Type Families and Elaboration

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 to 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, now as rewrite rules, but they lack one important feature of type classes: elaboration, that is, generating code from the derivation of a rewriting. This paper looks at the interplay of type classes and type families, how to deal with shortcomings in both of them, and discusses further relations on the assumption that type families support elaboration.

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, and are widely used and studied (Peyton Jones et al. 1997). The initial aim was to support ad-hoc polymorphism (Wadler and Blott 1989): a type *class* gives a name to a set of operations along with their types; subsequently, a type may become an *instance* of such 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 the instance for lists which illustrate these features in action:

```
class Show a where
  show :: a → String
instance Show a ⇒ Show [a] where
  show lst = "[" ++ intersperse ',' (map show lst) ++ "]"
```

For each call to an operation such as *show*, the compiler must resolve what code corresponds to that call. Note that the search is needed to find the correct code: above, *show* for type $[a]$ depends on the code for type a. The search and combination of code performed by the compiler is called *elaboration*.

We remark at this point that we consider type classes *without* support for *overlapping instances*. Overlapping instances are used to override an instance declaration in a more specific scenario. The best example is *Show* for strings, which are represented in Haskell as $[Char]$, and for which we usually want a different way to print them:

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

Overlapping instances make reasoning about programs more difficult, since the resolution of instances may change by later overlapping declarations. Furthermore, their common usage patterns can be express by using type families as shown in Section 4.

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

```
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 elements, it does not make sense to have more than one instance per collection type. Such constraints can be expressed using functional dependencies (Jones 2000), a concept borrowed from database theory:

```
class 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, you can only have one e.

Using functional dependencies, *functions* can also be defined at the level of types. Since their inception, functional dependencies have been abused in that way, and it is now common folklore how to do it: given a type level function of n parameters you want to encode, define a type class with an extra parameter (the result) and include a dependency of it on the rest. Each instance will then define a rule in the function. Here is the archetypical *Add* function defined as a type class:[1]

```
data Zero
data Succ a
```

[1] Note that this example needs the UndecidableInstances extension to work in GHC.

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

Type families (Schrijvers et al. 2007) 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 its return kind) and the rules for the function are stated in a series of **type instance** declarations. The *Add* function now becomes:

```
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.

However, when thinking in terms of functions, we are not used to wear our open-world hat. In a case like *Add*, we would want to define a complete function, with a restricted domain. Eisenberg et al. (2014) introduced *closed* type families to bridge this gap. Closed families are matched in order, each rule is only tried when the previous one is assured never to match. Thus, overlapping between rules is not a problem. On the other hand, these families cannot be extended in a different declaration. In GHC, closed type families are introduced using the following syntax:

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

As an aside, families can be *associated* with a type class. In that way, for each class instance you need to define also a set of types local to such instance. The *Collection* class is a good candidate to be given an associated type, namely the type of elements:

```
class Collection2 c where
  type Element c
  empty2 :: c
  add2   :: Element c → c → c
instance Collection2 [a] where
  type Element [a] = a
  empty2 = []
  add2   = (:)
```

The discussion above illustrates that type classes and type families have a lot of things in common, and in many cases choosing one over the other for a task is a matter of convenience or style. In other cases, though, their features differ. The following table summarizes the similarities and differences between classes and families:

	type classes	type families
open	✓	✓
closed		✓
elaboration	✓	
context	✓	

The goal of this paper is to discuss whether it is possible to bridge the gap, and bring type classes and type families even closer in terms of functionality (Sections 2 and 3). Most of the techniques presented in the those sections are folklore or have been used as part of a larger technique, but we expect to show the tight connection between them by focusing only in the tricks without a larger problem behind it. We have already seen how to simulate type families with functional dependencies.

Our *main contribution*, discussed in Section 4, is dealing with the opposite situation: *using type families to express type classes*. We shall see that a key ingredient for making type families as powerful as type classes is to equip type families with an *elaboration* mechanism. This extension does not only level the power of type classes and families, but yields new use cases that are impossible or difficult to express in terms of type classes.

2. Shortcomings of type families

Type families are usually described as a rewriting mechanism at the level of types. By writing family instances, the compiler is able to apply equalities between types to simplify them. As discussed above, the main distinguishing feature of type families is their support for closed definitions. At first sight, they lack the useful feature of elaboration, and also the ability to depend on contexts; here we show that we can simulate both of these aspects.

2.1 Elaboration

When the compiler resolves a specific instance of a type class, it checks that typing is correct, and also generates the corresponding code for the operations in the class. This second process is called *elaboration*, and is the main 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. Is it possible, however, to trick the compiler into elaborating a term from a family application?

The solution has already been pointed out in several places, e.g. by Bahr (2014), 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 $a \to b \to Bool$, we want a function $mkConst :: Bool \to (a \to b \to Bool)$.

To start, we need a type-level function which returns the result type of a curried function type of arbitrary arity:

```
type family Result f where
  Result (s → r) = Result r
  Result r       = r
```

This is the point where, if we could elaborate a function during rewriting, deriving our *mkConst* would be quite easy. Instead, we have to define an auxiliary type family that computes the *witness* of the rewriting of *Result*. The first step is creating a data type to encode such witness. By using data type promotion (Yorgey et al. 2012) we can move a common data type "one level up" such that its constructors are turned into types, and the type itself is turned into a kind. [2]

```
data ResultWitness = End | Step ResultWitness
```

We then define the closed type family *Result′*, which is responsible for computing the witness. Note the use of a kind signature to restrict its result to the types promoted before.

```
type family Result' f :: ResultWitness where
  Result' (s → m) = Step (Result' m)
  Result' r       = End
```

Here comes the trick: using a *type class* that elaborates the desired function in terms of the witness. The witness will be supplied via a zero-data constructor *Proxy*, which serves the purpose of recording the witness information:

```
data Proxy a = Proxy
class ResultE f r (w :: ResultWitness) where
  mkConstE :: Proxy w → r → f
```

[2] In GHC, this behavior is enabled by the DataKinds extension.

Each instance of *ResultE* will correspond to a way in which *ResultWitness* could have been constructed. Note that in the recurring cases, we need to provide a specific type argument using *Proxy*:

$$\textbf{instance } \mathit{ResultE}\ r\ r\ \mathit{End}\ \textbf{where}\\ \quad \mathit{mkConstE}\ \_\ r = r$$

$$\textbf{instance } \mathit{ResultE}\ m\ r\ l \Rightarrow \mathit{ResultE}\ (s \to m)\ r\ (\mathit{Step}\ l)\ \textbf{where}\\ \quad \mathit{mkConstE}\ \_\ r = \lambda(x :: s) \to \mathit{mkConstE}\ (\mathit{Proxy} :: \mathit{Proxy}\ l)\ r$$

However, we do not want the user to provide the value of *Proxy w* in each case, because we can construct it via the *Result′* type family. The final touch is thus to create the *mkConst* function which uses *mkConstElab* by providing the correct *Proxy*:

$$\mathit{mkConst} :: \forall\ f\ r\ w.(r \sim \mathit{Result}\ f, w \sim \mathit{Result}'\ f,\\ \qquad \mathit{ResultE}\ f\ r\ w) \Rightarrow r \to f\\ \mathit{mkConst}\ x = \mathit{mkConstE}\ (\mathit{Proxy} :: \mathit{Proxy}\ w)\ x$$

The main idea of this trick is to get hold of a *witness* for the type family rewriting. This is usually produced 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.

2.2 Context

Within Haskell, instances may depend on a certain context being available (for example, *Show* $[a]$ holds if and only if [3] *Show* a), whereas rewriting via type families does not allow any preconditions. But once again, we can encode it with a bit more work, assuming we are using closed type families. Let us consider the case of a serialization library. As part of its functionality, the library must decide which representation to use for a specific data type. Normally, the type will remain the same in this representation, but for some special cases of "list-like" types (which are to be encoded in the same way as lists) and "function-like" (whose domain and target types must be recursively encoded). Those special cases are recognized by the following families:[4]

$$\textbf{type family } \mathit{IsListLike}\ l :: \mathit{Maybe}\ *\\ \textbf{type instance } \mathit{IsListLike}\ [e] \quad = \mathit{Just}\ e\\ \textbf{type instance } \mathit{IsListLike}\ (\mathit{Set}\ e) = \mathit{Just}\ e$$

$$\textbf{type family } \mathit{IsFunctionLike}\ f :: \mathit{Maybe}\ (*, *)\ \textbf{where}\\ \quad \mathit{IsFunctionLike}\ (s \to r) = \mathit{Just}\ (s, r)\\ \quad \mathit{IsFunctionLike}\ t \qquad = \mathit{Nothing}$$

The type family that constructs representations is intuitively formulated by matching on the result of the previously introduced families:

$$\textbf{type family } \mathit{Repr}\ t\ \textbf{where}\\ \quad \mathit{IsFunctionLike}\ t \sim \mathit{Just}\ (s, r) \Rightarrow \mathit{Repr}\ t = \mathit{Repr}\ s \to \mathit{Repr}\ r\\ \quad \mathit{IsListLike}\ \ t \sim \mathit{Just}\ e \Rightarrow \mathit{Repr}\ t = [\mathit{Repr}\ e]\\ \qquad\qquad\qquad\qquad \mathit{Repr}\ t = t$$

But the above definition is not valid Haskell syntax. Instead we have to encode the conditional equations using a chain of auxiliary type families, each of which treats a single context. As extra arguments to the auxiliary type families, we incorporate the check that should be done next. The *Repr* type family thus becomes:

$$\textbf{type family } \mathit{Repr}\ t\ \textbf{where}\\ \quad \mathit{Repr}\ t = \mathit{Repr1}\ t\ (\mathit{IsFunctionLike}\ t)$$

$$\textbf{type family } \mathit{Repr1}\ t\ l\ \textbf{where}\\ \quad \mathit{Repr1}\ t\ (\mathit{Just}\ (s, r)) = \mathit{Repr}\ s \to \mathit{Repr}\ r\\ \quad \mathit{Repr1}\ t\ f \qquad\quad = \mathit{Repr2}\ t\ (\mathit{IsListLike}\ t)$$

$$\textbf{type family } \mathit{Repr2}\ t\ l\ \textbf{where}\\ \quad \mathit{Repr2}\ t\ (\mathit{Just}\ e) = [\mathit{Repr}\ e]\\ \quad \mathit{Repr2}\ t\ l \qquad = t$$

Even though the code becomes larger, the translation could be made automatically by the compiler. The main problem in this case is the error reporting. Let us define a simple function that only works on types which are already in their representative form:

$$\mathit{alreadyNormalized} :: (t \sim \mathit{Repr}\ t) \Rightarrow t \to t\\ \mathit{alreadyNormalized} = \mathit{id}$$

If we try to use it on a *Map*, the compiler will complain:

```
*> alreadyNormalized Data.Map.empty
<interactive>:7:1:
 Couldn't match expected type 'Map k0 a0'
 with actual type
  Repr2 (Map k0 a0) (IsListLike (Map k0 a0))'
 The type variables 'k0', 'a0' are ambiguous
```

The source of this problem is that we have not declared whether *Map* is *ListLike* or not. However, the inner details of our implementation now escape to the outside world in this error message. If contexts were added to type families, it would greatly benefit users to treat them especially in terms of error reporting.

2.3 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 enlarged, but where some extra magic happens at each specific type. As a guiding example, let us construct a type family to obtain the spiciness of certain type-level dishes:

$$\textbf{data } \mathit{Water}\\ \textbf{data } \mathit{Nacho}\\ \textbf{data } \mathit{TikkaMasala}\\ \textbf{data } \mathit{Vindaloo}$$

$$\textbf{data } \mathit{SpicinessR} = \mathit{Mild} \mid \mathit{BitSpicy} \mid \mathit{VerySpicy}\\ \textbf{type family } \mathit{Spiciness}\ f :: \mathit{SpicinessR}$$

The family instances for the dishes are straightforward to write:

$$\textbf{type instance } \mathit{Spiciness}\ \mathit{Water} \qquad\ = \mathit{Mild}\\ \textbf{type instance } \mathit{Spiciness}\ \mathit{TikkaMasala} = \mathit{Mild}\\ \textbf{type instance } \mathit{Spiciness}\ \mathit{Nacho} \qquad\ = \mathit{BitSpicy}\\ \textbf{type instance } \mathit{Spiciness}\ \mathit{Vindaloo} \quad\ = \mathit{VerySpicy}$$

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 which are a bit spicy, the final result be definitely be very spicy. To rule this special case, we defer the *Spiciness* of a list to an auxiliary type family *SpicinessL*:

$$\textbf{type instance } \mathit{Spiciness}\ [a] = \mathit{SpicinessL}\ (\mathit{Spiciness}\ a)$$

$$\textbf{type family } \mathit{SpicinessL}\ \mathit{lst}\ \textbf{where}\\ \quad \mathit{SpicinessL}\ \mathit{BitSpicy} = \mathit{VerySpicy}\\ \quad \mathit{SpicinessL}\ a \qquad\quad = a$$

This trick has been used for more mundane purposes, such as creating lenses at the type level (Izbicki 2014). The key point is that the non-overlapping rules for open type families allow us to add new instances for those types for which one is not yet defined. But by calling a closed type family at a type instance rule, you can

[3] We shall remind here that we are considering type classes without overlapping instances. If overlapping instances were allowed, the implication would hold only in one direction.

[4] In some cases, GHC needs a quote sign in front of type-level tuples to distinguish them from the term-level tuples.

refine the behaviour of a particular instance. Section 4 will show other interesting uses of this pattern.

3. Shortcomings of type classes

We have looked at one side of the coin, discussing idioms to deal with shortcomings of type families with respect to type classes. Looking back at our original table in Section 1, the only functionality unsupported by type classes is closedness. We shall see how taking into account our previous results on type families, we can handle that situation.

3.1 Closed type classes

In some cases, you know that for a certain type class only a limited and known set of instances should be available. This is a situation where Haskell does not have an inmediate solution: exposing a type class without also allowing new instances to be defined. However, this sort of functionality has received some attention in the literature: Heeren and Hage (2005) discuss a **close** type class directive with this specific purpose in the framework of better error diagnosis; and Morris and Jones (2010) illustrate that their instance chains also handle this case.

There is a handful of techniques to get closed type classes known by Haskell practitioners (StackOverflow 2013). These techniques boil down to the same idea: define a secret entity of some sort (we shall see that this entity can either be a type class or a type family), define an alias and export only this alias to the world.

Heeren and Hage (2005) present an example of closing the *Integral* type class to only admit *Int* and *Integer* as instances, which we use as a running example. Following the "secret class + type alias" idea, a first attempt is:[5]

```
module ClosedIntegral (Integral) where
class Integral' i
instance Integral' Int
instance Integral' Integer
type Integral i = Integral' i
```

Note that to create such an alias, we need the ConstraintKind extension in GHC, which allows treating instance and type equality constraints as simple elements of the kind *Constraint*. This solution works fine until the moment of writing a new instance:

```
instance Integral Char
```

At that point, synonyms are expanded, and that code effectively translates to a new instance of *Integral′*. In conclusion, this method does not work.

The core problem is that, by exposing *Integral′* via a synonym, we have given access to it. Instead, we can use another type class, and make the one we want to close be a prerequisite:

```
class Integral' i ⇒ Integral i
instance Integral' i ⇒ Integral i
```

If you now try to define a new instance of *Integral* in another file, you get an error message:

```
Not in scope: type constructor or class Integral'
```

Once again, a disadvantage of this method is that error messages are worded in terms of the internal elements, in this case *Integral′*.

Instead of another type class, you can use a type family. In that case, we combine the idea from Section 2.1 with the alias approach. The first thing to do is to write a type family *Integral′* responsible for building the witness – of type *IntegralW* – for the elaboration phase. This encoding allows us to use the fact that type families can be closed:

```
data IntegralW = None | IntW | IntegerW
type family Integral' i :: IntegralW where
  Integral' Int     = IntW
  Integral' Integer = IntegerW
  Integral' other   = None
```

Note that we have included a final catch-all case for those types which should not be in the type class. The next step is defining a new type class which takes care of elaboration. In our case, this is *IntegralE*:

```
class IntegralE i (witness :: IntegralW)
instance IntegralE Int     IntW
instance IntegralE Integer (IntegerW)
```

An important remark at this point is that we do not have any instance for the *None* case. Additionally, if the module in which *Integral* is defined does not export *IntegralE*, no new case can be added, effectively closing the set of possible cases, as we did before by hiding *Integral′*. The final step is generating the alias, the one visible to the user, which takes care of calling *Integral′* and elaborating based on the witness:

```
type Integral i = IntegralE i (Integral' i)
```

This alias connects the elaboration type class with the type family responsible of building the witness.

As previously, internals of the implementation escape to the outside world in case of error. For example, if a function *f* with an *Integral* constraint is used with a *Char* value, the message produced by GHC reads:

```
No instance for (IntegralE Char 'None)
arising from a use of 'f'
```

A solution which also involves type families, but in a different way, uses in its core the ConstraintKind extension found in GHC. Since *Constraint* is a kind like $*$ or any other promoted type, writing a type family which returns one constraint is possible. This family would work as an alias for a restricted set of types:

```
type family Integral i :: Constraint where
  Integral Int     = Integral' Int
  Integral Integer = Integral' Integer
```

For those types which are stated in the type family, having *Integral* is equivalent to *Integral′*. But for those which are not members, family rewriting gets stuck:

```
Could not deduce (Integral Float)
```

One nice effect of this type family is that error messages are termed using the *Integral* type family, so fewer internals are exposed to the programmer.

4. Type families with elaboration

In (Schrijvers et al. 2007), one of the earliest papers about type families in Haskell, the authors did already consider how to express type families using type classes and functional dependencies. Thus, the question whether both sorts of type-level programming are neccessary and desirable is posed since the very beginning. We sketch in this section a *new answer*: type classes may not be needed, given that we give type families some elaboration mechanism.

4.1 Encoding type classes

Let us skip for a moment the issue of elaborating functions in type classes, and just focus on the typing parts. The aim is to find a

[5] We elided the methods to be elaborated in order to make the presentation more concise.

translation of type classes into type families such that an instance for a type is found if and only if the corresponding type family rewrites to a certain type. For this latest type, which describes whether an instance is defined, we shall use the promoted version of *Defined*:

$$\textbf{data}\ \mathit{Defined} = \mathit{Yes} \mid \mathit{No}$$

For each type class C that we want to convert, we declare a new type family *IsC* whose result is of kind *Defined*. Throughout the section, *Eq* will be used as a guiding example:

$$\textbf{type family}\ \mathit{IsEq}\ (t :: *) :: \mathit{Defined}$$

Furthermore, each function which declares an instance constraint must be changed to work with the new *IsC* type family. Now, the constraint is an equality between an *IsEq* application and *Yes*. The following code declares an identity function whose domain is restricted only to those types which have *Eq*:

```
eqIdentity :: IsEq t ~ Yes => t -> t
eqIdentity = id
```

Of course, the whole point of declaring a type class is to populate it with instances. The most simple cases, such as *Char*, are dealt simply by defining a **type instance** which rewrites to *Yes*:

```
type instance IsEq Char = Yes
type instance IsEq Int  = Yes
type instance IsEq Bool = Yes
```

Those cases whose definition depend on a context, such as *Eq* on lists, can call *IsC* on a smaller argument to defer the choice:

```
type instance IsEq [a] = IsEq a
```

In the case of a more complex context, such as *Eq* on tuples, which needs to check both of its type variables, we introduce a type family *And* which checks for definedness of all its arguments:

```
type family And (a :: Defined) (b :: Defined) :: Defined where
  And Yes Yes = Yes
  And a   b   = No
type instance IsEq (a, b) = And (IsEq a) (IsEq b)
```

As with type classes, we are not constrained to ground types in our type families, we can also use type constructors. A translation of the *Functor* type class and some instances in this style reads:

```
type family IsFunctor (t :: * -> *) :: Defined
type instance IsFunctor []    = Yes
type instance IsFunctor Maybe = Yes
```

At this point it is important to remark that in some cases GHC needs explicit *kind signatures* on some of the arguments of a type class. If they are not included, GHC defaults to kind $*$ instead of giving an ambiguity error, so the problem may be unnoticed until later on. Having said so, 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 *Collection* class in the introduction:

```
type family Collection t e :: Defined
type instance Collection [e]     e = Yes
type instance Collection (Set e) e = Yes
```

We discuss the translation of functional dependencies into this new scheme in Section 4.6. For a formal treatment of the full translation, the reader is referred to Appendix A.

4.2 Elaboration at rewriting

The previous translation works well from a typing perspective, but does not generate any code, and we do expect so when we use a type class. Since our main goal is to get rid of classes, we cannot use the same trick as we did in Section 2. Furthermore, in that case type families rewrote to different witnesses depending on the rule that was applied. But in this case we want all instances to return the same *Yes* result. If that was not the case, we could not declare a constraint such as $\mathit{IsEq}\ t \sim \mathit{Yes}$ which would not depend on the type itself.

For those reasons, we propose the concept of *elaboration at rewriting*. The idea is that at 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. Part of this idea is already in place when GHC generates coercions from family applications.

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 dictionary in the same place within a type family. Without any special preference, we shall use the **dictionary** keyword to introduce it.[6] For example, the following declaration adds an *eq* function to the *IsEq* type family:

```
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 = Defined
  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 the value of its dictionaries to build its own. As concrete syntax, we propose using *name*@ 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 is seen in action in the declaration of *IsEq* for lists:

```
type instance IsEq [a] = e@(IsEq a) where
  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 which 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:

```
notEq :: e@(IsEq a) ~ Yes => a -> a -> Bool
notEq x y = ¬ (e@eq x y)
```

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

As we have seen, elaboration at rewriting is possible and opens new possibilities for type families. It is also the only piece missing that we cannot directly encode in type families. In the rest of the paper, though, we shall just focus on the typing perspective, which in contrast with elaboration *is* available in Haskell compilers.

4.3 Type class directives

The good news about our encoding of type classes is that it brings with it ways to encode some constraints over type classes that were previously considered separate extensions of Haskell. We shall

[6] We would have preferred the **where** keyword in consonance with type classes, but this syntax is already used for closed type families.

focus first on the type class *directives* of (Heeren and Hage 2005). In short, these directives introduce new constructs to describe more sharply the set of types which are instances of a type class, with the aim of producing better error messages for the programmers.

The first of these directives is **never**: as its name suggests, a declaration of the form **never** $Eq\ (a \to b)$ forbids any instance of Eq for a function. Since by convention we translated $Eq\ t$ as $IsEq\ t \sim Yes$, we only need to ensure that $IsEq\ (a \to b)$ does not rewrite to Yes. We can do that easily with the following:

type instance $IsEq\ (a \to b) = No$

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 compilers do not allow overlapping rules for a type family, this also disallows anybody to write an instance for any instantiation of $a \to b$, as we wanted.

The second directive is **close**, which limits the set of instances for a type class to those which have been defined until that point. We have already discussed how to deal with closed type classes in Section 3.1, but with this new encoding, it becomes even easier. We only need to define a closed type family which rewrites to No for any forbidden instance. The example used above where $Integral$ has only Int and $Integer$ is written as:

type family $IsIntegral\ t$ **where**
  $IsIntegral\ Int \quad\ = Yes$
  $IsIntegral\ Integer = Yes$
  $IsIntegral\ t \qquad = No$

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.

Another directive available in (Heeren and Hage 2005) is **disjoint** $C\ D$, which constraints 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 naive encoding of this directive is done as follows for $Integral$, with a similar structure for $Rational$:[7]

type family $IsIntegral\ t$ **where**
  $IsIntegral\ t = IsICheckR\ t\ (IsRational\ t)$

type family $IsICheckR\ t\ (isRational :: Defined) :: Defined$ **where**
  $IsICheckR\ t\ Yes = No$
  $IsICheckR\ t\ No\ \ = IsIntegral'\ t$

type family $IsIntegral'\ t :: Defined$

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.

type instance $IsIntegral'\ Int \quad\ = Yes$
type instance $IsIntegral'\ Integer = Yes$

[7] If we try to define $IsIntegral$ and $IsRational$ as type synonyms, we get a complaint of cyclic definition:

```
Cycle in type synonym declarations:
 type IsIntegral t = IsICheckR t (IsRational t)
 type IsRational t = IsRCheckI t (IsIntegral t)
```

Unfortunately, this naive encoding does not work. When trying to deduce $IsIntegral$, the compiler loops: indeed, $IsIntegral$ calls $IsRational$, which in turn calls $IsIntegral$ and so on. One possible solution is changing $IsIntegral$ to:

type family $IsIntegral\ t$ **where**
  $IsIntegral\ t = IsICheckR\ t\ (IsRational'\ t)$

The objective of this change is breaking the loop by directly detecting whether we have a $Rational$ instance. This works well in the case in which we do not have an $Integral$ instance because of a $Rational$ one, as GHCi shows:

```
*> :kind! IsIntegral Float
IsIntegral Float :: Defined
= 'No
```

But in those cases where an explicit $IsIntegral$ rule is provided, the system is unable to reduce the type, since it does not know what $IsRational'$ rewrites to:

```
*> :kind! IsIntegral Int
IsIntegral Int :: Defined
= IsICheckR Int (IsRational' Int)
```

As a last attempt, we might try to check $IsIntegral$ and $IsRational$ values at the same time. For this, we introduce an $OnlyFirstDefined$ closed family which describes the disjointess condition:

type $IsIntegral\ t = OnlyFirstDefined\ (IsIntegral'\ t)\ (IsRational'\ t)$

type family $OnlyFirstDefined\ yes\ no :: Defined$ **where**
  $OnlyFirstDefined\ Yes\ no\ \ = Yes$
  $OnlyFirstDefined\ yes\ Yes = No$

But once again we encounter the same problem: if the type does not have a defined $IsIntegral'$ rule, the system is not able to continue to the next branch in the type family. At this point, we admit defeat, and have not found a good way to encode **disjoint** directly as type families, as we have done for **never** and **close**.

4.4 Instance chains

Instance chains were introduced in (Morris and Jones 2010) 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 you can state negative information about instances.

One case where overlapping instances are needed in common Haskell is the definition of the $Show$ instance for lists: in this case, a special instance is used for strings, that is $[Char]$. With this extension, the exception will be handled as an instance chain:

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

$Show$ also gives us an example of explicit failure: in general, we cannot make an instance for functions $a \to b$. However, if the domain of the function supports the $Enum$ class, we can give an instance which traverse the entire set of input values. In any other case, we want the system to explicitly know that no instance is possible:

instance $Show\ (a \to b)$ **if** $(Enum\ a, Show\ a, Show\ b)$ **where**
  $show = ...$
else instance $Show\ (a \to b)$ **fails**

As we did for type class directives, we can encode these cases using our type family translation. The first thing we notice is that the $Show$ instance chain follows the pattern of the open-closed type families: we must allow adding new rules for those types not already covered by other rules, but for some cases we need to make some ordered distinction, which takes the form of a closed family. We also apply the transformation of contexts as seen in Section 2.2. Putting it all together, the corresponding *IsShow* type family reads:

type family *IsShow t* :: *Defined*

type instance *IsShow* [*a*] = *IsShowList a*
type family *IsShowList a* **where**
  IsShowList Char = *Yes*
  IsShowList a = *IsShow a*

type instance *IsShow* (*a* → *b*)
  = *IsShowFn* (*IsEnum a*) (*IsShow a*) (*IsShow b*)
type family *IsShowFn isEnum isShowA isShowB* **where**
  IsShowFn Yes Yes Yes = *Yes*
  IsShowFn e a b = *No*

The family works nicely given some initial *IsShow* rules for *Bool*:

type instance *IsShow Bool* = *Yes*

```
*> :kind! IsShow (Bool -> [Char])
IsShow (Bool -> [Char]) :: Defined
= 'Yes
```

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

```
*Main> :kind! IsShow (Maybe Bool -> [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 soudness* 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 (Eisenberg et al. 2014).

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.

4.5 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 *e* open, so each type class could have its own way to report errors. In the case of a closed one, it makes sense to have a specific closed data type. 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 instance like functions could benefit from reporting different errors depending on the constraint that failed: [8]

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"`
  IsShowFn e a (*No b*)
    = *No* `"Target type must be showable"`

The interpreter will now return the corresponding message if the function is known to be not showable:

```
*> :kind! IsShow (Float -> Bool)
IsShow (Float -> Bool) :: Defined Symbol
= 'No "Function with non-enumerable domain"
```

Currently, *Symbol* values cannot be easily manipulated. In a scenario where simple functions such as concatenation are present in the standard libraries, more complete error messages could be obtained by joining information from different sources. For example, when *IsEnum* returns *No*, its message could be combined in *IsShownFn*, assuming the presence of a (: ++ :) type family to perform string concatenation:

IsShowFn (*No e*) *a b*
  = *No* (`"Function with non-enumerable domain"`
    : ++ : `"\nbecause "` : ++ : *e*)

In conclusion, the extra control we get by explicitly describing how to search for *Show* instances via the *IsShow* type family also helps us to better pinpoint to the user where things go wrong. This is especially important in many scenarios, such as embedded domain-specific languages (Hage 2014).

4.6 Functional dependencies

There is one feature of type classes that we have not yet covered in the translation to type families, namely, *functional dependencies*. A simple functional dependency, such as that relating *c* and *e* in:

class *Collection c e* | *c* → *e* **where** ...

can be split, as shown in (Schrijvers et al. 2007), into a type class for the relation (which would in turn be translated into a type family as discussed in this section), and another type function for defining *e* in terms of *c*:

type family *IsCollection′ c e* :: *Defined*
type instance *IsCollection′* [*e*] *e* = *Yes*

type family *IsCollectionElement c*
type instance *IsCollectionElement* [*e*] = *e*

However, this split does not guarantee that the types related by *IsCollection′* and *IsCollectionElement* satisfy any constraint. Of course, you want the result of *IsCollectionElement* to be the same as the *e* in *IsCollection′*. This can be enforced by defining a synonym *IsCollection* which relates both type families via an equality constraint over the element type:

[8] As we discussed earlier, GHC needs kind signatures in some cases. Here, had we not included *Defined Symbol* on *IsShowFn* arguments, GHC would expect *Defined*∗ as its kinds, which is not correct.

$$\textbf{type}\ \mathit{IsCollection}\ c\ e = \mathit{And}\ (\mathit{IsCollection}'\ c\ e)\ (\mathit{EqDef}\ e\ (\mathit{IsCollectionElement}\ c))$$

The *EqDef* type family just reifies type equality into *Defined*:

```
type family EqDef a b :: Defined where
  EqDef a a = Yes
  EqDef a b = No
```

Most uses of functional dependencies can be translated by the above schema. The reason is that in most cases, functional dependencies are just used to define type-level functions with instance arguments.

Some cases are more difficult to cope with, though, like the dependencies that you may add to addition. Essentially, when you know two arguments that make up a sum, you know the other one by simply adding or by cancellation law:

```
class AddFD m n r | m n → r, r m → n, r n → m
```

Note that if you try to give instances for this type class, such as:

```
instance AddFD Zero     n    n
instance AddFD (Succ m) Zero (Succ m)
instance AddFD m n r
  ⇒ AddFD (Succ m) (Succ n) (Succ (Succ r))
```

the compiler will complain because of a conflict in functional dependencies: if the second and third arguments are given, it cannot deduce the first one, because there is always some overlap with the first rule. However, let us suppose for a moment that we could use functional dependencies in that way: how would it translate into type families?

To get a complete answer, we need to look at the two different ways in which functional dependencies influence the type system:[9]

- *FD-improvement*: if the context contains $\mathit{AddFD}\ m1\ n1\ r_1$ and $\mathit{AddFD}\ m2\ n2\ r2$, and we know that $m1 \sim m2$ and $n1 \sim n2$, then we have $r_1 \sim r2$;
- *Instance improvement*: if the context contains $\mathit{AddFD}\ m\ n\ r$, and for some substitution of m and n only one instance matches, then we can use it to rewrite r. For example, if we have $\mathit{AddFD}\ \mathit{Zero}\ n\ r$, we know inmediately that $n \sim r$.

In type family terms (where we define the corresponding *IsAddFD* family as shown above), FD-improvement translates into obtaining $r_1 \sim r2$ knowing that $m1 \sim m2$, $n1 \sim n2$ and, here comes the crux of the matter, $\mathit{IsAdd}\ m1\ n1\ r_1 \sim \mathit{IsAdd}\ m2\ n2\ r2$. Thus, the functional dependency constraint becomes a partial *injectivity* constraint in the family: if the results of a function, and some of its arguments (in this case, m and n) agree for two applications, we know that remaining argument (here, r) must also agree. A simple form of injectivity for type families has been considered for GHC, but has not been implemented as of version 7.8.[10]

On the other hand, instance improvements correspond to the ability of defining and *inverting* type-level functions from the instance relations. The functional dependency $m\ n \rightarrow r$ on *AddFD* is doing nothing more than defining the addition function in the type level (as shown in the Introduction), if we want to encode the other two, we need to invert addition:

```
type family IsAddRNToM where
  IsAddRNToM r        Zero     = r
  IsAddRNToM (Succ r) (Succ n) = IsAddRNToM r n
```

```
type family IsAddRMToN where
  IsAddRMToN r        Zero     = r
  IsAddRMToN (Succ r) (Succ m) = IsAddRMToN r m
```

While several approaches to bidirectionalization of functional programs have been proposed (Foster et al. 2012), it is not always possible or desirable to use bidirectionalization. Looking at type classes with our type family glasses can help decide when a certain functional dependency will be useful: if you cannot get the corresponding function out of it, the instance improvement rule may never be applied.

5. Comparison

5.1 Type families as functional dependencies

Sections 2 and 3 looked at how to deal with features not readily available in type classes or families. In Section 4 we turned to type families as an integrating framework for both concepts. In previous literature (Schrijvers et al. 2007) type classes with functional dependencies were used 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 4.6. 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 (Schrijvers et al. 2007; Diatchki 2007), 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.

5.2 Implicit arguments

In essence, in Section 4 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 (Devriese and Piessens 2011), which was also proposed to simulate type classes. Any argument marked as such in a function with double braces, like:

```
myFunction : {A : Set} → {{p : Show A}} → A → String
```

will be replaced by any value of the corresponding type in the environment in which it was called. Thus, if you think of *Show* of a class, you can provide an instance by constructing such a value:

```
showInt : Show Int
showInt x = ...   -- code for printing an integer
```

Since these values are constructed at the term level, you 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 search when looking for instance arguments, whereas our solution can simulate search with backtracking.

5.3 Tactics

The dependently type language Idris (Brady 2013) generalises the idea of Agda's instance arguments allowing the programmer to customise the search strategy for implicit arguments. Similarly to Coq, Idris has a tactic language to customise proof search. Unlike Coq, however, Idris allows the programmer to use the same machinery to customise the search for implicit arguments (The Idris Community 2014).

[9] Using the terminology from `https://ghc.haskell.org/trac/haskell-prime/wiki/FunctionalDependencies`.

[10] GHC Trac ticket on Injective type families: `https://ghc.haskell.org/trac/ghc/ticket/6018`.

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 $\mathit{Show}\ a$:

$$\mathit{myFunction} : \{\textbf{default}\ \mathit{tactics}\ \{t\}\ p : \mathit{Show}\ a\} \to a \to \mathit{String}$$

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

$$\mathit{myFunction} : \{\textbf{default}\ \mathit{tactics}\ \{\mathit{applyTactic}\ \mathit{findShow}; \mathit{solve}\}\ p : \mathit{Show}\ a\} \to a \to \mathit{String}$$

The search strategy is defined by *findShow*, which is an Idris function of that takes the goal type and the context as argument 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, *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.

6. Conclusion

Type classes and type families in Haskell have different sets of features. However, with a little work we can support elaboration and contexts in families, and closedness in instances. This suggests that there exists a framework for integrating the two as instances of a single concept: we show how type families can serve as such a concept. By creating type families which simulate classes, we get for free features such as type class directives, instance chains and control over the search procedure. We have argued that it is possible to add an elaboration mechanisms to type families to bridge the gap for its use in ad-hoc polymorphism.

References

P. Bahr. Composing and decomposing data types: A closed type families implementation of data types la carte. 10th ACM SIGPLAN Workshop on Generic Programming, to appear, 2014.

E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, 9 2013. ISSN 1469-7653. . URL http://journals.cambridge.org/article_S095679681300018X.

D. Devriese and F. Piessens. 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, pages 143–155, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0865-6. . URL http://doi.acm.org/10.1145/2034773.2034796.

I. S. Diatchki. *High-level abstractions for low-level programming*. PhD thesis, OGI School of Science & Engineering, May 2007.

R. A. Eisenberg, D. Vytiniotis, S. Peyton Jones, and S. Weirich. Closed type families with overlapping equations. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 671–683, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2544-8. . URL http://doi.acm.org/10.1145/2535838.2535856.

N. Foster, K. Matsuda, and J. Voigtlnder. Three complementary approaches to bidirectional programming. In J. Gibbons, editor, *Generic and Indexed Programming*, volume 7470 of *Lecture Notes in Computer Science*, pages 1–46. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-32201-3. . URL http://dx.doi.org/10.1007/978-3-642-32202-0_1.

J. Hage. Domain specific type error diagnosis (DOMSTED). Technical Report UU-CS-2014-019, Department of Information and Computing Sciences, Utrecht University, 2014.

B. Heeren and J. Hage. Type class directives. In *Proceedings of the 7th International Conference on Practical Aspects of Declarative Languages*, PADL'05, pages 253–267, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-24362-3, 978-3-540-24362-5. . URL http://dx.doi.org/10.1007/978-3-540-30557-6_19.

M. Izbicki. A neat trick for partially closed type families. Blog post available at http://izbicki.me/blog/a-neat-trick-for-partially-closed-type-families, 2014.

M. Jones. Type classes with functional dependencies. In G. Smolka, editor, *Programming Languages and Systems*, volume 1782 of *Lecture Notes in Computer Science*, pages 230–244. Springer Berlin Heidelberg, 2000. ISBN 978-3-540-67262-3. . URL http://dx.doi.org/10.1007/3-540-46425-5_15.

J. G. Morris and M. P. Jones. Instance chains: type class programming without overlapping instances. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, pages 375–386, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3. . URL http://doi.acm.org/10.1145/1863543.1863596.

S. Peyton Jones, M. Jones, and E. Meijer. Type classes: exploring the design space. In *Haskell Workshop*, 1997.

T. Schrijvers, M. Sulzmann, S. Peyton Jones, and M. Chakravarty. Towards open type functions for haskell. In *19th International Symposium on Implemantation and Application of Functional Languages*, 2007.

StackOverflow. Closed type classes. Question and answers available at http://stackoverflow.com/questions/17849870/closed-type-classes, 2013.

The Idris Community. Programming in Idris: A tutorial. Available from http://eb.host.cs.st-andrews.ac.uk/writings/idris-tutorial.pdf, 2014.

D. Vytiniotis, S. Peyton jones, T. Schrijvers, and M. Sulzmann. Outsidein(x) modular type inference with local assumptions, Sept. 2011. ISSN 0956-7968. URL http://dx.doi.org/10.1017/S0956796811000098.

P. Wadler and S. Blott. 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, pages 60–76, New York, NY, USA, 1989. ACM. ISBN 0-89791-294-2. . URL http://doi.acm.org/10.1145/75277.75283.

B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12, pages 53–66, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1120-5. . URL http://doi.acm.org/10.1145/2103786.2103795.

A. Formal translation from classes to families

In Section 4 we looked at the translation from type classes to families, but left out the technical details. This section deals with those details and the associated soundness and termination properties. We leave functional dependencies out of this discussion, since they come with their own set of difficulties, as shown in Section 4.6.

There are three Haskell constructs to translate: classes, contexts and instances. Type class declarations are of the form $\textbf{class}\ D\ t_1 \ldots tn$. Each of them gives raise to a new type family encoded as:

$$\textbf{type family}\ \mathit{IsD}\ t_1 \ldots t_m :: \mathit{Defined}$$

Here, *Defined* is the kind which represents whether an instance is available. It was introduced in Section 4.1 and refined in Section 4.5 to get better error messages. In addition, types t_1 to t_m may include kind annotations inferred from their use in the elaborated methods.

Note that we have not spoken about superclass contexts: they do not interfere with instance resolution, just impose a constraint of having to define an instance of each superclass. In this case, given a class $S \Rightarrow D$, the constraint would translate to having to define IsS to return Yes each time IsD returns Yes. Thus, superclasses impose their conditions on a prior stage to type checking.

The second construct to translate are context declarations of the form $Q\ s_1 .. s_j$, which may appear in function signatures, data types or other instance declarations. The translation is $\mathit{IsQ}\ s_1 \ldots s_j$.

Finally, we need to translate instance declarations. Each instance may have a number of context declarations, say n:

instance $(Q_1, \ldots, Q_n) \Rightarrow D\ t_1 \ldots t_m$

A type family instance is defined for each of them, of the form:

type instance $IsD\ t_1 \ldots t_m = And_n\ Q_1 \ldots Q_n$

For each number n of context declarations, we have a corresponding And_n closed type family which checks that all the arguments are Yes. More formally, we have:

type family $And_0 :: Defined$
  $And_0 = Yes$
type family $And_1\ d :: Defined$
  $And_1\ x = x$
type family $And_n\ d_1 \ldots d_n :: Defined$
  $And_n\ Yes \ldots Yes = Yes$ -- case everything Yes
  $And_n\ d_1 \ \ldots d_n \ = No$

In the translation, Q_1 to Q_n refer to the translation of instance constraints Q_1 to Q_n as given above.

A.1 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 (Vytiniotis et al. 2011). The authors describe the inference process in terms of a general framework, called OutsideIn(X), which is parametrized by a constraint system X. Each constraint system defines a concrete entailment $\mathcal{Q} \Vdash W$ which gives semantics to certain constraint Q under the axioms in the set $\mathcal{Q}$. Axioms are the generic name given to declarations such as class and family instances.

In particular, we are interested in the case X = type classes and type families, that is also discussed in (Vytiniotis et al. 2011). For this case, many rules are given for the concrete entailment $\Vdash$. Many of them deal, such are those dealing with reflexivity, symmetry and transitivity are quite straightforward:

$$\frac{}{\mathcal{Q} \Vdash \tau \sim \tau}\ \text{REFL} \qquad \frac{\mathcal{Q} \Vdash \tau_1 \sim \tau_2}{\mathcal{Q} \Vdash \tau_2 \sim \tau_1}\ \text{SYM}$$

$$\frac{\mathcal{Q} \Vdash \tau_1 \sim \tau_2 \qquad \mathcal{Q} \Vdash \tau_2 \sim \tau_3}{\mathcal{Q} \Vdash \tau_1 \sim \tau_3}\ \text{TRANS}$$

The rules related to type classes and type families are:

$$\frac{\mathcal{Q} \Vdash \bigwedge \overline{\tau_1 \sim \tau_2}}{\mathcal{Q} \Vdash F\ \overline{\tau_1} \sim F\ \overline{\tau_2}}\ \text{FCOMP}$$

$$\frac{\mathcal{Q} \Vdash D\ \overline{\tau_1} \qquad \mathcal{Q} \Vdash \bigwedge \overline{\tau_1 \sim \tau_2}}{\mathcal{Q} \Vdash D\ \overline{\tau_2}}\ \text{DICTEQ}$$

$$\frac{\forall \overline{a}. Q_1 \Rightarrow Q_2 \in \mathcal{Q} \qquad \mathcal{Q} \Vdash [\overline{a \mapsto \tau}] Q_1}{\mathcal{Q} \Vdash [\overline{a \mapsto \tau}] Q_2}\ \text{AXIOM}$$

The first two rules define how type equality distributes over instance constraints and type family applications. The last one describes the application of axioms: if we can prove the preconditions of an axiom for an specific substitution $[\overline{a \mapsto \tau}]$, then we can conclude the postcondition in the axiom. Note than in the case of type family instances, Q_1 is always empty, so the rule in that case reads:

$$\frac{\forall \overline{a}. F\ \overline{\rho} \sim \sigma \in \mathcal{Q}}{F\ [\overline{a \mapsto \tau}]\rho \sim [\overline{a \mapsto \tau}]\sigma}\ \text{AXIOM'}$$

A.2 Soundness of translation

In OutsideIn(X), entailment relations are parametrized by a set of axioms $\mathcal{Q}$, which can be either type class or type family instances. We define $\mathcal{Q}^{trans}$ as the set of axioms obtained by translating each instance axiom as defined above.

Lemma 1. *If* $\mathcal{Q} \Vdash \bigwedge_i D_i\ \overline{\tau_i} \sim Yes$*, then* $\mathcal{Q} \Vdash And_n\ \overline{D_i\ \overline{\tau_i}} \sim Yes$.

Proof. By case analysis of the definition of And_n. □

Theorem 1. *If* $\mathcal{Q} \Vdash D\ \overline{\tau}$*, then* $\mathcal{Q}^{trans} \Vdash IsD\ \overline{\tau} \sim Yes$.

Proof. By inversion of the rule applied to get $\mathcal{Q} \Vdash D\ \overline{\tau}$. There are only two interesting cases, DICTEQ and AXIOM.

For DICTEQ, taking into account the translation, proving $\mathcal{Q}^{trans} \Vdash IsD\ \overline{\tau} \sim Yes$ boils down to proving the soundness of this rule:

$$\frac{\mathcal{Q} \Vdash IsD\ \overline{\tau_1} \qquad \mathcal{Q} \Vdash \bigwedge \overline{\tau_1 \sim \tau_2}}{\mathcal{Q} \Vdash IsD\ \overline{\tau_2}}$$

The following derivation shows how to get it:

$$\frac{\mathcal{Q} \Vdash IsD\ \overline{\tau_1} \qquad \dfrac{\mathcal{Q} \Vdash \bigwedge \overline{\tau_1 \sim \tau_2}}{\mathcal{Q} \Vdash IsD\ \overline{\tau_1} \sim IsD\ \overline{\tau_2}}\ \text{FCOMP}}{\mathcal{Q} \Vdash IsD\ \overline{\tau_2}}\ \text{SYM, TRANS}$$

For AXIOM, first note that instance axioms of the form $\overline{Q} \Rightarrow Q^*$ get translated into type family axioms of the form $Q^* \sim And_n\ \overline{IsQ\ \overline{q}}$. Thus, we need to prove soundness of the rule:

$$\frac{\forall \overline{a}. IsD\ \overline{\sigma} \sim And_n\ \overline{IsQ\ \overline{q}} \in \mathcal{Q} \qquad \mathcal{Q} \Vdash \bigwedge IsQ\ [\overline{a \mapsto \tau}]\overline{q} \sim Yes}{\mathcal{Q} \Vdash IsD\ [\overline{a \mapsto \tau}]\overline{\sigma} \sim Yes}$$

We can derive the first premise by using AXIOM:

$$\frac{\forall \overline{a}. IsD\ \overline{\sigma} \sim And_n\ \overline{IsQ\ \overline{q}} \in \mathcal{Q}}{IsD\ [\overline{a \mapsto \tau}]\overline{\sigma} \sim And_n\ \overline{IsQ\ [\overline{a \mapsto \tau}]\overline{q}}}\ \text{AXIOM}$$

For the second premise, first apply the induction hypothesis to convert the proofs of the context of the rule. Then, use the previous lemma to get the version with And_n:

$$\frac{\mathcal{Q} \Vdash \bigwedge IsQ\ [\overline{a \mapsto \tau}]\overline{q} \sim Yes}{\mathcal{Q} \Vdash And_n\ \overline{IsQ\ [\overline{a \mapsto \tau}]\overline{q}} \sim Yes}$$

Using SYM and TRANS we get the desired result. □

A.3 Termination

An important issue to consider is whether termination characteristics of class instances are also carried over to the translated families. The most lenient conditions imposed by GHC over class instances[11] are the so-called *Paterson conditions*. 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.

In the case of type families $F\ t_1 \ldots t_m = s$, the conditions imposed by GHC ask that for each type family application $G\ \ r_1 \ldots r_k$ appearing in s, we have:

[11] If the user does not turn on the $UndecidableInstances$, which turns off any termination checking.

1. Each of the arguments $r_1 \mathinner{..} r_k$ do not contain any other type family applications.
2. The total number of data type constructors and variables in $r_1 \mathinner{..} r_k$ is strictly smaller than in $t_1 \mathinner{..} t_m$.
3. Each variable occurs in $r_1 \mathinner{..} r_k$ at most as often as in $t_1 \mathinner{..} t_m$.

The translation of a class instance which satisfies the Paterson conditions into a type family instance:

$$\textbf{type instance } \mathit{IsD}\ t_1 \ldots t_m = \mathit{And}_n\ Q_1 \ldots Q_n$$

satisfies the terminations conditions (2) and (3) of type families. However, condition (1) is not satisfied, because And_n contains nested family applications. Note that these are the only nested applications generated by the translation.

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