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, α, if the argument is of type α then the function returns a value of type α. 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 α, 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 3.3. We also give a brief description of type safety for this system in Section 3.4. We then describe two flavors of type families—closed type families[Eisenberg et al. 2014] in Section 4

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 5 with examples and their respective formalization in Section 4.2 and Section 5.3. We then give some details about the type safety of each system in Section 4.3 and Section 5.4 respectively. To conclude we draw some comparisons between the three systems while pointing towards some open questions in Section 6. 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 be 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`. We read this type as: For any type `a` that satisfies the predicate `Eq a`, if we are given two values of type `a`, then we can return a Boolen 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   e :: (Add Int Float b, Add b Int c) ⇒ c
          (+) a b = addInt a (toInt b)     e = (1 + 2.0) + 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 `e = (1 + 2.0) + 3`. Due to the use of the operator (+) in `e`, the most general type of `e` 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 due to incoherence. The compiler cannot choose an interpretation of the subterm `(1 + 2.0)`, it can either be an `Int` or a `Float`. Haskell, thus, disallows ambiguous types by reporting a type error. However, as a consequence, the type errors can cause confusion; although the issue is with instances of typeclass `Add m n p`, the type error is raised at the term that may be defined in a separate 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, $(x_1, \ldots, x_m \rightarrow y_1, \ldots, y_m)$. 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. It gives the compiler a way to detect inconsistent instances and report an error whenever it detects one. 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`.

Further, due to functional dependencies, we may also be able to determine the ambiguous type variables in a type. Let's reconsider the ambigouous 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, `e :: (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 final type synthesis `e :: Float`. It would be impossible to make such an improvement without the functional dependency.

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

```
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

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.

3.1 Formalizing Typeclasses with Functional Dependencies

To formalize the system, we first need to fix the language. The surface syntax, the one that the user writes, in shown in Figure 4. We will call this System **TCFD**. The syntax of types (τ) consists of type variables (α), functions ($\tau \longrightarrow \sigma$), and type constructors $\mathsf{T}\overline{\alpha}$. The qualified types ($\rho$) are given as $\mathsf{C}\overline{\alpha} \implies \tau$ where $\mathsf{C}\overline{\alpha}$ constrains the type τ. Type schemes (σ) are quantified constraint types. The terms or expressions in the language (e) consists of countably infinite set of variables (x, y), functions ($\lambda x{:}\tau.\ e$), function applications ($e_1\ e_2$), overloading of operators is achieved by (let $x = e_1$ in e_2), and data constructors (D) define values for a user defined type. Values in this system are data constructors and lambda expressions.

Type Variables	α, β
Term Variables	x, y
Class Constructors	C
Term Typing	$P \mid \Gamma \vdash e : \tau$

Predicates	π	$::= C\overline{\tau}$
Predicate Set	P	$::= \overline{\pi}$
Types	τ	$::= \alpha \mid \tau \longrightarrow \tau$
Qualified Types	ρ	$::= \tau \mid P \Rightarrow \tau$
Type Schemes	σ	$::= \forall\overline{\alpha}.\rho$
Terms	e	$::= x \mid e\ e \mid \lambda x{:}\tau.\ e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e$
Typing Environment	Γ	$::= \epsilon \mid \Gamma, x{:}\sigma$

Fig. 4. Excerpt of System **TCFD**

3.1.1 Notations. We will use some notations as follows. We use subscripts on objects $(\alpha_1, \ldots, \alpha_n)$ to distingush them. $\overline{\alpha}$ means a collection of $\alpha_1, \alpha_2, ..., \alpha_n$ items of arbitrary length. We use $S_1 \backslash S_2$ to denote the set difference operation. For an object X, $TV(X)$ is the set of variables that are free in X. We write $M[\overline{x}/\overline{y}]$ to denote the substitution where each variable x_i is mapped to y_i in M. Alternatively we also write ΩX for an substitution Ω applied to object X. We denote the most general unifier for two types τ_1 and τ_2 (if it exists), by $mgu(\tau_1, \tau_2)$[Robinson 1965]. For the sake of convenience, we would also write $mgu(\overline{\tau_1}, \overline{\tau_2})$ to give us a composition of most general unifier for each pairs of types (τ_{1i}, τ_{2i}). For a typeclass declaration we write **`class`** `P ⇒ C` $\overline{t}$, where $\overline{t}$ are the type parameters of the class and P are the constraints that must be satisfied. We denote the set of functional dependencies of class `C` with FD_{C}. We write $X \longrightarrow Y$ for an arbitrary functional dependency. The determinant of a functional dependency is denoted by t_X and the dependent is denoted by t_Y. $TV(C)$ denotes the set of type parameters of the class `C`. A predicate set P is satisfiable if there exists a substitution, Ω, such that $\emptyset \Vdash \Omega P$. We write $\lfloor P \rfloor = \{\Omega \mid \emptyset \Vdash \Omega P\}$ to mean the set of all substitutions that satisfy P.

For example, for a typeclass declaration **`class`** `Add m n p | m n ⟶ p`, we have, $t = (a, b, c)$, $\mathsf{FD}_{\mathsf{Add}} =$`{ m n ⟶ p }`, $TV(C) = \{m, n, p\}$. For the functional dependency `m n ⟶ 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,

TODO: : fix this definition? It seems broken

Z^+_{J}, on $Z \subseteq t$, to be equal to all the type parameters that are determined by set of the functional dependencies `J`. Thus, $\{p\}^+_{\mathsf{FD_{Add}}} = \{p\}$, $\{m\}^+_{\mathsf{FD_{Add}}} = \{m\}$ while $\{m, n\}^+_{\mathsf{FD_{Add}}} = \{m, n, p\}$.

We say that the tuple $P \mid \Gamma \vdash e : \tau$ to be a judgment that holds when there is a typing derivation that shows e has type τ with predicates P being satisfied and the free variables in e are given types by the typing environment Γ, that maps term variables to its types. The typing judgments for the language are shown in Figure 5. The generalize function $\mathrm{Gen}(\Gamma, \tau)$ quantifies all the free variables of the type τ that do not occur in the domain of the type environment Γ, i.e., $\mathrm{Gen}(\Gamma, \tau) = \forall (TV(\tau) \backslash dom(\Gamma)).\tau$. The instantiate function $\mathrm{Inst}(\sigma)$ maps each quantified variable in σ to a fresh variable, i.e. if $\sigma = \forall \overline{\alpha}.P \Rightarrow \tau$, then $\mathrm{Inst}(\sigma) = P[\overline{\alpha}/\overline{\beta}] \Rightarrow \tau[\overline{\alpha}/\overline{\beta}]$ where $\overline{\beta}$ are fresh. The variables need to be fresh to avoid any conflict with the existing type variables. The typing rule for function abstraction ($\longrightarrow$I) says that if we can show a typing derivation where the body of the function e has the type τ with the typing environment extended with the variable that represents the argument for the function with type τ_1, then we have a judgment that shows the lambda term has type $\tau_1 \longrightarrow \tau_2$.

The typing judgment for function application rule ($\longrightarrow$E) says that if we can show that the left hand term e_1 has type $\tau_2 \longrightarrow \tau$ and additionally we can show that the right hand term e_2 has type τ_2 then we can show that the term $e_1\ e_2$ has type τ. The rule (TLET) says that we can show that the type of e_2 with a variable x bound to a term e_1 is of type τ_2 only if we can build a derivation

$$\text{(T-ABS)}\ \frac{P \mid \Gamma, x{:}\tau_1 \vdash e : \tau}{P \mid \Gamma \vdash \lambda x{:}\tau_1.\, e : \tau_1 \to \tau} \qquad \text{(T-APP)}\ \frac{P \mid \Gamma \vdash e_1 : \tau_2 \to \tau \quad P \mid \Gamma \vdash e_2 : \tau_2}{P \mid \Gamma \vdash e_1\, e_2 : \tau}$$

$$\text{(T-VAR)}\ \frac{x{:}\sigma \in \Gamma \quad P \Rightarrow \tau = \mathrm{Inst}(\sigma)}{P \mid \Gamma \vdash x : \tau} \qquad \text{(T-LET)}\ \frac{P \mid \Gamma \vdash e_1 : \tau_1 \quad \sigma = \mathrm{Gen}(\Gamma, P \Rightarrow \tau_1) \quad P_1 \mid \Gamma, x{:}\sigma \vdash e_2 : \tau_2}{P_1 \mid \Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2}$$

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

that shows e_1 as τ_1 and we can show that e_2 has type τ_2 with typing environment extended with the variable x mapped to the generic instance of τ_1. The (TLET) is in essence overloading in action as the variable x is assigned a generic type. The rule (TVAR) says that to show that the type of the variable x is τ we need to look it up in our typing environment and instantiate it with fresh variables.

TODO: : Talk about principal type scheme?

3.2 Instance Validity and Inconsistency Detection

Each typeclass declaration introduces a new relation on types in the type system. With functional dependencies we introduce some additional constraints that the 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`** $P \Rightarrow$ `C t` **`where`** ... that the user 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 determinant, $TV(t_Y)$, should either already be in the set of dependent type variables, $TV(t_X)$, or should be fully determined using the functional dependencies induced by the class ($\mathsf{FD_C}$) or induced by the constraints ($\mathsf{FD_P}$).

(2) *Consistency Condition*: For each new instance of the form **`instance`** $Q \Rightarrow$ `C s` **`where`** ... along with **`instance`** $P \Rightarrow$ `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, 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.

3.3 Computing and Using Improving Substitution

An improving substitution, written as $impr(P)$, is a substitution that does not change the set of satisfiable instances of predicate set P. Computing an improving substitution is straightforward with the induced functional dependencies, $\mathsf{FD_P}$, on the predicate set, P. For each $(X \longrightarrow Y) \in \mathsf{FD_P}$ whenever we have $TV(t_X)$ we can infer $TV(t_Y)$. The rational behind improving substitution is that it helps simplifying the type by showing its true and concise characterization. Improving substitutions also affects the way ambiguous types are detected. 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 F_P due to P, the appropriate check would be $(\overline{\alpha} \cap TV(P)) \subseteq TV(\tau)^{+}_{F_P}$. We thus weaken the check to ensure that there might be some type variables α_i that are determined by the functional dependencies due to the class constraints. This improving substitution can be applied anywhere during the type inference algorithm without adversely affecting it.

3.4 Type safety of System TCFD

$$(\textsc{s-app})\ \frac{e_1 \rightsquigarrow e_1'}{e_1\ e_2 \rightsquigarrow e_1'\ e_2} \qquad (\textsc{s-}\beta)\ \frac{}{(\lambda x{:}\tau.\ e_1)\ e_2 \rightsquigarrow e_1[x/e_2]}$$

Fig. 6. Small Step Operational Semantics for System **TCFD**

The system that has been described has been in terms of its static semantics. For the static semantics to be of any real use, we also need to provide an important property of the system in the spirit of [Milner 1978]—"Well typed programs don't go wrong."—or if our type system says a program is well typed, then if we run the program, it should not crash (or get stuck). Before we can formalize type safety, we formalize what running the program means. We say a term e reduces e' or $e \rightsquigarrow e'$ using the rules shown in Figure 6. We define $\bullet \rightsquigarrow^* \bullet$ as a transitive closure of the $\bullet \rightsquigarrow \bullet$ relation.

Lemma 1 (Progress System **TCFD**). *If $\epsilon \mid \epsilon \vdash e_1 : \tau$ then $e_1 \in \mathcal{V}$ or $e_1 \rightsquigarrow e_2$.*

Lemma 2 (Preservation System **TCFD**). *If $\epsilon \mid \epsilon \vdash e_1 : \tau$ and $e_1 \rightsquigarrow e_2$ then $\epsilon \mid \epsilon \vdash e_2 : \tau$*

The (syntactic) type safety of the system can formally be given as

Lemma 3 (Type Safety System **TCFD**). *If $\epsilon \mid \epsilon \vdash e : \tau$ then either $e \in \mathcal{V}$ or there exists a term, e', such that $e \rightsquigarrow^* e'$ and $e' \in \mathcal{V}$*

ANI: Should i be talking about open type families?

4 CLOSED TYPE FAMILIES

Type family is powerful language feature of describing computation on types in a more natural style of functional programming rather than relations in the style of logic programming. For example, we can define addition over the two previously mentioned types `Z` and `S n` as shown in Figure 7. This style is a lot more palpable for programmers who are already used to writing equations with pattern matching at term level. It also has a cleaner view and has less code clutter compared to functional dependencies. The vector concatenation `concat_vec` also gets a cleaner and concise type as compared to previous definition.

```
data TT -- Type level True
data FF -- Type level False

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

```
type family TEq n m where
  TEq a a = TT
  TEq a b = FF
concat_vec :: Vec m e → Vec n e
                → Vec (Plus n m) e
concat_vec v1 v2 = ...
```

Fig. 7. Peano Arithmetic and Vector Operations with Closed Type Family

The `Add m n p` typeclass defined in Figure 2 can also be written in the type family style as shown in Figure 8. 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. As expected, the instance `Add Int Float` would raise a type error due to its conflict with the first type instance.

```
type family Result m n where
  Result Int Int = Int   -- (1)
  Result a Float = Float -- (2)
  Result Float a = Float -- (3)

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. 8. Add Typeclass using Closed Type Family

The system that supports closed type families (System μ**FC**), is an extension of System **FC** [Sulzmann et al. 2007a]. System **FC** is essentially System F[Girard et al. 1989; Reynolds 1974] with type equality coercions. The type equality coercions are special types, that act as proof or witness for equality between types. These type coercions are the workhorse of type rewriting; an essential component to support type level functions. A notable feature of GHC, an implementation of Haskell, is that it compiles the surface language to an explicitly type annotated core language based on System **FC**. The program transformation passes done after type checking also produce well typed System **FC** terms. The surface level expression language for GHC is close to Hindley-Milner language which may not mention any types or coercions. All the coercions thus have to be inferred by the type checker while compiling the source language into to core language.

TODO: : may be don't include this example? but i like them :(

```
data Leaf a
data Node a b

type family TMember e tree where
  TMember e (Leaf e') = TEq e e'
  TMember e (Node lt rt)
    = Or (TMember e lt) (TMember e rt)
```

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

type family CountArgs ty where
  CountArgs (a → b) = S (CountArgs b)
  CountArgs b       = Z
```

Fig. 9. Complex Typelevel Functions

The closedness of type families come from the syntax level restriction where the programmer is not allowed to add any more equations to the type family once they are defined. The utility of closed type families is further elucidated with the examples given in Figure 9. `CountArgs` computes the number of arguments that a type expects while `TMember` can be used to check if a given type exists in a complex type data structure. The advantage of such a restriction is that it makes it easier to define functions on types that would otherwise need some complex encoding, or a type checker which supports backtracking.

4.1 Type Matching and Apartness

Intuitively, the semantics of type family declarations is to lift the type family equations into axioms. Instantiating one of these axiom would give us coercions which can be used as an evidence for equality between two types. We say that a target type τ reduces to or evaluates to another type τ_1 when we can find such an evidence. We say a rule is fired when a certain type family rule is used to generate a coercion evidence term. before we dive into the formalization of this system, there are some subtle details about the static semantics of type reduction that we ought to mention.

The type rewriting is performed by using a top to bottom target matching procedure, where the first instance of the left hand side that matches is used to rewrite the type. This enables the type equations to possibly have have overlapping left hand sides of the equations. The type family definition of `TEq a b` indeed has overlapping equations with `Eq a a` and `Eq a b`. Overlapping equations introduce some complexity in the system, but it also enhances usability. There are two restrictions on the type patterns (1) it should have the same length as the arity of the type family constructor, and (2) the type patterns can not have occurrences of type family constructors. The first condition is necessary to avoid having partially applied type families in our system, while the second condition is to ensure soundness.

Consider `BadTyFam` shown in Figure 10, and a target type `BadTyFam (TEq Int Bool)`. The last equation is fired in this case and we reduce the target to `FF`. However, if we instead evaluate `TEq Int Bool` first we get `TT` and the second equation is fired. This shows that non-determinism in the target matching may introduce unsoundness. We also need to avoid eager target type reduction as shown with the `boom` example, that will case a crash at runtime. The crux of the problem is that, if we (erroneously) reduce `TyFam (TEq Bool d)` we get `()`, as `TEq Bool d` matches the second equation from `TEq a b`, and it reduces to `FF`. However, in the call to `boom`, we see that `d` is instantiated with `Bool` thus `TyFam (TEq Bool d)` it should reduce to `Int → Int`. This is indeed, unsound. A naïve notion of matching where we check for failure for a most general unifier of two

```
type family BadTyFam a where
  BadTyFam TT        = FF
  BadTyFam FF        = TT
  BadTyFam (TEq x y) = FF
type family TyFam b where
  TyFam TT = Int → Int
  TyFam FF = ()
```

```
fun :: d → TyFam (TEq Bool d)
fun _ = ()
boom :: Int
boom = fun True 5 -- 💣
```

Fig. 10. Bad Type Families

types is not a fool proof mechanism as it can lead to non-confluence and inconsistency in presence of type families. We instead use the following definition of apartness using type flattening defined below:

Definition 4 (Type Flattening)**.** We say a type τ is flattened to τ_1, or $\tau_1 = \texttt{flatten}(\tau)$, when every type family application of the from $\texttt{F}(\overline{\sigma})$ is replaced by a type variable, such that in the flattened type, every syntactically equivalent type family application in τ is replaced by same type fresh type variable and syntactically different type family applications in τ are replaced by distinct fresh type variables.

Now apartness of types and matching can be defined as:

Definition 5 (Apart and Matching)**.** We say two types, τ_1 and τ_2 are apart if $\mathit{unify}(\tau_1, \texttt{flatten}(\tau_2))\texttt{fails}$ and τ_1 does not mention type family constructors and dually, $\texttt{match}(\tau_1, \tau_2) = \neg\texttt{apart}(\tau_1, \tau_2)$

Simplification of type families just by using apartness criterion is too restrictive, for example if the two equations simplify to the same right hand side every time they have same left hand sides, they are sound and we should allow them. For example, the first and third equation in `Or a b` type family declaration example in Figure 9 are compatible. We formalize the notion of compatibility of equations below:

Definition 6 (Compatible Equations)**.** The two equations, p and q, are compatible if there exist two substitutions, Ω_1 and Ω_2 such that if $\Omega_1(\text{lhs}_p) = \Omega_2(\text{lhs}_q)$ then $\Omega_1(\text{rhs}_p) = \Omega_2(\text{rhs}_q)$.

Implementing compatibility is simple. For two equations p and q, we find $\Omega = \mathit{unify}(\text{lhs}_p, \text{lhs}_q,)$. If the call to unification fails, then the equations are compatible vacuously else if we do find a substitution, we check if $\Omega\text{rhs}_p = \Omega\text{rhs}_q$.

Definition 7 (Closed Type Family Simplification)**.** An equation, say p, given in the type family declaration can be used to simplify the the target $\mathtt{F}(\overline{\tau})$ to a type if the following two conditions hold:

(1) The type pattern on left hand side of the equation, $\mathtt{N}_p$ matches with the target $\overline{\tau}$ or $\mathtt{match}(\mathtt{N}_p, \overline{\tau})$
(2) Any equation q, that precedes p, is either compatible with p, $\mathtt{compat}(p, q)$, or it's pattern $\mathtt{N}_q$ is apart, $\mathtt{apart}(\mathtt{N}_q, \overline{\tau})$.

TODO: : talk about infitary unification?

4.2 Formalizing Closed Type Families

As the surface level language for GHC is too complex to formalize, we will formalize a small portion of interesting language constructs. The System μ**FC** has coercion types (γ, η), and type family constructors (F) which are different from type constructors as they differ in their static semantics. The axiom equations define the type rewriting strategy for a specific F. For example, TEq type family looks like $\mathtt{AxTEq} : [\forall\alpha.\ \mathtt{TEq}\ \alpha\ \alpha \sim \mathtt{TT}, \forall\alpha\beta.\ \mathtt{TEq}\ \alpha\ \beta \sim \mathtt{FF}]$.

The typing judgments for terms in this system is a triple $\Gamma \vdash e : \tau$ that asserts that there is a derivation such that term e has type τ under the typing context Γ. All the interesting rules are given in Figure 13. The typing environment (Γ) consists of two contexts, the variable context (Δ) which maps free variables to their types, and ground context (Σ) that stores all the type rewriting axioms and also necessary information for type and family constructors. The type validity judgments along with ground context and variable context validity are given in Figure 12. They essentially walk over the context structures and ensuring we do not add anything invalid and check for type and axiom well formedness by making sure the context is consistent by using the goodness criterion described above.

Type family Constructors	F, G
Type Constructors	T
Type Validity	$\Gamma \vdash \tau$ type
Proposition Validity	$\Gamma \vdash P$ prop
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	$\Gamma \vdash_{Co} \gamma : P$
One Hole Type Context	$C[\bullet]$

Types	τ, σ	$::= \alpha \mid \tau \to \tau \mid \forall \alpha.\, \tau \mid \mathsf{F}(\overline{\tau}) \mid \mathsf{T}(\overline{\tau})$
Ground Types	ω	$::= \tau \to \tau \mid \forall \alpha.\, \tau \mid \mathsf{T}\overline{\tau}$
Predicates	P	$::= \tau \sim \tau$
Type family Pattern	N	$::= \overline{\tau}$
Axiom Equations	Φ	$::= \forall \overline{\alpha}.\, \mathsf{F}(\mathsf{N}) \sim \sigma$
Axiom Types	Ψ	$::= \overline{\Phi}$
Coercions	γ, η	$::= \gamma \to \eta \mid \forall \alpha.\, \gamma \mid \gamma @ \tau \mid \mathsf{F}(\overline{\gamma}) \mid \mathsf{T}(\overline{\gamma})$ $\mid \mathrm{nth}_i\, \gamma \mid \langle \tau \rangle \mid \widehat{\gamma} \mid \gamma \circ \eta \mid \xi_i\, \overline{\tau}$
Terms	e	$::= x \mid \lambda x{:}\tau.\, e \mid e\, e \mid M \blacktriangleright \gamma \mid \Lambda \alpha.\, e \mid e\, \tau$ $\mid \mathsf{D}\overline{e}$
Values	$\mathcal{V}$	$::= \lambda x{:}\tau.\, e \mid \mathsf{D}\overline{e} \mid \Lambda \alpha.\, e$
Ground Context	Σ	$::= \epsilon \mid \Sigma, \xi : \Psi \mid \Sigma, \mathsf{F} : n \mid \Sigma, \mathsf{T} : n$
Variable Context	Δ	$::= \epsilon \mid \Delta, x{:}\tau \mid \Delta, \alpha$
Typing Context	Γ	$::= \Sigma; \Delta$

Fig. 11. System for Closed Type Families

$$(\textsc{v-tvar})\ \frac{\alpha \in \Delta \quad \vdash_{ctx} \Sigma; \Delta}{\Sigma; \Delta \vdash \alpha \text{ type}} \qquad (\textsc{v-arr})\ \frac{\Gamma \vdash \tau \text{ type}_1 \quad \Gamma \vdash \tau \text{ type}_2}{\Gamma \vdash (\tau_1 \to \tau_2) \text{ type}} \qquad (\textsc{v-tfa})\ \frac{\Sigma; \Delta, \alpha \vdash \tau \text{ type}}{\Sigma; \Delta \vdash (\forall \alpha.\, \tau) \text{ type}}$$

$$(\textsc{v-tctr})\ \frac{\mathsf{T}{:}n \in \Sigma \quad \vdash_{ctx} \Sigma; \Delta \quad \overline{\Sigma; \Delta \vdash \tau_i \text{ type}}^{i<n}}{\Sigma; \Delta \vdash \mathsf{T}\overline{\tau} \text{ type}} \qquad (\textsc{v-tfctr})\ \frac{\mathsf{F}{:}n \in \Sigma \quad \vdash_{ctx} \Sigma; \Delta \quad \overline{\Sigma; \Delta \vdash \tau_i \text{ type}}^{i<n}}{\Sigma; \Delta \vdash \mathsf{F}\overline{\tau} \text{ type}}$$

$$(\textsc{v-eqp})\ \frac{\Gamma \vdash \tau_1 \text{ type} \quad \Gamma \vdash \tau_2 \text{ type}}{\Gamma \vdash \tau_1 \sim \tau_2 \text{ prop}} \qquad (\textsc{v-tfp})\ \frac{\mathsf{F}{:}n \in \Sigma \quad \overline{\Sigma; \Delta, \overline{\alpha} \vdash N_i \text{ type}}^{i<n} \quad \Sigma; \Gamma, \overline{\alpha} \vdash \sigma \text{ type}}{\Sigma; \Delta \vdash \forall \overline{\alpha}.\, \mathsf{F}(N) \sim \sigma \text{ prop}}$$

$$(\textsc{v-gempt})\ \frac{}{\vdash_{gnd} \epsilon} \qquad (\textsc{v-gtc})\ \frac{\mathsf{T}\#\Sigma \quad \vdash_{gnd} \Sigma}{\vdash_{gnd} \Sigma, \mathsf{T}{:}n} \qquad (\textsc{v-gtf})\ \frac{\mathsf{F}\#\Sigma \quad \vdash_{gnd} \Sigma}{\vdash_{gnd} \Sigma, \mathsf{F}{:}n} \qquad (\textsc{v-gax})\ \frac{\begin{matrix}\vdash_{gnd}\Sigma \\ \xi\#\Sigma\end{matrix} \quad \overline{\Sigma; \epsilon \vdash \forall \overline{\alpha}.\, \mathsf{F}(\mathsf{N}) \sim \sigma \text{ prop}}}{\vdash_{gnd} \Sigma, \xi{:}\overline{\forall \overline{\alpha}.\, \mathsf{F}(\mathsf{N}) \sim \sigma}}$$

$$(\textsc{v-vempt})\ \frac{\vdash_{gnd} \Sigma}{\Sigma \vdash_{var} \epsilon} \qquad (\textsc{v-te})\ \frac{\Sigma \vdash_{var} \Delta}{\vdash_{ctx} \Sigma; \Delta} \qquad (\textsc{v-var})\ \frac{\Sigma \vdash_{var} \Delta \quad x\#\Delta \quad \Sigma; \Delta \vdash \tau \text{ type}}{\Sigma \vdash_{var} \Delta, x{:}\tau} \qquad (\textsc{v-tyvar})\ \frac{\Sigma \vdash_{var} \Delta \quad \alpha\#\Delta}{\Sigma \vdash_{var} \Delta, \alpha}$$

Fig. 12. Validity Judgments

We show the interesting term typing rules in Figure 13 the rest are standard and we skip them due to space constraints, except the interesting one (T-CAST). It says that a well typed term of type τ_1 can be typed using a new type τ_2 if there is a welltyped witness γ that casts it from type τ_1 to τ_2. The three rules (CO-REFL), (CO-SYM), and (CO-TRANS) say that the coercions form an equivalence relation. The four rules (CO-ARR), (CO-TYPE), (CO-FORALL) and (CO-FAM) says that if two types are equal then each of their respective components are also equal, or that they form a congruence relation. The rules (CO-NTHARR) and (CO-NTH) says that we can decompose type qualities into simpler ones. The rule (CO-INST) says that if we have a witness that says two polytypes are equal, then we can obtain a witness where their respective instantiations with a type are also equal types.

$$\text{(T-CAST)}\;\frac{\Gamma \vdash_{Co} \gamma : \tau_1 \sim \tau_2 \quad \Gamma \vdash e : \tau_1}{\Gamma \vdash e : \tau_2}$$

$$\text{(CO-REFL)}\;\frac{\Gamma \vdash \tau \text{ type}}{\Gamma \vdash_{Co} \langle\tau\rangle :: \tau \sim \tau} \qquad \text{(CO-SYM)}\;\frac{\Gamma \vdash_{Co} \gamma : \tau_1 \sim \tau_2}{\Gamma \vdash_{Co} \widehat{\gamma} : \tau_2 \sim \tau_1} \qquad \text{(CO-TRANS)}\;\frac{\Gamma \vdash_{Co} \gamma : \tau_1 \sim \tau_2 \quad \Gamma \vdash_{Co} \eta : \tau_2 \sim \tau_3}{\Gamma \vdash_{Co} \gamma \circ \eta : \tau_1 \sim \tau_3}$$

$$\text{(CO-ARR)}\;\frac{\Gamma \vdash_{Co} \gamma : \tau_1 \sim \sigma_1 \quad \Gamma \vdash_{Co} \eta : \tau_2 \sim \sigma_2}{\Gamma \vdash_{Co} \gamma \to \eta : (\tau_1 \to \tau_2) \sim (\sigma_1 \to \sigma_2)} \qquad \text{(CO-NTHARR}_i\text{)}\;\frac{\Gamma \vdash_{Co} \gamma : (\tau_1 \to \tau_2) \sim (\sigma_1 \to \sigma_2)}{\Gamma \vdash_{Co} \text{nth}_i\, \gamma : \tau_i \sim \sigma_i}$$

$$\text{(CO-TYPE)}\;\frac{\mathsf{T}{:}n \in \Sigma \quad \overline{\Sigma; \Delta \vdash_{Co} \gamma_i : \tau_i \sim \sigma_i}^{\,i<n}}{\Sigma; \Delta \vdash_{Co} \mathsf{T}\overline{\gamma} : \mathsf{T}\overline{\tau} \sim \mathsf{T}\overline{\sigma}} \qquad \text{(CO-NTH)}\;\frac{\Gamma \vdash_{Co} \gamma : \mathsf{T}\overline{\tau} \sim \mathsf{T}\overline{\sigma}}{\Gamma \vdash_{Co} \text{nth}_i\, \gamma : \tau_i \sim \sigma_i}$$

$$\text{(CO-FORALL)}\;\frac{\Gamma, \alpha \vdash_{Co} \gamma : \tau_1 \sim \tau_2}{\Gamma \vdash_{Co} \forall\alpha.\, \gamma : (\forall\alpha.\, \tau_1) \sim (\forall\alpha.\, \tau_2)} \qquad \text{(CO-INST)}\;\frac{\Gamma \vdash \tau \text{ type} \quad \Gamma \vdash_{Co} \gamma : \forall\alpha.\, \sigma_1 \sim \forall\alpha.\, \sigma_2}{\Gamma \vdash_{Co} \gamma@\tau : \sigma_1[\alpha/\tau] \sim \sigma_2[\alpha/\tau]}$$

$$\text{(CO-FAM)}\;\frac{\mathsf{F}{:}n \in \Sigma \quad \overline{\Sigma; \Delta \vdash_{Co} \gamma_i : \tau_i \sim \sigma_i}^{\,i<n}}{\Sigma; \Delta \vdash_{Co} \mathsf{F}\overline{\gamma} : \mathsf{F}\overline{\tau} \sim \mathsf{F}\overline{\sigma}} \qquad \text{(CO-AXIOM)}\;\frac{\begin{array}{c}\Psi = \overline{\forall\overline{\alpha}.\, \mathsf{F}(\mathsf{N}) \sim \sigma}\\ \xi{:}\Psi \in \Sigma\end{array} \quad \begin{array}{c}\overline{\Sigma;\Delta \vdash \tau_i \text{ type}}\\ \forall j<i.\ \texttt{no\_conflict}(\Psi, i, \overline{\tau}, j)\end{array} \quad \vdash_{ctx} \Sigma; \Delta}{\Sigma; \Delta \vdash_{Co} \xi_i\, \overline{\tau} : \mathsf{F}(\mathsf{N}\overline{[\alpha_i/\tau_i]}) \sim \sigma\overline{[\alpha_i/\tau_i]}}$$

Fig. 13. Typing Judgments System μ**FC**

The most interesting rule is the behemoth, (CO-AXIOM) which gives conditions as to when we can use a particular coercion axiom to rewrite a type. In our example of `TEq`, there are two possible ways in which the axiom could have been instantiated: $\texttt{AxTEq}_0\texttt{[Int]} : \texttt{TEq Int Int} \sim \texttt{TT}$ or $\texttt{AxTEq}_1\texttt{[Int, Int]} : \texttt{TEq Int Int} \sim \texttt{FF}$. But the second option would make the system unsound. The `no_conflict` check saves us from this disaster by allowing only the first option and rejecting the second. There are two ways in which the equations are in non-conflicting as shown in Figure 14—1) either the equations are compatible or 2) the equations are apart.

$$\text{(NC-APART)}\;\frac{\Psi = \overline{\mathsf{F}(\mathsf{N}) \sim \sigma} \quad \texttt{apart}(\mathsf{N}_j, \mathsf{N}_i[\overline{\tau}/\overline{\alpha}_i])}{\texttt{no\_conflict}(\Psi, i, \overline{\tau}, j)} \qquad \text{(COMPT-DIS)}\;\frac{\begin{array}{c}\Phi_1 = \mathsf{F}(\mathsf{N}_1) \sim \sigma_1\\ \Phi_2 = \mathsf{F}(\mathsf{N}_2) \sim \sigma_2\end{array} \quad \Omega = \mathit{unify}(\mathsf{N}_1, \mathsf{N}_2)\texttt{fails}}{\texttt{compat}(\Phi_1, \Phi_2)}$$

$$\text{(NC-COMPT)}\;\frac{\texttt{compat}(\Psi[i], \Psi[j])}{\texttt{no\_conflict}(\Psi, i, \overline{\tau}, j)} \qquad \text{(COMPT-INC)}\;\frac{\begin{array}{c}\Phi_1 = \mathsf{F}(\mathsf{N}_1) \sim \sigma_1\\ \Phi_2 = \mathsf{F}(\mathsf{N}_2) \sim \sigma_2\end{array} \quad \begin{array}{c}\Omega = \mathit{unify}(\mathsf{N}_1, \mathsf{N}_2)\\ \Omega\sigma_1 = \Omega\sigma_2\end{array}}{\texttt{compat}(\Phi_1, \Phi_2)}$$

Fig. 14. Non Conflicting Equations and Compatibility

4.2.1 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 ξ, $\forall\overline{\alpha}.\ \mathsf{F}(\mathsf{N}_i) \sim \sigma_i$ is used to reduce the target type $\mathsf{F}(\overline{\tau})$ to type τ_1. The `no_conflict` check is the same as discussed before and we use $C[\bullet]$ to mean that the rewrite can happen anywhere within the type.

$$(\textsc{ty-red})\ \frac{\begin{array}{c}\xi{:}\Psi\in\Sigma\\ \vdash_{gnd}\Sigma\end{array}\quad \begin{array}{c}\Psi=\overline{\forall\overline{\alpha}.\ \mathsf{F}(\mathsf{N})\sim\sigma}\\ \forall j<i.\ \texttt{no\_conflict}(\Psi,i,\overline{\tau},j)\end{array}\quad \begin{array}{c}\Omega=unify(N_i,\overline{\tau})\\ \tau_1=\Omega\sigma_i\end{array}}{\Sigma\vdash C[\mathsf{F}(\overline{\tau})]\rightsquigarrow C[\tau_1]}$$

4.2.2 *Consistency and Goodness of Context.*

Consistency means that we can never derive unsound equalities between ground types, such as $\gamma : \texttt{Int} \sim \texttt{Bool}$, in the system. Ground types (ω), 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 Σ is consistent, when for all coercions γ such that $\Sigma; \epsilon \vdash_{Co} \gamma : \omega_1 \sim \omega_2$ we have the following:

(1) if ω_1 is of the form $\mathsf{T}\overline{\tau}$ then so is ω_2
(2) if ω_1 is of the form $\tau_1 \longrightarrow \tau_2$ then so is ω_2
(3) if ω_1 is of the form $\forall\alpha.\ \tau$ then so is ω_2

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

Property 8 (Good Σ). *A ground context (Σ) is* Good, *written* $\mathsf{Good}\Sigma$ *when the following conditions are met for all $\xi{:}\Psi \in \Sigma$ and where Ψ is of the form $\overline{\forall\overline{\alpha}.\ \mathsf{F}_i(\mathsf{N}_i) \sim \sigma}$:*

(1) There exists an F such that $\forall i.\ \mathsf{F}_i = \mathsf{F}$ and none of the type pattern 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 $\mathsf{Good}\Sigma$ then, Σ 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 τ_1 and τ_2 have a common reduct, say τ_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 9 (Consistency). *If $\Sigma \vdash \bullet \rightsquigarrow \bullet$ is terminating and $\mathsf{Good}\Sigma$ then Σ is consistent.*

4.3 Type safety of System μFC

As our system has a variety of new terms, we give a new definition of the $\bullet \rightsquigarrow \bullet$ relation using the rules given in Figure 15. Similar to the previous Section 3.4, we should have type safety for System μFC which includes proving preservation and progress.

$$\text{(S-APP)}\ \frac{e_1 \rightsquigarrow e_1'}{e_1\ e_2 \rightsquigarrow e_1'\ e_2} \qquad \text{(S-TAPP)}\ \frac{e_1 \rightsquigarrow e_1'}{e_1\ \tau \rightsquigarrow e_1'\ \tau} \qquad \text{(S-CAPP)}\ \frac{e_1 \rightsquigarrow e_1'}{e_1\ \gamma \rightsquigarrow e_1'\ \gamma} \qquad \text{(S-CAST)}\ \frac{e_1 \rightsquigarrow e_1'}{e_1 \blacktriangleright \gamma \rightsquigarrow e_1' \blacktriangleright \gamma}$$

$$\text{(S-}\beta\text{)}\ \frac{}{(\lambda x{:}\tau.\ e_1)\ e_2 \rightsquigarrow e_1[x/e_2]} \qquad \text{(S-T}\beta\text{)}\ \frac{}{(\Lambda\alpha.\ e)\ \tau \rightsquigarrow e[\alpha/\tau]} \qquad \text{(S-TRANS)}\ \frac{}{(e \blacktriangleright \gamma) \blacktriangleright \eta \rightsquigarrow e \blacktriangleright \gamma \circ \eta}$$

$$\text{(S-PUSH)}\ \frac{\gamma_1 = \overparen{\text{nth}_0\ \gamma} \quad \gamma_2 = \text{nth}_1\ \gamma}{(\lambda x{:}\tau.\ e \blacktriangleright \gamma)\ e_1 \rightsquigarrow (\lambda x{:}\tau.\ e)\ (e_1 \blacktriangleright \gamma_1) \blacktriangleright \gamma_2} \qquad \text{(S-TPUSH)}\ \frac{}{(\Lambda\alpha.\ e \blacktriangleright \gamma)\ \tau \rightsquigarrow (\Lambda\alpha.\ e)\ \tau \blacktriangleright \gamma@\tau}$$

Fig. 15. Small Step Operational Semantics System μ**FC**

For proving preservation lemma, we would have to prove term substitution lemma which, in turn, will require that substitutions in coercions are sound. This is given by coercion substitution lemma.

Lemma 10 (Coercion Subst). *If* $\Sigma; \Delta, \alpha, \Delta' \vdash_{Co} \gamma : P$ *and* $\Sigma; \Delta \vdash \tau$ *type then,* $\Sigma; \Delta, \Delta' \vdash_{Co} \gamma[\alpha/\tau] : P[\alpha/\tau]$

The most interesting case would be to prove (C-AXIOM) case, but the restrictions due to `no_conflict` will be enough.

Lemma 11 (Preservation System μFC). *if* $\epsilon \vdash e : \tau$ *and* $e \rightsquigarrow e'$ *then* $\epsilon \vdash e' : \tau$

To have progress property, it is necessary to have consistency which assumes termination of type reduction.

Lemma 12 (Progress System μFC). *If* $\Sigma; \epsilon \vdash e : \tau$ *and* $\Sigma \vdash \bullet \rightsquigarrow \bullet$ *is terminating, then either* $e \in \mathcal{V}$*, or* e *is a coerced value of the form* $e' \blacktriangleright \gamma$ *where* $e' \in \mathcal{V}$ *or there exists a* e_1 *such that* $e \rightsquigarrow e_1$.

5 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 16. 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.

Next, consider the target type `TEq [a] a`, in System μ**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

```
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
```

Fig. 16. Partial Closed Type Family

`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.

5.1 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 typeclass world using `TEqC a b` and `PlusC m n` respectively as shown in Figure 17. In the constrained type families world, every closed type family will be associated with a closed typeclass. 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

```
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. 17. Closed Typeclasses Examples

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.

5.2 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.

5.3 Formalizing Constrained Type Families

This system is similar to System μ**FC** except for a few new constructors that we highlight in Figure 18. The ground context keeps track of two types of type family constructors, a total type family constructor of arity n (written $\mathsf{F}{:}_{\top} n$) and a partial type family constructor of arity n (written $\mathsf{F}{:}n$). The variable environment (Δ) along with variable type bindings also stores coercion constraint bindings $c{:}P$. Each equation that is introduced by axioms (ξ) in this system, are of the form $\forall \overline{\alpha}\ \overline{\chi}.\ \mathsf{F}(\overline{\tau}) \sim \sigma$. Both $\overline{\tau}$ and σ 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 $\alpha|c{:}\mathsf{F}(\overline{\tau}) \sim \alpha$ and read as "α such that c witnesses $\mathsf{F}(\overline{\tau})$ reduces to α". 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 χ to remind us that it is more specific than P; for any χ, the left hand side of the type equality is always a type family and right and side is a fresh variable. The `assume` χ `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
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	τ, σ	$::= \alpha \mid \tau \to \tau \mid \forall\alpha.\,\tau \mid \mathsf{F}(\overline{\tau}) \mid \mathsf{T}(\overline{\tau}) \mid P \Rightarrow \tau$
Ground Types	ω	$::= \tau \to \tau \mid \forall\alpha.\,\tau \mid \mathsf{T}(\overline{\tau})$
Predicates	P	$::= \tau \sim \tau$
Axiom Equations	Φ	$::= \forall\overline{\alpha}\ \overline{\chi}.\ \mathsf{F}(\overline{\tau}) \sim \sigma$
Axiom Types	Ψ	$::= \overline{\Phi}$
Coercions	γ, η	$::= c \mid \gamma \to \eta \mid \forall\alpha.\,\gamma \mid \gamma@\tau \mid \mathsf{F}(\overline{\gamma}) \mid \mathsf{T}(\overline{\gamma})$
		$\mid \mathrm{nth}_i\,\gamma \mid \langle\tau\rangle \mid \widehat{\gamma} \mid \gamma \circ \eta \mid \gamma_1 \sim \gamma_2 \Rightarrow \eta \mid \xi_i\,\overline{\tau}\,\overline{q}$
Eval Assumption	χ	$::= (\alpha \mid c : \mathsf{F}\overline{\tau} \sim \alpha)$
Eval Resolution	q	$::= (\tau \mid \gamma)$
Terms	e	$::= x \mid \lambda x{:}\tau.\,e \mid e\,e \mid M \blacktriangleright \gamma \mid \Lambda\alpha.\,e \mid e\,\tau \mid \mathsf{D}\overline{e}$
		$\mid \lambda c{:}P.\,e \mid e\,\gamma \mid$ `assume` χ `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	Σ	$::= \epsilon \mid \Sigma, \xi{:}\Psi \mid \Sigma, \mathsf{F}{:}_{\top}n \mid \Sigma, \mathsf{F}{:}n \mid \Sigma, \mathsf{T}{:}n$
Variable Context	Δ	$::= \epsilon \mid \Delta, \alpha \mid \Delta, x{:}\tau \mid \Delta, c{:}P$
Typing Context	Γ	$::= \Sigma; \Delta$

Fig. 18. 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 μ**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 5.3.2. 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.

$$\text{(v-qty)}\ \frac{\Gamma \vdash P \text{ prop} \quad \Gamma \vdash \tau \text{ type}}{\Gamma \vdash P \Rightarrow \tau \text{ type}} \qquad \text{(v-qtfp)}\ \frac{\mathsf{F}{:}n \in \Sigma \quad \overline{\Sigma; \Gamma \vdash \tau_i \text{ type}}^{i<n} \quad \Sigma; \Delta \vdash \sigma \text{ type}}{\Sigma; \Delta \vdash \mathsf{F}(\overline{\tau}) \sim \sigma \text{ prop}}$$

$$\text{(v-rese)}\ \frac{\vdash_{ctx} \Gamma}{\Gamma \vdash_{res} \epsilon : \epsilon} \qquad \text{(v-resc)}\ \frac{\Gamma \vdash \sigma \text{ type} \quad \Gamma \vdash_{Co} \gamma : \mathsf{F}(\overline{\tau}) \sim \sigma \quad \Gamma \vdash_{res} \overline{q} : \overline{\chi}[\alpha/\sigma]}{\Gamma \vdash_{res} (\sigma|\gamma), \overline{q} : (\alpha|c{:}\mathsf{F}(\overline{\tau}) \sim \alpha), \overline{\chi}}$$

$$\text{(v-assmn)}\ \frac{}{\Gamma \vdash \epsilon \text{ asmp}} \qquad \text{(v-assmc)}\ \frac{\mathsf{F}{:}n \in \Sigma \quad \overline{\Sigma; \Delta \vdash \tau_i \text{ type}}^{i<n} \quad \Sigma; \Delta, \alpha \vdash \overline{\chi} \text{ asmp}}{\Sigma; \Delta \vdash (\alpha|c{:}\mathsf{F}(\overline{\tau}) \sim \alpha), \overline{\chi} \text{ asmp}}$$

$$\text{(v-qgax)}\ \frac{\begin{array}{c}\vdash_{gnd}\Sigma \\ \xi\#\Sigma\end{array} \quad \dfrac{\overline{\Sigma;\overline{\alpha}_i, TV(\overline{\chi}) \vdash \tau_{0i} \text{ type}}}{\Sigma;\epsilon \vdash \forall\overline{\alpha}.\ \mathsf{F}(\overline{\tau}) \sim \sigma \text{ prop}} \quad \begin{array}{c}\overline{\Sigma;\overline{\alpha}_i \vdash \overline{\tau}_i \text{ type}} \\ \Sigma;\overline{\alpha}_i \vdash \overline{\chi}_i \text{ asmp}\end{array} \quad \mathsf{F}{:}n \in \Sigma}{\vdash_{gnd} \Sigma, \xi{:}\overline{\forall\overline{\alpha}\ \overline{\chi}.\ \mathsf{F}(\overline{\tau}_i) \sim \tau_{0i}}^{i<k}} \qquad \text{(v-qcovar)}\ \frac{\Sigma \vdash_{var} \Delta \quad \Sigma; \Gamma \vdash P \text{ prop} \quad c\#\Delta}{\Sigma \vdash_{var} \Delta, c{:}P}$$

Fig. 19. Validity Judgments for System *QFC*

$$\text{(co-var)}\ \frac{c{:}P \in \Delta}{\Sigma; \Delta \vdash c : P} \qquad \text{(co-qaxiom)}\ \frac{\begin{array}{c}\xi{:}\Psi \in \Sigma \\ \Psi = \overline{\forall\overline{\alpha}\ \overline{\chi}.\ \mathsf{F}(\overline{\tau}) \sim \sigma}\end{array} \quad \begin{array}{c}\overline{\Sigma;\Delta \vdash \tau_i \text{ type}} \\ \forall j<i.\ \mathtt{no\_conflict}(\Psi, i, \overline{\tau}, j)\end{array} \quad \begin{array}{c}\vdash_{ctx}\Sigma;\Delta \\ \Sigma;\Delta \vdash_{res} \overline{q}{:}\overline{\chi}[\alpha_i/\tau_i]\end{array}}{\Sigma; \Delta \vdash_{Co} \xi_i\ \overline{\tau}\ \overline{q} : \mathsf{F}(\overline{\sigma}\overline{[\alpha_i/\tau_i]}) \sim \sigma_0\overline{[\alpha_i/\tau_i]}}$$

$$\text{(t-coabs)}\ \frac{\Gamma, c{:}P \vdash e : \tau}{\Gamma \vdash \lambda c{:}P.\ e : P \Rightarrow \tau} \qquad \text{(co-qual)}\ \frac{\overline{\Gamma \vdash \gamma_i : \tau_i \sim \sigma_i}^{i<3}}{\Gamma \vdash_{Co} \gamma_1 \sim \gamma_2 \Rightarrow \gamma_3 : (\tau_1 \sim \tau_2 \Rightarrow \tau_2) \sim (\sigma_1 \sim \sigma_2 \Rightarrow \sigma_3)}$$

$$\text{(t-coapp)}\ \frac{\Gamma \vdash e : P \Rightarrow \tau \quad \Gamma \vdash \gamma : P}{\Gamma \vdash e\,\gamma : \tau} \qquad \text{(t-assum)}\ \frac{\overline{\Sigma; \Delta \vdash \tau_i \text{ type}} \quad \mathsf{F}{:}_{\top} n \in \Sigma \quad \Sigma; \Delta, \alpha, c{:}\mathsf{F}\overline{\tau} \sim \alpha \vdash e : \tau}{\Sigma, \Delta \vdash \mathtt{assume}\ (\alpha|c{:}\mathsf{F}\overline{\tau} \sim \alpha)\ \mathtt{in}\ e : \tau}$$

Fig. 20. Selected Typing Judgments System *QFC*

The typing judgments new to this system are shown in Figure 20. 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 α in validity assumptions $(\alpha|c{:}\mathsf{F}(\overline{\tau}) \sim \sigma)$ is instantiated to τ and γ 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.

5.3.1 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 (Ω_1) as well as evaluation resolutions (Ω_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.

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

Fig. 21. Type reduction

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

PROPERTY 13 (Good Σ RELAXED). *A ground context (Σ) is* Good*, written* GoodΣ *when the following conditions are met for all $\xi{:}\Psi \in \Sigma$ and where Ψ is of the form $\overline{\forall\overline{\alpha}\ \overline{\chi}.\ \mathsf{F}_i(\mathsf{N}_i) \sim \sigma}$:*

(1) There exists an 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 14 (CONSISTENCY). *If* GoodΣ *then Σ is consistent*

5.4 Type safety of System *Q*FC

Selected rules for term evaluation are shown in Figure 22. The rule (T-QRES) evaluates a constraint function application similar to function application while the rule (T-QPUSH) splits the inner coercion η so that it can be commuted with the coercion application γ. The rule (T-QRES) is the rule that illuminates the evaluation for total type family constructors. The clause $\mathsf{F}(\overline{\tau}) \Downarrow q$ says that we find a witness for $\mathsf{F}(\tau)$ reduction to a ground type to build an appropriate evaluation resolution and apply it to the enclosing term. We can thus state the preservation and progress for this system as follows:

$$(\textsc{t-qres})\ \frac{\chi = (\alpha | c{:}\mathsf{F}(\overline{\tau}) \sim \alpha) \qquad \mathsf{F}(\overline{\tau}) \Downarrow q}{\texttt{assume}\ \chi\ \texttt{in}\ e \rightsquigarrow e[\chi/q]} \qquad (\textsc{t-c}\beta)\ \frac{}{(\lambda c{:}P.\ e)\ \gamma \rightsquigarrow e[c/\gamma]} \qquad (\textsc{t-qpush})\ \frac{\begin{array}{cc} v{=}\lambda c{:}P.\ e & \eta_1{=}\widehat{\mathrm{nth}_1\ \eta} \\ \eta_0{=}\mathrm{nth}_0\ \eta & \eta_2{=}\mathrm{nth}_2\ \eta \end{array}}{(v \blacktriangleright \eta)\ \gamma \rightsquigarrow v\,(\eta_0 \circ \gamma \circ \eta_1) \blacktriangleright \eta_2}$$

Fig. 22. Small Step Operational Semantics System $\mathcal{Q}$**FC**

Lemma 15 (Preservation System $\mathcal{Q}$**FC**). *If $\epsilon \vdash e : \tau$ and $e \rightsquigarrow e'$ then $\epsilon \vdash e' : \tau$*

Lemma 16 (Progress System $\mathcal{Q}$**FC**). *If $\Sigma; \epsilon \vdash e : \tau$ then either e is a value, or e is a coerced value of the form $e' \blacktriangleright \gamma$ where $e' \in \mathcal{V}$ or there exists a e_1 such that $e \rightsquigarrow e_1$.*

6 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. 2007b].

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

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.

Jean-Yves Girard, Paul Taylor, and Yves Lafont. 1989. *Proofs and Types*. Cambridge University Press.

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

Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* 17, 3 (1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4

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

John C. Reynolds. 1974. Towards a theory of type structure. In *Programming Symposium* (1974) *(Lecture Notes in Computer Science)*, B. Robinet (Ed.). Springer Berlin Heidelberg, Syracuse, NY, USA, 408–425.

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, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. 2007a. System F with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation* (New York, NY, USA, 2007-01-16) *(TLDI '07)*. Association for Computing Machinery, 53–66. https://doi.org/10.1145/1190315.1190324

Martin Sulzmann, Gregory J. Duck, Simon Peyton-Jones, and Peter J. Stuckey. 2007b. 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