Constrained Type Families

J. GARRETT MORRIS, The University of Edinburgh, UK
RICHARD A. EISENBERG, Bryn Mawr College, USA

We present an approach to support partiality in type-level computation without compromising expressiveness or type safety. Existing frameworks for type-level computation either require totality or implicitly assume it. For example, type families in Haskell provide a powerful, modular means of defining type-level computation. However, their current design implicitly assumes that type families are total, introducing nonsensical types and significantly complicating the metatheory of type families and their extensions. We propose an alternative design, using qualified types to pair type-level computations with predicates that capture their domains. Our approach naturally captures the intuitive partiality of type families, simplifying their metatheory. As evidence, we present the first complete proof of consistency for a language with closed type families.

CCS Concepts: • **Theory of computation → Type structures**; • **Software and its engineering → Functional languages**;

Additional Key Words and Phrases: Type families, Type-level computation, Type classes, Haskell

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

1 INTRODUCTION

Indexed type families [Chakravarty et al. 2005; Schrijvers et al. 2008] extend the Haskell type system with modular type-level computation. They allow programmers to define and use open mappings from types to types. These have given rise to further extensions of the language, such as closed type families [Eisenberg et al. 2014] and injective type families [Stolarek et al. 2015], and they have many applications, including encoding units of measure in scientific calculation [Muranushi and Eisenberg 2014] and extensible variants [Bahr 2014; Morris 2015].

Nevertheless, some aspects of type families remain counterintuitive. Consider a unary type family F with no defining equations. A type expression such as $F\ \mathit{Int}$ should be meaningless—quite literally, as there are no equations for F to give it meaning. Nevertheless, not only is $F\ \mathit{Int}$ a type, but there are simple programs (such as divergence) that demonstrate its inhabitation. This apparent paradox has both practical and theoretical consequences. For example, we define a closed type family *Equ* such that $\mathit{Equ}\ \tau\ \sigma$ should be *True* iff τ and σ are the same type:[1]

[1]We use here the promoted *Bool* kind, as introduced by Yorgey et al. [2012].

2475-1421/2017/9-ART42
https://doi.org/10.1145/3110286

```
type family Equ a b :: Bool where
  Equ a a = True
  Equ a b = False
```

The type family application *Equ a* [*a*] does not reduce. This surprising fact stems from the use of *infinitary* unification during closed type family reduction [Eisenberg et al. 2014, §6.2]. This explanation raises more questions, however: Haskell does not have infinite types, so why use infinitary unification? Again, type families play a role. Consider the following:

```
type family Loop :: ⋆
type instance Loop = [Loop]
```

The type family application *Loop* will never rewrite to a ground type. But, *Equ Loop* [*Loop*] is equal to *Equ* [*Loop*] [*Loop*], and thus to *True*, justifying not rewriting *Equ a* [*a*] to *False*.

The complexity in this example arises from an underlying inconsistency in the notion of type families. Type families are used identically to other type constructors; that is, uses of type families come with an unstated assumption of totality, regardless of the equations in the program. For example, *Loop* will never reduce to any ground (i.e., type family-free) Haskell type, but still must be treated as a type for the purposes of reducing *Equ a* [*a*]. In essence, *Loop* is treated as a total 0-ary function on types, even though its definition makes it partial. Similar problems arise in injective type families and in interpreting definitions using open type families (§3).

We propose a refinement of indexed type families, *constrained type families*, which explicitly captures partiality in the definition and use of type families. In our design, partial type families be defined associated with type classes. Thus, the domain of a type family is naturally characterized by its corresponding type class predicate. We further insist that uses of type family be qualified by their defining class predicates, guaranteeing that they be well-defined. Non-terminating, or otherwise undefinable, type family applications will be guarded by unsatisfiable class predicates, assuring that they cannot be used to violate type safety.

The introduction of constraints simplifies the metatheory of type families, separating concerns about partiality from the machinery of rewriting. We demonstrate this by formally specifying a calculus with constrained closed type families and showing it is sound, relying on neither infinitary unification nor an assumption of termination, in contrast to previous work on type families [Eisenberg et al. 2014]. In terms of our earlier example, this means that we could safely rewrite *Equ a* [*a*] to *False* without risking type safety.

In summary, this paper contributes:

- An analysis of difficulties in the evolution of type families, including the need for infinitary unification in the semantics of closed type families and the inexpressiveness of injective type families. These warts on the type system belie a hidden assumption of totality (§3).
- The design of constrained type families (§4), which relaxes the assumption of totality by using type class predicates to characterize the domains of definition of partial type families.
- The design of closed type classes (§5), a simplification of instance chains [Morris and Jones 2010]. Closed type classes enable partial closed type families and increase the expressiveness of type classes, subsuming many uses of overlapping instances [Peyton Jones et al. 1997].
- A core calculus with constrained type families (§6), together with a proof of its soundness that requires neither an assumption of termination nor infinitary unification. This is a novel result for a calculus supporting type families with non-linear patterns. Even with infinitary unification, prior work [Eisenberg et al. 2014] was unable to fully prove consistency.
- A design that allows existing Haskell code to remain well typed, so long as it does not depend on the behavior of undefined type family applications (§7).

Although this paper is primarily concerned with Haskell, our work is applicable to any partial language that supports type-level computation. We hope that our work, among others', will encourage other languages to join in the type-level fun.

2 TYPE FAMILIES IN HASKELL

Associated type synonyms [Chakravarty et al. 2005] are a feature of the Haskell type system that allows the definition and use of extensible maps from types to types. They address many of the problems that arise in the use of multi-parameter type classes. One example is a class of collection types, *Collects*. In defining the *Collects* methods for a type c, we naturally need access to the types of its elements. To capture the types of collection elements, we could define the *Collects* class to have an associated type *Elem*:

```
class Collects c where
  type Elem c :: ⋆
  empty :: c
  insert :: Elem c → c → c
```

This declares both the *Collects* class and the type family *Elem*. Instances of the *Collects* class, must also specify instances of the *Elem* type family:

```
instance Collects [a] where
  type Elem [a] = a
  empty = [ ]
  insert = (:)
```

While associated types provide a natural syntactic combination of class and type family definitions, the class and type family components can actually be specified and formalized entirely separately [Schrijvers et al. 2008]. Instead of using an associated type synonym, we could have defined *Elem* as a distinct top-level entity.

```
type family Elem c :: ⋆
class Collects c where
  empty :: c
  insert :: Elem c → c → c
```

While there would then be no syntactic requirement to combine instances of the class and type family, it is easy to see that class instances would be undefinable without corresponding type family instances, while type family instances would be unusable without corresponding type class instances.

```
type instance Elem [a] = a
instance Collects [a] where
  empty = [ ]
  insert = (:)
```

These definitions are entirely equivalent to the original definitions; while it may be impractical to use a type family instance *Elem* τ without a corresponding instance *Collects* τ, it is not an error in either approach to do so.

Type families can express many type-level computations. However, some useful type-level functions cannot be expressed using open type families. One is the type family *Equ a b*, as appeared in the introduction. We might hope to characterize *Equ* using the following equations:

```
type family Equ a b :: Bool
type instance Equ a a = True
type instance Equ a b = False
```

However, type family instances are interpreted without any ordering, arising either from their source locations or from their relative generality. In this case, both equations apply to a type family application such as *Equ Int Int* but give different results, and so they are rejected as inconsistent. Closed type families [Eisenberg et al. 2014] address this problem by allowing ordered overlap among the instances in a type family definition, so long as the family cannot be further extended. We could write the equality function using a closed type family, as we did in the introduction. Closed type families cannot be extended later, even if their definitions do not cover all possible applications. For example, consider the following definition:

```
type family OnlyInt a :: Bool where
  OnlyInt Int = True
```

The type family application *OnlyInt Bool* does not rewrite to any ground type (i.e., type without type family applications), but the programmer is still prevented from adding further equations to *OnlyInt*.

In general, type families need not be injective. However, there are cases in which it would be useful to capture the natural injectivity of type-level definitions. For example: session types, which provide static typing for communication protocols, depend on a naturally injective notion of duality. We would expect that if the duals of two session types are equal, then the session types themselves are equal as well. Injective type families [Stolarek et al. 2015] can express such cases; duality could be characterized by the following type family:

```
type family Dual s = r | r → s
```

where the $s \rightarrow r$ annotation denotes the injectivity of duality. The compiler validates that the injectivity condition is upheld by the type family's defining equations.

The most recent version of the Glasgow Haskell Compiler, GHC 8.0, accepts all the varieties of type families described above.

3 THE TOTALITY TRAP

Recent developments in the theory and implementation of type families [Eisenberg et al. 2014; Stolarek et al. 2015] have relied on increasingly technical and confusing constraints, impeding their use in practice. In this section, we argue that these problems arise from a single source: an implicit assumption of totality for type families.

3.1 The Assumption of Totality

Type families are open and extensible, and so suggesting that they are assumed total seems counterintuitive. However, we can rephrase the question as follows. Suppose that we have a type family *F* with no equations. Is *F Bool* a type? It seems absurd that it should be—after all, the meaning of a type family is given by its equations, and *F* has no equations. Yet we can observe that it is:

```
type family F a :: ⋆
f x = fst (x, undefined :: F Bool)
```

This is a well-typed definition: *f* has type $a \rightarrow a$ and behaves like the identity function. So *F Bool* must be a type, even if all we can observe about it are properties true of all Haskell types (such as pointedness, or definition of *seq*). While it is possible that *F Bool* will be defined later in the program, this program would be equally valid were *F* replaced by a closed type family, such as

OnlyInt (above), in which case we could say with confidence that *OnlyInt Bool* would never become defined.

This illustrates that our intuitive understanding of type families is flawed. Rather than thinking of type families as partial functions on types, where individual instances extend the definition of the type family, a type family declaration should be thought of as initially introducing an infinite family of distinct types, one for each possible application of the type family, and individual instances as equating previously distinct types. But does this distinction cause actual problems? Consider the following definition:

$g\ x = x : x$

We might expect this definition to be ill-typed: x must have both type τ and type $[\tau]$, a seeming contradiction. But recall type family *Loop* (§1). If we must assume that *Loop* is a type, then it is clearly a satisfying instantiation of the constraint $\tau \sim [\tau]$, and so we can assign g the type $Loop \to Loop$. But worse, we expect Haskell terms to have principal types, and since g makes no reference to *Loop*, $Loop \to Loop$ cannot be its principal type. Instead, we conclude that g has principal type $a \sim [a] \Rightarrow a \to a$.

Now the true consequences of the totality assumption are revealed. It is not only a gap between the intuitive and actual meanings of type families, nor just an incompleteness in specifications of type checking and type inference with type families. Rather, we are left with a type system which must accept some (but not all) apparently erroneous definitions: we can reject $Int \sim Bool$, even if we must accept $a \sim [a]$. The specification of principal types for this system remains an open question.

It might seem that the problems illustrated here are not to do with the totality assumption itself, but rather in its interaction with the accepted equations for *Loop*, and therefore should be fixed simply by rejecting *Loop* (and other non-terminating type family definitions). However, this would burden the programmer with satisfying some termination checking algorithm, and does not reflect the realities of either type family practice or research (where significant effort has been devoted to accounting for non-terminating type families). Instead, we propose (§4) an approach that restores the intuitive interpretation of type families, preserves their current uses, and avoids introducing new constraints, such as termination checking.

3.2 Closed Type Families and the Infinity Problem

We have seen that assuming totality of type families introduces a variety of theoretical problems. With the development of closed type families, the totality assumption began causing practical problems as well, as demonstrated above in the unpleasant interaction between *Equ* and *Loop*.

Closed type families rely on a notion of *apartness* to determine when an equation cannot apply to a particular type family application. Intuitively, two types are apart if they have no common instantiations; for example, *Equ Int Bool* is apart from *Equ a a*, while *Equ a b* is not apart from *Equ a a*. This intuition can be formalized in terms of unification: two types are apart if they have no most general unifier. The problems with *Loop* arise from the apartness of *Equ a* $[a]$ and *Equ a a*: while these instances do not have any most general unifier in the typical sense, considering them apart leads to the unsoundness above. This problem is addressed in closed type families by defining apartness in terms of *infinitary* unification. As there is an infinite (i.e., non-idempotent) unifier of *Equ a* $[a]$ and *Equ a a* (namely $\{[a]/a\}$) *Equ a* $[a]$ does not rewrite to *False* until a is instantiated to some concrete type.

While the interaction between closed and infinite type families may seem like a theoretical concern, the solution causes confusion in practice. Programmers discovering that *Equ a* $[a]$ does not rewrite to *False* consider this a bug in the implementation rather than an expected behavior

of the type system.[2] It can also result in programs that use closed type families to require more complex type signatures than similar programs expressed using older techniques, like overlapping instances [Peyton Jones et al. 1997] or instance chains [Morris and Jones 2010].

3.3 Explosive Injectivity

We have seen that the totality assumption causes both theoretical and practical problems in the ongoing development of type families. These problems have depended on the interaction of other type system features with non-terminating type families. This might seem like a corner case, and one that programmers would not expect to encounter in practice. Recent work on injective type families bring the problems caused by the totality assumption into starker relief, without relying on non-terminating type families.

Some families of types are naturally injective; examples include duality relationships [Lindley and Morris 2016; Pucella and Tov 2008] or the pairing between mutable and immutable vectors types in the `vector` library.[3] However, because type families are not injective in general, expressing such examples required either the introduction of additional constraints to assure involutiveness or the use of either proxy arguments or type applications [Eisenberg et al. 2016] to fix type parameters. Injective type families [Stolarek et al. 2015] introduce annotations on type family declarations that characterize their injectivity. For example, the duality function for session types could be declared by

type family $Dual\ s = r \mid r \to s$

This declaration differs from traditional type family declarations in two ways: first, the result is named (r), and second, the annotation $r \to s$ specifies *Dual*'s injectivity: its result determines its argument.

Unfortunately, injective type families require seemingly arcane restrictions to preserve type safety. For example, consider the following apparently innocuous definitions:

type family $ListElems\ a = b \mid b \to a$
type instance $ListElems\ [a] = a$

ListElems is clearly injective: if $a \sim b$ then $[a] \sim [b]$. Nevertheless, this example is sufficient to derive a violation of type safety: by the definition of *ListElems*, we have that $ListElems\ [ListElems\ Int] \sim ListElems\ Int$, and then by injectivity, we have that $[ListElems\ Int] \sim Int$, an impossibility. In the previous sections, difficulties stemmed from the type family application *Loop*, which does not correspond to any ground type. In this case, problems arise from the type family application $ListElems\ Int$, which similarly cannot correspond to any ground type. Suppose that we could prove that $ListElems\ Int \sim \tau$ for some type τ; as, from the definition of *ListElems* we also have that $ListElems\ [\tau] \sim \tau$, the injectivity of *ListElems* has been violated.

Definitions like that of *ListElems* are ruled out by strict restrictions on the right-hand sides of injective type family equations; for example, the RHS of an injective type family equation cannot (in most cases) be a type variable or another type family application. These restrictions are necessary to assure the safety of injective type families, but have not yet been shown to be sufficient. They

[2] See, among others:

- https://ghc.haskell.org/trac/ghc/ticket/9082: Unexpected behavior involving closed type families and repeat occurrences of variables
- https://ghc.haskell.org/trac/ghc/ticket/9918: GHC chooses an instance between two overlapping, but cannot resolve a clause within the similar closed type family

[3] See https://github.com/haskell/vector/issues/34: Add immutable type family.

are also a significant limitation in expressiveness, especially in comparison with older approaches, such as functional dependencies [Jones 2000].

4 CONSTRAINING TYPE FAMILIES

In the previous section, we have seen that indexed type families are implicitly assumed to be defined at all their applications—that is, they represent total functions on types. We have seen how this totality assumption introduces practical and theoretical obstacles, both in preserving totality (such as in injective type families) or in accounting for its violations (such as in the interaction between non-terminating and closed type families).

We propose a new approach, *constrained type families*, which treats type families as partial maps between types. Our key observation is that Haskell already supplies a mechanism to limit the domain of polymorphism: qualified types with type classes. So we can capture partiality by associating each type family with a type class that characterizes its domain of definition. We will show that this approach naturally resolves the practical and theoretical issues with type families and restores their intuitive meaning, while adding little new complexity for programmer or implementer.

This section describes constrained open type families; we discuss the extension of our approach to closed type families in the following section.

4.1 Constrained Type Families

Our goal is a system of partial type families that sacrifices neither the expressiveness nor the ease of use of present type families. This introduces two challenges. First, we must retain the applicative syntax of type families, while taking their domains of definition into account. That is, a type family application such as $F\ \tau$ should be constrained by the domain of F. In particular, whether a type family application that contains type variables, such as $F\ a$, is well-defined depends on the instantiation of the type variable a. Second, we must keep type families easy to define, while simultaneously characterizing their domains of definition.

We address each of these problems using features already present in modern Haskell. Haskell already has a mechanism suited to capturing this kind of constrained polymorphism: qualified types and type classes [Wadler and Blott 1989]. Qualified types are currently used to track when type class methods are defined; for example, the equality operator is defined at all types $a \to a \to \mathit{Bool}$ such that the class predicate $\mathit{Eq}\ a$ is satisfiable. Our intention is to reuse the qualified types mechanism to account for partiality in type families as well. Haskell also supports a mechanism that combines type classes and type families: associated type synonyms. Our intention is to rely on associated types to simultaneously define type families and characterize their domains.

We propose combining these mechanisms to give an account of partial type families that matches both the intuitions and usage of type families in Haskell today. In doing so, we make two changes to the surface language. First, we require that type families be defined by associated types, disallowing free-standing type family declarations. This means that the well-definedness of type family instances follows from the satisfiability of the corresponding class predicates. In our previous example (§2), the type family application $\mathit{Elem}\ \tau$ is defined precisely when the predicate $\mathit{Collects}\ \tau$ is satisfiable. Second, we require that all uses of type families be well-defined, as enforced by their corresponding class predicates. That is, uses of the type family $\mathit{Elem}\ \tau$ must occur in a context that is sufficient to prove $\mathit{Collects}\ \tau$ (either because $\mathit{Collects}\ \tau$ is assumed or provable from the instances).

Our approach captures the natural interpretation and use of open type families. Open type families are already primarily useful in combination with type class constraints—we have no way to use a value of type $\mathit{Elem}\ \tau$ unless we have some additional information about that type, captured by the class constraint $\mathit{Collects}\ \tau$. Thus, our requirements do not reduce the expressiveness of the

language. The remainder of the section demonstrates informally that our approach addresses the difficulties and confusion with type families.

We begin with the behavior of undefined, or "stuck", type family instances (§3.1). As before, We define a type family, *F2*, now associated with a class *C2*:

class *C2 t* **where**
  type *F2 t* :: ⋆

Instances of the *F2* type family can be added only by adding instances to the *C2* class:

instance *C2 Int* **where**
  type *F2 Int* = *Bool*

Now, recall our function definition:

$f\ x = fst\ (x, undefined :: F2\ Bool)$

Is this definition still well-typed? The use of *F2 Bool* requires that *C2 Bool* be satisfiable to assure that it is well defined. However, without any instances of *C2 Bool* in scope, the constraint would be unsatisfiable, so the definition would be rejected. This account extends naturally to polymorphism. Suppose that we had some function *g* that used *F2*, with the following type:

$g :: C2\ a \Rightarrow a \rightarrow F2\ a$

(Note the requisite *C2 a* constraint.) Now, we could define an alternative version of *f* as follows:

$f'\ x = fst\ (x, g\ x)$

The definition of *f'* is not ill-typed, but its type, $C2\ a \Rightarrow a \rightarrow a$, includes the *C2 a* constraint to assures that the type of *g x* is well-defined.

The complications with closed type families arose from their interaction with non-terminating type families. We can already see how non-terminating type family definitions would play out in our system. As before, we define a type family *Loop*, but now as an associated type to a type class *Loopy*:

class *Loopy* **where**
  type *Loop* :: ⋆

As *Loop* is a 0-ary type family, *Loopy* is a 0-ary type class. This is not problematic; in particular, there are two canonical 0-ary type classes, one whose predicates are trivially true and another whose predicates are unsatisfiable. Now, suppose we want to add the equation $Loop \sim [Loop]$. We would need to do so via an instance of *Loopy*. However, we cannot add the instance

instance *Loopy* **where**
  type *Loop* = [*Loop*]

as the use of *Loop* on the right-hand side of the type definition does not have a corresponding constraint. We can add the instance

instance *Loopy* ⇒ *Loopy* **where**
  type *Loop* = [*Loop*]

but it is clear that the *Loopy* constraint cannot be satisfied. Thus, any attempt to use this *Loop* equation must be guarded by an unsatisfiable *Loopy* constraint, and so cannot compromise type safety.

Finally, we can give an informal description of constrained injective type families. We return to the *ListElems* example, now defining it by an associated type synonym:

$$\frac{\alpha \in \Gamma \qquad \vdash P \mid \Gamma \text{ ctx}}{P \mid \Gamma \vdash \alpha \text{ type}}\ \text{ST\_Var} \qquad \frac{P \mid \Gamma, \alpha \vdash \tau \text{ type}}{P \mid \Gamma \vdash \forall\, \alpha.\tau \text{ type}}\ \text{ST\_Forall} \qquad \frac{P, \pi \mid \Gamma \vdash \tau \text{ type}}{P \mid \Gamma \vdash \pi \Rightarrow \tau \text{ type}}\ \text{ST\_Qual}$$

$$\frac{\overline{P \mid \Gamma \vdash \tau_i \text{ type}}^{\,i<n} \qquad (H : n) \in \Sigma \qquad \vdash P \mid \Gamma \text{ ctx}}{P \mid \Gamma \vdash H\,\overline{\tau} \text{ type}}\ \text{ST\_TyCon}$$

$$\frac{(C \Rightarrow F : n) \in \Sigma \qquad \vdash P \mid \Gamma \text{ ctx} \qquad \overline{P \mid \Gamma \vdash \tau_i \text{ type}}^{\,i<n} \qquad P \Vdash C\,\overline{\tau}}{P \mid \Gamma \vdash F\,\overline{\tau} \text{ type}}\ \text{ST\_Family}$$

Fig. 1. Well-formedness rules for types

```
class Listy t where
  type ListElems t = u | u → t
instance Listy [t] where
  type ListElems [t] = t
```

Notice that we could not add an instance *Listy Int*, as that would require adding a corresponding instance to the type family and any such instance would be rejected for violating the injectivity constraint of *ListElems*. Consequently, inconsistencies arising from uses of the type family application *ListElems Int* must be guarded by the unsatisfiable class constraint *Listy Int*.

Constrained type families are not, in their simplest form, backward compatible. We will return to the question of compatibility with existing Haskell programs, and show how we can infer the requisite constraints to transition from current usage to the explicit use of constrained type families (§7).

4.2 Validating Constrained Type Families

In the previous section, we introduced an intuitive characterization of constrained type families. Later (§6), we will formalize a core calculus with constrained type families. However, our formalization will differ from Haskell-like surface languages in several significant ways. This section bridges the intuition of constrained type families and our core language, in the context of a simple, Haskell-like type system.

Figure 1 gives the syntax and formation rules for our surface type system. We omit kinds from our account, as they are an orthogonal concern from the use of type classes and type families. Our well-formedness judgment takes the form $P \mid \Gamma \vdash \sigma$ type, in which σ is a surface-language type, Γ is a type variable environment, and P is a predicate context. As we have omitted kinds, the environment Γ is simply a list of type variables. The form of the judgment and use of context P should be familiar from other formulations of qualified types [Jones 1994].

Our types include type variables (α), quantified types ($\forall\, \alpha.\tau$), qualified types ($\pi \Rightarrow \tau$), and applications of type constructors ($H\,\overline{\tau}$) and type families ($F\,\overline{\tau}$). The rules for variables, quantifiers, and qualifiers should all be unsurprising. Leaf nodes depend on an auxiliary well-formedness judgment $\vdash P \mid \Gamma$ ctx on contexts, which is entirely unsurprising. Our treatments of type constructors and type families depend on an ambient signature Σ, representing the top-level declarations. Arity n type constructors are captured by entries $(H : n) \in \Sigma$; the typing rule for constructors assures that they have the correct number of arguments. The interesting case is for type families. Constrained type families are represented by assertions $(C \Rightarrow F : n) \in \Sigma$; this denotes that type family F has arity n, and is associated with class C. Uses of the type family application $F\ \overline{\tau}$, then, should occur in a context strong enough to prove $C\ \overline{\tau}$. This is captured by ST_Family, in which we insist that the context P is strong enough to prove $C\,\overline{\tau}$; we omit the details of the standard type class entailment relation $\cdot \Vdash \cdot$. For a simple example, suppose that F is a unary type family declared in class C, and

class D is a subclass of C. Then we could prove any of the following judgments:

$$C\ \tau \mid \emptyset \vdash F\ \tau\ \text{type} \qquad D\ \tau \mid \emptyset \vdash F\ \tau\ \text{type} \qquad \emptyset \mid \emptyset \vdash C\ \tau \Rightarrow F\ \tau\ \text{type}$$

but, absent other instances of C, we could not prove $\emptyset \mid \emptyset \vdash F\ \tau$ type.

5 ACHIEVING CLOSURE

Closed type families are one of the most fruitful extensions of indexed type families. They allow type families to be specified by ordered sequences of overlapping equations, capturing many patterns of type-level computation that were previously inexpressible or required intricate indirect encodings. In this section, we discuss the extension of constrained type families to include closed type families. This introduces two challenges. First, there is no existing feature of type classes that mirrors closed type families. We introduce closed type classes, a simplification of instance chains [Morris and Jones 2010], and show how they can be used to constrain closed type families. Second, closed type families may be total, and so could be used without constraints. We discuss approaches to recognizing and supporting total closed type families. Finally, we illustrate the simplification our approach provides over previous formulations of closed type families.

5.1 Closed Type Classes

Closed type classes are our novel approach to introducing and resolving overlap among class instances. They closely follow the design of closed type families: just as closed type families allow type families to be defined by ordered sequences of overlapping equations, closed type classes allow type classes to be defined by ordered sequences of overlapping instances. Instance resolution begins with the first instance in the sequence, and proceeds to subsequent instances only if the first instance cannot match the goal predicate. In the next section, we will show that closed type classes can characterize the domain of definition of closed type families. We begin, however, by considering closed type classes as a feature on their own.

As an example, we consider heterogeneous lists, following the approach of Kiselyov et al. [2004]. We begin by introducing data types to represent heterogeneous lists:

```
data HNil = MkHNil
data HCons e l = MkHCons e l
```

For example, the declaration

```
hlst = MkHCons True (MkHCons 'c' MkHNil)
```

defines a heterogeneous list *hlst* with type *HCons Bool* (*HCons Char HNil*). Kiselyov et al. describe a number of operations on heterogeneous lists, and show how they can be used to build more complex data structures, such as extensible records. We will limit ourselves to some of the simpler operations. One such operation is *hOccurs*, which projects all elements of a given type from a heterogeneous list. We can define *hOccurs* using a closed type class as follows:

```
class HOccurs e l where
  hOccurs :: l → [e]

  instance HOccurs e HNil where
    hOccurs MkHNil = [ ]

  instance HOccurs e l ⇒ HOccurs e (HCons e l) where
    hOccurs (MkHCons e l) = e : hOccurs l

  instance HOccurs e l ⇒ HOccurs e (HCons e' l) where
    hOccurs (MkHCons _ l) = hOccurs l
```

HOccurs is a closed type class, as indicated by the sequence of instances inside the class declaration. The second two instances are overlapping—for example, both apply to the predicate *HOccurs Char* (*HCons Char HNil*)—but the ordering indicates that the first instance should apply in the common cases. Depending on its expected return type, *hOccurs hlst* could evaluate to [*True*], ['c'], or [].

Closed type classes bear a close resemblance to overlapping instances [Peyton Jones et al. 1997], a well-established extension of the Haskell class system. However, whereas the order of instances in closed type families is explicit in their declaration, overlapping instances have an implicit ordering, fixed by the compiler. This means that overlapping instances can lead to unintended ambiguity. For example, in Swierstra's [2008] encoding of extensible variants, he relies on a data type of functor coproducts:

```
data (f ⊕ g) e = Inl (f e) | Inr (g e)
```

He defines a class of polymorphic injectors, as follows:

```
class f ≤ g where
  inj :: f e → g e
instance f ≤ f where
  inj = id
instance f ≤(f ⊕ g) where
  inj = Inl
instance f ≤ h ⇒ f ≤(g ⊕ h) where
  inj = Inr ∘ inj
```

The intuition here is simple: these instances describe a recursive search of (right-grouped) coproduct types, in which the first two instances provide base cases and the third instance provides the recursive case. However, there is actually an unresolved overlap among the instances: the predicate $(f \oplus g) \leq (f \oplus g)$ could be resolved by either the first or third instance, and neither is more specific than the other. Consequently, GHC will report an error if such a predicate is encountered. An implementation of this class using closed type class (written simply by indenting the **instance** declarations to fit within the **class** body) would be unambiguous, and the predicate $(f \leq g) \leq (f \oplus g)$ would be resolved using the first instance.

5.2 Constrained Closed Type Families

Combining closed type classes and associated types gives us a way to introduce closed type families while accurately characterizing their domains of definition.

For an example, we turn again to the heterogeneous lists of Kiselyov et al. [2004]. Our new goal is to define an operation *hDelete*, which will remove all values of a given type from a heterogeneous list. In doing so, we must simultaneously define a mapping on types describing the type of the resulting list. We do this by defining an associated type *HWithout* such that, if *l* is a heterogeneous list type, then *HWithout e l* is the same list without any occurrences of element type *e*. Thus, we arrive at the following closed type class definition.

```
class HDelete e l where
  type HWithout e l :: ⋆
  hDelete :: Proxy e → l → HWithout e l
  instance HDelete e HNil where
    type HWithout e HNil = HNil
```

```
    hDelete _ MkHNil = MkHNil
  instance HDelete e l ⇒ HDelete e (HCons e l) where
    type HWithout e (HCons e l) = HWithout e l
    hDelete ep (MkHCons _ l) = hDelete ep l
  instance HDelete e l ⇒ HDelete e (HCons e' l) where
    type HWithout e (HCons e' l) = HCons e' (HWithout e l)
    hDelete ep (MkHCons e' l) = MkHCons e' (hDelete ep l)
```

The class *HDelete e l* has the *hDelete* method and the *HWithout* associated type synonym; to disambiguate the type of *hDelete*, we capture the type *e* using a *Proxy* argument. The *HDelete* class has three instances, following the same recursion scheme we used for *HOccurs*; again, the final two instances overlap. Like conventional closed type families, the associated type synonym equations are checked in the order in which they appear in the type class definition. For example, we have that *HWithout Char* (*HCons Bool* (*HCons Char HNil*)) ∼ *HCons Bool HNil*. Note that *HWithout* is not total: while it is defined for arbitrary *e*, it is only defined for *l* that are properly formed heterogeneous list types.

5.3 Closed Type Families and Totality

Unlike open type families, closed type families can be total.[4] For example, we could implement addition for type-level naturals using constrained closed type classes as follows:

```
data Nat = Z | S Nat
class PlusC (m :: Nat) (n :: Nat) where
  type Plus m n :: ⋆
  instance PlusC Z n where
    type Plus Z n = n
  instance PlusC m n ⇒ PlusC (S m) n where
    type Plus (S m) n = S (Plus m n)
```

This formulation behaves roughly as we expect: *Plus M N* evaluates to the sum of the naturals *M* and *N*, while the predicate *PlusC M N* is satisfied for arbitrary naturals *M* and *N*. However, in this case, the *PlusC M N* predicates are unnecessary: *Plus M N* is defined for arbitrary naturals *M* and *N*. Furthermore, the requirement to include these predicates could significantly complicate definitions using polymorphic recursion. For a simple example, consider the definition of the *append* function for length-indexed vectors. We might hope to write it as follows:

```
data Vec (a :: ⋆) (n :: Nat) where
  Nil  :: Vec a Z
  Cons :: a → Vec a n → Vec a (S n)
append :: PlusC m n ⇒ Vec a m → Vec a n → Vec a (Plus m n)
append Nil         ys = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

However, the type signature given here is not strong enough: in the second case, where we know that *m* is *S m'* for some *m'*, we also need to know that *PlusC m' n* holds. But this does not follow

[4] Open type families might also be total, with the right equations. Any such open type family can, however, be written as a closed family. We thus consider all open type families to be partial.

from the assumption *PlusC* (*S m'*) *n*. It would seem that we would have to define *append* itself via a type class:

```
class PlusC m n ⇒ Append m n where
  append :: Vec a m → Vec a n → Vec a (Plus m n)
  instance Append Z n where
    append Nil ys = ys
  instance Append m n ⇒ Append (S m) n where
    append (Cons x xs) ys = Cons x (append xs ys)
```

But this is verbose, and complicates what should be a simple definition. It also complicates uses of *append*, which will now have to include the *Append* constraint instead of the *PlusC* constraint or (even better) just an application of the *Plus* type family.

In essence, having recognized that most type families are partial, *some* are total, and users should be able to take advantage of this fact. If we could recognize *Plus* as total, then we could allow the following, much simpler definition of *append*:

```
append :: Vec a m → Vec a n → Vec a (Plus m n)
append Nil         ys = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

This definition needs no constraints, as the type-checker is aware that *Plus* is total, with no possibility for a usage outside its domain of definition.

We now have a new, challenging question: how do we know when a type family is total? Totality checking of functional programs is a hard problem, one we do not propose to solve here. This problem is well studied both in the context of dependently-typed programming[5] and partial evaluation [Lee et al. 2001; Sereni and Jones 2005]. In practice, an implementation of our ideas would use a totality checker to discover or check the totality of type families. Users could also have the capability to (unsafely) assert the totality of functions that lie beyond the abilities of the checker.

We can extend our type formation rules (§4.2) to take account of total type families. Intuitively, we can think of a total type family as a constrained type family for which the constraint is trivially provable. To formalize this notion, we extend our top-level environment Σ to include total type families $\top \Rightarrow F : n$ as well as partial type families $C \Rightarrow F : n$. Then, we can add a new rule that allows total type families regardless of the context:

$$\frac{(\top \Rightarrow F : n) \in \Sigma \qquad \vdash P \mid \Gamma \text{ ctx} \qquad \overline{P \mid \Gamma \vdash \tau_i \text{ type}}^{\,i<n}}{P \mid \Gamma \vdash F\,\overline{\tau} \text{ type}} \text{ ST\_TFamily}$$

While this rule is superficially similar to the rule for type constructors, it will have a different elaboration into our core calculus, which must explicitly account for the totality of *F*.

5.4 Simplifying Apartness

As introduced above (§3.2), closed type family reduction critically relies on a notion of apartness on types. The existing definition of apartness [Eisenberg et al. 2014, §3.3] is subtle, requiring both infinitary unification and a *flattening* operation to account for the possibility of type family applications in the arguments to another type family. Because type families cannot appear directly as arguments to other type families, the flattening operation—whose details thankfully no longer concern us—becomes redundant. In addition, because we require the caller of a function to provide

[5] E.g., https://coq.inria.fr/cocorico/CoqTerminationDiscussion

the ground type to which a type family reduces at every call site, we no longer have to worry about infinite types and infinitary unification. Thus, we can define apartness very simply: as the inverse of unifiability. Indeed, our formal development (§6) no longer contains a first-class notion of apartness, using unification directly.

6 TYPE SAFETY OF CONSTRAINED TYPE FAMILIES

For over a decade, GHC has compiled its variant of Haskell into System FC [Sulzmann et al. 2007], a variant of System F [Girard et al. 1989; Reynolds 1974] that supports explicit *coercions*, or proofs of equality between types. As type family instances introduce new such equalities (via axioms), type families are integrated into FC. Accordingly, proving the type safety of System FC requires careful reasoning about type family reduction. As the safety of Haskell itself rests on the safety of FC,[6] we must now show that our extension of constrained type families retains soundness.

Indeed we go further: by adding constrained type families and a new treatment of axioms, we can now prove that all type family reduction chains in System FC terminate, thus closing the gap in the proof presented by Eisenberg et al. [2014], which was unable to cope with the interaction of non-linear patterns and non-terminating type families.

This section presents an overview of our formalism and a sketch of our proofs. The full definitions and proofs can be found in our evaluated proof artifact.

6.1 System CFC

We will study a simplified version of System FC, called CFC ("constrained FC"). The grammar for the language is presented in Figure 2 and is checked by the judgments in Figures 3–7. Broadly speaking, CFC is like System F, but with explicit coercions witnessing equality between types and usable in type conversions (see rule E_Cast, Figure 4). The features in this system beyond those in System F are all driven by these coercions. Before describing the novelty of CFC, we take a quick tour of the grounds. Novel components are indicated in the following discussion; the rest of System CFC follows previous work (e.g., [Breitner et al. 2016; Eisenberg et al. 2014]).

Types in CFC are like those in System F, but with three additions: $H\,\overline{\tau}$ is a fully applied type constant H (allowing partial application would require reasoning about kinds), $\phi \Rightarrow \tau$ is a type τ qualified by an equality assumption ϕ, and $F\,\overline{\tau}$ is a fully applied type family F. Perhaps unexpectedly, classes are not included. The novel constrained nature of type families arises from CFC's differentiation between pretypes (any production of metavariable τ) and types (as validated by $\Gamma \vdash \tau$ type, Figure 3); proper types may mention type families only in a proposition ϕ. Examine the judgment $\Gamma \vdash \phi$ prop (Figure 3). Its rule P_Types allows the proposition to relate two proper types, while the rule P_Family allows a saturated type family application to be related to a type. Thus, in CFC, we would write $\mathit{insert} :: \forall a\ c.\ \mathit{Elem}\ c \sim a \Rightarrow a \to c \to c$ instead of the more typical $\mathit{insert} :: \forall c.\ \mathit{Collects}\ c \Rightarrow \mathit{Elem}\ c \to c \to c$. In effect, the type family equality assumption $\mathit{Elem}\ c \sim a$ takes the place of the class constraint $\mathit{Collects}\ c$: both assert that $\mathit{Elem}\ c$ can evaluate to a proper (type family-free) type.

The language omits any consideration of kinds, as the complexity of kinds does not illuminate the invention of constrained type families.

Expressions e are checked by the judgment $\Gamma \vdash e : \tau$ (Figure 4). There are two leaf forms, for variables x and constants (such as data constructors) K. In addition to System F's two forms of abstraction and application (over expressions and types), CFC contains abstraction and application over coercions. Accordingly, a function may assume an equality proposition ϕ relating two

[6] We are unaware of a precise semantics for the surface Haskell language that accounts for all the features of modern GHC/Haskell.

Metavariables.

α	type variables	x	term variables
c	coercion variables	ξ	axioms
F	type families	H	type constants
K	term constants (constructors)		

Notations.

- Substitutions application: $\tau[\theta]$
- Substitutions composition: $\theta = \theta_1 \circ \theta_2$
- $F : n$ stands for either $F :_{\mathcal{T}} n$ or $F :_{\top} n$
- Free variables of constructs: $fv(\cdot)$
- $tvs(\overline{\chi})$: bound type variables of $\overline{\chi}$
- Domains of contexts are denoted $dom(\Gamma)$

Grammar.

τ, σ, ρ	::=	$H\,\overline{\tau} \mid \tau_1 \to \tau_2 \mid \alpha \mid \forall\,\alpha.\tau \mid \phi \Rightarrow \tau \mid F\,\overline{\tau}$	types
ϕ	::=	$\tau_1 \sim \tau_2$	constraints
γ, η	::=	$\langle\tau\rangle \mid \mathbf{sym}\,\gamma \mid \gamma_1 \fatsemi \gamma_2 \mid H\,\overline{\gamma} \mid \gamma_1 \to \gamma_2 \mid \forall\,\alpha.\gamma$	coercions
	$\mid$	$\gamma_1 \sim \gamma_2 \Rightarrow \gamma_3 \mid F\,\overline{\gamma} \mid \mathbf{nth}_i\,\gamma \mid \gamma@\tau \mid c \mid \xi_i\,\overline{\tau}\,\overline{q}$	
e	::=	$x \mid K \mid \lambda x : \tau.e \mid e_1\,e_2 \mid \Lambda\alpha.e \mid e\,\tau$	expressions
	$\mid$	$\lambda c : \phi.e \mid e\,\gamma \mid e \triangleright \gamma \mid \mathbf{assume}\,\chi\,\mathbf{in}\,e$	
v	::=	$K \mid \lambda x : \tau.e \mid \Lambda\alpha.e \mid \lambda c : \phi.e$	values
χ	::=	$(\alpha \mid c : F\,\overline{\tau} \sim \alpha)$	evaluation assumption
q	::=	$(\tau \mid \gamma)$	evaluation resolution
E	::=	$\forall\,\overline{\alpha}\,\overline{\chi}.F\,\overline{\tau} \sim \tau_0$	type family equations
Σ	::=	$\emptyset \mid \Sigma, F :_{\mathcal{T}} n \mid \Sigma, F :_{\top} n \mid \Sigma, \xi : \overline{E}$	signatures
δ	::=	$\alpha \mid c{:}\phi \mid x{:}\tau$	bindings
Γ	::=	$\emptyset \mid \Gamma, \delta$	typing contexts
θ	::=	$\emptyset \mid \theta, \tau/\alpha \mid \theta, \gamma/c \mid \theta, e/x$	substitutions
$\mathcal{V}$	::=	$\ldots$	sets of variables
$C[\cdot]$	::=	$\ldots$	one-hole type contexts

Fig. 2. System CFC Design

types. The feature can be seen in the rules E_CLam and E_CApp (Figure 4). Though this language omits datatypes, generalized algebraic datatypes (GADTs) can be encoded using coercion abstractions [Sulzmann et al. 2007, §3.2]. Coercions are used in casts $e \triangleright \gamma$, which use the coercion to change the type of an expression (E_Cast, Figure 4). Lastly, expressions also contain a novel form **assume** χ **in** e used in our account of total type families (§6.3).

The small-step operational semantics (Figure 4) provides the relation $e \longrightarrow e'$. The definition for $\longrightarrow$ contains congruence forms to allow evaluation in applications and casts, β-reductions over the three application forms, and four push rules (counting S_Trans as a push rule for casts). The push rules allow us to move casts around when they get in the way—for example when a cast prevents us from reducing an applied λ-expression. Though somewhat intricate, these rules are derived straightforwardly simply by making choices in order to have the output expression preserve the type of the input expression. The novel rule S_Resolve is discussed with **assume** (§6.3). Values in CFC are unsurprisingly constants and abstractions.

Of the main productions in the grammar, we are left with coercions γ, checked by the judgment $\Gamma \vdash \gamma : \phi$ (Figure 5). A coercion is a witness of type equality; thus, the rules for coercion formation determine the equality relation underlying the type system.[7] The critical property of this relation is *consistency*—that we can never prove, for example, that *Int* equals *Bool*. We return to consistency

[7]In a similar system that leaves coercions out but has a conversion rule, the rules for $\Gamma \vdash \gamma : \phi$ would correspond to rules for definitional equality, often rendered with $\equiv$.

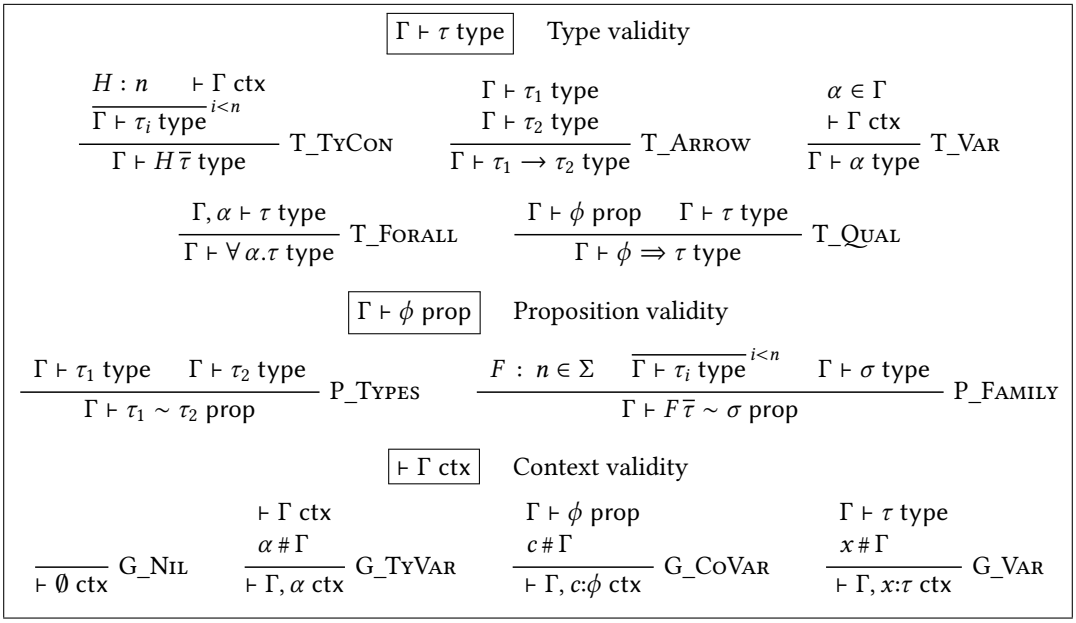

Fig. 3. Type validity judgments

and our proof thereof later in this section (§6.4). The equality relation as witnessed by these coercions has several properties:

- Our equality relation is indeed an equivalence, as witnessed by coercion forms for reflexivity ($\langle\tau\rangle$), symmetry (**sym** γ), and transitivity ($\gamma_1 \fatsemi \gamma_2$).
- Equality is congruent, as witnessed by a coercion for each recursive type form.
- Equality can be decomposed via the $\mathbf{nth}_i\ \gamma$ and $\gamma@\tau$ coercions. The former extracts equalities from applied type constants (C_Nth), function arrows (C_NthArrow), and qualified types (C_NthQual). The latter instantiates an equality between polytypes (C_Inst), giving us an equality between the two polytype bodies.
- Equality can be assumed, as witnessed by coercion variables c.
- Crucially, equality witnesses the reduction of type families through the form $\xi_i\ \overline{\tau}\ \overline{q}$ and the rule C_Axiom, as discussed in the next subsection.

Unlike in other developments of System FC, this system does *not* support a coercion regularity lemma; that is, $\Gamma \vdash \gamma : \phi$ does *not* imply that $\Gamma \vdash \phi$ prop. In other words, the two types related by a coercion may mention type families at arbitrary depths. The lemma was used primarily for convenience in prior proofs; its omission here does not bite.

6.2 Type Family Axioms and Signatures

Following prior work on System FC (initially that of Sulzmann et al. [2007]), we use *axioms* ξ to witness type family reductions. That is, if there is an equation **type** *F Int* = *Bool* in scope, then we have an axiom ξ that proves *F Int* $\sim$ *Bool*. An expression can then use this axiom to cast an expression of type *Bool* to one of type *F Int*.

In System CFC, axioms exist in an ambient signature Σ (which, more formally, should appear in every judgment; we omit this to reduce clutter). Signatures contain both declarations for type families $F : n$ and axiom declarations $\xi : \overline{E}$. The former has two forms: $F :_{\mathcal{T}} n$ declares a *partial*

$\boxed{\Gamma \vdash e : \tau}$ Expression typing

$$\frac{x{:}\tau \in \Gamma \qquad \vdash \Gamma \text{ ctx}}{\Gamma \vdash x : \tau}\ \text{E\_Var} \qquad \frac{K : H\,\overline{\tau} \qquad \vdash \Gamma \text{ ctx}}{\Gamma \vdash K : H}\ \text{E\_Const}$$

$$\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1.e : \tau_1 \to \tau_2}\ \text{E\_Lam} \qquad \frac{\Gamma, \alpha \vdash e : \tau}{\Gamma \vdash \Lambda\alpha.e : \forall\,\alpha.\tau}\ \text{E\_TLam} \qquad \frac{\Gamma, c{:}\phi \vdash e : \tau}{\Gamma \vdash \lambda c : \phi.e : \phi \Rightarrow \tau}\ \text{E\_CLam}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}\ \text{E\_App} \qquad \frac{\Gamma \vdash e : \forall\,\alpha.\tau \quad \Gamma \vdash \sigma \text{ type}}{\Gamma \vdash e\,\sigma : \tau[\sigma/\alpha]}\ \text{E\_TApp} \qquad \frac{\Gamma \vdash e : \phi \Rightarrow \tau \quad \Gamma \vdash \gamma : \phi}{\Gamma \vdash e\,\gamma : \tau}\ \text{E\_CApp}$$

$$\frac{\Gamma \vdash e : \tau_1 \quad \Gamma \vdash \gamma : \tau_1 \sim \tau_2 \quad \Gamma \vdash \tau_2 \text{ type}}{\Gamma \vdash e \triangleright \gamma : \tau_2}\ \text{E\_Cast} \qquad \frac{F :_\top n \in \Sigma \quad \overline{\Gamma \vdash \tau_i \text{ type}}^{\,i<n} \quad \Gamma, \alpha, c{:}F\,\overline{\tau} \sim \alpha \vdash e : \sigma \quad \alpha \notin fv(\sigma)}{\Gamma \vdash \textbf{assume}\ (\alpha | c : F\,\overline{\tau} \sim \alpha)\ \textbf{in}\ e : \sigma}\ \text{E\_Assume}$$

$\boxed{e \longrightarrow e'}$ Small-step operational semantics

$$\frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2}\ \text{S\_App} \qquad \frac{e \longrightarrow e'}{e\,\tau \longrightarrow e'\,\tau}\ \text{S\_TApp} \qquad \frac{e \longrightarrow e'}{e\,\gamma \longrightarrow e'\,\gamma}\ \text{S\_CApp} \qquad \frac{e \longrightarrow e'}{e \triangleright \gamma \longrightarrow e' \triangleright \gamma}\ \text{S\_Cast}$$

$$\frac{}{(\lambda x : \tau.e_1)\ e_2 \longrightarrow e_1[e_2/x]}\ \text{S\_Beta} \qquad \frac{}{(\Lambda\alpha.e)\ \tau \longrightarrow e[\tau/\alpha]}\ \text{S\_TBeta}$$

$$\frac{}{(\lambda c : \phi.e)\ \gamma \longrightarrow e[\gamma/c]}\ \text{S\_CBeta} \qquad \frac{v = \lambda c : \phi.e_0 \quad \eta_0 = \textbf{nth}_0\ \eta \quad \eta_1 = \textbf{sym}\,(\textbf{nth}_1\ \eta) \quad \eta_2 = \textbf{nth}_2\ \eta}{(v \triangleright \eta)\ \gamma \longrightarrow v\,(\eta_0\ ;\ \gamma\ ;\ \eta_1) \triangleright \eta_2}\ \text{S\_CPush}$$

$$\frac{v = \lambda x : \tau.e_0 \quad \gamma_1 = \textbf{sym}\,(\textbf{nth}_0\ \gamma) \quad \gamma_2 = \textbf{nth}_1\ \gamma}{(v \triangleright \gamma)\ e \longrightarrow v\,(e \triangleright \gamma_1) \triangleright \gamma_2}\ \text{S\_Push} \qquad \frac{v = \Lambda\alpha.e \quad \gamma' = \gamma@\tau}{(v \triangleright \gamma)\ \tau \longrightarrow v\,\tau \triangleright \gamma'}\ \text{S\_TPush}$$

$$\frac{}{(v \triangleright \gamma_1) \triangleright \gamma_2 \longrightarrow v \triangleright (\gamma_1\ ;\ \gamma_2)}\ \text{S\_Trans} \qquad \frac{\chi = (\alpha | c : F\,\overline{\tau} \sim \alpha) \quad F\,\overline{\tau} \Downarrow q}{\textbf{assume}\ \chi\ \textbf{in}\ e \longrightarrow e[q/\chi]}\ \text{S\_Resolve}$$

Fig. 4. Expression judgments

type family F that takes n arguments, and $F :_\top n$ declares a *total* type family. The difference is in the treatment of the **assume** construct (§6.3).

An axiom ξ is classified by a list of equations $\overline{E}$, where each equation has the form $\forall\,\overline{\alpha}\,\overline{\chi}.F\,\overline{\tau} \sim \tau_0$. Using a list of equations, as opposed to only one equation, is necessary to support closed type families, with their ordered lists of equations. However, the intricacies of closed type families do not affect our main contribution to this formalism (i.e., the constraining of type family applications via the distinction between pretypes and types). We will thus consider only singleton lists of equations E for now. We return to the full generality of closed type families below.

In an equation E, the types $\overline{\tau}$ and the type τ_0 are proper types, with no type family applications; the lack of type family application on the right-hand side (τ_0) is new in this work. As in prior work on type families, equations can be quantified over type variables $\overline{\alpha}$; this allows the equations to work at many types. For example, the equation $F\ (\mathit{Maybe}\ a) = a$ is quantified over the variable a.

$\boxed{\Gamma \vdash \gamma : \phi}$ Coercion validity

$$\frac{\Gamma \vdash \tau \text{ type}}{\Gamma \vdash \langle \tau \rangle : \tau \sim \tau}\ \text{C\_Refl} \qquad \frac{\Gamma \vdash \gamma : \tau_1 \sim \tau_2}{\Gamma \vdash \mathbf{sym}\,\gamma : \tau_2 \sim \tau_1}\ \text{C\_Sym} \qquad \frac{\Gamma \vdash \gamma_1 : \tau_1 \sim \tau_2 \quad \Gamma \vdash \gamma_2 : \tau_2 \sim \tau_3}{\Gamma \vdash \gamma_1 \fatsemi \gamma_2 : \tau_1 \sim \tau_3}\ \text{C\_Trans}$$

$$\frac{H : n \qquad \vdash \Gamma \text{ ctx} \qquad \overline{\Gamma \vdash \gamma_i : \tau_i \sim \sigma_i}^{\,i<n}}{\Gamma \vdash H\,\overline{\gamma} : H\,\overline{\tau} \sim H\,\overline{\sigma}}\ \text{C\_App} \qquad \frac{\Gamma \vdash \gamma_1 : \tau_1 \sim \sigma_1 \qquad \Gamma \vdash \gamma_2 : \tau_2 \sim \sigma_2}{\Gamma \vdash \gamma_1 \to \gamma_2 : (\tau_1 \to \tau_2) \sim (\sigma_1 \to \sigma_2)}\ \text{C\_Fun}$$

$$\frac{F : n \in \Sigma \qquad \vdash \Gamma \text{ ctx} \qquad \overline{\Gamma \vdash \gamma_i : \tau_i \sim \sigma_i}^{\,i<n}}{\Gamma \vdash F\,\overline{\gamma} : F\,\overline{\tau} \sim F\,\overline{\sigma}}\ \text{C\_Fam} \qquad \frac{\Gamma, \alpha \vdash \gamma : \tau_1 \sim \tau_2}{\Gamma \vdash \forall\,\alpha.\gamma : (\forall\,\alpha.\tau_1) \sim (\forall\,\alpha.\tau_2)}\ \text{C\_Forall}$$

$$\frac{\Gamma \vdash \gamma_1 : \tau_1 \sim \sigma_1 \qquad \Gamma \vdash \gamma_2 : \tau_2 \sim \sigma_2 \qquad \Gamma \vdash \gamma_3 : \tau_3 \sim \sigma_3}{\Gamma \vdash \gamma_1 \sim \gamma_2 \Rightarrow \gamma_3 : (\tau_1 \sim \tau_2 \Rightarrow \tau_3) \sim (\sigma_1 \sim \sigma_2 \Rightarrow \sigma_3)}\ \text{C\_Qual}$$

$$\frac{\Gamma \vdash \gamma : H\,\overline{\tau} \sim H\,\overline{\sigma}}{\Gamma \vdash \mathbf{nth}_i\,\gamma : \tau_i \sim \sigma_i}\ \text{C\_Nth} \qquad \frac{\Gamma \vdash \gamma : (\tau_0 \to \tau_1) \sim (\sigma_0 \to \sigma_1)}{\Gamma \vdash \mathbf{nth}_i\,\gamma : \tau_i \sim \sigma_i}\ \text{C\_NthArrow}$$

$$\frac{\Gamma \vdash \gamma : (\tau_0 \sim \tau_1 \Rightarrow \tau_2) \sim (\sigma_0 \sim \sigma_1 \Rightarrow \sigma_2)}{\Gamma \vdash \mathbf{nth}_i\,\gamma : \tau_i \sim \sigma_i}\ \text{C\_NthQual} \qquad \frac{\Gamma \vdash \gamma : (\forall\,\alpha.\sigma_1) \sim (\forall\,\alpha.\sigma_2) \quad \Gamma \vdash \tau \text{ type}}{\Gamma \vdash \gamma@\tau : \sigma_1[\tau/\alpha] \sim \sigma_2[\tau/\alpha]}\ \text{C\_Inst}$$

$$\frac{c{:}\phi \in \Gamma \quad \vdash \Gamma \text{ ctx}}{\Gamma \vdash c : \phi}\ \text{C\_Var} \qquad \frac{\xi : \overline{E} \in \Sigma \qquad E_i = \forall\,\overline{\alpha}\,\overline{\chi}.F\,\overline{\tau} \sim \tau_0 \qquad \vdash \Gamma \text{ ctx} \qquad \overline{\Gamma \vdash \sigma_j \text{ type}}^{\,j} \qquad \Gamma \vdash \overline{q} : \overline{\chi}[\overline{\sigma}/\overline{\alpha}] \qquad \forall\, n < i, \text{no\_conflict}(\overline{E}, i, \overline{\sigma}, n)}{\Gamma \vdash \xi_i\,\overline{\sigma}\,\overline{q} : F\,\overline{\tau}[\overline{\sigma}/\overline{\alpha}] \sim \tau_0[\overline{\sigma}/\overline{\alpha}, \overline{q}/\overline{\chi}]}\ \text{C\_Axiom}$$

$\boxed{\Gamma \vdash \overline{q} : \overline{\chi}}$ Evaluation resolution validity

$$\frac{\vdash \Gamma \text{ ctx}}{\Gamma \vdash \emptyset : \emptyset}\ \text{A\_Nil} \qquad \frac{\Gamma \vdash \sigma \text{ type} \qquad \Gamma \vdash \gamma : F\,\overline{\tau} \sim \sigma \qquad \Gamma \vdash \overline{q} : \overline{\chi}[\sigma/\alpha]}{\Gamma \vdash (\sigma|\gamma), \overline{q} : (\alpha|c : F\,\overline{\tau} \sim \alpha), \overline{\chi}}\ \text{A\_Cons}$$

Fig. 5. Coercion validity judgments

Also novel in this work is quantification over *evaluation assumptions* $\overline{\chi}$. The form for χ is $(\alpha|c : F\,\overline{\tau} \sim \alpha)$, read "$\alpha$ such that c witnesses that $F\,\overline{\tau}$ reduces to α". Quantification over evaluation assumptions is necessary to support type families that reduce to other type families. For example, we might have $F\ (\mathit{Maybe}\ a) = G\ a$; such an equation would compile to $\forall\, a\ (b \mid c : G\ a \sim b).\ F\ (\mathit{Maybe}\ a) \sim b$. Because of evaluation assumptions, we can continue to support equations such as $F\ (\mathit{Maybe}\ a) = G\ a$ even while disallowing type families on the right-hand sides of axioms. The assumptions in a type family equation bind a coercion variable c, though this variable is not used; the use of χ here (instead of a construct that does not bind c) is for simplicity and parallelism with the χ in the **assume** construct. Note that evaluation assumptions are more specific than arbitrary equality assumptions ϕ, requiring a type family on the left and requiring that the right-hand side be a fresh type variable. This restrictive form is critical in proving that type family reduction is confluent (§6.4).

Signatures, with their type family equations, are validated by the judgment $\vdash \Sigma$ ok and its auxiliary judgment $\Gamma \vdash \overline{\chi}$ assumps, both in Figure 6.

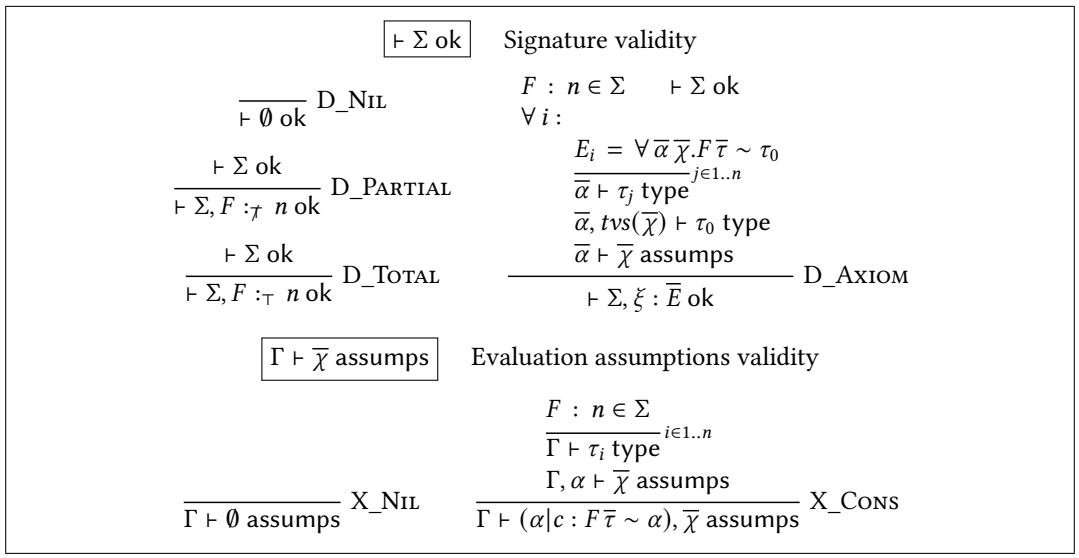

Fig. 6. Signature validity

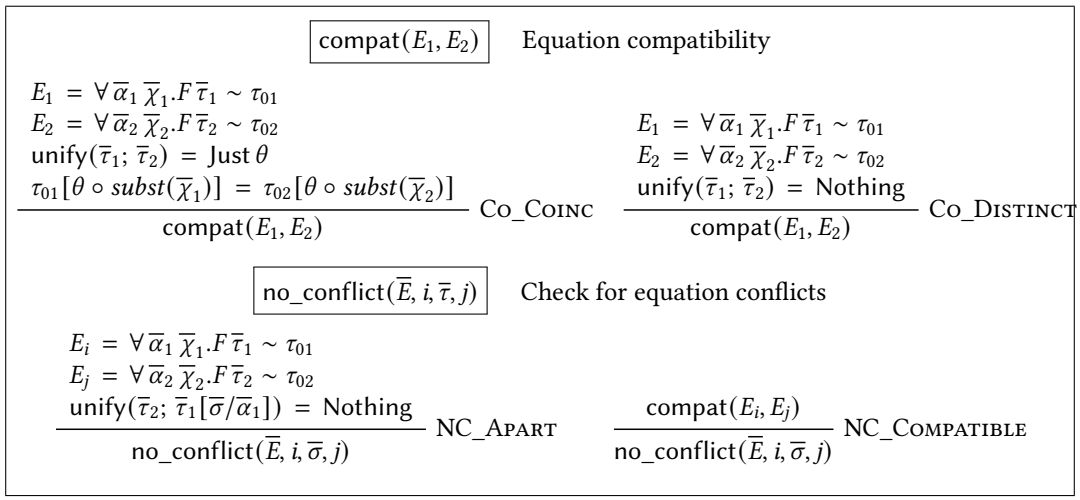

Fig. 7. Closed type family auxiliary judgments

The use of an axiom ξ to form a coercion has the form $\xi_i\, \overline{\tau}\, \overline{q}$, supplying the index i of the equation to use (for now, i will always be 0), a list of types $\overline{\tau}$ used to instantiate the type variables $\overline{\alpha}$, and a list of *evaluation resolutions* $\overline{q}$ used to instantiate the evaluation assumptions $\overline{\chi}$. An evaluation resolution q has the form $(\tau|\gamma)$, where the type τ can instantiate the type variable α in $(\alpha|c : F\, \overline{\tau} \sim \alpha)$, and the coercion γ proves the equality and instantiates the c. We write q/χ to mean a substitution that maps the type and coercion, respectively.

To understand the daunting rule C_Axiom, let's first simplify it to eliminate the possibility of multiple equations for the given axiom. Here is the simplified version:

$$\frac{\xi : \forall\, \overline{\alpha}\ \overline{\chi}.F\ \overline{\tau} \sim \tau_0 \in \Sigma \qquad \vdash \Gamma\ \mathsf{ctx} \qquad \overline{\Gamma \vdash \sigma_j\ \mathsf{type}}^{\,j} \qquad \Gamma \vdash \overline{q} : \overline{\chi}[\overline{\sigma}/\overline{\alpha}]}{\Gamma \vdash \xi_0\ \overline{\sigma}\ \overline{q} : F\ \overline{\tau}[\overline{\sigma}/\overline{\alpha}] \sim \tau_0[\overline{\sigma}/\overline{\alpha}, \overline{q}/\overline{\chi}]}\ \text{C\_Axiom (simplified)}$$

The rule looks up the axiom in the signature, checks to make sure the $\overline{\sigma}$ are proper (type family-free) types and that the $\overline{q}$ satisfy the assumptions $\overline{\chi}$ (using the auxiliary judgment $\Gamma \vdash \overline{q} : \overline{\chi}$, Figure 5). The $\Gamma \vdash \overline{q} : \overline{\chi}$ judgment is straightforward, matching up the $\overline{q}$ with the corresponding $\overline{\chi}$ and checking that the coercions in $\overline{q}$ prove the correct propositions.

Let's now generalize to full closed type families with an ordered list of equations.[8] Here is the full rule for axioms:

$$\frac{\xi : \overline{E} \in \Sigma \qquad E_i\ =\ \forall\, \overline{\alpha}\ \overline{\chi}.F\ \overline{\tau} \sim \tau_0 \qquad \vdash \Gamma\ \mathsf{ctx} \qquad \overline{\Gamma \vdash \sigma_j\ \mathsf{type}}^{\,j} \qquad \Gamma \vdash \overline{q} : \overline{\chi}[\overline{\sigma}/\overline{\alpha}] \qquad \forall\, n < i, \mathrm{no\_conflict}(\overline{E}, i, \overline{\sigma}, n)}{\Gamma \vdash \xi_i\ \overline{\sigma}\ \overline{q} : F\ \overline{\tau}[\overline{\sigma}/\overline{\alpha}] \sim \tau_0[\overline{\sigma}/\overline{\alpha}, \overline{q}/\overline{\chi}]}\ \text{C\_Axiom}$$

Compared to the rule above, this rule uses the index i to look up the right equation; it also adds an invocation of the no_conflict judgment (Figure 7). This check is substantively identical to the existing check for closed type families but with our simplified notion of apartness (see (§5.4)); the two necessary judgments appear in Figure 7. The only change from prior work is in the use of the *subst* operator in the premise to Co_Coinc. This rule detects when two type family equations are *compatible*. Recalling Eisenberg et al. [Eisenberg et al. 2014], two equations are compatible if, whenever they are both applicable to the same type, they will yield the same result. This can happen in two ways: if the two equations' left-hand sides are unifiable, then the right-hand sides coincide under the unifying substitution (Co_Coinc); or the two equations' left-hand sides have no overlap (Co_Distinct). In the former case, we must be careful, as the true right-hand sides of the equations may mention type families; we thus use *subst* to generate a substitution over the evaluation assumptions $\overline{\chi}$, expanding out the variables bound in the $\overline{\chi}$ to the type family applications they equal.

6.3 Totality and Assumptions

The challenge to totality in CFC is best understood by example. Consider again the *append* operation on length-indexed vectors (§5.3), repeated here:

```
append :: Vec a m → Vec a n → Vec a (Plus m n)
append Nil         ys = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

In CFC, the type of *append* would be rewritten to become

```
append :: Plus m n ~ p ⇒ Vec a m → Vec a n → Vec a p
```

But now we have a problem. In the *Cons* case, we have learned that *m ~ Succ m'* for some *m'*; *xs* has type *Vec a m'*. When we make the recursive call to *append*, we must provide a *p'* such that *Plus m' n ~ p'*. However, there is no way to get such a *p'* from the information to hand.

[8] The inclusion of closed type families in the formalization is to support our claim of a consistency proof in the presence of closed type families. However, the treatment of these families here is not novel, and our contributions have a minimal impact on the presentation of closed type families—it is the *metatheory* that is affected, not the *theory*. The intricacies of closed type families may therefore be skipped by readers not interested in reproducing our proof.

The solution to this problem is the **assume** construct. The idea of **assume** χ **in** e is that we are allowed to assume that arbitrary applications of a total type family reduce to proper types. Indeed, that's what *total* means!

Let's now examine the typing rule for assumptions:

$$\frac{F :_\top n \in \Sigma \quad \overline{\Gamma \vdash \tau_i \text{ type}}^{i<n} \quad \Gamma, \alpha, c{:}F\,\overline{\tau} \sim \alpha \vdash e : \sigma \quad \alpha \notin fv(\sigma)}{\Gamma \vdash \textbf{assume}\ (\alpha | c : F\,\overline{\tau} \sim \alpha)\ \textbf{in}\ e : \sigma}\ \text{E\_Assume}$$

This rule requires that the type family be total, according to the $\top$ subscript in the $F :_\top n \in \Sigma$ premise. It then checks the body e in a context where we have a type α and coercion c, as bound by χ. Finally, α is essentially existential, so the rule also does a skolem escape check to assure that α does not leak into the type of e.

Discharging such assumptions is straightforward:

$$\frac{\chi = (\alpha | c : F\,\overline{\tau} \sim \alpha) \quad F\,\overline{\tau} \Downarrow q}{\textbf{assume}\ \chi\ \textbf{in}\ e \longrightarrow e[q/\chi]}\ \text{S\_Resolve}$$

When an **assume** construct is ready to reduce, we are in an empty context—meaning that all type variables have concrete values. At this point, we simply evaluate the type family application at the concrete values. We are sure that this evaluation is possible, due to the totality of the type function. The $F\,\overline{\tau} \Downarrow q$ operation does the work for us, as defined in this property of total type families:

Property 6.1 (Total type families). *For all $F :_\top n \in \Sigma$ and all $\overline{\tau_i}^{i<n}$ such that $\overline{\emptyset \vdash \tau_i \text{ type}}$, there exists q such that $\emptyset \vdash q : (\alpha | c : F\,\overline{\tau} \sim \alpha)$. Define $F\,\overline{\tau} \Downarrow q$ to witness the above fact.*

This property must hold for any total type family, as accepted by any totality checker.

6.4 Metatheory: Consistency of Equality

System CFC admits the usual preservation and progress theorems.

Theorem 6.2 (Preservation). *If $\emptyset \vdash e : \tau$ and $e \longrightarrow e'$, then $\emptyset \vdash e' : \tau$.*

Theorem 6.3 (Progress). *If $\emptyset \vdash e : \tau$, then either e is a value v, e is a coerced value $v \triangleright \gamma$, or $e \longrightarrow e'$ for some e'.*

The proof of preservation is uninteresting. The hardest part is verifying that the push rules are correct, but the only challenge is attention to detail. The unusual choice to make the context empty in this proof is to support the S_Resolve rule, whose premise $F\,\overline{\tau} \Downarrow q$ is well-defined only in an empty context, according to Property 6.1.

On the other hand, proving progress requires reasoning about the consistency of our equality relation. This need arises in the case, among others, for E_App:

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}\ \text{E\_App}$$

We use the induction hypothesis to say that e_1 is a value v_1, a coerced value $v_1 \triangleright \gamma$, or steps to e_1'. In the case where $e_1 = v_1 \triangleright \gamma$, we then wish to use S_Push to show that the overall expression can step. However, this rule requires that v_1 have the form $\lambda x : \tau.e_0$. The only way to show this is that the coercion γ relates two functions.

The consistency lemma is what we need:

Lemma 6.4 (Consistency). *If $\emptyset \vdash \gamma : \tau_1 \sim \tau_2$, $\emptyset \vdash \tau_1$ type, and $\emptyset \vdash \tau_2$ type, then $\tau_1 = \tau_2$.*

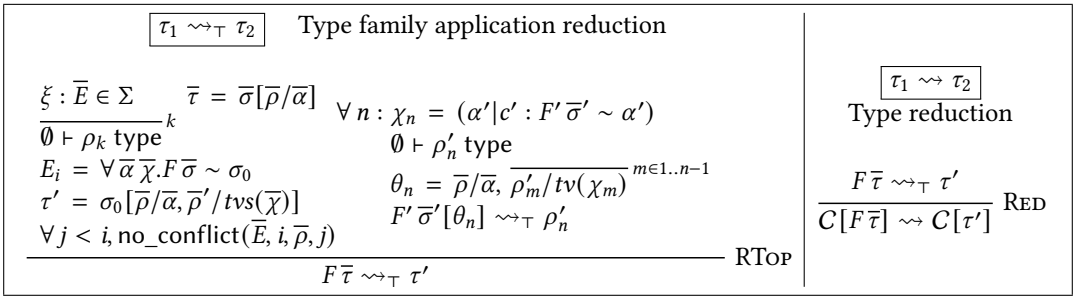

Fig. 8. Non-deterministic type reduction

In an empty context and when two types are type family free, if they are related by a coercion, then they must be the same. Using the following regularity lemma about expression typing, we can use consistency in the proof of progress to finish the E_App case, among others.

6.4.1 The Route to Consistency. Broadly speaking, we prove consistency in the same manner as in previous work.[9] First, we must restrict the set of available axioms to obey the following syntactic rules:

Assumption 6.5 (Good signature). *We assume that our implicit signature Σ conforms to the following rules, adapted from Eisenberg et al. [2014, Definition 18]:*

(1) For all $\xi : \overline{E} \in \Sigma$ where $E_i = \forall \overline{\alpha}_i\,\overline{\chi}_i.F_i\,\overline{\tau}_i \sim \tau_{0\,i}$, there exists F such that, for all i, $F_i = F$. That is, every equation listed within one axiom is over the same type family F.

(2) For all $\xi : \overline{E} \in \Sigma$ where $E_i = \forall \overline{\alpha}_i\,\overline{\chi}_i.F_i\,\overline{\tau}_i \sim \tau_{0\,i}$, for all i, $fv(\overline{\tau}_i) = \overline{\alpha}_i$. That is, every quantified type variable in an equation is mentioned free in a type on the equation's left-hand side.

(3) For all $\xi : \overline{E} \in \Sigma$, if $\text{length}(\overline{E}) > 1$ and the equations are over type family F, then no other axiom $\xi' : \overline{E}' \in \Sigma$ is over the same type family F. That is, all axioms with multiple equations are for closed *type families.*

(4) For all $\xi_1 : E_1 \in \Sigma$ and $\xi_2 : E_2 \in \Sigma$ (each with only one equation), if E_1 and E_2 are over the same type family F, then $\text{compat}(E_1, E_2)$. That is, equations for open type families are all pairwise compatible.

The conditions above are identical to the conditions in Eisenberg et al. [2014, Definition 18], but with one change: we here do not need to restrict the left-hand types of equations not to mention type families, because of the $\overline{\Gamma \vdash \tau_i \text{ type}}^{i<n}$ premise to D_Axiom describing the validity of axioms in the signature. Type family applications are not types.

Then, we define a non-deterministic rewrite relation on types $\tau_1 \rightsquigarrow \tau_2$ and prove both of the following:

Lemma 6.6 (Completeness of the rewrite relation). *If $\emptyset \vdash \gamma : \tau_1 \sim \tau_2$, then there exists τ_3 such that $\tau_1 \rightsquigarrow^* \tau_3 \leftsquigarrow^* \tau_2$.*

Lemma 6.7 (Proper types do not reduce). *If $\Gamma \vdash \tau$ type, then there exists no τ' such that $\tau \rightsquigarrow \tau'$.*

Taken together, these quickly prove the consistency lemma.

6.4.2 Type Reduction Relation. The type reduction relation $\rightsquigarrow$ is captured by the judgments in Figure 8. Rule Red says that a type σ can reduce by reducing a type family application occurring anywhere within σ. (The metavariable C denotes one-hole type contexts.) The intimidating RTop

[9] The best point of comparison is with Eisenberg et al. [2014], as that proof considers closed type families, as does ours here.

rule matches up with C_Axiom. The complication in the rule is in dealing with the evaluation assumptions $\overline{\chi}$ in a given type family equation; each needs to be satisfied with an evaluation resolution of a type paired with a coercion. The premises under the $\forall\, n$: roughly simulate the $\Gamma \vdash \overline{q} : \overline{\chi}$ judgment.

Unlike in prior proofs of the consistency of versions of System FC, when $\tau_1 \rightsquigarrow \tau_2$, there must be precisely one fewer type family application in τ_2 than in τ_1. This fact is borne of the use of evaluation assumptions $\overline{\chi}$ to model type family applications in the right-hand side of a type family equation instead of using type families there directly. It leads to this critical lemma:

Lemma 6.8 (Termination). *For all types τ, there exists a type σ such that $\tau \rightsquigarrow^* \sigma$ and σ cannot reduce.*

The fact that the reduction relation terminates means that we can use Newman's Lemma to prove confluence via local confluence, a necessary precursor to the proof of the completeness of the rewrite relation (Lemma 6.6):

Lemma 6.9 (Local confluence). *If $\tau_1 \leftsquigarrow \tau_0 \rightsquigarrow \tau_2$, then there exists τ_3 such that $\tau_1 \rightsquigarrow^* \tau_3 \leftsquigarrow^* \tau_2$.*

Lemma 6.10 (Confluence). *If $\tau_1 \leftsquigarrow^* \tau_0 \rightsquigarrow^* \tau_2$, then there exists τ_3 such that $\tau_1 \rightsquigarrow^* \tau_3 \leftsquigarrow^* \tau_2$.*

Eisenberg et al. [2014] also prove confluence via local confluence, but that proof must assume termination. The formulation here allows us to prove termination instead of assume it. The local confluence proof in the current work is also a simplification over the previous proof, as the location of occurrences of type family applications is restricted.

Conclusion. By using evaluation assumptions in our treatment of type families, we can easily prove the termination of type reduction and simplify the proof of confluence. The intricate definition of apartness from Eisenberg et al. [2014] is gone, as well. In short, our approach leads to a substantial simplification to the metatheory of type families.

7 PRACTICALITIES

We believe that constrained type families provide significant benefits compared to the previous approach to type families, with its underlying, implicit assumption of totality. As we are changing the type system of a language, not all current Haskell code is immediately supported in our design. For example, existing code may make use of non-associated open type families, or use incomplete type families as if they were total. In this section, we describe an approach for inferring constrained type families, and the corresponding constraints, from current declarations and uses of indexed type families. This is intended to allow a transition from current practice to the explicit use of constrained type families.

7.1 Inferring Type Family Constraints

We first consider uses of type families in types. Here, our approach is to read the well-formedness restrictions for constrained type families (§4.2) as inference rules rather than as a checking relation. Because the typing rules are syntax directed, given type environments Σ and Γ (known in advance), and a type τ, we can follow the rules to generate a P such that $P \mid \Gamma \vdash \tau$ type, if such a P exists. While there is not necessarily a unique P such that the derivation exists, it is easy to pick a minimal one such that it does. (In essence, we view the well-formedness rules as an attribute grammar, in which Σ, Γ and τ are given, and P is synthesized.) Then, we interpret each qualified type σ in context Γ in the program as instead denoting the type $P \Rightarrow \sigma$ where P is the minimal set of additional constraints such that $P \mid \Gamma \vdash \sigma$ type. Note that some programs may still fail to type check under this approach, if they explicitly make use of undefined type family applications. However, we view this

as an acceptable trade-off, as those programs arguably already contained (admittedly unreported) type errors.

7.2 Making Associations

We must also interpret top-level type family syntax in terms of constrained type families. Type family declarations themselves can be straightforwardly interpreted as declarations of constrained type families; for example, the declaration

type family $F\ t\ u :: \star$

would be interpreted as

class $CF\ t\ u$ **where**
  type $F\ t\ u :: \star$

where any other kind restrictions in the original declaration of F can be transferred straightforwardly to the declaration of CF. Connecting F to the compiler-generated CF would be a new special form (**class** F), entirely equivalent to CF.

Instance declarations are more interesting. For example, consider the instance declaration

type instance $F\ Int\ (Maybe\ t) = G\ Int\ t$

where we assume that G is a binary type family. We could not simply interpret this as the instance declaration

instance $CF\ Int\ (Maybe\ t)$ **where**
  type $F\ Int\ (Maybe\ t) = G\ Int\ t$

as the use of type family G lacks a suitable guarding constraint. Again, however, we can rely on interpreting the well-formedness rules for types to infer the necessary constraints. In this case, we would interpret the type instance as denoting the instance declaration

instance $P \Rightarrow CF\ Int\ (Maybe\ t)$ **where**
  type $F\ Int\ (Maybe\ t) = G\ Int\ t$

where P is the minimal set of constraints such that $P \mid t \vdash G\ Int\ t$ type holds. Again, so long as the original type instance declaration did not rely on undefined type family applications, the resulting instance declaration will be well-formed.

Finally, we turn to closed type families. Given a closed type family declaration, we initially check its totality (§5.3). If it is not total, we can then interpret it as a constrained closed type family, following the same approach as for open type families. For example, consider the following closed type family declaration:

type family $F\ t :: \star$ **where**
  $F\ (Maybe\ Int) = Bool$
  $F\ (Maybe\ t) \quad = G\ t$

This declaration is clearly not total. We would interpret this as a closed type family declaration:

class $CF\ t$ **where**
  type $F\ t :: \star$

  instance $CF\ (Maybe\ Int)$ **where**
    type $F\ (Maybe\ Int) = Bool$

  instance $P \Rightarrow CF\ (Maybe\ t)$ **where**
    type $F\ (Maybe\ t) = G\ t$

where P is the minimal set of constraints such that $P \mid t \vdash G\ t$ type holds.

The decision of whether or not to treat a top-level closed type family as constrained is based on the output from the totality checker. We expect users will want to override the compiler's decision in this matter, as any totality checker will be incomplete. We propose the new syntax **type family total** $F\ a$ **where**... to denote that F is intended to be total. Such a declaration would still be checked, but would never be packaged into an enclosing class. (A non-total definition would be reported as an error.) The user could additionally add a pragma {-# TOTAL F #-} to (unsafely) assert that F is total, circumventing the totality checker.

7.3 Runtime Efficiency

Constrained type families may also seem to have a non-trivial efficiency impact. For a simple example, suppose we have a type family F, and consider an existentially-packaged type family application:

data *FPack a* **where**
FPack :: $F\ a \to$ *FPack a*

We might expect an *FPack a* value to contain exactly a value of type $F\ a$. With constrained type families, however, the declaration above would be incorrect; we would need to add a predicate for its constraining class, say C:

data *FPack1 a* **where**
FPack1 :: $C\ a \Rightarrow F\ a \to$ *FPack a*

Now, a value of type *FPack1 a* does not just contain an $F\ a$ value, but must also carry a $C\ a$ dictionary, and uses of *FPack1* will be responsible for constructing, packing, and unpacking these dictionaries. Over sufficiently many uses of *FPack1*, this additional cost could be noticeable.

This efficiency impact can be mitigated, however. This issue can crop up only when we have a value of type $F\ a$ (or other type family application) without an instance of the associated class $C\ a$. But in order for the value of type $F\ a$ to be useful, parametricity tells us that $C\ a$, or some other class with a similar structure to the equations for $F\ a$ must be in scope. Barring this, it must be that $F\ a$ is used as a phantom type. In this case, we would want a "phantom dictionary" for $C\ a$, closely paralleling existing work on proof irrelevance in the dependently-typed programming community (e.g., Barras and Bernardo [2008]; Eisenberg [2016]; Mishra-Linger and Sheard [2008]; Tejiščák and Brady [2015]): the $C\ a$ dictionary essentially represents a proof that will never be examined. While we do not propose here a new solution to this problem, we believe that existing work will be applicable in our case as well.

8 RELATED WORK

The literature on type-level computation and the type system of Haskell is extensive; here, we summarize those parts most relevant to our work.

Type classes and functional dependencies. Partial type-level computation in Haskell was arguably first introduced with Jones's notion of functional dependencies [Jones 2000], which extended type classes with a notion of determined parameters. Indeed our treatment of requiring a class constraint to use type-level computation is inspired by functional dependencies. Functional dependencies build on Jones's theory of improvement for qualified types [Jones 1995], which allows the satisfiability of predicates to influence typing. While Jones's work does not focus on the computational interpretation of functional dependencies, many early examples highlighted it, such as those of Hallgren [2000] or Kiselyov et al. [2004]. Morris and Jones [2010] later introduced instance chains—closely

related to our closed type classes—which combined functional dependencies with explicit notions of negation and alternatives in class instances.

Associated types and type families. Chakravarty et al. [2005] introduced associated type synonyms to provide a more intuitive syntax for type-level computation in Haskell, while also addressing infelicities in the implementations of functional dependencies. Their type system requires that associated types appear only in contexts where their class predicates can be satisfied, matching our approach. However, this requirement was never implemented; GHC's translation to System FC [Sulzmann et al. 2007] showed that the constraint was never used at runtime and was thus deemed superfluous. The class constraints—that is, instance dictionaries [Hall et al. 1996]—are not needed at runtime, in contrast to ordinary class method invocation. Our work does not refute this conclusion, but instead observes that the design of type families and their metatheory are greatly simplified when we require the class constraint.

Recent work has focused on extending the expressiveness of type families themselves. Eisenberg et al. [2014] introduced closed type families, which allow overlapping equations in type family definitions, and Stolarek et al. [2015] introduced injective type families, recovering additional equalities from applications of injective type families. These features, particularly closed type families, have seen significant practical application.

Type classes and modules. An alternative approach to supporting type classes directly is to encode them using modules [Dreyer et al. 2007] or objects [Oliveira et al. 2010]. These approaches replace class predicates with module (or object) arguments, and use mechanisms for canonical values or implicit arguments to simulate instance resolution. Associated types arise naturally in these approaches, as type members of modules, and, as in our approach, can only appear when a suitable module is in scope. However, these approaches require significantly different underlying formalisms, and so it is not apparent how well they would accommodate other extensions, like closed and total type families, or closed classes.

Partial functions in logic. An interesting—and unexpected—parallel to our work arises in Scott's examination of identity and existence in intuitionistic logic [Scott 1979]. Scott considers the cases in which (first-order) terms in a logic may not be defined for arbitrary instantiations of their variables. For example, the term $1/a$ is not defined if a is instantiated to 0. Scott addresses this problem by introducing an additional predicate $E(\cdot)$ to track the existence of first-order terms, which plays a similar role to our requirement that uses of constrained type families mention their defining class predicates.

9 CONCLUSIONS

We have presented a new approach to type-level computation, relevant to any partial language, in which we permit partiality in types by using qualified types to capture their domains of definition. We have applied our approach to indexed type families in Haskell, showing that it aligns naturally with the intuitive semantics of type families and that it resolves many of the complexities in recent developments of type families. We have formalized our approach, and given the first complete proof of consistency for Haskell with closed type families.

Since their introduction, the theory and practice of functional dependencies and type families have diverged, although some uses of functional dependencies continue to seem more expressive than similar uses of type families. Our current work reunites type families with type classes. We believe it should provide an impetus to re-examine the role of functional dependencies. In particular, the use of equality constraints in our core language to prove that type families applications are

well-defined is evocative of the role that class predicates would play in a core calculus based on functional dependencies.

ACKNOWLEDGMENTS

Thanks to the anonymous referees for their helpful feedback. Morris was funded by EPSRC grant number EP/K034413/1.

REFERENCES

Patrick Bahr. 2014. Composing and decomposing data types: a closed type families implementation of data types à la carte. In *Proceedings of the 10th ACM SIGPLAN workshop on Generic programming, WGP 2014, Gothenburg, Sweden, August 31, 2014*, José Pedro Magalhães and Tiark Rompf (Eds.). ACM, 71–82.

Bruno Barras and Bruno Bernardo. 2008. The Implicit Calculus of Constructions as a Programming Language with Dependent Types. In *Foundations of Software Science and Computational Structures (FOSSACS 2008)*, Roberto Amadio (Ed.). Springer Berlin Heidelberg, Budapest, Hungary, 365–379.

Joachim Breitner, Richard A. Eisenberg, Simon Peyton Jones, and Stephanie Weirich. 2016. Safe Zero-cost Coercions for Haskell. *J. Funct. Program.* 26 (2016), 1–79.

Manuel M. T. Chakravarty, Gabriele Keller, and Simon L. Peyton Jones. 2005. Associated type synonyms. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005*, Olivier Danvy and Benjamin C. Pierce (Eds.). ACM, 241–253.

Derek Dreyer, Robert Harper, Manuel M. T. Chakravarty, and Gabriele Keller. 2007. Modular type classes. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '07)*. ACM, Nice, France, 63–70.

Richard A. Eisenberg. 2016. *Dependent Types in Haskell: Theory and Practice.* Ph.D. Dissertation. University of Pennsylvania.

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.

Richard A. Eisenberg, Stephanie Weirich, and Hamidhasan G. Ahmed. 2016. Visible Type Application. In *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016 (Lecture Notes in Computer Science)*, Peter Thiemann (Ed.), Vol. 9632. Springer, 229–254.

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

Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. 1996. Type Classes in Haskell. *ACM Trans. Program. Lang. Syst.* 18, 2 (March 1996).

Thomas Hallgren. 2000. Fun with functional dependencies, or (draft) types as values in static computations in Haskell. http://www.cse.chalmers.se/~hallgren/Papers/wm01.html. (2000).

Mark P. Jones. 1994. *Qualified Types: Theory and Practice.* Cambridge University Press.

Mark P. Jones. 1995. Simplifying and improving qualified types. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture (FPCA '95)*. ACM, La Jolla, California, USA, 160–169.

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.

Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. 2004. Strongly typed heterogeneous collections. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell (Haskell '04)*. ACM Press, Snowbird, Utah, USA, 96–107.

Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. 2001. The size-change principle for program termination. In *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, Chris Hankin and Dave Schmidt (Eds.). ACM, 81–92.

Sam Lindley and J. Garrett Morris. 2016. Embedding session types in Haskell. In *Proceedings of the 9th International Symposium on Haskell, Haskell 2016, Nara, Japan, September 22-23, 2016*, Geoffrey Mainland (Ed.). ACM, 133–145.

Nathan Mishra-Linger and Tim Sheard. 2008. Erasure and Polymorphism in Pure Type Systems. In *Foundations of Software Science and Computational Structures (FoSSaCS)*. Springer.

J. Garrett Morris. 2015. Variations on variants. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell (Haskell '15)*, Ben Lippmeier (Ed.). ACM, Vancouver, BC, 71–81.

J. Garrett Morris and Mark P. Jones. 2010. Instance chains: Type-class programming without overlapping instances. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming (ICFP '10)*. ACM, Baltimore, MD.

Takayuki Muranushi and Richard A. Eisenberg. 2014. Experience report: Type-checking polymorphic units for astrophysics research in Haskell. In *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, Wouter Swierstra (Ed.). ACM, 31–38.

Bruno C. d. S. Oliveira, Adriaan Moors, and Martin Odersky. 2010. Type classes as objects and implicits. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, William R. Cook, Siobhán Clarke, and Martin C. Rinard (Eds.). ACM, 341–360.

Simon Peyton Jones, Mark P. Jones, and Erik Meijer. 1997. Type classes: An exploration of the design space. In *Proceedings of the 1997 workshop on Haskell (Haskell '97)*. Amsterdam, The Netherlands.

Riccardo Pucella and Jesse A. Tov. 2008. Haskell session types with (almost) no class. In *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*. ACM, 25–36.

John C. Reynolds. 1974. Towards a theory of type structure. In *Paris Colloquium on Programming*. Springer-Verlag, 408–423.

Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. 2008. Type checking with open type functions. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming (IFCP '08)*. ACM, Victoria, BC, Canada, 51–62.

Dana Scott. 1979. Identity and existence in intuitionistic logic. In *Applications of Sheaves: Proceedings of the Research Symposium on Applications of Sheaf Theory to Logic, Algebra, and Analysis, Durham, July 9–21, 1977*, Michael Fourman, Christopher Mulvey, and Dana Scott (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 660–696.

Damien Sereni and Neil D. Jones. 2005. Termination Analysis of Higher-Order Functional Programs. In *Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings (Lecture Notes in Computer Science)*, Kwangkeun Yi (Ed.), Vol. 3780. Springer, 281–297.

Jan Stolarek, Simon L. Peyton Jones, and Richard A. Eisenberg. 2015. Injective type families for Haskell. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, Ben Lippmeier (Ed.). ACM, 118–128.

Martin Sulzmann, Manuel M. T. Chakravarty, Simon L. Peyton Jones, and Kevin Donnelly. 2007. System F with type equality coercions. In *Proceedings of TLDI'07: 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Nice, France, January 16, 2007*, François Pottier and George C. Necula (Eds.). ACM, 53–66.

Wouter Swierstra. 2008. Data types à la carte. *JFP* 18, 04 (2008), 423–436.

Matúš Tejiščák and Edwin Brady. 2015. Practical Erasure in Dependently Typed Languages. (2015). http://eb.host.cs.st-andrews.ac.uk/drafts/dtp-erasure-draft.pdf Draft.

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.

Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon L. Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. 2012. Giving Haskell a promotion. In *Proceedings of TLDI 2012: The Seventh ACM SIGPLAN Workshop on Types in Languages Design and Implementation, Philadelphia, PA, USA, Saturday, January 28, 2012*, Benjamin C. Pierce (Ed.). ACM, 53–66.