Type Families with Class, Type Classes with Family

Alejandro Serrano Jurriaan Hage

Department of Information and Computing Sciences
Utrecht University, The Netherlands
{A.SerranoMena, J.Hage}@uu.nl

Patrick Bahr

Department of Computer Science
University of Copenhagen, Denmark
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 [22]: 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:

```
class Show a where
  show :: a → String
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:

```
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:

```
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, 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*:[1]

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

[1] This example works only in GHC with UndecidableInstances extension.

Haskell'15, September 3-4, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3808-0/15/09...$15.00
http://dx.doi.org/10.1145/2804302.2804304

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 [7]. Major implementations such as the Glasgow Haskell Compiler (GHC) and the Utrecht Haskell Compiler (UHC) support these features.

Type families [15] 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:

```
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. [6] 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:

```
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:

```
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:

```
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:

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

and its corresponding type class translation:

```
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:

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

but not the one with functional dependencies:

```
g :: FunDep a b => Example' a -> b
g (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 [21] in Section 6. In this paper, we consider type classes without support for overlapping instances. However, as we argue in Section 3.1, 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* [23], 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:

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

Data type promotion produces the two types *False* and *True* of kind *Bool*[2]. Using type families we can define functions which operate on elements of a specific kind:

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

[2] 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 $*$ and $\rightarrow$, we can now have kind-level constructors coming from the promotion of a parametrized type. For example, the definition:

$$\textbf{data}\ \mathit{Maybe}\ a = \mathit{Nothing} \mid \mathit{Just}\ a$$

promotes into types *Nothing* and *Just* whose kind is parametric:

$$\begin{array}{ll} \mathit{Nothing} :: & \mathit{Maybe}\ k \\ \mathit{Just} & :: k \rightarrow \mathit{Maybe}\ k \end{array}$$

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

$$\begin{array}{l} \textbf{type family}\ \mathit{IsJust}\ x\ \textbf{where} \\ \quad \mathit{IsJust}\ (\mathit{Just}\ x) = \mathit{True} \\ \quad \mathit{IsJust}\ \mathit{Nothing} = \mathit{False} \end{array}$$

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

```
> :kind! IsJust
IsJust :: Maybe k -> Bool
= forall (k :: BOX). 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 *Show String* we write $\mathit{IsShow}\ \mathit{String} \sim \mathit{Yes}$, where $\sim$ is the notation for type equality. We follow the convention that a type class D gives rise to a corresponding type family *IsD*. Let us look at all the components of this construction via an example.

In principle, we could reuse the promoted data type *Bool* as result kind of these characteristic functions. Instead, we define a fresh kind *Defined*, given as follows:

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

There are two main reasons for defining a new kind instead of merely using *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 *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 *Defined* to include more information and defining a separate kind gives us this freedom.

Every type class declaration of the form **class** $C\ t_1 \ldots t_m$ is translated to a corresponding type family *IsC*:

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

For example, consider the definition of the *Eq* type class:

$$\begin{array}{l} \textbf{type class}\ \mathit{Eq}\ a\ \textbf{where} \\ \quad (\equiv), (\not\equiv) :: a \rightarrow a \rightarrow \mathit{Bool} \end{array}$$

Eq is translated to the following type family:

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

Note that we have included a kind signature $*$ for the argument t because the definition of the *Eq* type class restricts its instances to that kind. This is inferred by the compiler from the signatures of $(\equiv)$ and $(\not\equiv)$, but cannot be done automatically for *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 bye the type constraint $\mathit{IsC}\ t \sim \mathit{Yes}$. For example, say we want to declare an identity function whose domain is restricted to only those types that have an *Eq* instance:

$$\begin{array}{l} \mathit{eqIdentity} :: \mathit{Eq}\ t \Rightarrow t \rightarrow t \\ \mathit{eqIdentity} = \mathit{id} \end{array}$$

In the translation of type classes to type families, the type signature of *eqIdentity* is changed to the following:

$$\mathit{eqIdentity} :: \mathit{IsEq}\ t \sim \mathit{Yes} \Rightarrow t \rightarrow t$$

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

$$\begin{array}{l} \textbf{type instance}\ \mathit{IsEq}\ \mathit{Char} = \mathit{Yes} \\ \textbf{type instance}\ \mathit{IsEq}\ \mathit{Int} \ \ = \mathit{Yes} \\ \textbf{type instance}\ \mathit{IsEq}\ \mathit{Bool} = \mathit{Yes} \end{array}$$

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

$$\textbf{type instance}\ \mathit{IsEq}\ [a] = \mathit{IsEq}\ a$$

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

$$\begin{array}{l} \textbf{type family}\ \mathit{And}\ (a :: \mathit{Defined})\ (b :: \mathit{Defined})\ \textbf{where} \\ \quad \mathit{And}\ \mathit{Yes}\ \mathit{Yes} = \mathit{Yes} \\ \quad \mathit{And}\ a \quad b \quad = \mathit{No} \\ \textbf{type instance}\ \mathit{IsEq}\ (a, b) = \mathit{And}\ (\mathit{IsEq}\ a)\ (\mathit{IsEq}\ b) \end{array}$$

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

$$\begin{array}{l} \textbf{type family}\ \mathit{IsFunctor}\ (t :: * \rightarrow *) :: \mathit{Defined} \\ \textbf{type instance}\ \mathit{IsFunctor}\ [\,] \qquad = \mathit{Yes} \\ \textbf{type instance}\ \mathit{IsFunctor}\ \mathit{Maybe} = \mathit{Yes} \end{array}$$

Once again, we write a kind signature to prevent GHC from defaulting the kind of the t parameter to *IsFunctor* to $*$, 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 *Collection* class in the introduction:

$$\begin{array}{l} \textbf{type family}\ \mathit{IsCollection}\ t\ e :: \mathit{Defined} \\ \textbf{type instance}\ \mathit{IsCollection}\ [e] \qquad e = \mathit{Yes} \\ \textbf{type instance}\ \mathit{IsCollection}\ (\mathit{Set}\ e)\ e = \mathit{Yes} \end{array}$$

As in the case of one-parameter type classes, our *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*.[3] Overlapping instances can be used to override an instance declaration in a more specific scenario. The

[3] 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 $[\mathit{Char}]$, and for which we want a different way to print them:

$$\begin{array}{l}\textbf{instance}\ \mathit{Show}\ [\mathit{Char}]\ \textbf{where}\\ \quad \mathit{show}\ \mathit{str} = \ldots \quad \text{-- show between quotes}\end{array}$$

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. Given a type class $D\ t_1 .. t_m\ r\ s_1 \ldots s_k$, a functional dependency declaration has the form $t_1 \ldots t_m \to r$, expressing that type r is uniquely determined by the types $t_1, \ldots, t_m$. Examples of functional dependencies are given in the $\mathit{Collection}$ 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 $D\ t_1 .. t_m\ r\ s_1 \ldots s_k$ not only those occurring to the left of r; and the right-hand side may contain more than one type. But for simplicity, we assume that functional dependencies have this shape. We defer the more precise treatment until Section 6.

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 $D\ t_1 \ldots t_m\ r\ s_1 \ldots s_k$ with a functional dependency $t_1 \ldots t_m \to r$ comes into play:

- Suppose two different sequences of types are instances of D, that is, we have instances $D\ t'_1\ \ldots\ t'_m\ r'\ s'_1\ \ldots\ s'_k$ and $D\ t^*_1\ \ldots\ t^*_m\ r^*\ s^*_1\ \ldots\ s^*_k$. That means, whenever we have the equalities $t'_i \sim t^*_i$ for all $i = 1, \ldots, m$, then we also have the equality $r' \sim r^*$. Intuitively, this comes from the requirement of $t_1 \ldots t_m$ 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 $\mathit{Collection}\ [\mathit{Int}]\ x$ with a yet-unknown x. By using improvement with the dependency of the second argument over the first, we can deduce that $x \sim \mathit{Int}$ from the instance declaration $\mathit{Collection}\ [e]\ e$.

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 $\mathit{Collection}$ gives rise to a family:

$$\begin{array}{l}\textbf{type family}\ \mathit{CollectionElement}\ c\\ \textbf{type instance}\ \mathit{CollectionElement}\ [e] = e\end{array}$$

This technique is not new: the associated type family in the $\mathit{Collection2}$ 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 $\mathit{Collection}$ 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 $\mathit{IsCollection}$ type family is instantiated, a new rule is also added to $\mathit{CollectionElement}$ 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 $\mathit{IsCollection}$ type family, you would have to mention the $\mathit{CollectionElement}$ 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[4] for GHC. Syntactically, injectivity annotations are similar to functional dependencies:

$$\begin{array}{l}\textbf{type family}\ F\ t_1 \ldots t_m\ r\ s_1 .. s_k :: (\mathit{result} :: \kappa)\\ \qquad\qquad\qquad \mid \mathit{result}\ t_1 \ldots t_m \to r\end{array}$$

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 $\mathit{result} \to r$, the description coincides exactly on the function F being injective on the parameter r.[5]

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 $\mathit{result}\ t_1 \ldots t_m \to r$ in the translated IsC. In the translation the only possible value for the result of the type family IsC is Yes, and thus the addition of result in the injectivity annotation does not add any further information.[6]

The $\mathit{Collection}$ 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:

$$\begin{array}{l}\textbf{type family}\ \mathit{IsCollection}\ t\ e :: (\mathit{result} :: \mathit{Defined})\\ \qquad\qquad\qquad\qquad\qquad \mid \mathit{result}\ t \to e\end{array}$$

The injectivity constraint acknowledges the fact that when $\mathit{result} \sim \mathit{Yes}$ 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:

$$\textbf{class}\ C_1\ \overline{s}_1, \ldots, C_k\ \overline{s}_k \Rightarrow D\ t_1 \ldots t_m$$

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 $D\ t_1 \ldots t_m$ 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:

$$\begin{array}{l}\textbf{class}\ \mathit{Eq}\ a \Rightarrow \mathit{Ord}\ a\ \textbf{where} \ldots\\ \textbf{instance}\ \mathit{Eq}\ a \Rightarrow \mathit{Eq}\ [a]\ \textbf{where} \ldots\end{array}$$

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

[4] At `https://ghc.haskell.org/trac/ghc/wiki/InjectiveTypeFamilies`.

[5] At the moment there are only plans to implement injectivity annotations of the form $\mathit{result} \to t_1 \ldots t_m\ r\ s_1 .. s_k$, 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 $Eq\ a$,[7] 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:

$$D\ t_1\ \dots\ t_m \iff (D\ t_1\ \dots\ t_m, C_1\ \overline{s_1}, \dots, C_k\ \overline{s_k})$$

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

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

$$(>) :: Ord\ a \Rightarrow a \to a \to Bool$$

is translated to

$$(>) :: (IsOrd\ a \sim Yes, IsEq\ a \sim Yes) \Rightarrow a \to a \to Bool$$

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:

$$\begin{array}{l} Eq\ a :: Constraint \\ Eq\ \ :: * \to Constraint \\ IsEq\ a \sim Yes :: Constraint \end{array}$$

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:

$$\textbf{type}\ Serializable\ t = (Show\ t, Read\ t)$$

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:

$$\textbf{type}\ IsOrd^{\uparrow}\ a = (IsOrd\ a \sim Yes, IsEq^{\uparrow}\ a)$$

The $IsOrd\ a \sim Yes$ 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 $IsEq^{\uparrow}\ a$, 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

$$(>) :: Ord\ a \Rightarrow a \to a \to Bool$$

would be translated to a call to $IsOrd$:

$$(>) :: IsOrd\ a \sim Yes \Rightarrow a \to a \to Bool$$

now we use the type synonym we have just defined:

$$(>) :: IsOrd^{\uparrow}\ a \Rightarrow a \to a \to Bool$$

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

[7] Because any other instance for $[a]$ would overlap with the given one.

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 [17] 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** $Eq\ (a \to b)$ forbids any instance of Eq for a function type. Given that we translate $Eq\ t$ to $IsEq\ t \sim Yes$ (since it has no superclasses), we only need to ensure that $IsEq\ (a \to b)$ does not rewrite to Yes. We can do that easily with the following declaration:

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

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 rules for a type family may not overlap, this definition also disallows anybody to write an instance for any instantiation of $a \to b$, 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, 17], 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:

$$\begin{array}{l} \textbf{type family}\ IsIntegral\ t\ \textbf{where} \\ \quad IsIntegral\ Int \quad\quad = Yes \\ \quad IsIntegral\ Integer = Yes \\ \quad IsIntegral\ t \quad\quad\ \ = No \end{array}$$

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 o 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 naive encoding of this directive for *Integral* is achieved as follows:

```
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
type family IsRational t where
  IsRational t = IsICheckI t (IsIntegral t)
type family IsICheckI t (isIntegral :: Defined)
   :: Defined where
  IsICheckI t Yes  = No
  IsICheckI t No   = IsRational' t
type family IsRational' 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     = Yes
type instance IsIntegral' Integer = Yes
```

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

$$\begin{aligned} \mathit{IsIntegral\ Int} &\sim \mathit{IsCheckR}\ (\mathit{IsRational\ Int}) \\ &\sim \mathit{IsCheckR}\ (\mathit{IsCheckI}\ (\mathit{IsIntegral\ Int})) \\ &\sim \ldots \end{aligned}$$

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

```
type IsIntegral  t = IsICheckR t (IsRational t)
type IsRational t = IsRCheckI t (IsIntegral t)
```

the compiler itself would have detected this cycle in the definition and informed as 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:

```
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.

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

Examples of instances are:

```
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:

```
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 $\mathit{IsIntegral}\ t \sim \mathit{Yes}$ or $\mathit{IsRational}\ t \sim \mathit{Yes}$.

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:

$$\textbf{type}\ \mathit{IsIntegral}^{\uparrow}\ t = \mathit{IsIntegral}\ t \sim \mathit{Yes}$$

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

$$\textbf{type}\ \mathit{IsIntegral}^{\uparrow}\ t = \mathit{IsIntegralOrRational}\ t \sim \mathit{Integral}$$

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:

```
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:

```
type instance Spiciness Water       = Mild
type instance Spiciness TikkaMasala = Mild
type instance Spiciness Nacho       = BitSpicy
type 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*:

```
type instance Spiciness [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 $a \sim Char$, 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 $a \to b$. 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* ($a \to b$) **if** (*Enum a*, *Show a*, *Show b*) **where**
  show = ...
else instance *Show* ($a \to b$) **fails**

As in the previous case, when matching *Show* ($a \to b$), 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 And_3 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* ($a \to b$) = *IsShowFn a b*
type family *IsShowFn a b*
  = And_3 (*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*

```
*> :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 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 $IsEnum\ Int \sim Yes$, 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 *e* 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:[8]

type instance *IsShow* ($a \to b$)
  = *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"`

[8] 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.

$$IsShowFn\ e\ a\ (No\ b) = No\ \texttt{"Target type must be showable"}$$

The interpreter will now return the corresponding message if the function type is known to not be an instance of *Show*:

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

Currently, *Symbol* values cannot be easily manipulated. Once simple operations such as concatenation are present in the standard libraries, even more informative 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 operator to perform string concatenation:

$$\begin{aligned} &IsShowFn\ (No\ e)\ a\ b \\ &\quad = No\ (\texttt{"Function with non-enumerable domain"} \\ &\qquad\qquad +\!\!+\ \texttt{"\textbackslash nbecause "}\ +\!\!+\ e) \end{aligned}$$

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 inform the user where things go wrong. This is especially important in scenarios such as type error diagnosis for embedded domain-specific languages [7].

5. Elaboration at Rewriting

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, and 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 $a \to b \to Bool$, we want a function $mkConst :: Bool \to (a \to b \to Bool)$. To start, we need a type-level function that computes the result type of a curried function type of arbitrary arity:

$$\begin{aligned} &\textbf{type family}\ Result\ f\ \textbf{where} \\ &\quad Result\ (s \to r) = Result\ r \\ &\quad Result\ r \qquad\quad\ = r \end{aligned}$$

This is the point where, if we could elaborate a function during rewriting, implementing *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 to define a promoted data type to encode such witness on the type level.

$$\textbf{data}\ ResultWitness = End \mid Step\ ResultWitness$$

We then define the closed type family $Result'$, 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.

$$\begin{aligned} &\textbf{type family}\ Result'\ f :: ResultWitness\ \textbf{where} \\ &\quad Result'\ (s \to m) = Step\ (Result'\ m) \\ &\quad Result'\ r \qquad\quad\ = End \end{aligned}$$

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 *Proxy*, which serves the purpose of recording the witness information:

$$\textbf{data}\ Proxy\ a = Proxy$$

$$\begin{aligned} &\textbf{class}\ ResultE\ f\ r\ (w :: ResultWitness)\ \textbf{where} \\ &\quad mkConstE :: Proxy\ w \to r \to f \end{aligned}$$

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

$$\begin{aligned} &\textbf{instance}\ ResultE\ r\ r\ End\ \textbf{where} \\ &\quad mkConstE\ \_\ r = r \end{aligned}$$

$$\begin{aligned} &\textbf{instance}\ ResultE\ m\ r\ l \Rightarrow \\ &\qquad\qquad ResultE\ (s \to m)\ r\ (Step\ l)\ \textbf{where} \\ &\quad mkConstE\ \_\ r = \lambda s \to mkConstE\ (Proxy :: Proxy\ l)\ r \end{aligned}$$

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 *mkConstE* and provides it with the correct *Proxy*:

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

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 *Yes* result. If that was not the case, we could not declare a constraint such as $IsEq\ t \sim Yes$ 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 **dictionary** keyword to introduce it.[9] For example, the following declaration adds an *eq* function to the *IsEq* type family:

[9] 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 using the syntax *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 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:

```
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 is possible to drop the prefix 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 opens new possibilities for type families. It is also the only piece missing that we cannot directly encode in type families. Section 3 shows, though, that for the typing perspective our simulation can be encoded in *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 [19] 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 :+:, which combines constructors from different types, and a relation f :<: g, which specifies that the constructors in f are a subset of those in g. The relation :<: is defined as a type class, which provides a method to inject one type into the other:

```
inj :: f a → 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 [1] 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 :<: 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 ∼ Yes ⇒ t → ShowEverything t
  NoInst    ::                  t → ShowEverything t
```

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

```
showE :: ShowEverything t → String
showE (UsingInst x) = show x
showE (NoInst    x) = ""
```

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

```
showE :: ShowEverything t → ShowEverything t
showE (UsingInst x) = UsingInst x
showE (NoInst x)    = let type instance IsShow t = Yes
                            dictionary show x = ""
                          in UsingInst x
```

Now, when *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 the accompanying technical report [16] 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 *type class declaration* declares its name D and parameters $t_1, \ldots, t_m$, along with its superclasses $C_1, \ldots, C_k$ and a set of functional dependencies:

$$\textbf{class}\ C_1\ \overline{s}_1, \ldots, C_k\ \overline{s}_k \Rightarrow D\ t_1 \ldots t_m \mid \overline{u}_1 \to \overline{v}_1, \ldots, \overline{u}_n \to \overline{v}_n$$

where each $\overline{u}_1, \overline{v}_1 \ldots \overline{u}_n, \overline{v}_n$ 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:

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

Here, *Defined* is the kind which represents whether a type class instance is available. In addition, types t_1 to 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 *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, there 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

$$
\begin{array}{ll}
\textbf{type family} & f :<: g :: \mathit{Defined} \\
\textbf{dictionary} & \mathit{inj} :: f\ a \rightarrow g\ a \\
\textbf{where} & e :<: e \qquad\qquad = \mathit{Yes}\ \textbf{dictionary}\ \mathit{inj} = \mathit{id} \\
& f :<: (x :+: y) = d@(\mathit{Choose}\ f\ x\ y\ l@(f :<: x)\ r@(f :<: y)) \\
& \qquad\qquad\qquad\qquad \textbf{dictionary}\ \mathit{inj} = d@\mathit{choice}\ l@\mathit{inj}\ r@\mathit{inj} \\
& f :<: g \qquad\qquad = \mathit{No}
\end{array}
$$

$$
\begin{array}{ll}
\textbf{type family} & \mathit{Choose}\ f\ x\ y\ \mathit{fx}\ \mathit{fy} :: \mathit{Defined} \\
\textbf{dictionary} & \mathit{choice} :: (f\ a \rightarrow x\ a) \rightarrow (f\ a \rightarrow y\ a) \rightarrow f\ a \rightarrow (x :+: y)\ a \\
\textbf{where} & \mathit{Choose}\ f\ x\ y\ \mathit{Yes}\ \mathit{fy}\ \ = \mathit{Yes}\ \textbf{dictionary}\ \mathit{choice}\ x\ y = \mathit{Inl}\ \circ\ x \\
& \mathit{Choose}\ f\ x\ y\ \mathit{fx}\ \ \mathit{Yes} = \mathit{Yes}\ \textbf{dictionary}\ \mathit{choice}\ x\ y = \mathit{Inr}\ \circ\ y \\
& \mathit{Choose}\ f\ x\ y\ \mathit{fx}\ \ \mathit{fy}\ \ = \mathit{No}
\end{array}
$$

Figure 1. Subtyping for compositional data types

arguments determine some other left-hand side.[10] Using the syntax proposed in the aforementioned draft, the type family declaration needs to be changed to:

$$
\begin{array}{l}
\textbf{type family}\ \mathit{IsD}\ t_1\ ...\ t_m = (r :: \mathit{Defined}) \\
\qquad\qquad\qquad \mid r\ \overline{u}_1 \rightarrow \overline{v}_1, ..., r\ \overline{u}_n \rightarrow \overline{v}_n
\end{array}
$$

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:

$$
\begin{array}{l}
\textbf{type}\ \mathit{IsD}^{\uparrow}\ t_1\ ...\ t_m = (\mathit{IsD}\ t_1\ ...\ t_m \sim \mathit{Yes}, \\
\qquad\qquad\qquad\qquad\quad \mathit{IsC}_1^{\uparrow}\ \overline{s}_1 \sim \mathit{Yes}, ..., \mathit{IsC}_k^{\uparrow}\ \overline{s}_k \sim \mathit{Yes})
\end{array}
$$

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 signature

$$f :: D\ t_1\ ...\ t_m \Rightarrow r$$

is translated using the synonym as

$$f :: \mathit{IsD}^{\uparrow}\ t_1\ ...\ t_m \Rightarrow r$$

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 $\Rightarrow$) and a list of types to which the instance declaration applies (to the right of $\Rightarrow$):

$$\textbf{instance}\ (Q_1, ..., Q_n) \Rightarrow D\ t_1\ ...\ t_m$$

Each Q_i is of the form $E\ s_1\ ...\ s_k$ for some type class E. Each of the above type class instance declarations is translated into a type family instance of the following form:

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

where $Q_i' = \mathit{IsE}\ s_1\ ...\ s_k$, given that $Q_i = E\ s_1\ ...\ s_k$. Moreover, for each number n of context declarations, we have a corresponding closed type family And_n, which checks that all its arguments are Yes. More explicitly:

$$
\begin{array}{l}
\textbf{type family}\ \mathit{And}_0 :: \mathit{Defined} \\
\quad \mathit{And}_0 = \mathit{Yes}
\end{array}
$$

$$
\begin{array}{l}
\textbf{type family}\ \mathit{And}_1\ d :: \mathit{Defined} \\
\quad \mathit{And}_1\ x = x
\end{array}
$$

$$
\begin{array}{l}
\textbf{type family}\ \mathit{And}_n\ d_1\ ...\ d_n :: \mathit{Defined} \\
\quad \mathit{And}_n\ \mathit{Yes}\ ...\ \mathit{Yes} = \mathit{Yes}\quad \text{-- case everything Yes} \\
\quad \mathit{And}_n\ d_1\ \ ...\ d_n\ \ = \mathit{No}
\end{array}
$$

Examples. As a first example of this formal translation, let us consider the well-known type classes Eq and Ord along with some instance declarations and function type signatures.

$$
\begin{array}{l}
\textbf{class}\ \mathit{Eq}\ a \\
\textbf{instance}\ \mathit{Eq}\ \mathit{Int} \\
\textbf{instance}\ \mathit{Eq}\ a \Rightarrow \mathit{Eq}\ [a] \\
(\equiv) :: \mathit{Eq}\ a \Rightarrow a \rightarrow a \rightarrow \mathit{Bool}
\end{array}
$$

$$
\begin{array}{l}
\textbf{class}\ \mathit{Eq}\ a \Rightarrow \mathit{Ord}\ a \\
(>) :: \mathit{Ord}\ a \Rightarrow a \rightarrow a \rightarrow \mathit{Bool}
\end{array}
$$

The above declarations and type signatures are translated as follows:

$$
\begin{array}{l}
\textbf{type}\ \mathit{IsEq}^{\uparrow}\ a = \mathit{IsEq}\ a \sim \mathit{Yes} \\
\textbf{type family}\ \mathit{IsEq}\ a :: \mathit{Defined} \\
\textbf{type instance}\ \mathit{IsEq}\ \mathit{Int} = \mathit{Yes} \\
\textbf{type instance}\ \mathit{IsEq}\ [a] = \mathit{IsEq}\ a \\
(\equiv) :: \mathit{IsEq}^{\uparrow}\ a \Rightarrow a \rightarrow a \rightarrow \mathit{Bool}
\end{array}
$$

$$
\begin{array}{l}
\textbf{type}\ \mathit{IsOrd}^{\uparrow}\ a = (\mathit{IsOrd}\ a \sim \mathit{Yes}, \mathit{IsEq}^{\uparrow}\ a) \\
\textbf{type family}\ \mathit{IsOrd}\ a :: \mathit{Defined} \\
(>) :: \mathit{IsOrd}^{\uparrow}\ a \Rightarrow a \rightarrow a \rightarrow \mathit{Bool}
\end{array}
$$

The second example involves the $\mathit{Collection}$ type class which encodes the fact that a type c is a collection of elements of type e. This class is useful since not all collection types in Haskell are polymorphic like $[a]$ or $\mathit{Map}\ k\ v$, but only apply to a restricted set of types, like IntSet.

$$
\begin{array}{l}
\textbf{class}\ \mathit{Collection}\ c\ e \mid c \rightarrow e \\
\textbf{instance}\ \mathit{Collection}\ [a]\ a \\
\textbf{instance}\ \mathit{Ord}\ k \Rightarrow \mathit{Collection}\ (\mathit{Map}\ k\ v)\ v \\
\textbf{instance}\ \mathit{Collection}\ \mathit{IntSet}\ \mathit{Int}
\end{array}
$$

The above piece of code is translated as follows:

$$
\begin{array}{l}
\textbf{type}\ \mathit{IsCollection}^{\uparrow}\ c\ e = \mathit{IsCollection}\ c\ e \sim \mathit{Yes} \\
\textbf{type family}\ \mathit{IsCollection}\ c\ e :: (r :: \mathit{Defined}) \mid r\ c \rightarrow e \\
\textbf{type instance}\ \mathit{Collection}\ [a]\qquad\quad a\quad = \mathit{Yes} \\
\textbf{type instance}\ \mathit{Collection}\ (\mathit{Map}\ k\ v)\ v\ \ = \mathit{IsOrd}\ k \\
\textbf{type instance}\ \mathit{Collection}\ \mathit{IntSet}\qquad \mathit{Int} = \mathit{Yes}
\end{array}
$$

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

[10] This is called injectivity of type C in the draft, and it is not expected to be implemented in the short term.

build upon the OUTSIDEIN(X) framework [21], which underlies the GHC compiler from version 7, and which we describe thoroughly in the accompanying technical report [16]. Note that type class resolution and type family rewriting have been described within this framework, but without support for functional dependencies or injectivity. The technical report also includes our formalization of these concepts within OUTSIDEIN(X), based on the description as Constraint Handling Rules by Sulzmann et al. [18].

In OUTSIDEIN(X), typing depends on a concrete entailment judgment $\mathcal{Q} \Vdash Q$, which symbolizes that under axioms $\mathcal{Q}$, 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 $D\ \tau_1\ \dots\ \tau_n$. Our theorems are true with respect to that concrete entailment.

Theorem 1 (Soundness). *Let T be the derivation tree of $\mathcal{Q}^{trans} \Vdash \tau_1 \sim \tau_2$. Suppose that for each application of injectivity over a type family $IsD\ \overline{\tau}$ there exists a subtree that proves either $IsD\ \overline{\tau} \sim Yes$ or $Yes \sim IsD\ \overline{\tau}$. Then:*

- *If τ_1 and τ_2 are not of kind $Defined$, then $\mathcal{Q} \Vdash \tau_1 \sim \tau_2$.*
- *If $\tau_1 \equiv IsD\ \overline{\tau}$ and $\tau_2 \equiv Yes$ or vice versa, then $\mathcal{Q} \Vdash D\ \overline{\tau}$.*

Proof. See accompanying technical report [16]. □

This soundness result states that whenever we can derive $IsD\ \overline{\tau} \sim Yes$ in our translation, then we can derive $D\ \overline{\tau}$ 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 $IsD\ \overline{\tau} \sim Yes$. 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 No.

Theorem 2 (Completeness).

- *If $\mathcal{Q} \Vdash \tau_1 \sim \tau_2$, then $\mathcal{Q}^{trans} \Vdash \tau_1 \sim \tau_2$.*
- *If $\mathcal{Q} \Vdash D\ \overline{\tau}$, then $\mathcal{Q}^{trans} \Vdash IsD\ \overline{\tau} \sim Yes$.*

Proof. See accompanying technical report [16]. □

The completeness result is straightforward: our translation preserves all original type equalities, and any instance $D\ \overline{\tau}$ that is derivable in the original system, is derivable as $IsD\ \overline{\tau} \sim Yes$ 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*[11], which will serve as basis to prove termination in our setting. The Paterson conditions state that for each constraint $Q\ s_1 \dots s_j$ 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 $F\ t_1 \dots t_m = s$. In this case, the conditions require that for each type family application $G\ r_1 \dots r_k$ appearing in s, we have:

1. None of the arguments $r_1 \dots r_k$ contains any other type family applications.
2. The total number of data type constructors and variables in $r_1 \dots r_k$ is strictly smaller than in $t_1 \dots t_m$.
3. Each variable occurs in $r_1 \dots r_k$ at most as often as in $t_1 \dots t_m$.

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

$$\textbf{type instance}\ IsD\ t_1 \dots t_m = And_n\ Q'_1 \dots Q'_n$$

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 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.

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 3.2. 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 [4, 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 5.2 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:

$$myFunction : \{A : Set\} \to \{\{p : Show\ A\}\} \to A \to String$$

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

$showInt : Show\ Int$
$showInt\ x = \dots$ -- 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

[11] 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 [20].

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 $\mathit{Show}\ a$:

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

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:

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

The search strategy is defined by $\mathit{findShow}$, 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, $\mathit{findShow}$ 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 idioms 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 on "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

[1] P. Bahr. Composing and Decomposing Data Types: A Closed Type Families Implementation of Data Types à la Carte. In *Proceedings of the 10th ACM SIGPLAN Workshop on Generic Programming*, pages 71–82, New York, NY, USA, Aug. 2014. ACM.

[2] E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, 9 2013.

[3] 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*, pages 143–155, New York, NY, USA, 2011. ACM.

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

[5] A. Dijkstra, G. van den Geest, B. Heeren, and S. D. Swierstra. Modelling Scoped Instances with Constraint Handling Rules. Technical report, Department of Information and Computing Sciences, Utrecht University, 2007.

[6] 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*, pages 671–683, New York, NY, USA, 2014. ACM.

[7] J. Hage. DOMain Specific Type Error Diagnosis (DOMSTED). Technical Report UU-CS-2014-019, Department of Information and Computing Sciences, Utrecht University, 2014.

[8] B. Heeren and J. Hage. Type class directives. In *Proceedings of the 7th International Conference on Practical Aspects of Declarative Languages*, pages 253–267, Berlin, Heidelberg, 2005. Springer-Verlag.

[9] M. Izbicki. A neat trick for partially closed type families. Available at `http://izbicki.me/blog/a-neat-trick-for-partially-closed-type-families`, 2014.

[10] 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.

[11] O. Kiselyov. Stretching type classes. Retrieved from `http://okmij.org/ftp/Haskell/TypeClass.html`, May 2015.

[12] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, pages 96–107, New York, NY, USA, 2004. ACM.

[13] 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.

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

[15] T. Schrijvers, M. Sulzmann, S. Peyton Jones, and M. Chakravarty. Towards open type functions for Haskell. In O. Chitil, editor, *19th International Symposium on Implementation and Application of Functional Languages*, pages 233–251. Computing Laboratory, University of Kent, 2007.

[16] A. Serrano, J. Hage, and P. Bahr. Type Families with Class, Type Classes with Family. Technical report. To appear.

[17] P. J. Stuckey and M. Sulzmann. A theory of overloading. *ACM Trans. Program. Lang. Syst.*, 27(6):1216–1269, Nov. 2005. ISSN 0164-0925.

[18] M. Sulzmann, G. J. Duck, S. Peyton-Jones, and P. J. Stuckey. Understanding functional dependencies via constraint handling rules. *J. Funct. Program.*, 17(1):83–129, Jan. 2007. ISSN 0956-7968.

[19] W. Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436, July 2008.

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

[21] D. Vytiniotis, S. Peyton Jones, T. Schrijvers, and M. Sulzmann. OutsideIn(X): Modular Type Inference with Local Assumptions. *Journal of Functional Programming*, 21(4-5):333–412, Sept. 2011.

[22] 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*, pages 60–76, New York, NY, USA, 1989. ACM.

[23] 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*, pages 53–66, New York, NY, USA, 2012. ACM.