

# Dialects of Type Computations in Haskell

APOORV INGLE, University of Iowa, USA

In modern programming languages, it is a well perceived notion that static types have a two fold advantage (1) it serves as a guiding tool to help programmers write correct code and (2) the type checker can help identify code that does not have a correct behavior. An expressive type system can guarantee stronger claims about written programs. Enabling programmers to perform type level computations is a way to make the type system expressive. During the last three decades, there have been proponents of two different styles of type level computations—functional dependencies and type families—for languages that support typeclasses. In this paper we describe these two language features with examples, formalize them and finally compare them.

## 1 INTRODUCTION

Parametric polymorphism is a powerful technique 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  $\forall a. a \rightarrow a$ . We also need to have some way of taming unconstrained polymorphism. For example, an addition operation, `add`, on all types does not necessarily make sense, i.e. the type  $\forall a. a \rightarrow a \rightarrow a$  is too general for describing addition. Programming languages that support functions as first class citizens cannot get away with having unconstrained parametric polymorphism. Typeclasses[Wadler and Blott 1989] give a mechanism of having constrained polymorphic types. Theory of qualified types[Jones 1994] formalizes the system of typeclasses that justify the use of constraint polymorphism without compromising type safety by having predicates as a part of type syntax.

An alternative view of typeclasses is to define relations on types. As the type checker is guaranteed to terminate, it gives programmers a way to encode computations at type level. A typeclass with  $n$  parameters builds an  $n$ -tuple relation on types. However, using relations to encode computations is cumbersome due to its verbosity. A new language feature, type families[Schrijvers et al. 2007], was introduced in Haskell to enable expressing type functions. Type functions are stylistically more obvious for programmers. Naturally, enabling type families warrants a richer system of types which also means that ensuring type safety for such a language is nontrivial.

The scope of the current article is as follows. We give examples with intuitive set semantics for typeclasses in the beginning of Section 2, and then describe functional dependencies[Jones 2000] with some examples in Section 2.1. We formalize them in Section 2.2 and describe their consequences in Section 2.4. We also give a brief description of type safety for this system in Section 2.5. We then describe two flavors of type families—closed type families[Eisenberg et al. 2014] in Section 3 and constraint type families[Morris and Eisenberg 2017] in Section 4 with examples and their respective formalization in Section 3.2 and Section 4.3. We then give some details about the type safety of each system in Section 3.3 and Section 4.4 respectively. To conclude we give some related work in Section 5, and 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.

---

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

## 2 TYPECLASSES

Typeclasses can be thought of as collection of types. Each typeclass is accompanied by its member functions or operators that a particular instance of the typeclass supports. For example, equality can be expressed as a typeclass `Eq a` as follows:

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

The instances of `Eq` typeclass can be types such as integers (`Int`) and characters (`Char`) but it may not be meaningful to define equality on function types ( $a \rightarrow b$ ). Thus, the operator `(==)` is not truly polymorphic in this sense; it cannot operate on function types. Rather it is constrained to only those types that have an instance defined. We make this explicit by writing the type of the operator `(==) :: ∀a. Eq a ⇒ a → a → Bool`.

We can even have a hierarchy of typeclasses. An `Ord a` typeclass can be used to describe types that have an ordering relation on their values. A prerequisite of defining an ordering relation is that they should have a notion of equality. This is expressed as a predicate on the left hand side of the class declaration of `Ord a` declaration shown below. The typechecker would have to ensure every instance of `Ord a` has the necessary `Eq a` instance defined.

```
class Eq a ⇒ Ord a where
  (≤) :: a → a → Bool
instance Ord Int where
  a ≤ b = primLTInt a b
instance Eq Char where
  a ≤ b = primLTChar a b
```

There is nothing special about typeclasses having just one type parameter. It would be natural to have multiple type parameters while defining a typeclass. A multiparameter typeclass with  $n$  type parameters would represent a relation over  $n$  types. An example of such a typeclass would be `Add a b c` that represents all the types that support an add (+) operation. We would expect to have instances such as `Add Int Int Int`, `Add Int Float Float`, and so on. The instances of typeclass `TEq` would assert that types `a` and `b` that are equal. We can describe `Add` and `TEq` in simple

```
class Add m n p where
  (+) :: m → n → p
instance Add Int Float Float where
  (+) a b = addFloat (toFloat a) b
instance Add Int Int Int
  (+) a b = intAdd a b
class TEq a b
instance TEq Int Int
instance TEq Char Char
instance (TEq a1 b1, TEq a2 b2)
  ⇒ TEq (a1, a2) (b1, b2)
```

Fig. 1. Multiparameter Typeclasses

set semantics as,  $\text{Add} = \{(Int, Int, Int), (Int, Float, Float)\}$  and  $\text{TEq} = \{(Int, Int), (Char, Char)\} \cup \{((\tau_1, \tau_2), (\tau'_1, \tau'_2)) \mid (\tau_1, \tau'_1) \in \text{TEq} \wedge (\tau_2, \tau'_2) \in \text{TEq}\}$ .

Multiparameter typeclasses, while being powerful are problematic in practice. It is possible for a programmer to declare overlapping instances of such a typeclass. Declaring `Add Int Float Float` and `Add Int Float Int` would be conflicting as the type inference algorithm will no longer be able to resolve the overloaded operator `+`. The situation gets worse as the type error would be raised at the use of the overloaded operator confusing the programmer about the cause of the issue. However, there may be cases where we do would want to allow such overlapping definitions as well. Consider a class `Convert a b` that converts a type `a` to a type `b`. All four instances of this class—`Convert Int Char`, `Convert Char Int`, `Convert Int Int` and `Convert Char Char`—are valid. The issue is that the type system has no mechanism to identify if overlapping instances are inconsistent so that it can disallow them at compile time.

## 2.1 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 relation between the type parameters in the typeclass declaration. There is no change in the syntax of declaring instances for the typeclasses. As shown in Figure 2, `Convert a b` typeclass is just a binary relation on types while the new `Add m n p` typeclass relates the type parameters such that `m` and `n` determine `p`. We write  $x \rightarrow y$  to mean “ $x$  uniquely determines  $y$ ”. We can also have multiple functional dependencies as with `TEq` typeclass where `t` and `u` determine each other. In general we can have multiple parameters on both sides of the arrow,  $x_1, \dots, x_m \rightarrow y_1, \dots, y_m$ .

```

class Convert a b where
  to :: a → b

instance Add Int Float Float where
  ...

class TEq t u | t → u, u → t
instance TEq t t

class Add m n p | m n → p where
  (+) :: m → n → p
instance Add Int Int Int where
  ...

instance Add Int Float Float where
  ...

instance Add Float Int Float where
  ...

instance Add Int Float Int -- Error!

e :: (Add Int Float b, Add b Int c) ⇒ c
e = (1 + 2.0) + 3

```

Fig. 2. Add with Functional Dependency

The programmer can use functional dependencies to specify the intention of the typeclasses more accurately. Further, the compiler now has a way to detect inconsistencies with the declared instances and flag an error whenever it detects one. This improves error reporting as it flags errors at inconsistent declaration site rather than at the previously confusing operator use site. For example, the functional dependency on  $m \rightarrow p$  can now help the compiler flag the instance `Add Int Float Int` as a conflicting instance `Add Int Float Float` because `Int` and `Float` together should uniquely determine a type.

Consider the expression `e = (1 + 2.0) + 3`. Due to the use of the overloaded operator `(+)` in `e`, the principal or the most general type of `e` inferred by the type inference algorithm would be `(Add Int Float b, Add b Int c) ⇒ c`. But notice how `b` does not occur on the right and side of the  $\Rightarrow$ . This makes the type ambiguous. There is no way for the compiler to instantiate the type variable `b` to a specific value and be able to choose a particular instance of the operator. In general, ambiguous types do not have a well defined semantics in Haskell overloading. However, due to the use of functional dependency, this seemingly ambiguous type can be converted to an unambiguous type. We can infer that `b` has to be `Float` as it is determined by the other two type parameters 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, using the functional dependency to deduce that type type variable `c` has to be `Float`. Without the functional dependency on the class, it would not be impossible to make such an inference.

With functional dependencies at our disposal, We can even perform peano arithmetic at type level as shown in Figure 3. First we define two datatypes `Z` and `S n` that represent the number zero and successor of a number `n`, respectively. The

```

157      data Z    -- Type level Zero
158      data S n  -- Type level Successor
159
160      class (IsPeano m, IsPeano n, IsPeano p)
161        ⇒ Plus m n p | m n → p
162
163      instance IsPeano Z ⇒ Plus Z m m
164      instance Plus n m p ⇒ Plus (S n) m (S p)
165
166      class IsPeano c
167      instance IsPeano Z
168      instance IsPeano n ⇒ IsPeano (S n)
169
170      data Vector s e = Vec (List e)
171      concat_vec :: Plus m n p
172                  ⇒ Vector m e → Vector n e → Vector p e
173      concat_vec (Vec l1) (Vec l2) = Vec (append l1 l2)

```

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

instances of `IsPeano` assert that `Z` is a peano number also, if `n` is a peano number then `S n` is a Peano number. The instances of `Plus` typeclass relates three peano numbers where 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  $0 + m = m$ , while `Plus n m p ⇒ Plus (S n) m (S p)` asserts that if  $n + m = p$  then  $(1 + n) + m = (1 + p)$ . The `TEq` instance holds when there is an appropriate `Plus` instance defined. Notice how the functional dependency  $t \rightarrow u$ ,  $u \rightarrow t$  describes the fact that all instance of `TEq` should have the same type for `t` and `u`. Without this functional dependency it would impossible to enforce the criteria for a type class whose instances were intended for equal types. 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.

## 2.2 Formalizing Type classes with Functional Dependencies

To formalize the system, we first need to fix the language. The surface syntax, the one that the user writes, is shown in Figure 4. We will call this System **TCFD**. The syntax of types ( $\tau$ ) consists of type variables ( $\alpha$ ), functions ( $\tau \rightarrow \sigma$ ), and type constructors  $T\bar{\alpha}$ . The qualified types ( $\rho$ ) are given as  $C\bar{\alpha} \Rightarrow \tau$  where  $C\bar{\alpha}$  constrains the type  $\tau$ . Type schemes ( $\sigma$ ) 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.

		Predicates	$P ::= \overline{C\bar{\alpha}}$
	Term Variables	Types	$\tau ::= \alpha \mid \tau \rightarrow \tau \mid T\bar{\alpha}$
	Type Variables	Qualified Types	$\rho ::= \tau \mid P \Rightarrow \tau$
	Class Constructors	Type Schemes	$\sigma ::= \forall \bar{\alpha}. \rho$
	Data Constructors	Terms	$e ::= x \mid e e \mid \lambda x:\tau. e \mid \text{let } x = e_1 \text{ in } e_2$
	Type Constructors		$\mid D\bar{e} \mid \text{case } e \text{ of } \bar{M}$
		Values	$\mathcal{V} ::= \lambda x:\tau. e \mid D\bar{e}$
	Term Typing	Patterns	$M ::= D\bar{e} \rightarrow e$
		Typing Environment	$\Gamma ::= \epsilon \mid \Gamma, x:\sigma$

Fig. 4. System **TCFD**

**2.2.1 Notations and Definitions.** To avoid syntax clutter for formalization we will use some notations described as follows. We would use the subscripts on objects ( $\alpha_1, \dots, \alpha_n$ ) to mean they are distinct.  $\bar{\alpha}$  means a collection of  $\alpha_1, \alpha_2, \dots, \alpha_n$

items of arbitrary length. We use  $S_1 \setminus S_2$  to denote the set difference operation. For an object  $X$ ,  $TV(X)$  is the set of variables that are free in  $X$ . We write  $M[\bar{x}/\bar{y}]$  to denote the substitution where each variable  $x_i$  is mapped to  $y_i$  in  $M$ . Alternatively we also write  $\Omega X$  for an substitution  $\Omega$  applied to object  $X$ . We denote the most general unifier for two types  $\tau_1$  and  $\tau_2$  (if it exists), by  $mgu(\tau_1, \tau_2)$  [Robinson 1965]. For the sake of convinience, we would also write  $mgu(\bar{\tau}_1, \bar{\tau}_2)$  to give us a composition of most general unifier for each pairs of types  $(\tau_{1i}, \tau_{2i})$ . For a typeclass declaration we write **class**  $P \Rightarrow C \bar{t}$ , where  $\bar{t}$  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  $X \rightarrow Y$  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,  $\Omega$ , such that  $\emptyset \models \Omega P$ . We write  $\lfloor P \rfloor = \{\Omega \mid \emptyset \models \Omega P\}$  to mean the set of all substitutions that satisfy  $P$ .

For example, for a typeclass declaration **class**  $Add \ m \ n \ p \mid m \ n \rightarrow p$ , we have,  $t = (a, b, c)$ ,  $FD_{Add} = \{ m \ n \rightarrow p \}$ ,  $TV(C) = \{m, n, p\}$ . For the functional dependency  $m \ n \rightarrow p$ , we have,  $t_X = (m, n)$  and  $t_Y = (p)$ . 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,  $\{p\}_{FD_{Add}}^+ = \{p\}$ ,  $\{m\}_{FD_{Add}}^+ = \{m\}$  while  $\{m, n\}_{FD_{Add}}^+ = \{m, n, p\}$ .

**2.2.2 Type system for System TCFD.** 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  $\tau$  with predicates  $P$  being satisfied and the free variables in  $e$  are given types by the typing environment  $\Gamma$ , that maps term variables to its types. The typing judgments for the language are shown in Figure 5. The generalize function  $Gen(\Gamma, \tau)$  quantifies all the free variables of the type  $\tau$  that do not occur in the domain of the type environment  $\Gamma$ , i.e.,  $Gen(\Gamma, \tau) = \forall (TV(\tau) \setminus dom(\Gamma)). \tau$ . The instantiate function  $Inst(\sigma)$  maps each quantified variable in  $\sigma$  to a fresh variable, i.e. if  $\sigma = \forall \bar{\alpha}. P \Rightarrow \tau$ , then  $Inst(\sigma) = P[\bar{\alpha}/\bar{\beta}] \Rightarrow \tau[\bar{\alpha}/\bar{\beta}]$  where  $\bar{\beta}$  are fresh. The variables need to be fresh to avoid any conflict with the existing type variables. The typing rule for function abstraction ( $\rightarrow I$ ) says that if we can show a typing derivation where the body of the function  $e$  has the type  $\tau$  with the typing environment extended with the variable that represents the argument for the function with type  $\tau_1$ , then we have a judgment that shows the lambda term has type  $\tau_1 \rightarrow \tau$ . The typing judgment for function application rule ( $\rightarrow E$ ) says that if we can show that the left hand term  $e_1$  has type  $\tau_2 \rightarrow \tau$  and additionally we can show that the right hand term  $e_2$  has type  $\tau_2$  then we can show that the term  $e_1 \ e_2$  has type  $\tau$ . 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  $\tau_2$  only if we can build a derivation that shows  $e_1$  as  $\tau_1$  and we can show that  $e_2$  has type  $\tau_2$  with typing environment extended with the variable  $x$  mapped to the generic instance of  $\tau_1$ . 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  $\tau$  we need to look it up in our typing environment and instantiate it with fresh variables. The rule (TDATA) says that if need to show that the datatype constructor  $D\bar{e}_i$  has a user defined type  $T\bar{\tau}$  then we need to show that each of the arguments are well typed and the data constructor has the right type as given by its declaration. The data constructor case can be viewed as a generalized version of function types as  $\tau_1 \rightarrow \tau_2$  is a type constructor and  $\lambda x:\tau. e$  as its data constructor.

**TODO: : Talk about principal type scheme?**

$$\begin{array}{c}
\text{(T-ABS)} \frac{P \mid \Gamma, x:\tau_1 \vdash e : \tau}{P \mid \Gamma \vdash \lambda x:\tau_1. e : \tau_1 \rightarrow \tau} \quad \text{(T-APP)} \frac{P \mid \Gamma \vdash e_1 : \tau_2 \rightarrow \tau \quad P \mid \Gamma \vdash e_2 : \tau_2}{P \mid \Gamma \vdash e_1 e_2 : \tau} \\
\text{(T-VAR)} \frac{x:\sigma \in \Gamma \quad P \Rightarrow \tau = \text{Inst}(\sigma)}{P \mid \Gamma \vdash x : \tau} \quad \text{(T-LET)} \frac{P \mid \Gamma \vdash e_1 : \tau_1 \quad \sigma = \text{Gen}(\Gamma, P \Rightarrow \tau_1) \quad P_1 \mid \Gamma, x:\sigma \vdash e_2 : \tau_2}{P_1 \mid \Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2} \\
\text{(DI)} \frac{\overline{P \mid \Gamma \vdash e_i : \tau_i} \quad D:\overline{\tau_i} \rightarrow T\overline{\tau} \in \Gamma}{P \mid \Gamma \vdash D\overline{e_i} : T\overline{\tau}} \quad \text{(DE)} \frac{P \mid \Gamma \vdash e : T\overline{\tau'} \quad \overline{P \mid \Gamma \vdash M_i : \tau}}{P \mid \Gamma \vdash \text{case } e \text{ of } \overline{M} : \tau} \\
\text{(T-MATCH)} \frac{P \mid \Gamma \vdash D\overline{x_i} : T\overline{\tau_i} \quad P \mid \Gamma, \overline{x_i}:\overline{\tau_i} \vdash e : \tau}{P \mid \Gamma \vdash D\overline{x_i} \rightarrow e : \tau}
\end{array}$$

Fig. 5. Typing judgments for System TCFD Terms

### 2.3 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**  $P \Rightarrow C \ t \text{ where } \dots$  that the user writes, we need to check that, for each functional dependency  $f = (X \rightarrow Y) \in \text{FD}_C$ ,  $TV(t_Y) \subseteq TV(t_X)_f^+$ . Consider the class declarations **class** C1 a b | a  $\rightarrow$  b and **class** C2 a b | b  $\rightarrow$  a and an instance declaration, **instance** C1 Int x  $\Rightarrow$  C2 x Int. This instance declaration violates the above condition, as x cannot be determined by the dependency b  $\rightarrow$  a. It turns out, that this condition is weak [Jones and Diatchki 2008]. An instance might have additional dependencies induced by the predicates  $P$  that we need to take into account. A more appropriate check would be  $TV(t_Y) \subseteq TV(t_X)_{\text{FD}_{P,C}}^+$ , where  $\text{FD}_{P,C}$  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 ( $\text{FD}_C$ ) or induced by the constraints ( $\text{FD}_P$ ).
- (2) *Consistency Condition*: For each new instance of the form **instance**  $Q \Rightarrow C \ s \text{ where } \dots$  along with **instance**  $P \Rightarrow C \ t \text{ where } \dots$  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 with an identity unifier.

### 2.4 Computing and Using Improving Substitution

An improving substitution, written as  $\text{impr}(P)$ , is a substitution that does not change the set of satisfiable instances of predicate set  $P$ . Thus, if  $\Omega = \text{impr}(P)$  then  $\lfloor P \rfloor = \lfloor \Omega P \rfloor$ . Now, computing an improving substitution is straightforward with the induced functional dependencies,  $\text{FD}_P$ , on the predicate set,  $P$ . For each  $(X \rightarrow Y) \in \text{FD}_P$  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. For example, for a given type `Add Int Float p ⇒ p` we know that there is a unique choice for `p` due to the functional dependency, thus we have an improving substitution  $\Omega = [p/Float]$  which fixes the seemly polymorphic type variable `p` without changing its indented meaning. Improving substitutions also affects the way ambiguous types are detected. For a qualified type,  $\forall \bar{\alpha}. P \Rightarrow \tau$ , the usual ambiguity check is  $(\bar{\alpha} \cap TV(P)) \subseteq TV(\tau)$ . However, with induced functional dependencies  $F_P$  due to  $P$ , the appropriate check would be  $(\bar{\alpha} \cap TV(P)) \subseteq TV(\tau)_{F_P}^+$ . We thus weaken the check to ensure that there might be some type variables  $\alpha_i$  that are determined by the functional dependencies due to the class constraints. This improving substitution can also be applied anywhere during the type inferencing algorithm without adversely affecting it.

## 2.5 Type safety of 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.

$$\begin{array}{c} \text{(S-APP)} \frac{e_1 \rightsquigarrow e'_1}{e_1 e_2 \rightsquigarrow e'_1 e_2} \quad \text{(S-}\beta\text{)} \frac{}{(\lambda x:\tau. e_1) e_2 \rightsquigarrow e_1[x/e_2]} \end{array}$$

Fig. 6. Small Step Operational Semantics for System TCFD

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  $\epsilon \mid \epsilon \vdash e_2 : \tau$*

For preservation, we need to also prove a helper lemma, that says substitution is sound.

LEMMA 3 (TERM SUBSTITUTION SYSTEM TCFD). *If  $P \mid \Gamma, x:\tau_2 \vdash e_1 : \tau_1$  and  $P \mid \Gamma \vdash e_2 : \tau_2$  then  $P \mid \Gamma \vdash e_1[x/e_2] : \tau_1$*

We define  $\bullet \rightsquigarrow^* \bullet$  as a transitive closure of the  $\bullet \rightsquigarrow \bullet$  relation. The (syntactic) type safety of the system can formally be given as

LEMMA 4 (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?

### 3 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  $\mathbb{Z}$  and  $\mathbb{S}$  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 TEq n m where
  TEq a a = TT
  TEq a b = FF

type family Plus n m where
  Plus  $\mathbb{Z}$  m = m
  Plus ( $\mathbb{S}$  n) m =  $\mathbb{S}$  (Plus n m)

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.

Before we go further, we first need to fix the taxonomy of the building blocks of type family declaration. Taking the example of `Plus m n`, the header—`type family Plus m n`—says that we are declaring a new type family constructor named `Plus` of arity 2. The equations are listed after the `where` keyword. Each equation's left hand side mentions the name of the type family and a type pattern, and the right hand side mentions the result type.

```

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  $\mu\text{FC}$ ), 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. Type equality coercions are special types, that are proofs 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.

The closedness of type families comes 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 key take away is that closed type families make it easier to define functions on types that would otherwise need some complex encoding, or a type checker which supports backtracking.

<pre> <b>data</b> Leaf a <b>data</b> Node a b  <b>type family</b> TMember e tree <b>where</b>   TMember e (Leaf e') = TEq e e'   TMember e (Node lt rt) = Or (TMember e lt) (TMember e rt) </pre>	<pre> <b>type family</b> Or a b <b>where</b>   Or FF a = a   Or TT a = TT   Or a TT = TT  <b>type family</b> CountArgs ty <b>where</b>   CountArgs (a → b) = S (CountArgs b)   CountArgs b = Z </pre>
---	---

Fig. 9. Complex Typelevel Functions

### 3.1 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  $\tau$  reduces to or evaluates to another type  $\tau_1$  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. For example, the target type  $\text{Int} \rightarrow \text{Float} \rightarrow \text{Result Int Float}$  reduces to or evaluates to  $\text{Int} \rightarrow \text{Float} \rightarrow \text{Float}$  with the rule  $\text{Result } a \text{ Float} = \text{Float}$  being fired. 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 overlapping left hand sides of the equations. The type family definition of  $\text{TEq } a \text{ } b$  indeed has overlapping equations with  $\text{Eq } a \text{ } a$  and  $\text{Eq } a \text{ } b$ . The type  $\text{TEq } a \text{ } a$  is non-linear as it mentions the type variable  $a$  twice. Overlapping equations and non linear patterns 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  $\text{BadTyFam } (\text{TEq Int Bool})$ . The last equation is fired in this case and we reduce the target to  $\text{FF}$ . However, if we instead evaluate  $\text{TEq Int Bool}$  first we get  $\text{TT}$  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 cause a crash at runtime. The crux of the problem is that, if we (erroneously) reduce  $\text{TyFam } (\text{TEq Bool } d)$  we get  $()$ , as  $\text{TEq Bool } d$  matches the second equation from  $\text{TEq } a \ b$ , and it reduces to  $\text{FF}$ . However, in the call to boom, we see that  $d$  is instantiated with  $\text{Bool}$  thus  $\text{TyFam } (\text{TEq Bool } d)$  it should reduce to  $\text{Int} \rightarrow \text{Int}$ . This is indeed, unsound. A naïve notion of matching where we

```

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

check for failure for a most general unifier of two 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 5** (Type Flattening). We say a type  $\tau$  is flattened to  $\tau_1$ , or  $\tau_1 = \text{flatten}(\tau)$ , when every type family application of the form  $F(\bar{\sigma})$  is replaced by a type variable, such that in the flattened type, every syntactically equivalent type family application in  $\tau$  is replaced by same type fresh type variable and syntactically different type family applications in  $\tau$  are replaced by distinct fresh type variables. For example, if we have a type  $\tau = F \ b \ c \rightarrow F \ a \ b \rightarrow F \ a \ b \rightarrow F \ c \ b$  then  $\text{flatten}(\tau) = \alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_2 \rightarrow \beta$  with  $[\alpha_1/F \ b \ c, \alpha_2/F \ a \ b, \beta/F \ c \ b]$ .

Now apartness of types and dually matching can be defined as:

**Definition 6** (Apart and Matching). We say two types,  $\tau_1$  and  $\tau_2$  are apart if  $\text{mgu}(\tau_1, \text{flatten}(\tau_2))$  fails and  $\tau_1$  does not mention type family constructors and dually,  $\text{match}(\tau_1, \tau_2) = \neg \text{apart}(\tau_1, \tau_2)$

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  $\text{Or } a \ b$  type family declaration example in Figure 9 are compatible. We formalize the notion of compatibility of equations below:

**Definition 7** (Compatible Equations). The two equations,  $p$  and  $q$ , are compatible if there exist two substitutions,  $\Omega_1$  and  $\Omega_2$  such that if  $\Omega_1(\text{lhs}_p) = \Omega_2(\text{lhs}_q)$  then  $\Omega_1(\text{rhs}_p) = \Omega_2(\text{rhs}_q)$ .

Implementing compatibility is simple. For two equations  $p$  and  $q$ , we find  $\Omega = \text{mgu}(\text{lhs}_p, \text{lhs}_q)$ . 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 8** (Closed Type Family Simplification). An equation, say  $p$ , given in the type family declaration can be used to simplify the target  $F(\bar{\tau})$  to a type if the following two conditions hold:

- (1) The type pattern on left hand side of the equation,  $N_p$  matches with the target  $\bar{\tau}$  or  $\text{match}(N_p, \bar{\tau})$
- (2) Any equation  $q$ , that precedes  $p$ , is either compatible with  $p$ ,  $\text{compat}(p, q)$ , or it's pattern  $N_q$  is apart,  $\text{apart}(N_q, \bar{\tau})$ .

**TODO:** : talk about infinitary unification

### 3.2 Formalizing Closed Type Families

		Types	$\tau, \sigma ::= \alpha \mid \tau \rightarrow \tau \mid \forall \alpha. \tau \mid F(\bar{\tau}) \mid T(\bar{\tau})$
		Ground Types	$\omega ::= \tau \rightarrow \tau \mid \forall \alpha. \tau \mid T\bar{\tau}$
		Predicates	$P ::= \tau \sim \tau$
		Type family Pattern	$N ::= \bar{\tau}$
		Axiom Equations	$\Phi ::= \forall \bar{\alpha}. F(N) \sim \sigma$
		Axiom Types	$\Psi ::= \bar{\Phi}$
		Coercions	$\gamma, \eta ::= \gamma \rightarrow \eta \mid \forall \alpha. \gamma \mid \gamma @ \tau \mid F(\bar{\gamma}) \mid T(\bar{\gamma})$ $\mid \text{nth}_i \gamma \mid \langle \tau \rangle \mid \hat{\gamma} \mid \gamma \circ \eta \mid \xi_i \bar{\tau}$
		Terms	$e ::= x \mid \lambda x:\tau. e \mid e e \mid M \blacktriangleright \gamma \mid \Lambda \alpha. e \mid e \tau$ $\mid D\bar{e} \mid \text{case } e \text{ of } \bar{M}$
		Values	$\mathcal{V} ::= \lambda x:\tau. e \mid D\bar{e} \mid \Lambda \alpha. e$
Type family Constructors	$F$	Ground Context	$\Sigma ::= \epsilon \mid \Sigma, \xi : \Psi \mid \Sigma, F : n \mid \Sigma, T : n$
Type Constructors	$T$	Variable Context	$\Delta ::= \epsilon \mid \Delta, x:\tau \mid \Delta, \alpha$
Type Validity	$\Gamma \vdash \tau \text{ type}$	Typing Context	$\Gamma ::= \Sigma; \Delta$
Proposition Validity	$\Gamma \vdash P \text{ prop}$		
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	$C[\bullet]$		

Fig. 11. System for 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  $\mu\text{FC}$  has coercion types  $(\gamma, \eta)$ , 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,  $\text{TEq}$  type family looks like  $\text{axiomTEq} : [\forall \alpha. \text{TEq } \alpha \alpha \sim \text{TT}, \forall \alpha \beta. \text{TEq } \alpha \beta \sim \text{FF}]$ .

The typing judgments for terms in this system is a quadruple  $\Gamma \vdash e : \tau$  that asserts that there is a derivation such that term  $e$  has type  $\tau$  under the typing context  $\Gamma$ . All the interesting rules are given in Figure 13. The typing environment  $(\Gamma)$  consists of two contexts, the variable context  $(\Delta)$  which maps free variables to their types, and ground context  $(\Sigma)$  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 are essentially walking over the context structures and ensuring we do not add anything invalid.

The term typing rules are standard and we skip them due to space constraints, except the three interesting ones. As System  $\mu\text{FC}$  is extension of System  $\text{FC}$ , we have two rules at term level, one for type abstraction ( $T\text{-TYABS}$ ) and another for type application ( $T\text{-TYAPP}$ ). These rules are similar to previously described ( $T\text{-ABS}$ ) and ( $T\text{-APP}$ ) respectively but describe type and term interaction. The rule ( $T\text{-CAST}$ ) says that a well typed term of type  $\tau_1$  can be typed using a new type  $\tau_2$  if there is a welltyped witness  $\gamma$  that casts it from type  $\tau_1$  to  $\tau_2$ . The three rules ( $\text{CO-REFL}$ ), ( $\text{CO-SYM}$ ), and

$$\begin{array}{c}
\text{(v-TVAR)} \frac{\alpha \in \Delta \quad \vdash_{ctx} \Sigma; \Delta}{\Sigma; \Delta \vdash \alpha \text{ type}} \quad \text{(v-ARR)} \frac{\Gamma \vdash \tau \text{ type}_1 \quad \Gamma \vdash \tau \text{ type}_2}{\Gamma \vdash (\tau_1 \rightarrow \tau_2) \text{ type}} \quad \text{(v-TFA)} \frac{\Sigma; \Delta, \alpha \vdash \tau \text{ type}}{\Sigma; \Delta \vdash (\forall \alpha. \tau) \text{ type}} \\
\text{(v-TCTR)} \frac{\Gamma \vdash n \in \Sigma \quad \vdash_{ctx} \Sigma; \Delta \quad \overline{\Sigma; \Delta \vdash \tau_i \text{ type}}^{i < n}}{\Sigma; \Delta \vdash \tau \text{ type}} \quad \text{(v-TFCTR)} \frac{F \vdash n \in \Sigma \quad \vdash_{ctx} \Sigma; \Delta \quad \overline{\Sigma; \Delta \vdash \tau_i \text{ type}}^{i < n}}{\Sigma; \Delta \vdash F \tau \text{ type}} \\
\text{(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}} \quad \text{(v-TFF)} \frac{F \vdash n \in \Sigma \quad \overline{\Sigma; \Delta, \bar{\alpha} \vdash N_i \text{ type}}^{i < n} \quad \Sigma; \Gamma, \bar{\alpha} \vdash \sigma \text{ type}}{\Sigma; \Delta \vdash \forall \bar{\alpha}. F(N) \sim \sigma \text{ prop}} \\
\text{(v-GEMPT)} \frac{}{\vdash_{gnd} \epsilon} \quad \text{(v-GTC)} \frac{\Gamma \vdash \Sigma \quad \vdash_{gnd} \Sigma}{\vdash_{gnd} \Sigma, \Gamma \vdash n} \quad \text{(v-GTF)} \frac{F \vdash \Sigma \quad \vdash_{gnd} \Sigma}{\vdash_{gnd} \Sigma, F \vdash n} \quad \text{(v-GAX)} \frac{\vdash_{gnd} \Sigma \quad \overline{\Sigma; \epsilon \vdash \forall \bar{\alpha}. F(N) \sim \sigma \text{ prop}}_{\xi \# \Sigma}}{\vdash_{gnd} \Sigma, \xi \vdash \forall \bar{\alpha}. F(N) \sim \sigma} \\
\text{(v-VEPMT)} \frac{\vdash_{gnd} \Sigma}{\Sigma \vdash_{var} \epsilon} \quad \text{(v-TE)} \frac{\Sigma \vdash_{var} \Delta}{\vdash_{ctx} \Sigma; \Delta} \quad \text{(v-VAR)} \frac{\Sigma \vdash_{var} \Delta \quad x \# \Delta \quad \Sigma; \Delta \vdash \tau \text{ type}}{\Sigma \vdash_{var} \Delta, x : \tau} \quad \text{(v-TYVAR)} \frac{\Sigma \vdash_{var} \Delta \quad \alpha \# \Delta}{\Sigma \vdash_{var} \Delta, \alpha}
\end{array}$$

Fig. 12. Validity Judgments

(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.

$$\begin{array}{c}
\text{(T-CAST)} \frac{\Gamma \vdash_{Co} \gamma : \tau_1 \sim \tau_2 \quad \Gamma \vdash e : \tau_1}{\Gamma \vdash e : \tau_2} \quad \text{(T-TYABS)} \frac{\Gamma \vdash \alpha \text{ type} \quad \Gamma, \alpha \vdash e : \tau}{\Gamma \vdash \Lambda \alpha. e : \forall \alpha. \tau} \quad \text{(T-TYAPP)} \frac{\Gamma \vdash \tau_1 \text{ type} \quad \Gamma \vdash e : \forall \alpha. \tau}{\Gamma \vdash e \tau_1 : \tau[\tau_1/\alpha]} \\
\text{(CO-REFL)} \frac{\Gamma \vdash \tau \text{ type}}{\Gamma \vdash_{Co} \langle \tau \rangle :: \tau \sim \tau} \quad \text{(CO-SYM)} \frac{\Gamma \vdash_{Co} \gamma : \tau_1 \sim \tau_2}{\Gamma \vdash_{Co} \hat{\gamma} : \tau_2 \sim \tau_1} \quad \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 \rightarrow \eta : (\tau_1 \rightarrow \tau_2) \sim (\sigma_1 \rightarrow \sigma_2)} \quad \text{(CO-NTHARR)} \frac{\Gamma \vdash_{Co} \gamma : (\tau_1 \rightarrow \tau_2) \sim (\sigma_1 \rightarrow \sigma_2)}{\Gamma \vdash_{Co} nth_i \gamma : \tau_i \sim \sigma_i} \\
\text{(CO-TYPE)} \frac{\Gamma \vdash n \in \Sigma \quad \overline{\Sigma; \Delta \vdash_{Co} \gamma_i : \tau_i \sim \sigma_i}^{i < n}}{\Sigma; \Delta \vdash_{Co} \overline{\gamma} : \overline{\tau} \sim \overline{\sigma}} \quad \text{(CO-NTH)} \frac{\Gamma \vdash_{Co} \gamma : \overline{\tau} \sim \overline{\sigma}}{\Gamma \vdash_{Co} 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)} \quad \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{F \vdash n \in \Sigma \quad \overline{\Sigma; \Delta \vdash_{Co} \gamma_i : \tau_i \sim \sigma_i}^{i < n}}{\Sigma; \Delta \vdash_{Co} F \overline{\gamma} : F \overline{\tau} \sim F \overline{\sigma}} \quad \text{(CO-AXIOM)} \frac{\Psi = \overline{\forall \alpha. F(N) \sim \sigma} \quad \overline{\Sigma; \Delta \vdash \tau_i \text{ type}} \quad \forall j < i. \text{no\_conflict}(\Psi, i, \tau_i, j) \quad \vdash_{ctx} \Sigma; \Delta}{\Sigma; \Delta \vdash_{Co} \xi_i \overline{\tau} : F(N[\alpha_i/\tau_i]) \sim \sigma[\alpha_i/\tau_i]}
\end{array}$$

Fig. 13. Typing Judgments System  $\mu\text{FC}$ 

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:  $\text{TEq}_0[\text{Int}] : \text{TEq Int Int} \sim \text{TT}$  or  $\text{axiomTEq}_1[\text{Int}, \text{Int}] : \text{TEq Int Int} \sim \text{FF}$ . 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 not in conflict, 1) either the equations are compatible or 2) the equations are apart. The rules are shown in Figure 14

$$\begin{array}{c}
 \text{(NC-APART)} \frac{\Psi = \overline{F(N)} \sim \sigma \quad \text{apart}(N_j, N_i[\bar{\tau}/\bar{\alpha}_i])}{\text{no\_conflict}(\Psi, i, \bar{\tau}, j)} \qquad \text{(COMPT-DIS)} \frac{\Phi_1 = F(N_1) \sim \sigma_1 \quad \Phi_2 = F(N_2) \sim \sigma_2 \quad \Omega = \text{mgu}(N_1, N_2) \text{ fails}}{\text{compat}(\Phi_1, \Phi_2)} \\
 \text{(NC-COMPT)} \frac{\text{compat}(\Psi[i], \Psi[j])}{\text{no\_conflict}(\Psi, i, \bar{\tau}, j)} \qquad \text{(COMPT-INC)} \frac{\Phi_1 = F(N_1) \sim \sigma_1 \quad \Phi_2 = F(N_2) \sim \sigma_2 \quad \Omega = \text{mgu}(N_1, N_2) \quad \Omega\sigma_1 = \Omega\sigma_2}{\text{compat}(\Phi_1, \Phi_2)}
 \end{array}$$

Fig. 14. Non Conflicting Equations and Compatibility

**3.2.1 Type Reduction.** We can formally specify type reduction or rewrite rule, written as  $\Gamma \vdash \bullet \rightsquigarrow \bullet$ , using the rule (TY-RED). This rule says that the  $i$ -th equation of axiom  $\xi, \forall \bar{\alpha}. F(N_i) \sim \sigma_i$  is used to reduce the target type  $F(\bar{\tau})$  to type  $\tau_1$ . The use of  $C[\bullet]$  is to mean that the rewrite can happen anywhere within the type.

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

**3.2.2 Consistency and Goodness of Context.** Consistency means that we can never derive unsound equalities between ground types, such as  $\gamma : \text{Int} \sim \text{Bool}$ , in the system. Ground types ( $\omega$ ), in our system are nothing but  $\text{TEq}, \tau_1 \rightarrow \tau_2$  and  $\forall \alpha. \tau$ . In short, they are types that do not contain type family constructors. We say that a ground context  $\Sigma$  is consistent, when for all coercions  $\gamma$  such that  $\Sigma; \epsilon \vdash_{C_0} \gamma : \omega_1 \sim \omega_2$  we have the following:

- (1) if  $\omega_1$  is of the form  $\text{TEq}$  then so is  $\omega_2$
- (2) if  $\omega_1$  is of the form  $\tau_1 \rightarrow \tau_2$  then so is  $\omega_2$
- (3) if  $\omega_1$  is of the form  $\forall \alpha. \tau$  then so is  $\omega_2$

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

**PROPERTY 9 (Good  $\Sigma$ ).** A ground context ( $\Sigma$ ) is Good, written  $\text{Good}\Sigma$  when the following conditions are met for all  $\xi: \Psi \in \Sigma$  and where  $\Psi$  is of the form  $\overline{\forall \bar{\alpha}. F_i(N_i)} \sim \sigma$ :

- (1) There exists an  $F$  such that  $\forall i. F_i = F$  and none of the type pattern  $N_i$  mentions a type family constructor.
- (2) The binding variables  $\bar{\alpha}$  occur atleast 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  $\text{Good}\Sigma$  then,  $\Sigma$  is consistent. One way to prove this is via confluence of type reduction relation. Whenever we have  $\Sigma \vdash \tau_1 \rightsquigarrow \tau_2$  then we would know that  $\tau_1$  and  $\tau_2$  have a common reduct, say  $\tau_3$ . As type reduction relation (TY-RED) only works on type family's and does not reduce any ground type heads, confluence would be sufficient prove consistency. However, to prove confluence, it is necessary to assume termination of type reduction. We get our consistency lemma as follows.

LEMMA 10 (CONSISTENCY). *If  $\Sigma \vdash \bullet \rightsquigarrow \bullet$  is terminating and  $\text{Good}\Sigma$  then  $\Sigma$  is consistent.*

### 3.3 Type safety of System $\mu\text{FC}$

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 2.5, we should have type safety for System  $\mu\text{FC}$  which includes proving preservation and progress.

$$\begin{array}{c}
\text{(S-APP)} \frac{e_1 \rightsquigarrow e'_1}{e_1 e_2 \rightsquigarrow e'_1 e_2} \quad \text{(S-TAPP)} \frac{e_1 \rightsquigarrow e'_1}{e_1 \tau \rightsquigarrow e'_1 \tau} \quad \text{(S-CAPP)} \frac{e_1 \rightsquigarrow e'_1}{e_1 \gamma \rightsquigarrow e'_1 \gamma} \quad \text{(S-CAST)} \frac{e_1 \rightsquigarrow e'_1}{e_1 \blacktriangleright \gamma \rightsquigarrow e'_1 \blacktriangleright \gamma} \\
\text{(S-}\beta\text{)} \frac{}{(\lambda x:\tau. e_1) e_2 \rightsquigarrow e_1[x/e_2]} \quad \text{(S-T}\beta\text{)} \frac{}{(\Lambda \alpha. e) \tau \rightsquigarrow e[\alpha/\tau]} \quad \text{(S-TRANS)} \frac{}{(e \blacktriangleright \gamma) \blacktriangleright \eta \rightsquigarrow e \blacktriangleright \gamma \circ \eta} \\
\text{(S-PUSH)} \frac{\gamma_1 = \widehat{\text{nth}_0 \gamma} \quad \gamma_2 = \text{nth}_1 \gamma}{(\lambda x:\tau. e \blacktriangleright \gamma) e_1 \rightsquigarrow (\lambda x:\tau. e) (e_1 \blacktriangleright \gamma_1) \blacktriangleright \gamma_2} \quad \text{(S-TPUSH)} \frac{}{(\Lambda \alpha. e \blacktriangleright \gamma) \tau \rightsquigarrow (\Lambda \alpha. e) \tau \blacktriangleright \gamma @ \tau}
\end{array}$$

Fig. 15. Small Step Operational Semantics System  $\mu\text{FC}$

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 11 (COERCION SUBSTITUTION). *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 12 (PRESERVATION SYSTEM  $\mu\text{FC}$ ). *if  $\epsilon \vdash e : \tau$  and  $e \rightsquigarrow e'$  then  $\epsilon \vdash e' : \tau$*

To state progress, it is necessary to show consistency and assume termination of type reduction. We then get our progress lemma as follows:

LEMMA 13 (PROGRESS SYSTEM  $\mu\text{FC}$ ). *If  $\Sigma; \epsilon \vdash e : \tau$  and  $\Sigma \vdash \bullet \rightsquigarrow \bullet$  is terminating, 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  $e \rightsquigarrow e_1$ .*

## 4 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 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  $\mu\text{FC}$ , 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

```

type family PTyFam a where   loopy :: forall 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

both  $[a]$  and  $a$ . As `Loop` unwinds infinitely to become  $[[[...[Loop]...]]]$  we will have  $\text{TEq } [Loop] \text{ Loop}$  evaluate to  $\text{TEq } [Loop] \text{ [Loop]}$  which is  $\text{TT}$ . This justifies the reason for not evaluating  $\text{TEq } [a] \text{ a}$  to  $\text{FF}$ . However, Haskell does not have infinite ground types, and we thus would expect  $\text{TEq } [a] \text{ a}$  to reduce to  $\text{FF}$ . 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  $\text{TEq } [a] \text{ a}$ . 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  $\text{Loop} \rightarrow \text{Loop}$ . But this is not its principle type, as we can generalize it to be  $(a \sim [a]) \Rightarrow a \rightarrow a$ . Haskell however rejects this program on the basis of infinitary unification of  $a \sim [a]$  is not possible. We are left in a position where we accept some problematic definitions, like `Loop`, but not the others like,  $(a \sim [a])$ .

One possible solution to this conundrum is to reject the definition of infinite type families like `Loop`. This would burden the programmers to provide evidence to guarantee termination we thus defer going in that direction. Instead, we leverage the existing infrastructure that Haskell already has—qualified types and typeclasses—to make the totality assumption explicit.

#### 4.1 Closed Typeclasses

Typeclasses as discussed in previous section can be extended to have new instances. Closed typeclasses 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  $\text{TEq } a \text{ b}$  and  $\text{Plus } m \text{ n}$  can be expressed in the closed typeclass world using  $\text{TEqC } a \text{ b}$  and  $\text{PlusC } m \text{ n}$  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  $\text{forall } a. a \rightarrow a$  but instead it is  $\text{forall } a. \text{PTyFamC Bool} \Rightarrow a \rightarrow a$ , 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  $\text{Loop} \sim [\text{Loop}]$  we need to specify `LoopC` to be a constraint on the instance. This makes the use of `Loop` no longer threaten the type soundness as `LoopC` is unsatisfiable.

```

class LoopC where
  type Loop
  instance LoopC ⇒ LoopC where
    type Loop = [Loop]
class {-TOTAL-} TEqC a b where
  type TEq a b
  instance TEqC a a where
    type TEq a a = TT
  instance TEqC a b where
    type TEq a b = FF

class PTyFamC a where
  type PTyFam a
  instance PTyFamC Int where
    type PTyFam Int = Bool
class {-TOTAL-} PlusC m n where
  type Plus m n
  instance PlusC Z m where
    type Plus Z m = Z
  instance PlusC m n ⇒ PlusC (S m) n where
    type Plus (S m) n = S (Plus m n)

```

Fig. 17. Closed Typeclasses Examples

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.

## 4.2 Type matching and Apartness

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.

## 4.3 Formalizing Constrained Type Families

This system is an extension of System  $\mu\text{FC}$ . We have some new constructs which 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  $F:\top n$ ) and a partial type family constructor of arity  $n$  (written  $F:n$ ). Each equation that is introduced by axioms ( $\xi$ ) in this system, are of the form  $\forall \bar{\alpha} \bar{\chi}. F(\bar{\tau}) \sim \sigma$ . Both  $\bar{\tau}$  and  $\sigma$  do not have occurrence of any family type constructors. The equations are quantified by type variables  $\bar{\alpha}$  and also over a new term collection evaluation assumptions  $\bar{\chi}$ . These evaluation assumptions are of the form  $\alpha|c:F(\bar{\tau}) \sim \alpha$  and read as “ $\alpha$  such that  $c$  witnesses  $F(\bar{\tau})$  reduces to  $\alpha$ ”. We use these evaluation assumptions to allow type families on the left hand side of the type equations written in the source program. For example,  $F(T a) = G a$  where  $G$  is a type family constructor and  $T$  is a vanilla type constructor will be compiled into an equation  $\forall a (b|c : Ga \sim b). F(T a) \sim b$ . Evaluation assumptions thus are used to support axioms that mention type families on the right hand side of the equations in source. We use  $\chi$  to remind us that it is more specific than  $P$ ; for any  $\chi$ , the left hand side of the type equality is always a type family and right hand side is a fresh variable.

The ground types in this system can mention type family constructors only in propositions  $P$ . For example, the type  $\forall m n. \text{Add } m n \Rightarrow m \rightarrow n \rightarrow \text{Result } m n$  would instead be written as  $\forall m n. \text{Result } m n \sim p \Rightarrow m \rightarrow n \rightarrow p$ . This is an assertion that  $\text{Result } m n$  evaluates to a type family free type.



		Types	$\tau, \sigma ::= \alpha \mid \tau \rightarrow \tau \mid \forall \alpha. \tau \mid F(\bar{\tau}) \mid T(\bar{\tau}) \mid P \Rightarrow \tau$
833	Type family Constructors	Ground Types	$\omega ::= \tau \rightarrow \tau \mid \forall \alpha. \tau \mid T(\bar{\tau})$
834	Type Constructors	Predicates	$P ::= \tau \sim \tau$
835		Axiom Equations	$\Phi ::= \forall \bar{\alpha} \bar{\chi}. F(\bar{\tau}) \sim \sigma$
836		Axiom Types	$\Psi ::= \Phi$
837	Type Validity	Coercions	$\gamma, \eta ::= \gamma \rightarrow \eta \mid \forall \alpha. \gamma \mid \gamma @ \tau \mid F(\bar{\gamma}) \mid T(\bar{\gamma}) \mid \text{nth}_i \gamma$
838	Proposition Validity		$\mid \langle \tau \rangle \mid \bar{\gamma} \mid \gamma \circ \eta \mid \gamma_1 \sim \gamma_2 \Rightarrow \eta \mid c \mid \xi_i \bar{\tau} \bar{q}$
839	Assumption Validity	Evaluation Assumption	$\chi ::= (\alpha \mid c : F \bar{\tau} \sim \alpha)$
840	Ground Context Validity	Evaluation Resolution	$q ::= (\tau \mid \gamma)$
841	Variable Context Validity		
842	Context Validity	Terms	$e ::= x \mid \lambda x : \tau. e \mid e e \mid M \blacktriangleright \gamma \mid \Lambda \alpha. e \mid e \tau \mid \lambda c : P. e$
843		Values	$\mathcal{V} ::= \lambda x : \tau. e \mid D \bar{e} \mid \Lambda \alpha. e \mid \lambda c : P. e$
844	Term Typing		
845	Coercion Typing		
846	Resolution Validity		
847		Ground Context	$\Sigma ::= \epsilon \mid \Sigma, \xi : \Psi \mid \Sigma, F :_{\top} n \mid \Sigma, F :_{\top} n \mid \Sigma, T : n$
848	One Hole Type Context	Variable Context	$\Delta ::= \epsilon \mid \Delta, x : \tau \mid \Delta, c : P \mid \Delta, \alpha$
849		Typing Context	$\Gamma ::= \Sigma; \Delta$
850			
851			

Fig. 18. System for Constraint Type Families

The new validity judgements are ...

$$\begin{array}{c}
\text{(V-RESE)} \frac{\vdash_{ctx} \Gamma}{\Gamma \vdash_{res} \epsilon : \epsilon} \qquad \text{(V-RESC)} \frac{\Gamma \vdash \sigma \text{ type} \quad \Gamma \vdash_{Co} \gamma : F(\bar{\tau}) \sim \sigma \quad \Gamma \vdash_{res} \bar{q} : \bar{\chi}[a/\sigma]}{\Gamma \vdash_{res} (\sigma \mid \gamma), \bar{q} : (\alpha \mid c : F(\bar{\tau}) \sim \alpha), \bar{\chi}} \\
\\
\text{(V-GQAX)} \frac{\vdash_{gnd} \Sigma \quad \overline{\Sigma; \bar{\alpha}_i, TV(\bar{\chi}) \vdash \tau_{0i} \text{ type}} \quad \overline{\Sigma; \bar{\alpha}_i \vdash \bar{\tau}_i \text{ type}} \quad F : n \in \Sigma}{\vdash_{gnd} \Sigma, \xi : \forall \bar{\alpha} \bar{\chi}. F(\bar{\tau}_i) \sim \tau_{0i}}^{i < k} \\
\\
\text{(V-ASSMN)} \frac{}{\Gamma \vdash \epsilon \text{ asmp}} \quad \text{(V-ASSMC)} \frac{F : n \in \Sigma \quad \overline{\Sigma; \Delta \vdash \tau_i \text{ type}}^{i < n} \quad \Sigma; \Delta, \alpha \vdash \bar{\chi} \text{ asmp}}{\Sigma; \Delta \vdash (\alpha \mid c : F(\bar{\tau}) \sim \alpha), \bar{\chi} \text{ asmp}}
\end{array}$$

Fig. 19. Validity Judgements for System **QFC**

The typing judgments new to this system are shown in Figure 20. The rest are very similar to System  $\mu\text{FC}$ . 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  $\bar{q}$  that instantiate the validity assumptions  $\bar{\chi}$ . The validity resolutions are of the form  $(\tau \mid \gamma)$  where the type  $\alpha$  in validity assumptions  $(\alpha \mid c : F(\bar{\tau}) \sim \sigma)$  is instantiated to  $\tau$  and  $\gamma$  proves the equality and instantiates  $c$ . This is exactly what the rule (V-RESC) does. We write  $[\chi/q]$  for a substitution where the assumption  $\chi$  is substituted by  $q$ .

The rule (T-ASSUM) is a special rule that says we are allowed to assume arbitrary applications of a type family that is total. We can indeed do this as a total type family is expected to reduce to a type family free type.

**4.3.1 Type reduction.** The type reduction relation is given using two rules (QTY-RED-TOP) and (QTY-RED). The reduction relation is different from System  $\mu\text{FC}$  in the sense that it is always guaranteed to terminate. This is due to the fact that

$$\begin{array}{c}
\text{(CO-VAR)} \frac{c:P \in \Delta}{\Sigma; \Delta \vdash c : P} \quad \text{(CO-QAXIOM)} \frac{\frac{\xi:\Psi \in \Sigma}{\Psi = \sqrt{\alpha} \bar{\chi}. F(\bar{\tau}) \sim \sigma} \quad \frac{\Sigma; \Delta \vdash \tau_i \text{ type}}{\forall j < i. \text{no\_conflict}(\Psi, i, \bar{\tau}, j)} \quad \frac{\vdash_{ctx} \Sigma; \Delta}{\Sigma; \Delta \vdash_{res} \bar{q}; \bar{\chi}[\alpha_i/\tau_i]}}{\Sigma; \Delta \vdash_{Co} \xi_i \bar{\tau} \bar{q} : F(\bar{\sigma}[\alpha_i/\tau_i]) \sim \sigma_0[\alpha_i/\tau_i]} \\
\text{(T-COABS)} \frac{\Gamma, c:P \vdash e : \tau}{\Gamma \vdash \lambda c.P. e : P \Rightarrow \tau} \quad \text{(CO-QUAL)} \frac{\overline{\Gamma \vdash \gamma_i : \tau_i \sim \sigma_i}^{i < 3}}{\Gamma \vdash_{Co} \gamma_1 \sim \gamma_2 \Rightarrow \gamma_3 : (\tau_1 \sim \tau_2 \Rightarrow \tau_2) \sim (\sigma_1 \sim \sigma_2 \Rightarrow \sigma_3)} \\
\text{(T-COAPP)} \frac{\Gamma \vdash e : P \Rightarrow \tau \quad \Gamma \vdash \gamma : P}{\Gamma \vdash e \gamma : \tau} \quad \text{(T-ASSUM)} \frac{\frac{\Sigma; \Delta \vdash \tau_i \text{ type}}{F : \tau \vdash n \in \Sigma} \quad \Sigma; \Delta, \alpha, c:F\bar{\tau} \sim \alpha \vdash e : \tau}{\Sigma, \Delta \vdash \text{assume } (\alpha|c:F\bar{\tau} \sim \alpha) \text{ in } e : \tau}
\end{array}$$

Fig. 20. Typing Judgments System *QFC*

we ensure applying type arguments to type families only when they satisfy proper constraints, thus guaranteeing a type reduction to finally obtain a type family free type. For total type families we are automatically guaranteed this by their definition. And due to the fact that type family free types do not reduce (there is no matching rule) we have termination for the type reduction relation.

$$\begin{array}{c}
\text{(QTY-RED)} \frac{\Sigma \vdash F(\bar{\tau}) \rightsquigarrow \tau_1}{\Sigma \vdash C[F(\bar{\tau})] \rightsquigarrow C[\tau_1]} \quad \text{(QTY-RED-TOP)} \frac{\frac{\xi:\Psi \in \Sigma}{\vdash_{gnd} \Sigma} \quad \frac{\Psi = \sqrt{\alpha} \bar{\chi}. F(\bar{\tau}) \sim \sigma}{\forall j < i. \text{no\_conflict}(\Psi, i, \bar{\tau}, j)} \quad \text{fix me}}{\Sigma \vdash F(\bar{\tau}) \rightsquigarrow \tau_1}
\end{array}$$

Fig. 21. Type reduction

4.3.2 *Goodness and Consistency.* The definition of Goodness can now be relaxed as follows:

PROPERTY 14 (Good  $\Sigma$  RELAXED). *A ground context  $(\Sigma)$  is Good, written  $\text{Good}\Sigma$  when the following conditions are met for all  $\xi:\Psi \in \Sigma$  and where  $\Psi$  is of the form  $\sqrt{\alpha}. F_i(N_i) \sim \sigma$ :*

- (1) *There exists an  $F$  such that  $\forall i. F_i = F$ .*
- (2) *The binding variables  $\bar{\alpha}$  occur atleast once in the type arguments  $\bar{\tau}$ , 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 to prove that type reduction is confluence. Thus we get our relaxed consistency lemma as follows:

LEMMA 15 (CONSISTENCY). *If  $\text{Good}\Sigma$  then  $\Sigma$  is consistent*

#### 4.4 Type safety of System *QFC*

Selected rules for term evalutaion 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  $\eta$  so that it can be commuted with the coercion application  $\gamma$ . The rule (T-QRES) is the rule that illuminates the evaluation for total type family constructors. The clause  $F(\bar{\tau}) \Downarrow q$  does what?

ANI: start here

$$\begin{array}{c}
\text{(T-QRES)} \frac{\chi = (\alpha | c:F(\bar{\tau}) \sim \alpha) \quad F(\bar{\tau}) \Downarrow q}{\text{assume } \chi \text{ in } e \rightsquigarrow e[\chi/q]} \quad \text{(T-C}\beta\text{)} \frac{}{(\lambda c:P. e) \gamma \rightsquigarrow e[c/\gamma]} \quad \text{(T-QPUSH)} \frac{\begin{array}{c} v = \lambda c:P. e \quad \eta_1 = \overline{\text{nth}_1} \eta \\ \eta_0 = \text{nth}_0 \eta \quad \eta_2 = \text{nth}_2 \eta \end{array}}{(v \blacktriangleright \eta) \gamma \rightsquigarrow v(\eta_0 \circ \gamma \circ \eta_1) \blacktriangleright \eta_2}
\end{array}$$

Fig. 22. Small Step Operational Semantics System **QFC**

LEMMA 16 (PRESERVATION SYSTEM **QFC**). *If  $\epsilon \vdash e : \tau$  and  $e \rightsquigarrow e'$  then  $\epsilon \vdash e' : \tau$*

Due to constrained type family applications we can weaken the precondition on progress lemma

LEMMA 17 (PROGRESS SYSTEM **QFC**). *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  $e \rightsquigarrow e_1$ .*

## 5 RELATED WORK

Functional dependencies have also been formalized using constraint handling rules (CHR), a technique from logic programming [Sulzmann et al. 2007b]. There is no known implementation of functional dependencies using CHR.

There are several variations of type families that have been explored and implemented in GHC. Open type families [Schrijvers et al. 2008], which predates closed type families, are extendable in a sense that the programmer can add more equations. To maintain both, extensibility and compatibility, open type families disallow overlapping equations. A system with both closed type families and open type families can co-exist without hiccups. This is indeed the current implementation of GHC. The semantics of the instance equations is similar to that of unordered collection of type equations, unlike closed type families, where the equations are considered in a top to bottom fashion. Associated types [Chakravarty et al. 2005] are a syntactic variation of open type families. Each typeclass has an associated type parameterized over typeclass parameters. Each instance of such typeclass specifies what the associated type's interpretation in the context. Injective type families [Stolarek et al. 2015] uses the idea from functional dependencies to specify additional injective constraints that the type family instance should satisfy to aid type inference.

## 6 CONCLUSION AND FUTURE WORK

All three systems described in the paper have a common property that they are based on the principle of type erasure. The type analysis is performed at compile time and the types are erased from the programs at runtime. This ensures that there is no runtime overheads because of types. Due to type safety properties of each of these systems, we are guaranteed that if a program passes the type check then it does not crash at runtime. A feature that we have not included in each of the systems is the system of kinds, or type of types. They provide a way to have partially applied data constructors. The reason to not include them is that they are orthogonal to our discussion. Adding them to the system will not change type safety of the language.

Type computation is an attractive feature for programmers with a condition that type checking and inference is guaranteed termination. If functional dependencies are morally equivalent to type families one would expect to have a translations that can go from one style to another. There are two ways to do this, 1) have a translation to a common intermediate language or 2) have translation procedures that converts one style to another. Recent work by [Karachalias

and Schrijvers 2017] explores the first idea by giving an elaboration of functional dependencies to type families. There is no known implementation of the algorithm given. 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. However, there is no implementation of constraints type families for the users to test it in the wild.

In conclusion, the motivation of constraint type families is to reunite the idea of functional dependencies and type families. 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

- 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.
- 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](https://doi.org/10.1007/3-540-46425-5_15)
- Mark P. Jones and Iavor S. Diatchki. 2008. Language and program design for functional dependencies. In *Proceedings of the first ACM SIGPLAN symposium on Haskell* (New York, NY, USA, 2008-09-25) (Haskell '08). Association for Computing Machinery, New York, NY, USA, 87–98. <https://doi.org/10.1145/1411286.1411298>
- 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](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>
- 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, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. 2008. Type checking with open type functions. *ACM SIGPLAN Notices* 43, 9 (2008), 51–62. <https://doi.org/10.1145/1411203.1411215>
- 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/>
- Jan Stolarek, Simon Peyton Jones, and Richard A. Eisenberg. 2015. Injective type families for Haskell. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell* (New York, NY, USA, 2015-08-30) (Haskell '15). Association for Computing Machinery, New York, NY, USA, 118–128. <https://doi.org/10.1145/2804302.2804314>
- 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>