top of Figure [5]. In this case, we build a reshaped derivation tree, given underneath. This new derivation tree is a valid proof of $Q^{trans} \vdash IsD \top \sim Yes$. Furthermore, we have reduced the number of occurrences of types of the form $And_{\alpha} \xi$ or $IsE \top$ (even if $\beta \equiv IsE \top$ or $\beta \equiv And_{\alpha} \xi$). Thus, we can apply induction over this new derivation tree to get a proof of $Q \vdash D \top$.

The last case is given at the top of Figure [6] and involves the usage of SYM in the left argument. As in the previous case, we reshape the tree in order to obtain one with a smaller number of applications of a type family $IsE \top$ for some $E$. Note that in this case we rebuild from a proof of $Q^{trans} \vdash IsD \top \sim Yes$ to one of $Q^{trans} \vdash Yes \sim IsD \top$. In any case, the induction hypothesis applies and we can get a proof of $Q \vdash D \top$.

- $\alpha \equiv And_{\alpha} \top \top$; as in the previous case, we have further branching depending on the rule that is applied in the left. The cases for REFLEX, TRANS and SYM are worked out in the same way as above.

The case given in Figure [7] corresponds to the application of an axiom. We know an axiom of the form

\[\forall x. IsD \top \sim And_{\alpha} IsQ \top \top\]

in $Q^{trans}$ must come from a corresponding axiom in $Q$:

\[\forall x. Q \top \top \Rightarrow D \top \]

In addition, by Lemma [1] it must be the case that we have derivation trees $T_i'$ proving in each case that

\[Q^{trans} \vdash IsQ \top \top \sim Yes\]

We apply the induction hypothesis to those derivation trees to obtain proofs $T_i''$ for $Q \vdash Q \top \top$. Then, just apply the AXIOM rule to those constraints, as shown at the bottom of Figure [7].

In this last case we need to treat $And_{\alpha}$ differently from $And_{\alpha}$ for $n < 1$, since Lemma [1] does not apply in such a situation. However, in this case we can completely forget about the right argument whose consequent is $Q^{trans} \vdash And_{\alpha} \top \top \sim Yes$, since we can get the proof we want directly from the left argument.

The argument for the TRANS case in which the consequent is of the form $Yes \sim IsD \top$ is symmetric to the argument above: simply swap the roles of the left and right antecedent of the TRANS rule.

**Theorem 2** (Completeness).

- If $Q \vdash T_1 \sim T_2$, then $Q^{trans} \vdash T_1 \sim T_2$.
- If $Q \vdash D \top \top \sim IsD \top \top \sim Yes$, then $Q^{trans} \vdash IsD \top \top \sim Yes$.

**Proof.** By induction on the derivation tree $T$ for $Q \vdash T_1 \sim T_2$.

Case $Q \vdash T_1 \sim T_2$

Most of the cases transfer as-is to $Q^{trans}$. The exceptions are those related to functional dependencies.

Case FD. Suppose this rule is applied to constraints $D \top \top \sim D \top'$. By the induction hypothesis we can obtain trees $T_1$ and $T_2$ corresponding to $Q^{trans} \vdash IsD \top \top \sim Yes$ and $Q^{trans} \vdash IsD \top \top \sim Yes$.

Then, apply rule SYM to $T_2$ to obtain a derivation of $Q^{trans} \vdash IsD \top \top \sim Yes$ and then TRANS to obtain $Q^{trans} \vdash IsD \top \top \sim IsD \top'$. It is now possible to apply rule FAMJ11 to obtain the desired conclusion.

Case INSTIMP. By induction hypothesis we can get a derivation tree of $Q^{trans} \vdash IsD \top \top \sim Yes$. Then, apply the rule FAMJMR using the injectivity condition imposed on $IsD$ by the translation.

Case $Q \vdash D \top$.

There are only two relevant cases, DICTEQ and AXIOM.
Figure 4. Derivation trees for case TRANS/FCOMP

Figure 5. Derivation trees for case TRANS/TRANS

Figure 6. Derivation trees for case TRANS/SYM
Case DICTEQ. We have to show that $Q^{\text{trans}} \models IsD \bar{\pi} \sim Yes$ given that $Q \models \bigwedge \bar{\pi}_1 \sim \bar{\pi}_2$ and $Q \models D \bar{\pi}_1$. By the induction hypothesis we have that $Q^{\text{trans}} \models IsD \bar{\pi} \sim Yes$ and by the induction hypothesis applied repeatedly to $Q \models \bigwedge \bar{\pi}_1 \sim \bar{\pi}_2$ we have that $Q^{\text{trans}} \models \bigwedge \bar{\pi}_1 \sim \bar{\pi}_2$. Thus we may derive $Q^{\text{trans}} \models IsD \bar{\pi} \sim Yes$ as follows:

$$Q^{\text{trans}} \models \bigwedge \bar{\pi}_1 \sim \bar{\pi}_2$$

$$Q^{\text{trans}} \models IsD \bar{\pi}_1 \sim IsD \bar{\pi}_2$$

$$Q^{\text{trans}} \models IsD \bar{\pi} \sim IsD \bar{\pi}_1$$

$$Q^{\text{trans}} \models IsD \bar{\pi} \sim Yes$$

Case AXIOM. We need to consider two cases, depending on whether the axiom being applied comes from an instance declaration or superclasses in a class declaration.

- From an instance: we need to show that $Q^{\text{trans}} \models IsD \bar{\pi} \sim Yes$, given that $\forall \bar{\pi} \bigwedge Q \bar{\pi} \Rightarrow D \bar{\pi} \in Q$ and $Q \models \bigwedge Q \bar{\pi} \Rightarrow [\bar{\tau} \mapsto \bar{\sigma}] \bar{\pi}$. Note that instance axioms of the form $\bigwedge Q \bar{\pi} \Rightarrow D \bar{\pi}$ are translated into type family axioms of the form $IsD \bar{\pi} \sim And_\bar{\pi} Q' \bar{\pi}$. Thus, we have that $\forall \bar{\pi} IsD \bar{\pi} \sim And_\bar{\pi} Q' \bar{\pi} \in Q^{\text{trans}}$. Using the AXIOM rule we can thus derive that

$$Q^{\text{trans}} \models IsD \bar{\tau} \sim And_\bar{\pi} IsQ [\bar{\tau} \mapsto \bar{\sigma}] \bar{\pi}$$  \hspace{1cm} (1)

In the case $n = 0$, (1) has the specific form

$$Q^{\text{trans}} \models IsD \bar{\tau} \sim And_\bar{\pi} IsQ [\bar{\tau} \mapsto \bar{\sigma}] \bar{\pi}$$

By the definition of And$_0$, it is also the case that $Q^{\text{trans}} \models And_0 \sim Yes$. Using TRANS, we can combine those proofs to obtain a derivation of $Q^{\text{trans}} \models IsD \bar{\tau} \sim Yes$.

Now, for the case $n \geq 1$, by induction hypothesis we have that

$$Q^{\text{trans}} \models \bigwedge IsQ [\bar{\tau} \mapsto \bar{\sigma}] \bar{\pi} \sim Yes,$$

which by Lemma implies that

$$Q^{\text{trans}} \models And_\bar{\pi} IsQ [\bar{\tau} \mapsto \bar{\sigma}] \bar{\pi} \sim Yes$$  \hspace{1cm} (2)

Using TRANS, we can then combine (1) and (2) to conclude that

$$Q^{\text{trans}} \models IsD \bar{\tau} \sim Yes.$$