# Dialects of Type Computations in Haskell

APOORV INGLE, University of Iowa, USA

Static types have two advantages: (1) they serve as a guiding tool to help programmers write correct code, and (2) the typechecker can help identify code that does not behave correctly. An expressive type system can guarantee stronger claims about programs. Type level computations make the type system more expressive. In Haskell, there are two styles of type level computation—functional dependencies and type families. In this report we describe these two language features with examples, formalize and compare them.

## 1 INTRODUCTION

Parametric polymorphism is a powerful technique that allows programs to work on a wide variety of types. The identity function, `id`, that takes an input and returns it without modification has the type $\forall\alpha.\ \alpha \rightarrow \alpha$, We read this type as follows: for all types, $\alpha$, if the argument is of type $\alpha$ then the function returns a value of type $\alpha$. We also need to tame unconstrained polymorphism. A division function on all types does not make sense. We cannot divide a function that multiplies two numbers by a function that adds two numbers. The type $\forall\ \alpha.\ (\alpha \times \alpha) \rightarrow \alpha$, that accepts a pair of values of type $\alpha$, and returns the first component divided by the second component is too general to describe division. A constrained polymorphic type, $\forall\ \alpha.\ (\texttt{Dividable}\ \alpha) \Rightarrow (\alpha \times \alpha) \rightarrow \alpha$, more accurately describes the functions intention. Intuitively, the predicate, $\texttt{Dividable}\ \alpha$, means: only those types that satisfy this predicate have a meaningful divide function. Typeclasses[Wadler and Blott 1989] give a mechanism of having such constrained polymorphic types. Theory of qualified types[Jones 1994] formalizes typeclasses and justifies constrained polymorphism without compromising type safety by having predicates as a part of type syntax.

A typeclass also defines relations on types. This gives programmers a way to encode computations at type level. However, using relations to encode type computations is cumbersome. A new language feature, type families[Schrijvers et al. 2007], was introduced in Haskell to enable type functions. They are stylistically more obvious for functional programmers. Naturally, type families warrants a richer system of types and ensuring type safety for such a language is nontrivial.

**TODO: :** rework this at the end

The scope of the current article is as follows: we first give examples and intuitive set semantics for typeclasses in the beginning of Section 2, and then describe functional dependencies[Jones 2000] with some examples in Section 3. We formalize them in Section 3.1 and describe their consequences in Section 3.2. We also give a brief description of type safety for this system in Section **??**. We then describe two flavors of type families—closed type families[Eisenberg et al. 2014] in Section 5

Author's address: Apoorv Ingle, University of Iowa, Department of Computer Science, McLean Hall, Iowa City, Iowa, USA.

and constrained type families[Morris and Eisenberg 2017] in Section 6 with examples and their respective formalization in Section 5.2 and Section 6.3. We then give some details about the type safety of each system in Section ?? and Section ?? respectively. To conclude we draw some comparisons between the three systems while pointing towards some open questions in Section 7. To be concrete about the examples we will use a Haskell like syntax.

## 2 TYPECLASSES

Typeclasses can be thought of as collection of types. Each typeclass is accompanied by its member functions that all the instances ought to support. For example, equality can be expressed as a typeclass `Eq a` as follows:

```
class Eq a where                  instance Eq Int where           instance Eq Char where
  (==) :: a → a → Bool              a == b = primEQInt a b          a == b = primEQChar a b
```

The instances of `Eq` typeclass can be types such as integers (`Int`) and characters (`Char`) but defining equality on function types (a→b) is not meaningful. The operator `(==)` is not truly polymorphic: it cannot operate on function types. Rather it is constrained to only those types that have an `Eq` instance defined. We make this explicit in the type of the operator `(==)`, by saying it's defined only on those types `a` that satisfy the `Eq a` predicate, or `(==) :: ∀a. Eq a ⇒ a → a → Bool`. The reading for this type is: for any type `a` that satisfies the predicate `Eq a`, if we are given two values of type `a`, then we can return a Boolean value indicating if the two arguments are equal.

There is nothing special about typeclasses having just one type parameter. A multiparameter typeclass with $n$ type parameters represents a relation on $n$ types. An example of such a typeclass, `Add a b c` is shown in Figure 1. It represents a relation of types `a`, `b` and `c` such that adding values of type `a` and type `b` gives us a value of type `c`. The type of the operator, `(+)`, that performs this add operation will be `∀a b c. Add a b c ⇒ a → b → c`. Instances of such a typeclass would be `Add Int Int Int`, `Add Int Float Float`, and so on.

```
class Add m n p where    instance Add Int Float Float where    instance Add Int Int Int
  (+) :: m → n → p         (+) a b = addFloat (toFloat a) b      (+) a b = intAdd a b

instance Add Int Float Int where                 expr :: (Add Int Float b, Add b Int c) ⇒ c
  (+) a b = addInt a (toInt b)                   expr = (1 + 2.5) + 3
```

Fig. 1. Multiparameter Typeclasses

Multiparameter typeclasses, however, are difficult to use in practice. Suppose the programmer defines two instances: `Add Int Float Float` and `Add Int Float Int` and writes an term `expr = (1 + 2.5) + 3`. Due to the use of the operator `(+)` in `expr`, its most general type synthesized by the type inference algorithm will be `∀ b c. (Add Int Float b, Add b Int c) ⇒ c`. Notice how the type variable `b` occurs only in the predicate set `(Add Int Float b, Add b Int c)`. Such types are called ambiguous types and the type variables, such as `b`, are called ambiguous type variables. Ambiguous types do not have well defined semantics in Haskell. The compiler cannot choose a unique interpretation of the subterm `(1 + 2.5)` as it can very well be of the value 4 of type `Int` or the value `2.5` of type `Float`. Haskell, thus, disallows ambiguous types by reporting a type error. However, the type errors can cause confusion; although the issue can be attributed to conflicting instances of typeclass `Add m n p`, the type error is raised at the term which may be defined in a different location.

## 3 FUNCTIONAL DEPENDENCIES WITH EXAMPLES

Typeclasses with functional dependencies[Jones 2000] is a generalization of multiparameter typeclasses. It introduces a new syntax where the user can specify a dependency between the type parameters in the typeclass declaration. There is no change in the syntax of declaring instances. `Add m n p` typeclass, as shown in Figure 2, now has a functional dependency between the type parameters such that types `m` and `n` determine the type `p`. In general we can have multiple parameters on both sides of the arrow, $(x_1, \ldots, x_m \to y_1, \ldots, y_m)$. We write `X → Y` to mean “the parameters `X` uniquely determine the parameters `Y`”.

```
class Add m n p | m n → p where   instance Add Int Float Float where
  (+) :: m → n → p                  ...
                                  instance Add Int Float Int where -- Error!
                                    ...
```

Fig. 2. `Add m n p` with Functional Dependency and Conflicting Instances

The programmer can use functional dependencies to specify the intention of the typeclasses more accurately, and in turn it gives the compiler a way to detect and report inconsistent instances. For example, the functional dependency on `m n → p` can now help the typechecker flag the instance `Add Int Float Int` to be in conflict with the instance `Add Int Float Float` which was indeed the real intention of the typeclass.

Further, we may also be able to determine the ambiguous type variables using the functional dependencies. Let’s reconsider the ambiguous type of term `e` from the previous section, we can now determine that `b` has to be `Float`. It is determined by the types `Int` and `Float` of the class instance. Thus, `expr :: (Add Int Float Float, Add Float Int c) ⇒ c`. We can even go a step further and improve this seemingly polymorphic type. The type variable `c` can be determined to be `Float`, giving us the type `expr :: Float`. It would be impossible to make such an improvement without the functional dependency.

```
data Z   -- Type level Zero
data S n -- Type level Successor

class Plus m n p | m n → p
instance IsPeano m ⇒ Plus Z m m
instance Plus n m p ⇒ Plus (S n) m (S p)
```

```
class IsPeano c
instance IsPeano Z
instance IsPeano n ⇒ IsPeano (S n)
data Vector s e = Vec (List e)
concat_vec :: Plus m n p
     ⇒ Vector m e → Vector n e → Vector p e
concat_vec (Vec l1) (Vec l2) = Vec (append l1 l2)
```

Fig. 3. Peano Arithmetic and Vector Operations with Functional Dependencies

With functional dependencies at our disposal, we can even perform Peano arithmetic at type level, as shown in Figure 3. The two datatypes `Z` and `S n` represent the number zero and successor of a number `n`, respectively. The instances of `IsPeano` assert that: `Z` is a peano number, and if `n` is a peano number then `S n` is a Peano number. The instances of `Plus` typeclass relates three peano numbers such that the relation holds if the first two peano numbers add up to be equal to the third. Thus, `Peano Z m m` asserts the relation `0 + m = m`, and `Plus n m p ⇒ Plus (S n) m (S p)` asserts that if n + m = p then (1 + n) + m = (1 + p). The `concat_vec` function demonstrates why type level computation would be useful for a linear algebra library. The type of `concat_vec` says that the size of the resulting vector is the sum of the sizes of the argument vectors. The above examples are meant to demonstrate how functional dependencies help in encoding type level functions that multiparameter type classes cannot not due to absence of the capability to restrict the class relation that the programmer needs.

## 3.1 Formalizing Typeclasses with Functional Dependencies

In previous sections we made a case for why ambiguous types are problematic and how functional dependencies can help us solve it. We now formalize this intuition to make our claim precise. We organize the our language as shown in Figure 4 and call it System **TCFD**. The types ($\tau$) consist of type variables ($\alpha$), functions ($\tau \longrightarrow \tau$), and type constants ($\mathsf{T}$) such as `Int`, `Float` etc. The qualified types ($\rho$) are $P \Rightarrow \rho$ where $P$ constrains the type $\rho$. Type schemes ($\sigma$) are quantified constraint types. The terms or expressions in the language ($e$) consists term level variables ($x$, $y$), functions ($\lambda x{:}\tau.\ e$), function applications ($e_1\ e_2$).

*3.1.1 Notations.* We will use notations as follows. Subscripts on objects ($\alpha_1, \ldots, \alpha_n$) are used to distinguish them. A collection of $\alpha_1, \alpha_2, \ldots, \alpha_n$ items of arbitrary length is written as $\overline{\alpha}$ We use $S_1 \backslash S_2$ to denote the set difference operation. For an object $X$, $TV(X)$ is the set of type variables that are free in $X$. We write $[\overline{x} \mapsto \overline{y}]e$ to denote the substitution where each variable $x_i$ is mapped to $y_i$

| | | | | |
|---|---|---|---|---|
| | | Predicates | $\pi$ | $::= C\overline{\tau}$ |
| Type Variables | $\alpha, \beta$ | Predicate Set | $P, Q$ | $::= \overline{\pi}$ |
| Term Variables | $x, y$ | Types | $\tau$ | $::= \alpha \mid \tau \to \tau \mid \mathtt{T}$ |
| Class Constructors | $\mathtt{C}$ | Qualified Types | $\rho$ | $::= \tau \mid P \Rightarrow \tau$ |
| Type Constants | $\mathtt{T}$ | Type Schemes | $\sigma$ | $::= \forall\overline{\alpha}.\rho$ |
| | | Terms | $e$ | $::= x \mid e\, e \mid \lambda x{:}\tau.\, e$ |
| Term Typing | $P \mid \Gamma \vdash e : \tau$ | Values | $v$ | $::= \lambda x{:}\tau.\, e$ |
| | | Typing Environment | $\Gamma$ | $::= \epsilon \mid \Gamma, x{:}\sigma$ |

Fig. 4. System **TCFD**

in $e$. Alternatively, we also write $\Omega X$ for an substitution $\Omega$ applied to object $X$. We denote the most general unifier for two types $\tau_1$ and $\tau_2$ (if it exists), by $mgu(\tau_1, \tau_2)$[Robinson 1965]. We write $mgu(\overline{\tau_1}, \overline{\tau_2})$ to give us a composition of most general unifier for each pair of types $(\tau_{1i}, \tau_{2i})$.

For a typeclass declaration we write **`class`** $P \Rightarrow$ `C t`, where `t` are the type parameters of the class and $P$ are the constraints that must be satisfied, and for an instance of typeclass `C`, we write **`instance`** $P \Rightarrow$ `C t`, where length of `t` matches the typeclass arity with additional constraints on `t` are introduced by $P$. We denote the set of functional dependencies of class `C` with $\mathtt{FD}_{\mathtt{C}}$. For an arbitrary functional dependency $X \to Y$ the determinant of a functional dependency is denoted by $t_X$ and the dependent is denoted by $t_Y$. For example, for a typeclass declaration **`class`** `Add m n p | m n` $\to$ `p`, we have, $t = (m, n, p)$, $\mathtt{FD}_{\mathtt{Add}} = \{$ `m n` $\to$ `p` $\}$. For the functional dependency `m n` $\to$ `p`, we have, $t_X = (\mathtt{m}, \mathtt{n})$ and $t_Y = (\mathtt{p})$. Given a set of functional dependencies `J`, we define the closure operation, $Z^+_{\mathtt{J}}$, on $Z \subseteq t$, to be equal to all the type parameters that are determined by set of the functional dependencies `J`. Thus, $\{m\}^+_{\mathtt{FD}_{\mathtt{Add}}} = \{m\}$ and $\{m, n\}^+_{\mathtt{FD}_{\mathtt{Add}}} = \{m, n, p\}$.

*3.1.2 Type System.* The typing environment $\Gamma$ is a mapping between term variables to types such that a term variable appears at most once. We write $dom(\Gamma)$ to mean the set of all term variables in $\Gamma$ i.e. $\{x \mid (x \mapsto \tau) \in \Gamma\}$. We denote $\Gamma_x$ to be $\Gamma$ obtained after removing the binding for $x$ (if it existed) in $\Gamma$. The typing judgements are of the form $P \mid \Gamma \vdash e : \sigma$. They assert existence of a typing derivation that shows $e$ has type $\sigma$ with predicates $P$ being satisfied and the free variables in $e$ are given types by the typing environment $\Gamma$. The typing rules that build these typing derivations in our system are shown in Figure 5 that we will go over next.

The left column lists the standard term typing rules. Each term is associated with one typing rule that specifies when the term is well typed. The rule (VAR) says that if the variable $x$ has type $\sigma$ then typing environment $\Gamma$ should contain that binding to confirming. The rule ($\to I$) says that if a term, $e$, has type $\tau_2$ with a free variable $x$ of type $\tau_1$ then the term $\lambda x{:}\tau_1.\, e$ has a type $\tau_1 \to \tau_2$.

$$(\textsc{var})\ \frac{x{:}\sigma \in \Gamma}{P \mid \Gamma \vdash x : \sigma} \qquad (\forall\, I)\ \frac{P \mid \Gamma \vdash e : \rho \quad \alpha \notin TV(\Gamma) \cup TV(P)}{P \mid \Gamma \vdash e : \forall\alpha.\, \rho}$$

$$(\to I)\ \frac{P \mid \Gamma_x, x{:}\tau_1 \vdash e : \tau_2}{P \mid \Gamma \vdash \lambda x{:}\tau_1.\, e : \tau_1 \to \tau_2} \qquad (\forall\, E)\ \frac{P \mid \Gamma \vdash e : \forall\alpha.\, \rho \quad \beta \text{ fresh}}{P \mid \Gamma \vdash e : [\alpha \mapsto \beta]\rho}$$

$$(\to E)\ \frac{P \mid \Gamma \vdash e_1 : \tau_2 \to \tau \quad P \mid \Gamma \vdash e_2 : \tau_2}{P \mid \Gamma \vdash e_1\, e_2 : \tau} \qquad (\Rightarrow I)\ \frac{P, Q \mid \Gamma \vdash e : \rho}{P \mid \Gamma \vdash e : Q \Rightarrow \rho}$$

$$(\Rightarrow E)\ \frac{P \mid \Gamma \vdash e : Q \Rightarrow \rho \quad P \Vdash Q}{P \mid \Gamma \vdash e : \rho}$$

Fig. 5. Typing judgments for System **TCFD** Terms

The rule $(\to E)$ justifies function application that says if a term (function) $e_1$ is of type $\tau_1 \to \tau_2$ and another term (argument) $e_2$ is of type $\tau_2$, then the term $e_1\ e_2$ is of type $\tau_2$.

The right column contains rules that involve the predicate set of the typing judgement. The rule $(\forall\, I)$ generalizes the type and the rule $(\forall\, E)$ instantiates it. It is necessary for $\beta$ to be a fresh type variable to ensure it doesn't clash with existing free type variables. The rule $(\Rightarrow I)$ moves the global predicate $Q$ in to constrain the type $\rho$ while the $(\Rightarrow E)$ moves the constraint out of the type $\rho$. The condition $P \Vdash Q$, read as "$P$ entails $Q$", means that whenever $Q$ is satisfied $P$ is also satisfied.

**TODO: :** entailment relation properties

## 3.2 Improving Substitution and Ambiguous Types

The typing rules shown in Figure 5 does not included the use of functional dependencies. We will now formalize it using an extra typing rule (impr) that we motivate below.

$$(\textsc{impr})\ \frac{P \mid \Gamma \vdash e : \rho \quad \Omega = impr(P)}{\Omega P \mid \Omega\Gamma \vdash e : \Omega\rho}$$

An improving substitution, written as $impr(P)$, is a substitution that does not change the set of satisfiable instances of predicate set $P$. The rational behind improving substitution is that it helps simplifying the type by showing its true and concise characterization. Computing an improving substitution in our case is to find if any of the type variables can be determined using the functional dependency given in the predicate set. For each functional dependency, `X` $\to$ `Y`, induced by the predicate set, $P$, or $(X \to Y) \in \mathsf{FD}_\mathsf{P}$, for a particular class `C t` $\in P$, whenever we know $TV(t_X)$, then we can determine $TV(t_Y)$. For example, consider the type of `expr :: (Add Int Float b, Add b Float c)` $\Rightarrow$ `c`. The improvement is done using the functional dependency $\mathsf{FD}_\mathsf{Add} = \{$`m n` $\to$ `p`$\}$ to obtain an improving substitution $[b \mapsto$ `Float`$, c \mapsto$ `Float`$]$.

We can also modify detection of ambiguous types. For a qualified type, $\forall\overline{\alpha}.P \Rightarrow \tau$, the usual ambiguity check is $(\overline{\alpha} \cap TV(P)) \subseteq TV(\tau)$. However, with induced functional dependencies $\mathsf{FD_P}$ due to $P$, the appropriate check would be $(\overline{\alpha} \cap TV(P)) \subseteq TV(\tau)^{+}_{\mathsf{FD_P}}$. We thus weaken the check as there can be some type variables which are determined by the functional dependencies induced by the class constraints.

## 3.3 Instance Validity and Inconsistency Detection

Every typeclass introduces a new realtion on types. With functional dependencies, we have additional constraints which every instances should satisfy. We need to ensure that the instances declared are compatible with the functional dependencies associated with the typeclass. There are two necessary conditions to ensure this:

(1) *Covering Condition*: For each new instance declaration **instance** $P$ ⇒ `C t` **where** ... that the programmer writes, we need to check that,$TV(t_Y) \subseteq TV(t_X)^{+}_{\mathsf{FD_{P,C}}}$, where $\mathsf{FD_{P,C}}$ are the functional dependencies of $C$ and additional dependencies induced by the instance context $P$. Intuitively, this condition says that all the type variables of the dependent, $TV(t_Y)$, should either already be in the set of determinant type variables, $TV(t_X)$, or they should be fully determined using the functional dependencies induced by the class ($\mathsf{FD_C}$) or the functional dependencies induced by the constraints ($\mathsf{FD_P}$). For example, for a typeclass declaration **class** `C a b | a → b` the instance declaration **instance** `C Int a` fails the coverage test, while **instance** `C a Int` passes it.

(2) *Consistency Condition*: For each new instance of the form **instance** $Q$ ⇒ `C s` **where** ... along with **instance** $P$ ⇒ `C t` **where** ... we need to ensure whenever $t_Y = s_Y$ we also have $t_X = s_X$. It is straightforward to check this condition. We first find the most general unifier for $t_X$ and $s_X$, say $U = mgu(t_X, s_X)$, and then check that $Ut_Y = Us_Y$. If we cannot find such a unifier, then we know that the instances are consistent. For example, **instance** `C1 Int a` is consistent with **instance** `C1 Char a` as there is no unifier for `Int` and `Char`. However, **instance** `Add Int Float Float` and **instance** `Add Int Float Int` are inconsistent.

## 4 ASSOCIATED TYPES

Another way to express type computation is by having the typeclass with a special associated type[Chakravarty et al. 2005]. In this style, each instance of the typeclass specifies how the associated type should be interpreted. For example, Peano arithmetic can be expressed using `PlusC m n` typeclass with an associated type `Plus m n` as shown in Figure 6.

The type `Plus m n` has two interpretations due to the two typeclass instances, `Plus Z n = Z` means $0 + n = 0$, while `Plus (S m) n = S (Plus m n)` means $(1 + m) + n = 1 + (m + n)$. The associated type can be viewed as a type function; depending on the use context one of the

```
class PlusC m n where                    instance PlusC m n ⇒ PlusC (S m) n where
  type Plus m n                            type Plus (S m) n = S (Plus m n)

instance PlusC Z n where                 concat_vec :: Vec m e → Vec n e
  type Plus Z n = Z                                → Vec (Plus n m) e
```

Fig. 6. Plus as an Associated Type

interpretations would be picked. The type of `concat_vec` changes as `Plus n m` is now a type. We no longer specify it in the predicate set of the type as it does not have a relational style reading. As visible from the code snippet, attractiveness of this style is that type computation is no longer limited in the predicates.

An enthusastic programmer might now try write type equality using a typeclasses with an associated type as shown below. We use `TT` and `FF` to represent type level true and false respectively while the associated type `TEq a b` will be used to compute if the two types `a` and `b` are equal.

```
class TEqC a b where          instance TEqC a a where          instance TEqC a b where -- Error
  type TEq a b                  type TEq a a = TT                type TEq a b = FF
```

To the programmers disappointment this definition is rejected by the type system. The presense of both instances `TEqC a a` and `TEqC a b` is conflicting due to their overlap. Two instances overlap when both instances provide a match while resolving a constraint. For example, both the instances `TEqC a a` and `TEqC a b` can be used to match the constraint `TEqC Int Int`. There is no way for the compiler to choose one instance over the other. Haskell rejects programs that have overlapping instances as it assumes that there can only be a unique typeclass instance that matches the constraint.

## 5 CLOSED TYPE FAMILIES

Closed type family is a generalization of associated typeclasses. We notice from the above `PlusC m n` typelcass example that we could separate out the associated type `Plus m n` into its own entity and collect each of its type equations defined by the instance together as shown below.

```
type family Plus n m where
  Plus Z m = m
  Plus (S n) m = S (Plus n m)
```

We call this declaration a type family as it can be thought of as a family of types indexed by the type parameters. In our case, the type family `Plus m n` has two equations, each corresponding to the two, previously discussed, instances of the typeclass. This style of writing type functions is palatable for functional programmers. It resembles a familiar notion of writing term level equations with pattern matching. The `Add m n p` typeclass defined in Figure 2 can also be written in a closed

```
type family Result m n where
  Result Int Int = Int
  Result a Float = Float
  Result Float a = Float

class Add m n where
  (+) :: m → n → Result m n

instance Add Int Int where
  (+) = intAdd
```

```
instance Add Int Float where
  i + f = addFloat (int2Float i) f

instance Add Float Int where
  f + i = addFloat (int2Float i) f

instance Add Int Float where -- Error
  i + f = addInt i (float2int f)
```

Fig. 7. Add Typeclass using Closed Type Family

type family style as shown in Figure 7. The change is that the new `Add` typeclass takes only two parameters in this setting while the result type of (+) function now returns a special type `Result m n`. This `Result m n` type is defined for each instance we expect the typeclass `Add` to be defined at. The second typeclass instance `Add Int Float` would raise a type error due to its overlap with the first typeclass instance. Even if the compiler did accept overlapping instances, the expected type of the operator (+) will not match the type of the term; the type `Result Int Float` always reduces to type `Float`.

## 5.1 Type Matching, Apartness and Reduction

The role of type famlies is to compute the representation type by instantiating the type indices during type checking phase. In the `Add` example, the representation type of `Result Int Float` is `Float` while, the representation type of `Plus Z m` is `Z`. The semantics of equations has certain subtleties. Reduction an occurance of a type family in a type signature is done using a top to bottom matching technique. The first equation that matches is used, or fired, for reduction. For example, the type `Add Float Float` matches with the second equation `Add a Float`—`Int` and `Float` don't match—to reduce to `Float`. The substitution $[a \mapsto \texttt{Float}]$, which does the trick, can be computed using the most general unifier function $mgu(a, \texttt{Float})$.

```
type family TEq a b where
  TEq a a = TT
  TEq a b = FF
```

The real advantage of closed type families over associated types, is the ability to express type functions that were previously not possible such as `TEq a b` as shown above. Fixing the order of equation matching liberates us from the restriction of having non-overlapping equations; the type `TEq Int Int` will be matched with the first equation.

Now, let us consider the function `funTrick`, as shown in Figure 8. The type `Tricky a` matches the second equation and reduces to `Bool`. This reduction will, however, can crash the program, 💣, which uses `funTrick` and instantiates `d` to `Bool`, and `Tricky Bool` in turn reduces to `Int → Int`. Type soundness bugs, which cause runtime crashes, can be introduced while using overlapping equations and a naïve type reduction strategy. We need to identify the necessary conditions to use type reductions and ensure type soudness.

```
type family Tricky a where   funTrick :: a → Tricky a    💣 :: Int
  Tricky Bool = Int → Int    funTrick _ = True           💣 = funTrick True 5
  Tricky a = Bool
```

Fig. 8. Unsoundess due to Tricky Type Family

Due to term rewriting systems literature[Bezem et al. 2003], a necessary condition for type soundness with type reduction is to have confluence. If a type has multiple ways of reducing, it should not matter what way we choose, we should obtain the same final type. Indeed, in the above example, we did see a consequence of non-confluence, one way of reduction gave us `Int → Int` while the the other gave us `Bool`. We fix this behavior by incorporating flattening and apartness into the type reduction criteron. First we fix our notion of matching:

**Definition 1** (Matching)**.** We say a type $\tau$ matches $\sigma$ if there is a substitution $\Omega$ such that $\Omega\tau \sim \sigma$ with $dom(\Omega) \subseteq TV(\tau)$.

We use $\tau_1 \sim \tau_2$ to remind us that $\tau_1$ and $\tau_2$ are equal in the sense of propositional equality rather than definitional equality.

**Definition 2** (Type Flattening)**.** We say a type $\tau$ is flattened to $\tau_1$, or $\tau_1 = \mathtt{flatten}(\tau)$, when, every type family application of the from $\mathsf{F}(\overline{\sigma})$ in $\tau$, if $\sigma$ of the form $\mathsf{F}'(\overline{\sigma}')$ it is replaced by a type variable, such that in the flattened type, every syntactically equivalent type family application in $\tau$

is replaced by same type fresh type variable and syntactically different type family applications in $\tau$ are replaced by distinct fresh type variables.

**Definition 3** (Apartness)**.** For an equation, say $p$, and a type, say $\tau$, $p$ is apart from $\tau$, if or we cannot find a most general unifier that unifies both the left hand side of the equation and the flattened type of $\tau$, or $mgu(\text{lhs}_p, \texttt{flatten}(\tau))$ fails.

Finally, the equation selection criterion for type reduction can be defined follows:

**Definition 4** (Closed Type Family Reduction Criteron)**.** An equation, say $p$, given in the type family declaration can be used to simplify the type $\texttt{F}(\overline{\tau})$, if the following two conditions hold:

(1) The left hand side of the equation, $\text{lhs}_p$ matches the the type $\texttt{F}(\overline{\tau})$ or $\texttt{match}(\text{lhs}_p, \texttt{F}(\overline{\tau}))$.
(2) All equations, $q$, preceding $p$, are apart from $\texttt{F}(\overline{\tau})$, or $\texttt{apart}(q, \texttt{F}(\overline{\tau}))$.

Let us reconsider the type `a → Tricky a`, with the above definition of type familiy reduction. Now, `Tricky Bool` does not match `Tricky a`, as there is no substitution, $\Omega$, such that $\Omega$`Bool` $\sim$ `Int`. However, `Tricky Bool` is not apart from `Tricky d`—the substitution $[d \mapsto \texttt{Bool}]$ does unify them, hence we cannot use the second equation for type reduction; the second criteron from Definition 4 is not satisfied.

```
type family G a b where      type family F a where       G (F a) (F a) ⇝ Bool
  G Int Bool = Int             F Int = Char                G (F a) (F b) ⇝̸ Bool
  G a a = Bool                 F a = Bool
```

Fig. 9. Type family Reduction Example

Type flattening with sharing is necessary in apartness check to ensure type soundess. For example, consider the type families shown in Figure 9, if we have a type `G (F a) (F a)`, flattening the type would give us `G c c`, and we will be able to match it with the second equation `G a a` and it will be apart from `G Int Bool`. The type `G (F a) (F b)`, however, after flattening becomes `G c d` and it is not apart from `G Int Bool`.

Simplification of type families by using apartness criterion will ensure soundness but it has two shortcomings: 1) it is overly restrictive, and 2) it is inefficient due to the calls to unification. We will illustrate them by using a couple examples: Consider second and third equations from `Add` type family, `Add a Float = Float` and `Add Float a = Float`, and the type `Add Float b`, which we wish to reduce. In this case, reduction of `Add Float b` to `Float` is not possible as the second equation is not apart from `Add Float b`. But we know that both the equations reduce to the same final type, which is `Float`. We call such equations conincident equations. We would like to allow such reductions as they will not threaten soundness. Next, consider the equations of the `G a b` type family, `G Int Bool` and `G a a`. It is easy to see that if any type matches `G a a` it will always be apart from `G Int Bool`, and we can thus skip on the apartness check in such cases.

We can overcome on both the described shortcomings by using the notion of compatibility of equations defined below in our reduction criteron.

**Definition 5** (Compatible Equations)**.** Two equations, $p$ and $q$, are compatible if and only if there exist two substitutions, say $\Omega_1$ and $\Omega_2$, such that if their application to the left hand sides of the equations equates them then the right hand sides after applying the respective substitutions also equates them. More succinctly, $\mathtt{compat}(p, q)$ iff there exist substitutions $\Omega_1$ and $\Omega_2$ such that if $\Omega_1(\text{lhs}_p) = \Omega_2(\text{lhs}_q)$ then $\Omega_1(\text{rhs}_p) = \Omega_2(\text{rhs}_q)$.

**Definition 6** (Closed Type Family Reduction Criteron Optimized)**.** An equation, say $p$, given in the type family declaration can be used to simplify the type $\mathsf{F}(\overline{\tau})$, if the following two conditions hold:

(1) The left hand side of the equation, $\text{lhs}_p$ matches the the type $\mathsf{F}(\overline{\tau})$ or $\mathtt{match}(\text{lhs}_p, \mathsf{F}(\overline{\tau}))$.
(2) All equations, $q$, preceding $p$, are either apart from $\mathsf{F}(\overline{\tau})$, or the equations are compatible. Symbolically, $\mathtt{compat}(p, q) \vee \mathtt{apart}(\text{lhs}_q, \mathsf{F}(\overline{\tau}))$

The two key insights here are that compatibility check loosens up the restriction to enable type reductions without threatening soundess, and compatibility of equations can be precomputed as it does not depend on the type we want to reduce and giving us a cheaper check as compared to checking for apartness.

## 5.2 Formalizing Type Soundess and Type Reductions

In the previous section, we alluded that type soundess can be achieved using confluence. In this section we will formalize it using System $\mu$**FC**. The portion of the system necessary to show type soundness is shown in Figure 10. We have special syntax category for type family consructors $(\mathsf{F}, \mathsf{G})$ along with the usual types. Predicates in this system are type equalites $\tau_1 \sim \tau_2$ asserts that types $\tau_1$ and $\tau_2$ are equal. The axioms $(\xi{:}\Psi)$ are a list of type family equations which are declared along with the type family constructors. For example, the type family declaration `TEq a b` will be represented in our system as $\mathsf{AxTEq} : [\forall\alpha.\ \mathsf{TEq}\ \alpha\ \alpha \sim \mathsf{TT}, \forall\alpha\beta.\ \mathsf{TEq}\ \alpha\ \beta \sim \mathsf{FF}]$.

*5.2.1 Type Reduction.* We can formally specify type reduction, written as $\Sigma \vdash \bullet \rightsquigarrow \bullet$, using the rule (TY-RED). This rule says that the $i$-th equation of axiom $\xi$, $\forall\overline{\alpha}.\ \mathsf{F}(\mathsf{N}_i) \sim \sigma_i$ is used to reduce the target type $\mathsf{F}(\overline{\tau})$ to type $\tau_0$. The `no_conflict` check is the same as discussed before and we use

| | |
|---|---|
| Type family Constructors | $\mathsf{F}, \mathsf{G}$ |
| Type Constants | $\mathsf{T}$ |
| Type Validity | $\Gamma \vdash \tau$ type |
| Proposition Validity | $\Gamma \vdash P$ prop |
| Ground Context Validity | $\vdash_{gnd} \Sigma$ |
| Type reduction | $\Sigma \vdash \bullet \rightsquigarrow \bullet$ |
| One Hole Type Context | $C[\bullet]$ |

| | | |
|---|---|---|
| Types | $\tau, \sigma$ | $::= \alpha \mid \tau \longrightarrow \tau \mid \mathsf{F}(\overline{\tau}) \mid \mathsf{T}$ |
| Ground Types | $\omega$ | $::= \tau \longrightarrow \tau \mid \mathsf{T}$ |
| Predicates | $P$ | $::= \tau \sim \tau$ |
| Type family Pattern | $\mathsf{N}$ | $::= \overline{\tau}$ |
| Axiom Equations | $\Phi$ | $::= \forall \overline{\alpha}.\ \mathsf{F}(\mathsf{N}) \sim \sigma$ |
| Axiom Types | $\Psi$ | $::= \overline{\Phi}$ |
| Ground Context | $\Sigma$ | $::= \epsilon \mid \Sigma, \xi : \Psi \mid \Sigma, \mathsf{F} : n$ |

Fig. 10. Excerpt of System for Closed Type Families (System $\mu$**FC**)

$$(\textsc{nc-apart})\ \frac{\Psi = \overline{\mathsf{F}(\mathsf{N}) \sim \sigma} \qquad \mathtt{apart}(\mathsf{N}_j, \mathsf{N}_i[\overline{\tau}/\overline{\alpha}_i])}{\mathtt{no\_conflict}(\Psi, i, \overline{\tau}, j)} \qquad (\textsc{compt-dis})\ \frac{\begin{array}{c}\Phi_1{=}\mathsf{F}(\mathsf{N}_1){\sim}\sigma_1\\ \Phi_2{=}\mathsf{F}(\mathsf{N}_2){\sim}\sigma_2\end{array} \qquad \Omega = mgu(\mathsf{N}_1, \mathsf{N}_2)\mathtt{fails}}{\mathtt{compat}(\Phi_1, \Phi_2)}$$

$$(\textsc{nc-compt})\ \frac{\mathtt{compat}(\Psi[i], \Psi[j])}{\mathtt{no\_conflict}(\Psi, i, \overline{\tau}, j)} \qquad (\textsc{compt-inc})\ \frac{\begin{array}{c}\Phi_1{=}\mathsf{F}(\mathsf{N}_1){\sim}\sigma_1\\ \Phi_2{=}\mathsf{F}(\mathsf{N}_2){\sim}\sigma_2\end{array} \qquad \begin{array}{c}\Omega{=}mgu(\mathsf{N}_1,\mathsf{N}_2)\\ \Omega\sigma_1{=}\Omega\sigma_2\end{array}}{\mathtt{compat}(\Phi_1, \Phi_2)}$$

Fig. 11. Non Conflicting Equations and Compatibility

$C[\bullet]$ to mean that the rewrite can happen anywhere within the type.

$$(\textsc{ty-red})\ \frac{\begin{array}{c}\xi{:}\Psi{\in}\Sigma\\ \vdash_{gnd}\Sigma\end{array} \qquad \begin{array}{c}\Psi{=}\overline{\forall\overline{\alpha}.\ \mathsf{F}(\mathsf{N}){\sim}\sigma}\\ \forall j{<}i.\ \mathtt{no\_conflict}(\Psi{,}i{,}\overline{\tau}{,}j)\end{array} \qquad \begin{array}{c}\Omega{=}mgu(N_i{,}\overline{\tau})\\ \tau_0{=}\Omega\sigma_i\end{array}}{\Sigma \vdash C[\mathsf{F}(\overline{\tau})] \rightsquigarrow C[\tau_0]}$$

*5.2.2 Consistency and Goodness of Context.* Consistency means that we can never derive unsound equalities between ground types, such as $\gamma : \mathtt{Int} \sim \mathtt{Bool}$, in the system. Ground types ($\omega$), in our system are nothing but $\mathsf{T}\overline{\tau}$, $\tau_1 \longrightarrow \tau_2$ and $\forall \alpha.\ \tau$. We say that a ground context $\Sigma$ is consistent, when for all coercions $\gamma$ such that $\Sigma; \epsilon \vdash_{Co} \gamma : \omega_1 \sim \omega_2$ we have the following:

(1) if $\omega_1$ is of the form $\mathsf{T}\overline{\tau}$ then so is $\omega_2$
(2) if $\omega_1$ is of the form $\tau_1 \longrightarrow \tau_2$ then so is $\omega_2$
(3) if $\omega_1$ is of the form $\forall \alpha.\ \tau$ then so is $\omega_2$

In general context consistency is difficult to prove. We take a conservative approach and enforce syntactic restrictions on the ground context.

Property 7 (Good $\Sigma$). *A ground context ($\Sigma$) is* Good, *written* Good$\Sigma$ *when the following conditions are met for all $\xi{:}\Psi \in \Sigma$ and where $\Psi$ is of the form $\overline{\forall \overline{\alpha}.\ \mathsf{F}_i(\mathsf{N}_i) \sim \sigma}$:*

*(1) There exists an $\mathsf{F}$ such that $\forall i.\ \mathsf{F}_i = \mathsf{F}$ and none of the type pattern $\mathsf{N}_i$ mentions a type family constructor.*

*(2) The binding variables $\overline{\alpha}$ occur at least once in the type pattern* N*, on the left hand side of the equation.*

Given our characterization of type reduction in the previous section, we now have to show that if we have Good$\Sigma$ then, $\Sigma$ is consistent. One way to prove this is via confluence of type reduction relation. Whenever we have $\Sigma \vdash \tau_1 \rightsquigarrow \tau_2$ then we would know that $\tau_1$ and $\tau_2$ have a common reduct, say $\tau_3$. As type reduction relation (TY-RED) only works on type family's and does not reduce any ground type heads, confluence would be sufficient prove consistency. However, to prove confluence, it is necessary to assume termination of type reduction. We get our consistency lemma as follows.

Lemma 8 (Consistency). *If $\Sigma \vdash \bullet \rightsquigarrow \bullet$ is terminating and* Good$\Sigma$ *then $\Sigma$ is consistent.*

## 6 CONSTRAINT TYPE FAMILIES

In the previous section for closed type families, we have made an implicit assumption— the type families are all total, in the sense their domain is all the types. This is problematic in theory as as well as in practice. Consider the case where a closed type family does not have an equation for a particular type argument as shown in Figure 12. We know that `PTyFam Bool` has no satisfying equations associated with it that gives it a meaning and never will in the future as it is closed. In our current setup a program that diverges—`loopy`—can be given this nonsensical type and much worse, the system treats it like a valid type.

```
type family PTyFam a where   loopy :: ∀ a. a
  PTyFam Int = Bool          loopy = loopy
type family Loop where       sillyFst x = fst (x, loopy :: PTyFam Bool)
  Loop = [Loop]              sillyList x = x : x
```

**TODO: :** fix this example

Fig. 12. Partial Closed Type Family

Next, consider the target type `TEq [a] a`, in System $\mu$**FC**, this type is not evaluated to `FF` even though `[a]` and `a` don't have no most general unifier. The reason being there may be an infinite type such as `Loop` that does unify both `[a]` and `a`. As `Loop` unwinds infinitely to become `[[[...[Loop]...]]]` we will have `TEq [Loop] Loop` evaluate to `TEq [Loop] [Loop]` which is `TT`. This justifies the reason for not evaluating `TEq [a] a` to `FF`. However, Haskell does not have infinite ground types, and we thus would expect `TEq [a] a` to reduce to `FF`. The crux of the problem here is that we treat type families as they are type constructors (ground types). `Loop` will never reduce to a ground type but we must treat it like one while trying to reduce `TEq [a] a`. This non-uniform treatment of type family constructors is confusing for programmers.

There also seems to be a mismatch in our intuitive semantics of type families. We think of them as partial functions on types where each new equation extends its definition. Instead we should be thinking about them as introducing a family of distinct types and each new equation equates types that were previously not equal. This distinction in semantics does matter in practice as illuminated by `sillyList`. If `Loop` is a type then we can give `sillyList` a type that is `Loop → Loop`. But this is not its principle type, as we can generalize it to be `(a ~ [a]) ⇒ a → a`. Haskell however rejects this program on the basis of infinitary unification of `a ~ [a]` is not possible. We are left in a position where we accept some problematic definitions, like `Loop`, but not the others like, `(a ~ [a])`. To solve this problem we leverage the existing infrastructure that Haskell already has—qualified types and typeclasses—to make the totality assumption explicit.

## 6.1 Closed Typeclasses

Typeclasses can be extended to have new instances. Closed typeclasses, on the contrary, are classes that will not be able to be extended once they are defined. They essentially mirrors closed type families in the sense the programmer cannot add more instances after they are defined. We can define overlapping instances for a closed typeclass and their resolution will be performed at operator's use site in a top to bottom order on instance declarations.

For example, the type families `TEq a b` and `Plus m n` can be expressed in the closed type-class world using `TEqC a b` and `PlusC m n` respectively as shown in Figure 13. In the con-

```
class LoopC where                      class PTyFamC a where
  type Loop                              type PTyFam a
  instance LoopC ⇒ LoopC where           instance PTyFamC Int where
    type Loop = [Loop]                     type PTyFam Int = Bool
class {-TOTAL-} TEqC a b where         class {-TOTAL-} PlusC m n where
  type TEq a b                           type Plus m n
  instance TEqC a a where                instance PlusC Z m where
    type TEq a a = TT                      type Plus Z m = Z
  instance TEqC a b where                instance PlusC m n ⇒ PlusC (S m) n where
    type TEq a b = FF                      type Plus (S m) n = S (Plus m n)
```

Fig. 13. Closed Typeclasses Examples

strained type families world, every closed type family will be associated with a closed type-class. Any type family without an associated typeclass will be disallowed. For example, see **`class`** `PTyFamC`. The type for `sillyFst` in this system is no longer `∀ a. a → a` but instead it is `∀ a. PTyFamC Bool ⇒ a → a`, and the type checker will flag it as an error wherever it is used; there is no way to satisfy the instance `PTyFamC Bool`. The type family `Loop` will also need to have an associated typeclass `LoopC`. To declare an instance of `LoopC` where `Loop ~ [Loop]` we need to specify `LoopC` to be a constraint on the instance. This makes the use of `Loop` no longer threatens the type soundness as `LoopC` is unsatisfiable.

Most type families are partial, only some are total and we would want the users to take advantage of this fact by allowing programmers to specify it. In general checking for or inferring totality for a given closed type family is a hard problem, thus we would also give the users a way to let the type checker accept it without checking it. It would otherwise be inconvenient to express total type families.

## 6.2 Type matching and Apartness Simplified

The type rewriting in closed type families had a complex criterion for apartness that included flattening and then checking if they had a unifier using infinitary unification. In constrained type families we neither have to depend on infinitary type unification nor flattening of types. We can also relax the restriction that type families cannot appear in the left hand side of the type rewrite equations due to the class constraints associated with the use of each type family constructor. The constraint of allowing only terminating type families can also be lifted as seen from the `LoopC` example, it no longer threaten type soundness. Apartness in this system is just checking for failure of unification.

## 6.3 Formalizing Constrained Type Families

This system is similar to System $\mu$**FC** except for a few new constructors that we highlight in Figure 14. The ground context keeps track of two types of type family constructors, a total type family constructor of arity $n$ (written $\mathtt{F}{:}_{\top} n$) and a partial type family constructor of arity $n$ (written $\mathtt{F}{:}n$). The variable environment ($\Delta$) along with variable type bindings also stores coercion constraint bindings $c{:}P$. Each equation that is introduced by axioms ($\xi$) in this system, are of the form $\forall \overline{\alpha}\ \overline{\chi}.\ \mathtt{F}(\overline{\tau}) \sim \sigma$. Both $\overline{\tau}$ and $\sigma$ do not have occurrence of any family type constructors. The equations are quantified by type variables $\overline{\alpha}$ and also over a new term collection evaluation assumptions $\overline{\chi}$. These evaluation assumptions are of the form $\alpha | c{:}\mathtt{F}(\overline{\tau}) \sim \alpha$ and read as "$\alpha$ such that $c$ witnesses $\mathtt{F}(\overline{\tau})$ reduces to $\alpha$". We use these evaluation assumptions to allow type families on the left hand side of the type equations written in the source program. For example, the user written type equation `F (F' a) = G a`, where $G$ and $F'$ are type family constructors, will be compiled into an equation $\forall a\ (b | c : G\ a \sim b)(d | c : F'\ a \sim d).\ F(d) \sim b$. We use $\chi$ to remind us that it is more specific than $P$; for any $\chi$, the left hand side of the type equality is always a type family and right and side is a fresh variable. The `assume` $\chi$ `in` $e$ is the construct that is used while working with total type families. It provides a sort of an escape hatch as we are guaranteed to obtain a type family free type after reducing a total type family.

The ground types in this system can mention type family constructors only in propositions $P$. For example, the type `∀m n. Add m n ⇒ m → n → Result m n` would instead be written as `∀m n. Result m n ~ p ⇒ m → n → p`. This is an assertion that `Result m n` evaluates to a type family free type.

| | | |
|---|---|---|
| Type Validity | $\Gamma \vdash \tau$ type | |
| Proposition Validity | $\Gamma \vdash P$ prop | |
| Assumption Validity | $\Gamma \vdash \overline{\chi}$ asmp | $\mid \xi_i\, \overline{\tau}\, \overline{q}$ |
| Ground Context Validity | $\vdash_{gnd} \Sigma$ | |
| Variable Context Validity | $\Sigma \vdash_{var} \Delta$ | |
| Context Validity | $\vdash_{ctx} \Gamma$ | |
| Term Typing | $\Gamma \vdash e : \tau$ | |
| Coercion Typing | $\Gamma \vdash_{Co} \gamma : P$ | |
| Resolution Validity | $\Gamma \vdash_{res} \overline{q} : \overline{\chi}$ | |
| One Hole Type Context | $C[\bullet]$ | |

| | | |
|---|---|---|
| Types | $\tau, \sigma$ | $::= \alpha \mid \tau \to \tau \mid \mathsf{F}(\overline{\tau}) \mid \mathsf{T} \mid P \Rightarrow \tau$ |
| Ground Types | $\omega$ | $::= \tau \to \tau \mid \mathsf{T}$ |
| Predicates | $P$ | $::= \tau \sim \tau$ |
| Axiom Equations | $\Phi$ | $::= \forall \overline{\alpha}\, \overline{\chi}.\, \mathsf{F}(\overline{\tau}) \sim \sigma$ |
| Axiom Types | $\Psi$ | $::= \overline{\Phi}$ |
| Eval Assumption | $\chi$ | $::= (\alpha \mid c : \mathsf{F}\overline{\tau} \sim \alpha)$ |
| Eval Resolution | $q$ | $::= (\tau \mid \gamma)$ |
| Terms | $e$ | $::= x \mid \lambda x{:}\tau.\, e \mid e\, e \mid M \blacktriangleright \gamma \mid \Lambda\alpha.\, e \mid e\, \tau \mid \mathsf{D}\overline{e} \mid \lambda c{:}P.\, e \mid e\, \gamma \mid \texttt{assume}\ \chi\ \texttt{in}\ e$ |
| Values | $\mathcal{V}$ | $::= \lambda x{:}\tau.\, e \mid \mathsf{D}\overline{e} \mid \Lambda\alpha.\, e \mid \lambda c{:}P.\, e$ |
| Ground Context | $\Sigma$ | $::= \epsilon \mid \Sigma, \xi{:}\Psi \mid \Sigma, \mathsf{F}{:}_{\top} n \mid \Sigma, \mathsf{F}{:}n \mid \Sigma, \mathsf{T}{:}n$ |
| Variable Context | $\Delta$ | $::= \epsilon \mid \Delta, \alpha \mid \Delta, x{:}\tau \mid \Delta, c{:}P$ |
| Typing Context | $\Gamma$ | $::= \Sigma; \Delta$ |

Fig. 14. System for Constraint Type Families

The new validity judgments reflect the above discussion. The rule (V-QTY) ensures that type equality coercions can appear only in $P$. We do not have the rule (V-TFCTR) in this system to ensure ground types do not mention type family constructors. The rule (V-QTFP) replaces the rule (V-TFP). This means that arguments to type family constructors can mention type families and we no longer have to use special type patterns as in System $\mu$**FC**'s (V-TFP). The rule (V-QGAX) replaces the (V-GAX) that checks axiom equations are valid. This checks that the context is consistent by making sure it is `Good`, as discussed in 6.3.2. We also have two new classes of validity judgments, (V-ASSMN) and (V-ASSMC) check that the evaluation assumptions that appear in the axioms are valid, while rules (V-RESE) and (V-RESC) ensure that the evaluation resolutions are valid.

The typing judgments new to this system are shown in Figure **??**. The rule (T-COABS) abstracts over coercion variable while (T-COAPP) applies a coercion argument to a term. We have a new version of axiom application rule (CO-QAXIOM). It is very similar to (CO-AXIOM), except that we need to provide extra validity resolutions $\overline{q}$ that instantiate the validity assumptions $\overline{\chi}$. The validity resolutions are of the form $(\tau|\gamma)$ where the type $\alpha$ in validity assumptions $(\alpha|c{:}\mathsf{F}(\overline{\tau}) \sim \sigma)$ is instantiated to $\tau$ and $\gamma$ proves the equality and instantiates $c$. This is exactly what the rule (V-RESC) does. Finally, The rule (T-ASSUM) is the special rule that says we are allowed to assume arbitrary applications of a type family would give us a type free type. We can indeed do this by the definition of total type family.

*6.3.1 Type reduction.* The type reduction relation is given using two rules (QTY-RED-TOP) and (QTY-RED). The rule (QTY-RED-TOP) does the heavy lifting of producing the correct substitutions for types ($\Omega_1$) as well as evaluation resolutions ($\Omega_2$). The correct equation selection is done by `no_conflict` criterion. The specialty of this relation is that we ensure applying type arguments to type families only when they satisfy proper constraints with the use of evaluation resolutions, thus guaranteeing every type reduction to eventually obtain a type family free type. And due to the fact that type family free types do not reduce, we can prove termination, for the type reduction relation.

$$\text{(QTY-RED)}\ \frac{\Sigma \vdash \mathsf{F}(\overline{\tau}) \rightsquigarrow \tau_1}{\Sigma \vdash C[\mathsf{F}(\overline{\tau})] \rightsquigarrow C[_1]} \qquad \text{(QTY-RED-TOP)}\ \frac{\begin{array}{c}\xi{:}\Psi \in \Sigma \\ \vdash_{gnd} \Sigma \\ \Psi_i = \forall \overline{\alpha}_i\ \overline{\chi}_i.\ \mathsf{F}(\overline{\sigma_i}) \sim \sigma_0 \\ \forall j < i.\ \mathtt{no\_conflict}(\Psi, i, \overline{\tau}, j)\end{array} \quad \begin{array}{c}\overline{\chi}_i = \overline{(\alpha' \,|\, c{:}\mathsf{G}(\overline{\tau}') \sim \alpha')} \\ \Sigma \vdash \overline{\mathsf{G}(\Omega_1 \overline{\tau}')} \rightsquigarrow \overline{\tau_0}\end{array} \quad \begin{array}{c}\Omega_1 = mgu(\overline{\sigma}_i, \overline{\tau}) \\ \Omega_2 = [\overline{\chi_i}/\overline{\mathsf{G}(\Omega_1 \overline{\tau}') \sim \tau_0}] \\ \tau_1 = \Omega_1 \Omega_2 \sigma_0\end{array}}{\Sigma \vdash \mathsf{F}(\overline{\tau}) \rightsquigarrow \tau_1}$$

Fig. 15. Type reduction

*6.3.2 Goodness and Consistency.* The definition of Goodness can now be relaxed due to simplifying apartness criteria for types.

PROPERTY 9 (Good $\Sigma$ RELAXED). *A ground context ($\Sigma$) is* Good, *written* Good$\Sigma$ *when the following conditions are met for all* $\xi{:}\Psi \in \Sigma$ *and where* $\Psi$ *is of the form* $\overline{\forall \overline{\alpha}\ \overline{\chi}.\ \mathsf{F}_i(\mathsf{N}_i) \sim \sigma}$*:*

*(1) There exists an* $\mathsf{F}$ *such that* $\forall i.\ \mathsf{F}_i = \mathsf{F}$*.*
*(2) The binding variables* $\overline{\alpha}$ *occur at least once in the type arguments* $\overline{\tau}$*, on the left hand side of the equation.*

With the reduction relation defined in previous section, and knowing that it is always terminating, we can use Newman's lemma[Newman 1942] to prove that type reduction is confluent. Thus we get our relaxed consistency lemma as follows:

LEMMA 10 (CONSISTENCY). *If* Good$\Sigma$ *then* $\Sigma$ *is consistent*

## 7 CONCLUSION AND FUTURE WORK

Type computation, either using functional dependencies or type families is an attractive feature for programmers as it considerably improves language expressivity. If functional dependencies are morally equivalent to type families one would expect to have either of the two cases to be true 1) translation mechanism that can go from one style to another or 2) translation of both the language features into a common intermediate language. As of now both of these remain an active area or research[Karachalias and Schrijvers 2017; Sulzmann et al. 2007].

The type safety formalization of closed type families hinges on the assumption that type family reduction are terminating. This problem is effectively solved by using constrained type families. In conclusion, the motivation of constraint type families is to reunite the idea of functional dependencies and type families that had previously diverged. The use of equality constraints to ensure that type family applications are well defined is reminiscent of the use of functional dependencies to ensure typeclass instances are well defined.

## REFERENCES

M. Bezem, J.W. Klop, E. Barendsen, R. de Vrijer, and Terese. 2003. *Term Rewriting Systems*. Cambridge University Press, UK.

Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. 2005. Associated types with class. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2005-01-12) *(POPL '05)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/1040305.1040306

Richard A. Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. 2014. Closed Type Families with Overlapping Equations. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, San Diego, California, USA, 671–683.

Mark P. Jones. 1994. *Qualified Types: Theory and Practice*. Cambridge University Press, Cambridge, UK. https://doi.org/10.1017/CBO9780511663086

Mark P. Jones. 2000. Type Classes with Functional Dependencies. In *Proceedings of the 9th European Symposium on Programming Languages and Systems (ESOP '00)*. Springer-Verlag, Berlin, Germany, 230–244. https://doi.org/10.1007/3-540-46425-5_15

Georgios Karachalias and Tom Schrijvers. 2017. Elaboration on functional dependencies: functional dependencies are dead, long live functional dependencies. *ACM SIGPLAN Notices* 52, 10 (2017), 133–147. https://doi.org/10.1145/3156695.3122966

J. Garrett Morris and Richard A. Eisenberg. 2017. Constrained Type Families. *Proc. ACM Program. Lang.* 1, ICFP, Article 42 (Aug. 2017), 28 pages. https://doi.org/10.1145/3110286

M. H. A. Newman. 1942. On Theories with a Combinatorial Definition of "Equivalence". *Annals of Mathematics* 43, 2 (1942), 223–243. https://doi.org/10.2307/1968867

J. A. Robinson. 1965. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM* 12, 1 (1965), 23–41. https://doi.org/10.1145/321250.321253

Tom Schrijvers, Martin Sulzmann, Simon Peyton Jones, and Manuel Chakravarty. 2007. Towards open type functions for Haskell. (2007). https://www.microsoft.com/en-us/research/publication/towards-open-type-functions-haskell/

Martin Sulzmann, Gregory J. Duck, Simon Peyton-Jones, and Peter J. Stuckey. 2007. Understanding functional dependencies via constraint handling rules. *Journal of Functional Programming* 17, 1 (2007), 83–129. https://doi.org/10.1017/S0956796806006137

Philip Wadler and Stephen Blott. 1989. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '89)*. ACM, Austin, Texas, USA, 60–76. https://doi.org/10.1145/75277.75283