Dialects for Type Computations

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 compiler 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 illustrate, formalize, and compare these two language features.

1 INTRODUCTION

Parametric polymorphism is a powerful technique that allows programs to work on a wide variety of types. The identity function, which takes an input and returns it without modification has the type $\forall\alpha.\ \alpha \to \alpha$. We read this type as: for all types α, if the argument is of type α then the function returns a value of the type α. We also need to tame unconstrained polymorphism. A division function does not make sense on all types. We cannot divide a function, which multiplies two numbers, by another function, which adds two numbers. The type $\forall\ \alpha.\ (\alpha \times \alpha) \to \alpha$, that accepts a pair of values of type α, and returns the first argument divided by the second argument is too general to describe division. A constrained polymorphic type, $\forall\ \alpha.\ \texttt{(Dividable}\ \alpha\texttt{)} \Rightarrow (\alpha \times \alpha) \to \alpha$, more accurately describes the functions intention. Intuitively, the predicate, `Dividable` α, means: only those types have a meaningful divide function that satisfy this predicate. Typeclasses[Wadler and Blott 1989] give a mechanism of having such constrained polymorphic types. The 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 defines a relation on types. This gives programmers a way to perform computations at type level. However, using relations to perform type computations is cumbersome. A new language feature, type families[Schrijvers et al. 2007], was introduced in Haskell to enable partial type functions. They are stylistically more obvious for functional programmers. Naturally, type families warrants a richer system of types, and reasoning about type safety for such systems is nontrivial.

The scope of the current article is as follows: we first describe typeclasses in Section 2, and then describe functional dependencies[Jones 2000] with some examples in Section 3.1. We formalize them in Section 3.2 and describe their consequences in Section 3.3. We have a brief discussion about associated types as a way to have type functions in Section 4. We then describe two flavors

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

of type families—closed type families[Eisenberg et al. 2014] in Section 5 and constrained type families[Morris and Eisenberg 2017] in Section 6. We formalize the important portions of both the systems in Section 5.3 and Section 6.2 respectively to state the type consistency property. To conclude, we point towards some open questions in Section 7. For presenting code examples, we will be using Haskell[Marlow and Peyton Jones 2010] like syntax.

2 TYPECLASSES

Typeclasses can be thought of as predicates on types. Each typeclass is accompanied by its member functions that all the instances support. For example, equality can be expressed as a typeclass `Equal a` shown below:

```
class Equal a where
  eq::(a×a) → Bool

instance Equal Int where
  eq(a,b) = primEQInt(a,b)

instance Equal Char where
  eq(a,b) = primEQChar(a,b)
```

The instances of `Equal a` typeclass can be types such as integers (`Int`) and characters (`Char`). We say that `Int` and `Char` satisfy the predicate `Equal a`. An equality instance on function types (`a→b`) is not meaningful. The function `eq` is not truly polymorphic: it cannot operate on function types. Rather, it is constrained to only those types that have an `Equal` instance defined. We make it explicit in the type of this function, `∀a.Equal a ⇒ (a×a) → Bool`. We read this type as: for any type `a` satisfying the predicate `Equal a`, the function when given an argument of a pair type, `(a×a)`, returns a value of type `Bool`.

```
class Addition m n p where
  add :: (m × n) → p

instance Addition Float Float Float
  add (f1, f2) = primAddF (f1, f2)

instance Addition Int Float Float where
  add (i, f) = primAddF ((toFlt i), f)

instance Addition Int Float Int where
  add (i, f) = primAddI (i, (trucFtoI f))

-- expr :: (Addition Int Float b, Addition b Int c) ⇒ c
expr = add (add (1, 2.5), 3)
```

Fig. 1. Multiparameter Typeclasses

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, `Addition m n p` is shown in Figure 1. It represents a relation on types `m`, `n` and `p` such that, adding values of type `m`

and type `n` returns a value of type `p`. The type of the function, `add`, which performs the addition operation, is given as `∀a b c. Addition a b c ⇒ (a × b) → c`. Instances of such a typeclass would be `Addition Float Float Float`, `Addition Int Float Float`, and so on.

Although more general than single parameter typeclasses, multiparameter typeclasses can be difficult to use in practice. Suppose, the programmer defines two instances: `Addition Int Float Float` and `Addition Int Float Int` and writes an term `expr = add(add(1,2.5), 3)`. Due to the use of the function `add` in `expr`, its most general type synthesized by the type inference algorithm will be `∀ b c. (Addition Int Float b, Addition b Int c) ⇒ c`. Notice how the type variable `b` occurs only in the predicate set `(Addition Int Float b, Addition 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. The compiler cannot choose a unique interpretation of the sub-term `add(1, 2.5)` as it can very well be evaluated to a value `4` of type `Int`, or a value `3.5` of type `Float`. Haskell disallows such definitions with ambiguous types and reports a type error. To resolve these type errors, the programmer needs to provide explicit type annotations as hints to the type inference algorithm.

3 FUNCTIONAL DEPENDENCIES

3.1 Examples

```
class Addition m n p | m n → p where
  add :: (m×n) → p

instance Addition Int Float Int where -- Error!
  add(a,b) = ...

instance Addition Int Float Float where
  add(a,b) = ...
```

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

Typeclasses with functional dependencies[Jones 2000] generalizes multiparameter typeclasses. It introduces a new syntax for typeclass declarations allowing users to specify a dependency between the type parameters. The syntax for instance declarations does not change. `Addition m n p` typeclass, as shown in Figure 2, has a functional dependency between the type parameters such that types `m` and `n` determine the type `p`. In other words, fixing the type `m` and `n` will also fix 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`".

Functional dependencies aid the programmers to specify the intention of the typeclasses more accurately and in turn gives the compiler a way to detect and report inconsistent instances. For example, the functional dependency `m n → p` on the type class `Addition m n p`, helps the compiler flag the instance `Addition Int Float Int` to be in conflict with the instance `Addition Int Float Float`.

An advantage of functional dependencies is that it can aid in determining ambiguous type variables using an improvement technique. Let's reconsider the ambiguous type of term `expr` from the previous section, `(Addition Int Float b, Addition Float Int c) ⇒ c`. We can assert that `b` has to be equal to `Float`, as it is determined by the types `Int` and `Float` using the typeclass instance, `Addition Int Float Float`. Thus, `expr` no longer has an ambiguous type, as it is inferred as `Addition Float Int c ⇒ c`. We can go a step further, and improve this seemingly polymorphic type as well. The type variable `c` can be determined to be `Float`, giving us the type of `expr` to be `Float`. It would be impossible to make such an improvement without the functional dependency on the typeclass.

```
data Z    -- 0
data S n  -- 1 + n

class IsPeano p
instance IsPeano Z
instance IsPeano n ⇒ IsPeano (S n)

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)
```

```
data Vector s e = Vec ([e])

concat_vec :: Plus m n p
           ⇒ Vector m e
           → Vector n e
           → Vector p e
concat_vec (Vec l1) (Vec l2)
     = Vec (l1 ++ l2)
```

Fig. 3. Peano Arithmetic and Vector Concatenation

We can also perform Peano arithmetic at type level using functional dependencies, as shown in Figure 3. The two datatypes `Z` and `S n` represent the number zero and successor of a number `n`, respectively at type level. The instances of `IsPeano p` assert that: `Z` is a Peano number, and if `n` is a Peano number then `S n` is a Peano number. The intention of `Plus m n p` typeclass is to build a relation such that the addition of first two type parameters is equal to the third. The instances of `Plus m n p` type class represent the axioms of Peano arithmetic: `Plus Z m m` represents the axiom $0 + m = m$, and `Plus n m p ⇒ Plus (S n) m (S p)` represents the axiom, if $n + m = p$ then $(1 + n) + m = (1 + p)$ at type level. The functional dependency `m n → p` plays an important role here: it ensures the result of addition of two numbers is a unique number. The `concat_vec` function

demonstrates why such type level computations might 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 demonstrate how functional dependencies help in encoding type level functions. The vanilla multiparameter typeclasses lacked the capability of restricting the relation on types that is sometimes necessary to accurately represent the programmers intention.

3.2 Formalizing Typeclasses with Functional Dependencies

		Class Predicates	π	$::= C\overline{\tau}$
Type Variables	α, β	Predicate Set	P, Q	$::= \overline{\pi}$
Term Variables	x, y	Types	τ	$::= \alpha \mid \tau \rightarrow \tau \mid \mathsf{T}$
Class Constructors	C	Qualified Types	ρ	$::= \tau \mid P \Rightarrow \rho$
Type Constants	T	Type Schemes	σ	$::= \forall\overline{\alpha}.\rho$
		Terms	e	$::= x \mid e\ e \mid \lambda x.\ e$
Term Typing	$P \mid \Gamma \vdash e : \tau$			
		Typing Environment	Γ	$::= \epsilon \mid \Gamma, x{:}\sigma$

Fig. 4. System **TCFD**

In the previous section, 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 language as shown in Figure 4, and call it System **TCFD**. The types (τ) can be type variables (α), function types ($\tau \rightarrow \tau$), o r type constants (T), such as `Int`, `Float` etc. The qualified types (ρ) have the form $P \Rightarrow \rho$ where the predicate set, P, constrains the type ρ. Type schemes (σ) are quantified constraint types. The terms in the language (e) are term variables (x, y), functions ($\lambda x.\ e$), where x is the argument to the function body e, and function applications ($e_1\ e_2$), which represents calling a function e_1 with an argument e_2.

We write **`class`** $P \Rightarrow$ `C t` for a typeclass declaration, where `t` are the type parameters of the class and P are the constraints that the type parameters additionally should satisfy. For an instance of typeclass `C`, we write **`instance`** $P \Rightarrow$ `C t`, where length of `t` matches the typeclass arity and additional constraints given by P on `t` need to be satisfied. We denote the set of functional dependencies of class `C` with $\mathcal{F}_{\mathsf{C}}$. For an arbitrary functional dependency $X \rightarrow Y$, the determinant of a functional dependency is denoted by t_X, and the dependent by t_Y. Given a set of functional dependencies `F`, we define the closure operation, Z^+_{F}, on $Z \subseteq t$, to be equal to all the type parameters that are determined by the functional dependencies in `F`.

For example, for a typeclass declaration **`class`** `Addition m n p | m n` $\rightarrow$ `p`, we have, $t = (m, n, p)$, $\mathcal{F}_{\texttt{Addition}} = \{$ `m n` $\rightarrow$ `p` $\}$. For the functional dependency `m n` $\rightarrow$ `p`, we have, $t_X = (\mathtt{m}, \mathtt{n})$ and $t_Y = (\mathsf{p})$, and $\{m\}^+_{\mathcal{F}_{\mathsf{Add}}} = \{m\}$ and $\{m, n\}^+_{\mathcal{F}_{\mathsf{Add}}} = \{m, n, p\}$.

The type environment, Γ, is a mapping between term variables to types such that any term variable appears at most once. We write $dom(\Gamma)$ to mean the set of all term variables in Γ, or $dom(\Gamma) = \{x \mid (x{:}\tau) \in \Gamma\}$. We denote Γ_x to be Γ obtained after removing the binding for x in Γ. The typing judgments are of the form $P \mid \Gamma \vdash e : \sigma$. They assert existence of a typing derivation that shows the term e, has the type σ, satisfying the predicates P, and the free term variables in e are assigned types by the typing environment Γ. The typing rules that build the typing derivations in System **TCFD** are summarized in Figure 5. In their present form, they do not use the functional dependencies.

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

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

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

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

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

The left column lists the term typing rules that do not interact with predicates. Each term structure has a typing rule specifying conditions that make the term well typed. The rule (VAR) says that if the variable x has type σ then typing environment Γ should contain the binding $x{:}\sigma$. The rule $(\rightarrow I)$ says that if a term, e, has type τ_2 with a free variable x of type τ_1 then the term $\lambda x.\ e$ has a type $\tau_1 \rightarrow \tau_2$. The rule $(\rightarrow E)$ that says if a term (function) e_1 is of type $\tau_2 \rightarrow \tau$ and another term (argument) e_2 is of type τ_2, then the term $e_1\ e_2$ is of type τ.

The right column lists typing rules that involve the predicate set. The rule $(\forall\, I)$ generalizes the type. We use $TV(X)$ to denote the set of free type variables in object X. The rule $(\forall\, E)$ instantiates the type. It is necessary for β to be a fresh type variable to ensure it does not conflict with the existing free type variables. We write $[x \mapsto y]X$ to denote a substitution where the variable x in the object X is mapped to y. The rule $(\Rightarrow I)$ moves the global predicate Q in to constrain the type ρ while the $(\Rightarrow E)$ moves the constraint out of the type ρ. The condition $P \Vdash Q$, read as "P entails Q", means that whenever Q is satisfied P is also satisfied. The theory of qualified types[Jones 1994] mandates the entailment relation to satisfy the three following properties:

(1) Reflexivity: $P \Vdash P$
(2) Transitivity: if $P \Vdash Q_1$ and $Q_1 \Vdash Q_2$ then $P \Vdash Q_2$

(3) Regularity: if $P \Vdash Q$ then $\Omega P \Vdash \Omega Q$

In the case of typeclasses, a predicate π is of the form $\mathtt{C}\overline{\tau}$, and it is satisfied when we can find a typeclass instance that matches π. The above three properties indeed hold for the System **TCFD**'s formulation of predicate satisfiability. The typing rules that we just discussed do not use any functional dependencies induced by the predicates. The use of the rules ($\Rightarrow I$) followed by ($\forall I$) in the typing derivation can introduce ambiguous types.

3.3 Ambiguous Types and Improving Substitution

The usual ambiguity check, for a qualified type, $\forall\overline{\alpha}.P \Rightarrow \tau$, is to ensure that all quantified type variables in the predicates P also appear in the type τ, or $(\overline{\alpha} \cap TV(P)) \subseteq TV(\tau)$. With induced functional dependencies, $\mathcal{F}_{\mathtt{P}}$, due to P, the appropriate check would be $(\overline{\alpha} \cap TV(P)) \subseteq TV(\tau)^{+}_{\mathcal{F}_{\mathtt{P}}}$. We weaken the check as there may be some free type variables that can be determined using the induced functional dependencies by the constraint set P. For example, the type of `expr` will no longer be flagged as ambiguous as the type variable `b` in `(Addition Int Float b, Addition b Int c) ⇒ c` is actually determined using the functional dependency `m n → p`.

Further, we may also want to show what the concise type is for a term is. This enhances usability of the system as the programmer would prefer the type of `expr` to be shown as `Float` rather than the above inferred type `(Add Int Float b, Add b Int c) ⇒ c`. We need to have an extra component in the type inference that lets us use the functional dependencies to determine the seemingly free type variables. 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 simplify the type by showing its true and concise characterization. In System **TCFD**, computing an improving substitution is to find if any type variables can be determined using the functional dependencies induced by the predicate set. For each functional dependency `X→Y`, induced by a class constraint `C t` in the predicate set P, whenever we know $TV(t_X)$, then we can determine $TV(t_Y)$. For example, in case of `expr`, where we inferred the most general type to be `(Addition Int Float b, Addition b Float c) ⇒ c`, the improvement substitution, $[b \mapsto \mathtt{Float}, c \mapsto \mathtt{Float}]$, obtained by using the functional dependency, $\mathcal{F}_{\mathtt{Add}} = \{\mathtt{m\ n} \rightarrow \mathtt{p}\}$, gives us the improved type `Float`.

3.4 Instance Validity and Inconsistency Detection

Every typeclass introduces a new relation on types. Functional dependencies give additional conditions that every instance should satisfy so that the instances declared are compatible with each other. There are two necessary conditions to ensure this:

(1) *Covering Condition*: For each new instance declaration **`instance`** $P \Rightarrow$ `C t` **`where`** ... that the programmer writes, we need to check that for each functional dependency of the form

$(X \rightarrow Y) \in \mathcal{F}_{\mathtt{P},\mathtt{C}}$, $TV(t_Y) \subseteq TV(t_X)^{+}_{\mathcal{F}_{\mathtt{P},\mathtt{C}}}$. $\mathcal{F}_{\mathtt{P},\mathtt{C}}$ are the functional dependencies induced by the class C and additional dependencies induced by the instance context P. Intuitively, this condition says that all the type variables in the dependent parameter, $TV(t_Y)$, of a functional dependency $(X \rightarrow Y) \in \mathcal{F}_{\mathtt{P},\mathtt{C}}$, 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 `C`, or those that are induced by the instance constraints 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 pair of instance of the form **`instance`** $Q \Rightarrow$ `C s` **`where`** ... and **`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 $U = mgu(t_X, s_X)$, and then check that $Ut_Y = Us_Y$. If we cannot find such a unifier, then we know that the instances are consistent. The most general unifier, $mgu(\tau_1, \tau_2,)$, can be computed using Robinson's algorithm[Robinson 1965].
For example, **`instance`** `C Int [Int]` is consistent with **`instance`** `C Char [Char]` as there is no unifier for `Int` and `Char`. However, **`instance`** `C a Int` together with **`instance`** `C a Float` fails the consistency condition.

4 ASSOCIATED TYPES

Another way to express type computation is by having the typeclass with an associated type[Chakravarty et al. 2005]. In this style, each instance of the typeclass specifies how the associated type should be represented. For example, `Addition m n p` typeclass can be written using an associated type as shown in Figure 6.

```
class Addition m n where
  type Result m n
  add :: (m×n) → Result m n

instance Addition Float Float where
  type Result Float Float = Float
  add(f, f) =...

instance Addition Int Float where -- Error
  add(i, f) = primAddI(i, (truncFtoI f))

instance Addition Int Float where
  type Result Int Float = Float
  add(i, f) =...

instance Addition Float Int where
  type Result Float Int = Float
  add(f, i) =...
```

Fig. 6. `Addition m n` typeclass with an Associated Type `Result m n`

The typeclass `Addition m n` has two type parameters in this style. The type of `add` changes slightly: it returns a value of type `Result m n`. The `Result m n` associated type can be thought of as a placeholder type that gets reduced to the representation type during the type analysis phase once the type parameters `m` and `n` are specified. The logic of reduction is given in each typeclass instance declaration using type equations. This effectively solves the problem of ambiguous types; the instantiation of `Result m n` reduces to a specific type. The type of the subterm `(add (1, 2.5))` in `expr` will now be inferred as `Float`, as `Result Int Float` reduces to `Float`. The associated type it can be thought as a type function in a way; depending on its use context, it reduces to an appropriate representation type.

```
class Tricky a where
  type Trick a

instance Tricky Bool where
  type Trick Bool = Int → Int

instance Tricky a where
  type Trick a = Bool
```

```
funTrick :: a → Trick a
funTrick _ = True

💣 :: Int
💣 = funTrick True 5
```

Fig. 7. `Tricky a` Typeclass and Type Inconsistency

Now suppose, a programmer writes a `Tricky a` typeclass shown in Figure 7. It has two instances where, `Trick Bool` reduces to a function type `Int → Int`, and `Trick a` reduces to `Bool`. Next, consider the function `funTrick`. The type `Trick a`, reduces to `Bool` matching the instance `Tricky a`. The use of `funTrick` in 💣, however, instantiates the `funTrick` to type `Trick Bool`, and `Trick Bool`, in turn, reduces to `Int → Int`. The type analysis finds no type errors. At runtime, the value returned by `funTrick` is `True`, and `True 5` crashes the program.

Haskell rejects such programs as it can cause type inconsistency as described above. The check that Haskell uses is that of overlapping instances; `Tricky a` and `Tricky Bool` overlap. The intention of the programmer was to be able to reduce `Tricky a` to `Int → Int` when `a` is `Bool`, and reduce it to `Bool` in all other cases. However, there is no way for the compiler to know this, and there is no language construct that supports such type reductions.

5 CLOSED TYPE FAMILIES

5.1 Examples

Closed type families is a generalization of associated typeclasses. We notice from the above `Tricky a` typeclass example that we could separate out the associated type `Trick a` into its own entity and collect each of its type equations defined in the instances in one place as shown below.

```
type family Tricky a where
  Trick Bool = Int → Int
  Trick a = Bool
```

One can think of a type family to be a family of types indexed its type parameters. In this case, the type family `Tricky a` has one type parameter, `a`, that it is indexed by. Each equation specifies what the type reduces to for specific type parameters; `Tricky Bool = Int → Int` means that `Tricky Bool` will reduce to `Int → Int`. This style of writing type functions is more palatable for functional programmers. It resembles a familiar style of writing term level equations with pattern matching.

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

class Addition m n where
  add :: (m×n) → Result m n

instance Addition Float Float where
  add(f1, f2) = ...

instance Addition Int Float where
  add(i, f) = ...

instance Addition Float Int where
  add(f,i) = ...

instance Addition Int Int where
  add(i1, i2) = ...
```

Fig. 8. Add Typeclass using `Result` Closed Type Family

The `Addition m n` typeclass, defined using an associated type in Figure 6, can also be written in a closed type family style as shown in Figure 8. The type equations written in the instances now move into a closed type family declaration `Result m n`. The advantage of closed type families over associated types, is the ability to express type functions that were previously not possible such as `Result a a = a` or in case of `Tricky a` type family, `Tricky a = Bool`. Fixing the order

of equation matching liberates us from the restriction of allowing only non-overlapping type equations; the first equation will match the type `Result Int Int` and reduce it to `Int`. As another example, previously shown Peano arithmetic using functional dependency style, is now shown using closed type family in Figure 9. The first equation encodes the axiom $0 + n = n$, while the second equation encodes the axiom, $(1 + n) + m = 1 + (n + m)$. As a cherry on top, the second equation computes the addition of two type level Peano numbers using recursion, which functional programmers adore.

```
type family Plus m n where
  Plus Z n = n                -- 0 + n = n
  Plus (S n) m = S (Plus n m) -- (1 + n) + m = 1 + (n + m)
```

Fig. 9. Peano Arithmetic using Closed Type Family

5.2 Type Matching, Apartness and Reduction

The role of type families is to compute, or reduce to, the representation the using instantiations type indices during the type analysis phase. For example, the representation type of `Result Int Float` is `Float` while, the representation type of `Plus Z n` is `n`. The semantics of such equations have subtleties. The reduction of an occurrence of a type family type is done using a top to bottom matching technique. The first equation that matches is used for reduction. For example, to reduce type `Addition Float Float`, first equation `Addition a a = a` matches it and reduces it to `Float`.

```
type family Tricky a where
  Tricky Bool = Int → Int
  Tricky a = Bool

funTrick :: a → Tricky a
funTrick _ = True

💣 :: Int
💣 = funTrick True 5
```

Fig. 10. Inconsistency due to Tricky Type Family

Now, let us reconsider the function `funTrick` and the closed type family `Tricky a`, shown in Figure 10. The type `Tricky a` does not match the first equation but matches the second equation and reduces to `Bool`. This reduction, however, will crash the program, 💣, which uses `funTrick` and instantiates `d` to `Bool`, and `Tricky Bool` in turn reduces to `Int → Int`. Type inconsistency bugs can be introduced if a naïve type reduction strategy is used with overlapping equations. We need to identify a reduction strategy that does not violate type consistency.

Fortunately, having a confluent type reduction relation is not only a necessary but also a sufficient condition to achieve type consistency[Bezem et al. 2003]. If a type has multiple ways of reducing, it should not matter what way we choose, the final type obtained should always the same. 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`, resulting in a runtime crash. We fix this behavior by incorporating flattening and apartness into the type reduction strategy. First we define the notion of matching:

Definition 1 (Matching, $\texttt{match}(\tau, \sigma)$)**.** We say a type τ matches σ, if and only if there is a substitution Ω such that, $dom(\Omega) \subseteq TV(\tau)$ and $\Omega\tau = \sigma$.

Definition 2 (Type Flattening, $\texttt{flatten}(\tau)$)**.** We say a type τ is flattened to τ_1, when every type family application of the form $\texttt{F}(\overline{\sigma})$ in τ is replaced by a type variable, such that in the flattened type, all syntactically equivalent type family applications are replaced by a same type variable, and syntactically different type family applications are replaced by distinct fresh type variables.

Definition 3 (Apartness, $\texttt{apart}(p, \tau)$)**.** For an equation, p, and a type, τ, p is apart from τ, if and only if we cannot unify the left hand side of the equation and the flattened τ.

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

Definition 4 (System μ**FC** Type Reduction Strategy)**.** An equation, p, given in the type family declaration of `F`, can be used to simplify the type $\texttt{F}(\overline{\tau})$, if the two following conditions hold:

(1) The left hand side of the equation matches the type $\texttt{F}(\overline{\tau})$.
(2) All equations that precede p, are apart from $\texttt{F}(\overline{\tau})$.

Let us reconsider the type `a → Tricky a`, with the above definition of type family reduction. Now, `Tricky Bool` does not match `Tricky a`, as there is no substitution, Ω, such that Ω`Bool = 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; doing so will violate the second criterion from Definition 4.

Type flattening with sharing optimizes the type reduction process in some cases. For example, consider the type: `Result (Result c d) (Result c d)`. We cannot reduce this type without flattening. However, with flattening, this type is transformed to `Result b b`, which matches the first equation, `Result a a = a`, and we can reduce the original type to `Result c d`. However, if we did not have sharing, we would flatten it to `Result a b`, and we would can no longer be able to use the first equation for reduction.

Reduction of type families by apartness check will ensure consistency but it has two shortcomings: 1) it is overly restrictive, and 2) it is inefficient due to the excessive calls to unification. We will illustrate them by using a couple examples: Consider second and third equations

from `Add` type family— `Addition a Float = Float` and `Addition Float a = Float`—, and the type `Addition Float b`, which we wish to reduce. In this case, reduction of `Addition Float b` to `Float` is not possible as the second equation is not apart from `Addition Float b`. But we know that both the equations reduce to the same final type, which is `Float`. We call such equations coincident equations. We would like to allow such reductions as they will not threaten consistency. Next, consider the equations of the `Plus m n` type family, `Plus Z m` and `Plus (S n) m`. It is easy to see that if any type matches `Plus (S m) m`, it will always be apart from `Plus Z m`, and we can thus safely skip on the apartness check. We can overcome on both of these shortcomings by using the notion of compatibility of equations defined below and incorporate it in our reduction strategy.

Definition 5 (Compatible Equations, $\texttt{compat}(p, q)$)**.** Two equations are compatible if and only if, whenever the left hand sides of the equations unify then so do the right hand sides of the equations. More formally, equations p and q are compatible if and only if whenever there exist substitutions Ω_p and Ω_q such that, if $\Omega_p(\text{lhs}_p) = \Omega_q(\text{lhs}_q)$ then $\Omega_p(\text{rhs}_p) = \Omega_p(\text{rhs}_q)$.

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

(1) The left hand side of the equation matches the the type $\texttt{F}(\overline{\tau})$
(2) All equations preceding p are either compatible with p, or they are apart from $\texttt{F}(\overline{\tau})$.

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

5.3 Formalizing Type Reduction and Type Consistency

In the previous section, we alluded that type consistency can be achieved using confluence. In this section we will formalize it using System μ**FC**. The portion of the system necessary to show type consistency is shown in Figure 11. We have special syntax category for type family constructors ($\texttt{F}, \texttt{G}$) along with the usual types. A type pattern ($\texttt{N}$) are tuple of types none of which contain type family constructors. The size of the pattern is same as that of the type family constructor arity. Predicates in this system are of the form $\tau_1 \sim \tau_2$ and assert that types τ_1 and τ_2 are equal. We also call them type equality predicates. The axioms, (ξ:Ψ), are named list of type family equations that are declared with the type family constructors. For example, the type family declaration for `Plus m n` type family will be represented as $\texttt{AxPlus} : [\forall\alpha.\ \texttt{Plus}\ Z\ \alpha \sim \alpha, \forall\alpha\beta.\ \texttt{Plus}\ (\texttt{S}\ \alpha)\ \beta \sim \texttt{S}(\texttt{Plus}\ \alpha\ \beta)]$ We refer to the collection of all axioms as the ground context (Σ).

We formally state type reduction as follows:

Type family Constructors	F, G	Types	τ, σ	$::= \alpha \mid \tau \to \tau \mid \mathsf{F}(\overline{\tau}) \mid \mathsf{T}$
Type Constants	T	Value Types	ω	$::= \tau \to \tau \mid \mathsf{T}$
		Type Equality Predicate	π	$::= \tau \sim \tau$
		Predicates	P	$::= \overline{\pi}$
		Type family Pattern	N	$::= \overline{\alpha \mid \omega}$
Type reduction	$\Sigma \vdash \bullet \rightsquigarrow \bullet$	Axiom Equations	Φ	$::= \forall \overline{\alpha}.\ \mathsf{F}(\mathsf{N}) \sim \sigma$
One Hole Type Context	$\mathbb{C}[\![\bullet]\!]$	Axiom Types	Ψ	$::= \overline{\Phi}$
		Ground Context	Σ	$::= \epsilon \mid \Sigma, \xi : \Psi$

Fig. 11. Excerpt of System for Closed Type Families (System μ**FC**)

Definition 7 (Type Reduction: $\Sigma \vdash \bullet \rightsquigarrow \bullet$, $\Sigma \vdash \bullet \rightsquigarrow^* \bullet$)**.** A type τ reduces to type σ, written $\Sigma \vdash \tau \rightsquigarrow \sigma$, if we can build a predicates π, of the form $\tau \sim \sigma$, using the axioms in Σ. We define $\Sigma \vdash \bullet \rightsquigarrow^* \bullet$ to be a reflexive and transitive closure of $\Sigma \vdash \bullet \rightsquigarrow \bullet$.

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

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

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

Fig. 12. Non Conflicting Equations, Compatibility and Type Reduction

We are essentially giving semantics to type reduction using type equality predicates. The compiler has two main ways to generate equality predicates. It can either find an appropriate axiom to instantiate from the ground context, or it can use one of the following axioms: reflexivity ($\epsilon \Vdash \tau_1 \sim \tau_1$), symmetry ($\tau_1 \sim \tau_2 \Vdash \tau_2 \sim \tau_1$), transitivity ($\tau_1 \sim \tau_2, \tau_2 \sim \tau_3 \Vdash \tau_1 \sim \tau_3$), and congruence ($(\tau_{11} \sim \tau_{12}, \tau_{21} \sim \tau_{22}) \Vdash (\tau_{11} \to \tau_{21}) \sim (\tau_{12} \to \tau_{22})$). These extra axioms are due to the fact that type equality forms an equivalence relation and a congruence relation. The appropriateness of axioms is exactly what Definition 6 describes. We formalize the definition in a rule format as shown in Figure 12. The rule ($\textsc{ty-}\beta$) packages everything together. This rule says that the i-th equation of axiom ξ, $\forall \overline{\alpha_i}.\ \mathsf{F}(\mathsf{N}_i) \sim \sigma_i$, is used to reduce the target type $\mathsf{F}(\overline{\tau})$ to type τ_0. The clause $\forall j < i.\ \mathtt{no\_conflict}(\Psi, i, \overline{\tau}, j)$ ensures that we do not perform unsound reductions. The $\mathbb{C}[\![\bullet]\!]$ says that the type reduction can occur anywhere within the type structure.

To get an idea of how these rules work out for the type reduction, let us consider the type `Plus (S Z) (Plus Z (S Z))`, along with the axioms in `AxPlus`. First, we see that `Plus Z (S Z)` $\rightsquigarrow$ `S Z`, as it matches the first equation, `AxPlus[1]`. Then, `Plus (S Z) (S Z)` reduces to `S (Plus Z (S Z))` as it second equation matches the type and the first equation is compatible by being disjoint using the rule (COMPAT-DIS). Finally, `S (Plus Z (S Z))` reduces to `S (S Z)`, giving us the expected result. The predicates that we built in this process were: `Plus Z (S Z)` $\sim$ `S Z`, `Plus (S Z) (S Z)` $\sim$ `S (Plus Z (S Z))`, `S (Plus Z (S Z))` $\sim$ `S (S Z)` and `Plus (S Z) (Plus Z (S Z))` $\sim$ `S (S Z)`.

The above exercise also illuminates an important property of System μ**FC**: if we have a type reduction from τ_1 to τ_2 then we can build a type predicate $\tau_1 \sim \tau_2$. We abuse the notation for entailment, $\Sigma \Vdash \tau_1 \sim \tau_2$, to mean we can obtain the type equality predicate $\tau_1 \sim \tau_2$ by using all necessary instantiations of the axioms in Σ.

LEMMA 8 (COMPLETENESS OF TYPE REDUCTION). *if* $\Sigma \vdash \tau_1 \rightsquigarrow^* \tau_2$ *then* $\Sigma \Vdash \tau_1 \sim \tau_2$

Consistency means we can never derive unsound equalites between value types, such as `Int` $\sim$ `Bool`, in the system. Value types (ω), in System μ**FC** are type constants, `T`, and function types $\tau_1 \rightarrow \tau_2$. Consistency of the type system hinges on the requirement that we have a consistent context. We say that a ground context Σ is consistent, when for all predicates $\omega_1 \sim \omega_2$, which we can build from Σ, or $\Sigma \Vdash \omega_1 \sim \omega_2$, the two following properties holds:

(1) if ω_1 is `T` then, ω_2 is also `T`
(2) if ω_1 is $\tau_1 \rightarrow \tau_2$ then, ω_2 is also $\tau_1 \rightarrow \tau_2$.

LEMMA 9 (TYPE CONSISTENCY). *If* Σ *is consistent then, System* μ***FC*** *is type consistent.*

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

Definition 10 (`Good` Σ)**.** A ground context (Σ) is good, written `Good` Σ when the following conditions are met for all $\xi{:}\Psi \in \Sigma$ and where Ψ is of the form $\overline{\forall \alpha_i.\ \mathsf{F}_i(\mathsf{N}_i) \sim \sigma}$:

(1) There exists an `F` such that $\forall i.\ \mathsf{F}_i = \mathsf{F}$, and no type pattern, N_i, mentions a type family constructor.
(2) The binding variables, $\overline{\alpha_i}$, occur at least once in the type pattern N_i, 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` Σ then, Σ is consistent. One way to prove this is via confluence of the type reduction relation. Type reduction relation given by the rule (TY-β) works only on type families and does not reduce any value types; making confluence a sufficient condition to ensure consistency. Now to prove confluence for type reduction, we first prove that the rewrite relation has local

confluence—whenever we have $\tau_1 \sim \tau_2$, then then we can always find a common reduct type τ_3 such that $\Sigma \vdash \tau_1 \rightsquigarrow^* \tau_3$ and $\Sigma \vdash \tau_2 \rightsquigarrow^* \tau_3$—and finally, we appeal to Newman's lemma[Newman 1942] given below that completes the proof of confluence of type reduction.

Lemma 11 (Newman). *If a rewrite system is terminating and locally confluent then it is confluent.*

The design of System μ**FC** does leave the door open for non-terminating type family definitions and we have to make do with a weaker consistency lemma than what we originally hoped for.

Lemma 12 (Σ Consistency). *If* $\Sigma \vdash \bullet \rightsquigarrow \bullet$ *is terminating, and* Good Σ *then* Σ *is consistent.*

6 CONSTRAINED TYPE FAMILIES

In the previous section for closed type families, we made an implicit assumption that all type families are total, in the sense their domain is all the types. This has philosophical and practical consequences. Consider the example as shown in Figure 13.

```
type family PTyFam a where
  PTyFam Int = Bool

type family Loop a where
  Loop a = [Loop a]

type family TEq a b where
  TEq a a = Int → Int
  TEq a b = Bool
```

```
g x = x : x -- Error

TEq [a] a  ⇝ ???
```

Fig. 13. Partial Closed Type Family

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. So, is `PTyFam Bool` a type? The implementation of System μ**FC** in GHC[The GHC Team 2020], a compiler for Haskell, it is treated as a type. This has practical consequences. Consider the type `TEq [a] a` as shown above. Our intuition is to reduce `TEq [a] a` to `Bool` as we can never have a list type, `[a]`, equal to its element type, `a`. Given our type reduction strategy described in Definition 6, the first equation does not match `TEq [a] a` and the second equation does. However, we cannot assert that `TEq [a] a` is apart from `TEq a b`. The reason being that there might exist a type family `Loop a` that reduces to `[Loop a]`. Thus, `TEq [Loop Bool] (Loop Bool)` can reduce to `TEq [Loop Bool] [Loop Bool]`, and eventually rewrite to `Int → Int`. Hence, GHC does not reduce `TEq [a] a` to `Bool`; it uses an apartness check

based on unification for infinite types[Jaffar 1984] rather than Robinson's unification. But GHC also not support infinite types; a declaration such as, `g x = x : x`, does not type check as `g` is of type `∀ a. ([a] ~ a) ⇒ a → a`; but the infinite type `Loop a` is accepted by GHC. We are thus left in a confusing situation where we accept some problematic types, like `Loop a`, but not all like `[a] ~ a ⇒ a → a`.

The nub of the issue is that there is a mismatch in our intuitive semantics of type families and their behavior. We think of them as partial functions on types where each new equation extends its definition. Instead we should be thinking of them as introducing a family of distinct types and each new equation equates types that were previously not equal. One possible solution to this conundrum is to reject the definition of infinite type families like `Loop a` but this burdens the programmers by enforcing them to provide an evidence to guarantee termination. Instead, we leverage the infrastructure that Haskell already has—qualified types—to solve this problem using constrained type families (System ***Q*FC**).

6.1 Type matching and Apartness Simplified

The type reduction in closed type families had a complex criterion for apartness check, which included flattening and then checking if they had a unifier. In constrained type families we neither have to depend on infinitary type unification nor flattening of types. Apartness in this system is just checking for failure of unification using Robinson's algorithm.

Definition 13 (System ***Q*FC** Type Reduction Strategy)**.** An equation, say p, given in the type family declaration can be used to simplify the type $\mathsf{F}(\overline{\tau})$, if the two following conditions hold:

(1) The left hand side of the equation matches the the type $\mathsf{F}(\overline{\tau})$.
(2) All equations preceding p are compatible with p or do not match $\mathsf{F}(\overline{\tau})$.

6.2 Formalizing Type reduction and Type Consistency

System ***Q*FC** is similar to System μ**FC** except for a few new constructs that we highlight in Figure 14. The type can be qualified with a predicate $(P \Rightarrow \tau)$ and the predicates are nothing but type equalites, of the form $\tau \sim \tau$ as before. Each type equality equation is of the form $\forall\overline{\alpha}\ \overline{\chi}.\ \mathsf{F}(\overline{\tau}) \sim \sigma$ where both $\overline{\tau}$ and σ do not have occurrence of any family type constructors. The equations are quantified not only by the type variables $\overline{\alpha}$ but also over evaluation assumptions $\overline{\chi}$. The 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) a b = G a b`, where `G` and `F'` are type family constructors, will be compiled to an equation: as $\forall\alpha\beta\ (b|c : \mathsf{G}\ \alpha\beta \sim b)(d|c' : \mathsf{F}'\ \alpha \sim d).\ \mathsf{F}d\ \alpha\ \beta \sim b$. We use χ to remind us that it is

more specific than π; for any χ, the left hand side of the type equality is always a type family and right and side is a fresh variable.

		Types	τ, σ	$::= \alpha \mid \tau \to \tau \mid \mathsf{F}(\overline{\tau}) \mid \mathsf{T} \mid P \Rightarrow \tau$
		Value Types	ω	$::= \tau \to \tau \mid \mathsf{T}$
		Type Equality Predicate	π	$::= \tau \sim \tau$
Type family Constructors	F, G	Predicates	P	$::= \overline{\pi}$
Type Constants	T	Axiom Equations	Φ	$::= \forall \overline{\alpha}\ \overline{\chi}.\ \mathsf{F}(\overline{\tau}) \sim \sigma$
		Axiom Types	Ψ	$::= \overline{\Phi}$
Type reduction	$\Sigma \vdash \bullet \rightsquigarrow \bullet$	Eval Assumption	χ	$::= (\alpha \mid c : \mathsf{F}(\overline{\tau}) \sim \alpha)$
One Hole Type Context	$\mathbb{C}[\![\ \bullet\ ]\!]$	Eval Resolution	q	$::= (\tau \mid \gamma)$
		Ground Context	Σ	$::= \epsilon \mid \Sigma, \xi{:}\Psi$

Fig. 14. Excerpt of System ***Q*FC**

The valid types in this system are only those that mention type family constructors in the predicate set P. For example, the type `∀m n. Addition 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.

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

Fig. 15. Type reduction

The type reduction relation is formalized using the rule (QTY-β). It 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 which is slightly different than before, as we have a weaker apartness check. 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. Type family free types, or value types, do not reduce; none of the equations in the ground context have left hand sides as a value type. We can thus prove termination for the type reduction relation.

Lemma 14 (System ***Q*FC** Type Reduction Terminates). *For all types* τ, $\Sigma \vdash \tau \rightsquigarrow^* \omega$

The definition of goodness can now be relaxed from the restriction that type families cannot appear in the left hand side of the type family equations. The compilation of equations into axioms will ensure that the type family occurrences will be abstracted out as evaluation assumptions χ.

Definition 15 (Good Σ relaxed)**.** A ground context (Σ) is good, written Good Σ, when the following conditions are met for all $\xi{:}\Psi \in \Sigma$ and Ψ is of the form $\overline{\forall \overline{\alpha}_i\ \overline{\chi}_i.\ \mathsf{F}_i(\overline{\tau}_i) \sim \sigma}$:

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

With the new reduction relation defined above, and knowing that it is always terminates, we can use Newman's lemma[Newman 1942] to prove that type reduction is confluent using local confluence just as we did in System μ**FC**. However, we get a stronger type consistency lemma as we do not have to assume termination.

Lemma 16 (Σ Consistency). *If* Good Σ*, then* Σ *is consistent*

Lemma 17 (System Q**FC** Type Consistency). *If* Σ *consistent, then System* Q**FC** *is type consistent.*

7 CONCLUSION AND FUTURE WORK

Type computation, either by functional dependencies or by type families is an attractive feature for programmers as it considerably improves language expressivity. From the above examples, we might think that functional dependencies are morally equivalent to type families. In such a case we would expect to have one 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 consistency 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 constrained 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. https://doi.org/10.1145/2535838.2535856

Joxan Jaffar. 1984. Efficient unification over infinite terms. *New Generation Computing* 2, 3 (1984), 207–219. https://doi.org/10.1007/BF03037057

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

Simon Marlow and Simon Peyton Jones. 2010. Haskell Language Report 2010. (2010). https://www.haskell.org/onlinereport/haskell2010/

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

The GHC Team. 2020. The Glasgow Haskell Compiler. (2020). https://downloads.haskell.org/ghc/8.8.4/

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