Constrained Type Families (extended version), preprint

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

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.

ACM Reference format:
J. Garrett Morris and Richard A. Eisenberg. 2017. Constrained Type Families (extended version), preprint. *PACM Progr. Lang.* 1, 1, Article 1 (January 2017), 36 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Indexed type families [3, 22] 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 [5] and injective type families [25], and they have many applications, including encoding units of measure in scientific calculation [17] and extensible variants [1, 15].

Nevertheless, some aspects of type families and their extensions remain counterintuitive. For example, 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. We can demonstrate them using one of the original motivating examples for closed type families. 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]

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

Given this declaration, it is surprising that the type family application $\mathit{Equ}\ a\ [a]$ does not reduce to *False*. The immediate explanation is that closed type families use infinitary unification, and there is an infinite (i.e., non-idempotent) unifying substitution $\{[a]/a\}$ that makes $\mathit{Equ}\ a\ [a]$ match the first equation, and so rewrite to *True*. But this explanation only raises more questions: Haskell does not have infinite types, so why should closed type families rely on infinitary unification? Again, type families play a role. Consider the following:

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

[1] We use here the promoted *Bool* kind, as introduced by Yorgey et al. [30].

2017. 2475-1421/2017/1-ART1 $15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

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 the decision not to rewrite *Equ a* [*a*] to *False*.

The complexity in this example arises not from closed type families per se, but from their interaction with 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 (or lack thereof) in the program. Our use of *Loop* exploits this inconsistency. The type expression *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. Closed type families are not the only place that such actually-partial type families cause problems: 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. We begin with associated type synonyms [3], the original motivation for type families. As in associated types, we insist that type families be defined via type classes. Thus, the domain of a type family is naturally characterized by its corresponding type class predicate. Unlike associated types, we 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 must be guarded by unsatisfiable class predicates, assuring that they cannot be used to violate type safety.

One of the most promising extensions of type families is closed type families, which permits overlapping equations in type family definitions. We introduce *closed type classes*, a parallel feature which is a simplification of Morris and Jones's instance chains [16]. These allow type classes to be defined by ordered, potentially overlapping lists of instances. Closed type classes allow us to characterize the domains of partial closed type families. They also provide a more expressive, modular approach to many uses of overlapping instances, such as those explored by Kiselyov et al. [11].

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 [5]. 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 [16]. Closed type classes enable partial closed type families and increase the expressiveness of type classes, subsuming many uses of overlapping instances [18].
- 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 [5] 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, we believe that our analysis and design extend beyond this one language. Instead, our work is applicable to any partial language that supports type-level computation.

Currently, Haskell is the canonical member of this set of languages, but 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 [3] 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 [22]. 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 [5] 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 [25] can express such cases; duality could be characterized by the following type family:

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

where the $s \to r$ 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 [5, 25] 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 $a \to a$ 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 $\mathit{Loop} \to \mathit{Loop}$. But worse, we expect Haskell terms to have principal types, and since g makes no reference to *Loop*, $\mathit{Loop} \to \mathit{Loop}$ 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 $\mathit{Int} \sim \mathit{Bool}$, 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, $\mathit{Equ}\ \mathit{Int}\ \mathit{Bool}$ is apart from $\mathit{Equ}\ a\ a$, while $\mathit{Equ}\ a\ b$ is not apart from $\mathit{Equ}\ a\ a$. 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 $\mathit{Equ}\ a\ [a]$ and $\mathit{Equ}\ a\ a$: 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 $\mathit{Equ}\ a\ [a]$ and $\mathit{Equ}\ a\ a$ (namely $\{[a]/a\}$) $\mathit{Equ}\ a\ [a]$ 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 $\mathit{Equ}\ a\ [a]$ 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 [18] or instance chains [16].

[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

3.3 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 (as above [13, 19]) 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 proxy arguments to fix type parameters. Injective type families [25] 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* $s = r \mid r \rightarrow s$

This declaration differs from traditional type family declarations in two ways: first, the result is named (r), and second, the annotation $r \rightarrow s$ 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* $a = b \mid b \rightarrow a$
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*] ∼ *ListElems Int*, and then by injectivity, we have that [*ListElems Int*] ∼ *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 are also a significant limitation in expressiveness, especially in comparison with older approaches, such as functional dependencies [10].

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.

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

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 [29]. 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 ⇒ a → 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* ⇒ *a* → *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* ∼ [*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:

```
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).

$$\frac{\alpha \in \Gamma \qquad \vdash P \mid \Gamma \text{ ctx}}{P \mid \Gamma \vdash \alpha \text{ type}}\ \text{ST\_}\textsc{Var} \qquad \frac{P \mid \Gamma, \alpha \vdash \tau \text{ type}}{P \mid \Gamma \vdash \forall\, \alpha.\tau \text{ type}}\ \text{ST\_}\textsc{Forall} \qquad \frac{P, \pi \mid \Gamma \vdash \tau \text{ type}}{P \mid \Gamma \vdash \pi \Rightarrow \tau \text{ type}}\ \text{ST\_}\textsc{Qual}$$

$$\frac{\begin{array}{cc}(H : n) \in \Sigma & \vdash P \mid \Gamma \text{ ctx} \\ \overline{P \mid \Gamma \vdash \tau_i \text{ type}}^{\,i<n} & \end{array}}{P \mid \Gamma \vdash H\,\overline{\tau} \text{ type}}\ \text{ST\_}\textsc{TyCon} \qquad \frac{\begin{array}{cc}(C \Rightarrow F : n) \in \Sigma & \vdash P \mid \Gamma \text{ ctx} \\ \overline{P \mid \Gamma \vdash \tau_i \text{ type}}^{\,i<n} & P \Vdash C\,\overline{\tau}\end{array}}{P \mid \Gamma \vdash F\,\overline{\tau} \text{ type}}\ \text{ST\_}\textsc{Family}$$

Fig. 1. Well-formedness rules for types

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 [8].

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 an 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 Morris and Jones's instance chains [16], 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 allows 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. [11]. We begin by introducing data types to represent heterogeneous lists:

```
data HNil = HNil
data HCons e l = HCons e l
```

For example, the declaration

```
hlst = HCons True (HCons 'c' HNil)
```

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 HNil = [ ]
  instance HOccurs e l ⇒ HOccurs e (HCons e l) where
    hOccurs (HCons e l) = e : hOccurs l
  instance HOccurs e l ⇒ HOccurs e (HCons e’ l) where
    hOccurs (HCons _ 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 [18], 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 encoding of extensible variants [27], 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 $(f \oplus g) \preceq (f \oplus g)$ 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 $(f \preceq g) \preceq (f \oplus g)$ 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. [11]. 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 _ HNil = HNil
  instance HDelete e l ⇒ HDelete e (HCons e l) where
    type HWithout e (HCons e l) = HWithout e l
    hDelete ep (HCons _ 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 (HCons e' l) = HCons 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 *HExists*; again, the final two instances overlap. Like conventional closed type families, the associated type synonym equations are checked in the order in which the 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:

[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.

```
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 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 [12, 24]. 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:

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

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 [5, §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 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 [26], a variant of System F [6, 20] 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. [5], 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 the appendix.

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 3–6. 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 6). The features in this system beyond those in System F are all driven by these coercions.

Critically, CFC allows *coercion assumptions*—or abstractions over coercions. This feature allows a function to assume an equality proposition ϕ relating two types. The feature can be seen in the rules T_Qual (which allows a

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

[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 are applied postfix; e.g., $\tau[\theta]$
- Substitutions may be composed: $\theta = \theta_1 \circ \theta_2$
- $F : n$ stands for either $F :_{\not\top} n$ or $F :_{\top} n$
- Free variables of constructs are denoted $fv(\cdot)$
- $tvs(\overline{\chi})$ extracts the bound type variables from $\overline{\chi}$
- 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	$::=$	$x \mid K \mid \lambda x : \tau.e \mid e_1\,e_2 \mid \Lambda\alpha.e \mid e\,\tau \mid \lambda c : \phi.e \mid e\,\gamma \mid e \triangleright \gamma \mid \mathbf{assume}\,\chi\,\mathbf{in}\,e$	expressions
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	$::=$	$\emptyset \mid \Sigma, F :_{\not\top} n \mid \Sigma, F :_{\top} n \mid \Sigma, \xi : \overline{E}$	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

type to have the shape $\phi \Rightarrow \tau$, Figure 3) and E_CLam (which is the typing rule for a λ-abstraction over a coercion, Figure 6). Though this language omits datatypes, generalized algebraic datatypes (GADTs) can be encoded using coercion abstractions.

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

Perhaps unexpectedly, classes, too, are omitted. Instead, CFC differentiates between pretypes (any production of metavariable τ) and types (as validated by $\Gamma \vdash \tau$ type, Figure 3); proper types may not mention type families at all. The only place type families may appear is in a proposition ϕ. Examine the judgment $\Gamma \vdash \phi$ prop (Figure 3). Its rule P_Types allows the proposition to be between 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 $insert :: \forall a\ c.\ Elem\ c \sim a \Rightarrow a \to c \to c$ instead of the more typical $insert :: \forall\, c.\ Collects\ c \Rightarrow Elem\ c \to c \to c$. In effect, the type family equality assumption $Elem\ c \sim a$ takes the place of the class constraint $Collects\ c$: both assert that $Elem\ c$ can evaluate to a proper (type family-free) type.

Before describing the novelty of CFC, we take a quick tour of the grounds.

6.2 System CFC Primer

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

$$\boxed{\Gamma \vdash \tau\ \text{type}} \quad \text{Type validity}$$

$$\frac{H : n \qquad \overline{\Gamma \vdash \tau_i\ \text{type}}^{\,i<n} \qquad \vdash \Gamma\ \text{ctx}}{\Gamma \vdash H\,\overline{\tau}\ \text{type}}\ \textsc{T\_TyCon}
\qquad
\frac{\Gamma \vdash \tau_1\ \text{type} \qquad \Gamma \vdash \tau_2\ \text{type}}{\Gamma \vdash \tau_1 \to \tau_2\ \text{type}}\ \textsc{T\_Arrow}
\qquad
\frac{\alpha \in \Gamma \qquad \vdash \Gamma\ \text{ctx}}{\Gamma \vdash \alpha\ \text{type}}\ \textsc{T\_Var}$$

$$\frac{\Gamma, \alpha \vdash \tau\ \text{type}}{\Gamma \vdash \forall\,\alpha.\tau\ \text{type}}\ \textsc{T\_Forall}
\qquad
\frac{\Gamma \vdash \phi\ \text{prop} \qquad \Gamma \vdash \tau\ \text{type}}{\Gamma \vdash \phi \Rightarrow \tau\ \text{type}}\ \textsc{T\_Qual}$$

$$\boxed{\Gamma \vdash \phi\ \text{prop}} \quad \text{Proposition validity}$$

$$\frac{\Gamma \vdash \tau_1\ \text{type} \qquad \Gamma \vdash \tau_2\ \text{type}}{\Gamma \vdash \tau_1 \sim \tau_2\ \text{prop}}\ \textsc{P\_Types}
\qquad
\frac{F\ :\ n \in \Sigma \qquad \overline{\Gamma \vdash \tau_i\ \text{type}}^{\,i<n} \qquad \Gamma \vdash \sigma\ \text{type}}{\Gamma \vdash F\,\overline{\tau} \sim \sigma\ \text{prop}}\ \textsc{P\_Family}$$

$$\boxed{\vdash \Gamma\ \text{ctx}} \quad \text{Context validity}$$

$$\frac{}{\vdash \emptyset\ \text{ctx}}\ \textsc{G\_Nil}
\qquad
\frac{\vdash \Gamma\ \text{ctx} \qquad \alpha \mathbin{\#} \Gamma}{\vdash \Gamma, \alpha\ \text{ctx}}\ \textsc{G\_TyVar}
\qquad
\frac{\Gamma \vdash \phi\ \text{prop} \qquad c \mathbin{\#} \Gamma}{\vdash \Gamma, c{:}\phi\ \text{ctx}}\ \textsc{G\_CoVar}
\qquad
\frac{\Gamma \vdash \tau\ \text{type} \qquad x \mathbin{\#} \Gamma}{\vdash \Gamma, x{:}\tau\ \text{ctx}}\ \textsc{G\_Var}$$

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

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

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

$$\frac{\Gamma, \alpha \vdash \gamma : \tau_1 \sim \tau_2}{\Gamma \vdash \forall\,\alpha.\gamma : (\forall\,\alpha.\tau_1) \sim (\forall\,\alpha.\tau_2)}\ \textsc{C\_Forall}
\qquad
\frac{\Gamma \vdash \gamma_1 : \tau_1 \sim \sigma_1 \qquad \Gamma \vdash \gamma_2 : \tau_2 \sim \sigma_2 \qquad \Gamma \vdash \gamma_3 : \tau_3 \sim \sigma_3}{\Gamma \vdash \gamma_1 \sim \gamma_2 \Rightarrow \gamma_3 : (\tau_1 \sim \tau_2 \Rightarrow \tau_3) \sim (\sigma_1 \sim \sigma_2 \Rightarrow \sigma_3)}\ \textsc{C\_Qual}$$

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

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

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

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

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

Fig. 3. Type-level validity judgments

$$\boxed{\vdash \Sigma \ \mathsf{ok}} \quad \text{Signature validity}$$

$$\frac{}{\vdash \emptyset \ \mathsf{ok}}\ \textsc{D\_Nil} \qquad \frac{\vdash \Sigma \ \mathsf{ok}}{\vdash \Sigma, F :_{\mathcal{T}} n \ \mathsf{ok}}\ \textsc{D\_Partial} \qquad \frac{\vdash \Sigma \ \mathsf{ok}}{\vdash \Sigma, F :_{\top} n \ \mathsf{ok}}\ \textsc{D\_Total}$$

$$\frac{\begin{array}{l} F : n \in \Sigma \qquad \vdash \Sigma \ \mathsf{ok} \\ \forall\, i : \\ \quad E_i \ = \ \forall \overline{\alpha}\, \overline{\chi}. F\, \overline{\tau} \sim \tau_0 \\ \quad \overline{\overline{\alpha} \vdash \tau_j \ \mathsf{type}}^{\,j \in 1..n} \\ \quad \overline{\alpha}, \mathit{tvs}(\overline{\chi}) \vdash \tau_0 \ \mathsf{type} \\ \quad \overline{\alpha} \vdash \overline{\chi} \ \mathsf{assumps} \end{array}}{\vdash \Sigma, \xi : \overline{E} \ \mathsf{ok}}\ \textsc{D\_Axiom}$$

$$\boxed{\Gamma \vdash \overline{\chi} \ \mathsf{assumps}} \quad \text{Evaluation assumptions validity}$$

$$\frac{}{\Gamma \vdash \emptyset \ \mathsf{assumps}}\ \textsc{X\_Nil}$$

$$\frac{\begin{array}{l} F : n \in \Sigma \\ \overline{\Gamma \vdash \tau_i \ \mathsf{type}}^{\,i \in 1..n} \\ \Gamma, \alpha \vdash \overline{\chi} \ \mathsf{assumps} \end{array}}{\Gamma \vdash (\alpha | c : F\, \overline{\tau} \sim \alpha), \overline{\chi} \ \mathsf{assumps}}\ \textsc{X\_Cons}$$

Fig. 4. Signature validity

$$\boxed{\mathsf{no\_conflict}(\overline{E}, i, \overline{\tau}, j)} \quad \text{Check for equation conflicts}$$

$$\frac{\begin{array}{l} E_i \ = \ \forall \overline{\alpha_1}\, \overline{\chi}_1. F\, \overline{\tau}_1 \sim \tau_{01} \\ E_j \ = \ \forall \overline{\alpha_2}\, \overline{\chi}_2. F\, \overline{\tau}_2 \sim \tau_{02} \\ \mathsf{unify}(\overline{\tau}_2;\ \overline{\tau}_1[\overline{\sigma}/\overline{\alpha}_1]) \ = \ \mathsf{Nothing} \end{array}}{\mathsf{no\_conflict}(\overline{E}, i, \overline{\sigma}, j)}\ \textsc{NC\_Apart}$$

$$\frac{\mathsf{compat}(E_i, E_j)}{\mathsf{no\_conflict}(\overline{E}, i, \overline{\sigma}, j)}\ \textsc{NC\_Compatible}$$

$$\boxed{\mathsf{compat}(E_1, E_2)} \quad \text{Equation compatibility}$$

$$\frac{\begin{array}{l} E_1 \ = \ \forall \overline{\alpha_1}\, \overline{\chi}_1. F\, \overline{\tau}_1 \sim \tau_{01} \\ E_2 \ = \ \forall \overline{\alpha_2}\, \overline{\chi}_2. F\, \overline{\tau}_2 \sim \tau_{02} \\ \mathsf{unify}(\overline{\tau}_1;\ \overline{\tau}_2) \ = \ \mathsf{Just}\, \theta \\ \tau_{01}[\theta \circ \mathit{subst}(\overline{\chi}_1)] \ = \ \tau_{02}[\theta \circ \mathit{subst}(\overline{\chi}_2)] \end{array}}{\mathsf{compat}(E_1, E_2)}\ \textsc{Co\_Coinc}$$

$$\frac{\begin{array}{l} E_1 \ = \ \forall \overline{\alpha_1}\, \overline{\chi}_1. F\, \overline{\tau}_1 \sim \tau_{01} \\ E_2 \ = \ \forall \overline{\alpha_2}\, \overline{\chi}_2. F\, \overline{\tau}_2 \sim \tau_{02} \\ \mathsf{unify}(\overline{\tau}_1;\ \overline{\tau}_2) \ = \ \mathsf{Nothing} \end{array}}{\mathsf{compat}(E_1, E_2)}\ \textsc{Co\_Distinct}$$

Fig. 5. Closed type family auxiliary judgments

ϕ, and $F\,\overline{\tau}$ is a fully applied type family F. This last element in the grammar may be surprising, as we have just said that types do *not* contain type family applications. Indeed they do not, but this is not a syntactic restriction in our formulation; instead it is guaranteed by the judgment $\Gamma \vdash \tau$ type (Figure 3), which checks that a type τ is well scoped and mentions no type families. Putting $F\,\overline{\tau}$ in the grammar of types simplifies, for example, the coercion judgment $\Gamma \vdash \gamma : \phi$.

The grammar for propositions ϕ, checked by the judgment $\Gamma \vdash \phi$ prop (Figure 3), contains only equality assumptions. Well-formed ϕs can relate two types, or a type family application to a type.

Expressions e, checked by the judgment $\Gamma \vdash e : \tau$ (Figure 6), are also fairly routine. 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, as introduced above. The grammar for expressions also has a form of cast $e \triangleright \gamma$ as explained above. Finally, it contains a novel form **assume** χ **in** e used in our account of total type families (§6.4).

The small-step operational semantics (Figure 6) 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** (§6.4). Values in CFC are unsurprisingly constants and abstractions.

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

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

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

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

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

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

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

$$\frac{}{(\lambda x : \tau . e_1)\, e_2 \longrightarrow e_1[e_2/x]}\ \textsc{S\_Beta} \qquad \frac{}{(\Lambda \alpha . e)\, \tau \longrightarrow e[\tau/\alpha]}\ \textsc{S\_TBeta} \qquad \frac{}{(\lambda c : \phi . e)\, \gamma \longrightarrow e[\gamma/c]}\ \textsc{S\_CBeta}$$

$$\frac{\begin{array}{rcl} v & = & \lambda x : \tau . e_0 \\ \gamma_1 & = & \mathbf{sym}\,(\mathbf{nth}_0\ \gamma) \\ \gamma_2 & = & \mathbf{nth}_1\ \gamma \end{array}}{(v \triangleright \gamma)\ e \longrightarrow v\ (e \triangleright \gamma_1) \triangleright \gamma_2}\ \textsc{S\_Push} \qquad \frac{\begin{array}{rcl} v & = & \Lambda \alpha . e \\ \gamma' & = & \gamma @ \tau \end{array}}{(v \triangleright \gamma)\ \tau \longrightarrow v\ \tau \triangleright \gamma'}\ \textsc{S\_TPush} \qquad \frac{\begin{array}{rcl} v & = & \lambda c : \phi . e_0 \\ \eta_0 & = & \mathbf{nth}_0\ \eta \\ \eta_1 & = & \mathbf{sym}\,(\mathbf{nth}_1\ \eta) \\ \eta_2 & = & \mathbf{nth}_2\ \eta \end{array}}{(v \triangleright \eta)\ \gamma \longrightarrow v\ (\eta_0 \fatsemi \gamma \fatsemi \eta_1) \triangleright \eta_2}\ \textsc{S\_CPush}$$

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

Fig. 6. Expression judgments

Of the main productions in the grammar, we are left with coercions γ, checked by the judgment $\Gamma \vdash \gamma : \phi$ (Figure 3). A coercion is a witness of type equality; thus, the rules for coercion formation determine the equality relation underlying the type system. The critical property of this relation is *consistency*—that we can never prove, for example, that *Int* equals *Bool*. We return to consistency and our proof thereof later in this section (§6.5). 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 ($\mathbf{sym}\,\gamma$), 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.

6.3 Type Family Axioms and Signatures

Following prior work on System FC (initially that of Sulzmann et al. [26]), we use *axioms* ξ to witness type family reductions. That is, if there is an equation **type** $F\ \mathit{Int} = \mathit{Bool}$ in scope, then we have an axiom ξ that proves $F\ \mathit{Int} \sim \mathit{Bool}$. An expression can then use this axiom to cast an expression of type Bool to one of type $F\ \mathit{Int}$.

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: $F :_{\not\top} n$ declares a *partial* 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 (§6.4).

An axiom ξ is classified by a list of equations $\overline{E}$, where each equation has the form $\forall\,\overline{\alpha}\,\overline{\chi}.F\,\overline{\tau} \sim \tau_0$. In these equations, 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. We need a *list* of equations to support closed type families, with potentially overlapping equations; this echoes the treatment in the original work on closed type families [5]. 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.

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 (§6.5).

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 4.

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, 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. (This notation is used in S_Resolve, §6.4.)

We can now explain C_Axiom:

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

For a coercion $\xi_i\,\overline{\sigma}\,\overline{q}$, the rule looks up the axiom in the signature, checks to make sure the $\overline{\sigma}$ are proper (type family-free) types, that the $\overline{q}$ satisfy the assumptions $\overline{\chi}$ (using the auxiliary judgment $\Gamma \vdash \overline{q} : \overline{\chi}$, Figure 3), and that there is no previous equation in the axiom that might have applied. This last 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 5.[7] 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.

6.4 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:

$$\begin{array}{l} \mathit{append} :: \mathit{Vec}\ a\ m \to \mathit{Vec}\ a\ n \to \mathit{Vec}\ a\ (\mathit{Plus}\ m\ n) \\ \mathit{append}\ \mathit{Nil}\qquad\qquad\ \mathit{ys} = \mathit{ys} \\ \mathit{append}\ (\mathit{Cons}\ x\ \mathit{xs})\ \mathit{ys} = \mathit{Cons}\ x\ (\mathit{append}\ \mathit{xs}\ \mathit{ys}) \end{array}$$

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

$$\mathit{append} :: \mathit{Plus}\ m\ n\ \sim\ p \Rightarrow \mathit{Vec}\ a\ m \to \mathit{Vec}\ a\ n \to \mathit{Vec}\ a\ p$$

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

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:

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

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:

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

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* $\overline{\emptyset \vdash \tau_i\ \mathsf{type}}$, *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.

[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. [5], 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.

$\boxed{\tau_1 \rightsquigarrow_\top \tau_2}$ Type family application reduction

$$\frac{\begin{array}{llc} \xi : \overline{E} \in \Sigma & E_i \ = \ \forall \overline{\alpha}\,\overline{\chi}.F\,\overline{\sigma} \sim \sigma_0 & \forall\, n : \chi_n \ = \ (\alpha' | c' : F'\,\overline{\sigma}' \sim \alpha') \\ \overline{\tau} \ = \ \overline{\sigma}[\overline{\rho}/\overline{\alpha}] & \tau' \ = \ \sigma_0[\overline{\rho}/\overline{\alpha}, \overline{\rho}'/tvs(\overline{\chi})] & \emptyset \vdash \rho'_n \text{ type} \\ \overline{\emptyset \vdash \rho_k \text{ type}}^{k} & \forall\, j < i, \text{no\_conflict}(\overline{E}, i, \overline{\rho}, j) & \theta_n \ = \ \overline{\rho}/\overline{\alpha},\ \overline{\rho'_m/tv(\chi_m)}^{m \in 1..n-1} \\ & & F'\,\overline{\sigma}'[\theta_n] \rightsquigarrow_\top \rho'_n \end{array}}{F\,\overline{\tau} \rightsquigarrow_\top \tau'}\ \textsc{RTop}$$

$\boxed{\tau_1 \rightsquigarrow \tau_2}$ Type reduction

$$\frac{F\,\overline{\tau} \rightsquigarrow_\top \tau'}{C[F\,\overline{\tau}] \rightsquigarrow C[\tau']}\ \textsc{Red}$$

Fig. 7. Non-deterministic type reduction

6.5 Metatheory: Consistency of Equality

System CFC admits the usual preservation and progress theorems; proofs are in the appendix.

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.

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:

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}\ \textsc{E\_App}$$

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 gives us this critical fact:

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 $\tau_1 \ = \ \tau_2$.*

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.

Lemma 6.5 (Type regularity). *If $\Gamma \vdash e : \tau$, then $\Gamma \vdash \tau$ type.*

6.5.1 *The route to consistency.*

Broadly speaking, we prove consistency in the same manner as in previous work.[8] 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.

6.5.2 *Type reduction relation.*

The type reduction relation $\rightsquigarrow$ is captured by the judgments in Figure 7. 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 rule matches up with C_Axiom. The

[8] The best point of comparison is with Eisenberg et al. [5], as that proof considers closed type families, as does ours.

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. [5] also prove confluence via local confluence, but that proof must assume termination. The formulation here allows us to prove this property. 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. [5] 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. That is, we can interpret the rules of $P \mid \Gamma \vdash \tau$ type as an attribute grammar, in which Γ and τ are inherited while P is synthesized. While there is not necessarily a unique P such that a derivation $P \mid \Gamma \vdash \tau$ type holds, it is easy to pick a minimal P such that it does. 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 :: ★
```

would be interpreted as

```
class CF t u where
  type F t u :: ★
```

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 ⇒ 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 :: ★where
  F (Maybe Int) = Bool
  F (Maybe t)   = 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 :: ★

  instance CF (Maybe Int) where
    type F (Maybe Int) = Bool

  instance P ⇒ 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 → 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 ⇒ F a → 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., [2, 4, 14, 28]): 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 [10], 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 [9], 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 [7] or Kiselyov et al. [11]. Morris and Jones [16] 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. [3] 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. Sulzmann et al. [26] presented a variant of System F [6, 21] extended with first-class equality coercions, and showed that it was sufficient to expression associated types (among other type system features). Schrijvers et al. [22] introduced type families as a distinct source-language feature of Haskell and show how associated types reduce to type families.

Recent work has focused on extending the expressiveness of type families themselves. Eisenberg et al. [5] introduced closed type families, which allow overlapping equations in type family definitions, and Stolarek et al [25] introduced injective type families, recovering additional equalities from applications of injective type families. These features, particularly closed type families, have seen significant practical application.

Partial functions in logic. An interesting—and unexpected—parallel to our work arises in Scott's examination of identity and existence in intuitionistic logic [23]. 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.

REFERENCES

[1] P. Bahr. Composing and decomposing data types: a closed type families implementation of data types à la carte. In J. P. Magalhães and T. Rompf, editors, *Proceedings of the 10th ACM SIGPLAN workshop on Generic programming, WGP 2014, Gothenburg, Sweden, August 31, 2014*, pages 71–82. ACM, 2014.

[2] B. Barras and B. Bernardo. The implicit calculus of constructions as a programming language with dependent types. In R. Amadio, editor, *Foundations of Software Science and Computational Structures*, FOSSACS 2008, pages 365–379, Budapest, Hungary, 2008. Springer Berlin Heidelberg.

[3] M. M. T. Chakravarty, G. Keller, and S. L. Peyton Jones. Associated type synonyms. In O. Danvy and B. C. Pierce, editors, *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005*, pages 241–253. ACM, 2005.

[4] R. A. Eisenberg. *Dependent Types in Haskell: Theory and Practice.* PhD thesis, University of Pennsylvania, September 2016.

[5] R. A. Eisenberg, D. Vytiniotis, S. Peyton Jones, and S. Weirich. Closed type families with overlapping equations. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 671–683, San Diego, California, USA, 2014. ACM.

[6] J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types.* Cambridge University Press, New York, NY, USA, 1989.

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

[8] M. P. Jones. *Qualified Types: Theory and Practice.* Cambridge University Press, 1994.

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

[10] M. P. Jones. Type classes with functional dependencies. In *Proceedings of the 9th European Symposium on Programming Languages and Systems*, ESOP '00, pages 230–244, Berlin, Germany, 2000. Springer-Verlag.

[11] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, Haskell '04, pages 96–107, Snowbird, Utah, USA, 2004. ACM Press.

[12] C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In C. Hankin and D. Schmidt, editors, *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, pages 81–92. ACM, 2001.

[13] S. Lindley and J. G. Morris. Embedding session types in Haskell. In G. Mainland, editor, *Proceedings of the 9th International Symposium on Haskell, Haskell 2016, Nara, Japan, September 22-23, 2016*, pages 133–145. ACM, 2016.

[14] N. Mishra-Linger and T. Sheard. Erasure and polymorphism in pure type systems. In *Foundations of Software Science and Computational Structures (FoSSaCS).* Springer, 2008.

[15] J. G. Morris. Variations on variants. In B. Lippmeier, editor, *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell*, Haskell '15, pages 71–81, Vancouver, BC, 2015. ACM.

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

[17] T. Muranushi and R. A. Eisenberg. Experience report: Type-checking polymorphic units for astrophysics research in Haskell. In W. Swierstra, editor, *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, pages 31–38. ACM, 2014.

[18] S. Peyton Jones, M. P. Jones, and E. Meijer. Type classes: An exploration of the design space. In *Proceedings of the 1997 workshop on Haskell*, Haskell '97, Amsterdam, The Netherlands, 1997.

[19] R. Pucella and J. A. Tov. 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*, pages 25–36. ACM, 2008.

[20] J. C. Reynolds. Towards a theory of type structure. In *Paris Colloquium on Programming*, pages 408–423. Springer-Verlag, 1974.

[21] J. C. Reynolds. Syntactic control of interference. In A. V. Aho, S. N. Zilles, and T. G. Szymanski, editors, *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, pages 39–46. ACM Press, 1978.

[22] T. Schrijvers, S. Peyton Jones, M. Chakravarty, and M. Sulzmann. Type checking with open type functions. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, IFCP '08, pages 51–62, Victoria, BC, Canada, 2008. ACM.

[23] D. Scott. Identity and existence in intuitionistic logic. In M. Fourman, C. Mulvey, and D. Scott, editors, *Applications of Sheaves: Proceedings of the Research Symposium on Applications of Sheaf Theory to Logic, Algebra, and Analysis, Durham, July 9–21, 1977*, pages 660–696. Springer Berlin Heidelberg, Berlin, Heidelberg, 1979.

[24] D. Sereni and N. D. Jones. Termination analysis of higher-order functional programs. In K. Yi, editor, *Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings*, volume 3780 of *Lecture Notes in Computer Science*, pages 281–297. Springer, 2005.

[25] J. Stolarek, S. L. Peyton Jones, and R. A. Eisenberg. Injective type families for Haskell. In B. Lippmeier, editor, *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, pages 118–128. ACM, 2015.

[26] M. Sulzmann, M. M. T. Chakravarty, S. L. Peyton Jones, and K. Donnelly. System F with type equality coercions. In F. Pottier and G. C. Necula, editors, *Proceedings of TLDI'07: 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Nice, France, January 16, 2007*, pages 53–66. ACM, 2007.

[27] W. Swierstra. Data types à la carte. *JFP*, 18(04):423–436, 2008.

[28] M. Tejiščák and E. Brady. Practical erasure in dependently typed languages. Draft, 2015. URL http://eb.host.cs.st-andrews.ac.uk/drafts/dtp-erasure-draft.pdf.

[29] P. Wadler and S. Blott. 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, pages 60–76, Austin, Texas, USA, 1989. ACM.

[30] B. A. Yorgey, S. Weirich, J. Cretin, S. L. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving Haskell a promotion. In B. C. Pierce, editor, *Proceedings of TLDI 2012: The Seventh ACM SIGPLAN Workshop on Types in Languages Design and Implementation, Philadelphia, PA, USA, Saturday, January 28, 2012*, pages 53–66. ACM, 2012.

A PROOFS

A.1 Evaluation assumptions

Definition A.1 (Evaluation assumption substitution). Define the substitution q/χ to mean $\sigma/\alpha, \gamma/c$, where $q = (\sigma|\gamma)$ and $\chi = (\alpha|c : F\,\overline{\tau} \sim \alpha)$.

Definition A.2 (Evaluation assumption scoping). Define $fv(\overline{\chi})$ inductively as follows:

$$\begin{aligned} fv(\emptyset) &= \emptyset \\ fv((\alpha|c : F\,\overline{\tau} \sim \alpha), \overline{\chi}) &= fv(\overline{\tau}) \cup (fv(\overline{\chi}) \backslash \{\alpha\}) \end{aligned}$$

Definition A.3 (Evaluation assumption substitution). Define $subst(\overline{\chi})$ inductively as follows:

$$\begin{aligned} subst(\emptyset) &= \emptyset \\ subst(\overline{\chi}, (\alpha|c : F\,\overline{\tau} \sim \alpha)) &= subst(\overline{\chi}) \circ (F\,\overline{\tau})/\alpha \end{aligned}$$

A.2 Assumptions about environment

Assumption A.4 (Declarations). *We assume that if decl* $\in \Sigma$, *then* $\vdash$ *decl* ok.

Assumption A.5 (Good signature). *We assume that our implicit signature Σ conforms to the following rules (adapted from Eisenberg et al. [5, 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 in 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* $\mathsf{length}(\overline{E}) > 1$ *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* $\mathsf{compat}(E_1, E_2)$. *That is, equations for open type families are all compatible.*

Remark. The conditions above are identical to the conditions in Eisenberg et al. [5, 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 $\overline{\Gamma \vdash \tau_i \ \mathsf{type}}^{\,i<n}$ premise to D_Axiom describing the validity of axioms in the signature. Type family applications are not types.

A.3 Unification

Property A.6 (unify correct). *(Eisenberg et al. [5, Property 11]) If there exists a substitution θ such that $\overline{\sigma}[\theta] = \overline{\tau}[\theta]$, then* $\mathsf{unify}(\overline{\sigma}; \overline{\tau})$ *succeeds. If* $\mathsf{unify}(\overline{\sigma}; \overline{\tau}) = \mathsf{Just}\,\theta$, *then θ is a most general unifier of $\overline{\sigma}$ and $\overline{\tau}$.*

Lemma A.7 (Apartness is stable under type substitution). *(Eisenberg et al. [5, Property 12]) If* $\mathsf{unify}(\overline{\tau}_1; \overline{\tau}_2) = \mathsf{Nothing}$, *then for all substitutions θ,* $\mathsf{unify}(\overline{\tau}_1; \overline{\tau}_2[\theta]) = \mathsf{Nothing}$.

Proof. We prove the contrapositive: that if $\mathsf{unify}(\overline{\tau}_1; \overline{\tau}_2[\theta]) = \mathsf{Just}\,\theta'$, then there exists θ'' such that $\mathsf{unify}(\overline{\tau}_1; \overline{\tau}_2) = \mathsf{Just}\,\theta''$. By Property A.6, we have $\overline{\tau}_1[\theta'] = \overline{\tau}_2[\theta' \circ \theta]$. Since we assume that $fv(\overline{\tau}_1) \cap fv(\overline{\tau}_2) = \emptyset$, we can rewrite this as $\overline{\tau}_1[\theta' \circ \theta] = \overline{\tau}_2[\theta' \circ \theta]$. Thus, $\theta'' = \theta' \circ \theta$ and we are done. □

A.4 Structural properties

Definition A.8 (Subset on contexts). Define $\Gamma \subseteq \Gamma'$ to mean that Γ' contains at least all the bindings in Γ, possibly in a different order.

Lemma A.9 (Weakening/Permutation). *Suppose* $\Gamma \subseteq \Gamma'$ *and* $\vdash \Gamma'$ ctx.

(1) If $\Gamma \vdash \tau$ type, *then* $\Gamma' \vdash \tau$ type.
(2) If $\Gamma \vdash \phi$ prop, *then* $\Gamma' \vdash \phi$ prop.
(3) If $\Gamma \vdash \gamma : \phi$, *then* $\Gamma' \vdash \gamma : \phi$.
(4) If $\Gamma \vdash \overline{q} : \overline{\chi}$, *then* $\Gamma' \vdash \overline{q} : \overline{\chi}$.
(5) If $\Gamma \vdash x : \tau$, *then* $\Gamma' \vdash x : \tau$.

Proof. By straightforward mutual induction, renaming bound variables if necessary to satisfy freshness conditions. □

Lemma A.10 (Context strengthening). *Suppose* $dom(\delta) \notin fv(\Gamma')$.

(1) If $\Gamma, \delta, \Gamma' \vdash \tau$ type, *and* $dom(\delta) \notin fv(\tau)$, *then* $\Gamma, \Gamma' \vdash \tau$ type.
(2) If $\Gamma, \delta, \Gamma' \vdash \phi$ prop *and* $dom(\delta) \notin fv(\phi)$, *then* $\Gamma, \Gamma' \vdash \phi$ prop.
(3) If $\vdash \Gamma, \delta, \Gamma'$ ctx, *then* $\vdash \Gamma, \Gamma'$ ctx.

Proof. Straightforward mutual induction. □

Lemma A.11 (Scoping).

(1) If $\Gamma \vdash \tau$ type, *then* $fv(\tau) \subseteq dom(\Gamma)$.
(2) If $\Gamma \vdash \phi$ prop, *then* $fv(\phi) \subseteq dom(\Gamma)$.
(3) If $\Gamma \vdash \gamma : \phi$, *then* $fv(\gamma) \subseteq dom(\Gamma)$ *and* $fv(\phi) \subseteq dom(\Gamma)$.
(4) If $\Gamma \vdash \overline{q} : \overline{\chi}$, *then* $fv(\overline{q}) \subseteq dom(\Gamma)$ *and* $fv(\overline{\chi}) \subseteq dom(\Gamma)$.
(5) If $\Gamma \vdash e : \tau$, *then* $fv(e) \subseteq dom(\Gamma)$ *and* $fv(\tau) \subseteq dom(\Gamma)$.
(6) If $\vdash \Gamma$ ctx, *then* $fv(\Gamma) = \emptyset$.

Proof. By induction, using a mutual induction between $\Gamma \vdash \tau$ type, $\Gamma \vdash \phi$ prop, followed by the others. We appeal to Assumption A.4 in the C_Axiom case. □

A.5 Regularity I

Lemma A.12 (Context formation). *If* $\vdash \Gamma$ ctx *and* Γ' *is a prefix of* Γ, *then* $\vdash \Gamma'$ ctx.

Proof. Straightforward induction on Γ. □

Lemma A.13 (Context regularity). *If any of*

(1) $\Gamma \vdash \tau$ type, *or*
(2) $\Gamma \vdash \phi$ prop, *or*
(3) $\Gamma \vdash \gamma : \phi$, *or*
(4) $\Gamma \vdash \overline{q} : \overline{\chi}$, *or*
(5) $\Gamma \vdash x : \tau$,

then $\vdash \Gamma$ ctx.

Proof. Straightforward mutual induction on typing judgments, appealing to Lemma A.12 in the cases that bind a new variable. □

A.6 Substitution

Lemma A.14 (Type substitution in no_conflict). *If* $\xi : \overline{E} \in \Sigma$ *and* no_conflict$(\overline{E}, i, \overline{\sigma}, k)$, *then* no_conflict$(\overline{E}, i, \overline{\sigma}[\tau_0/\alpha], k)$.

Proof. By case analysis on no_conflict$(\overline{E}, i, \overline{\sigma}, k)$. We have two cases:

Case NC_Apart:

$$\frac{\begin{array}{l} E_i \;=\; \forall \overline{\alpha}_1\,\overline{\chi}_1 . F\,\overline{\tau}_1 \sim \tau_{01} \\ E_j \;=\; \forall \overline{\alpha}_2\,\overline{\chi}_2 . F\,\overline{\tau}_2 \sim \tau_{02} \\ \mathsf{unify}(\overline{\tau}_2;\ \overline{\tau}_1[\overline{\sigma}/\overline{\alpha}_1]) \;=\; \mathsf{Nothing} \end{array}}{\mathsf{no\_conflict}(\overline{E}, i, \overline{\sigma}, j)}\ \textsc{NC\_Apart}$$

Inversion tells us $\mathsf{apart}(\overline{\tau}_2;\ \overline{\tau}_1[\overline{\sigma}/\overline{\alpha}_1])$. We must show $\mathsf{apart}(\overline{\tau}_2;\ \overline{\tau}_1[\overline{\sigma}[\tau_0/\alpha]/\overline{\alpha}_1])$. From the assumption that equations are well-scoped (Assumption A.4 and D_Axiom), we see that $\alpha \notin fv(\overline{\tau}_1)$ and we can thus rewrite as $\mathsf{apart}(\overline{\tau}_2;\ \overline{\tau}_1[\overline{\sigma}/\overline{\alpha}_1][\tau_0/\alpha])$. We have this by Lemma A.7 and we are thus done.

Case NC_Compatible :

$$\frac{\mathsf{compat}(E_i, E_j)}{\mathsf{no\_conflict}(\overline{E}, i, \overline{\sigma}, j)}\ \textsc{NC\_Compatible}$$

The substitution has no effect on the premise and we are thus done.

□

Lemma A.15 (Type substitution in types). *Assume* $\Gamma \vdash \tau$ type.

(1) If $\Gamma, \alpha, \Gamma' \vdash \sigma$ type, *then* $\Gamma, \Gamma'[\tau/\alpha] \vdash \sigma[\tau/\alpha]$ type.
(2) If $\Gamma, \alpha, \Gamma' \vdash \phi$ prop, *then* $\Gamma, \Gamma'[\tau/\alpha] \vdash \phi[\tau/\alpha]$ prop.
(3) If $\vdash \Gamma, \alpha, \Gamma'$ ctx, *then* $\vdash \Gamma, \Gamma'[\tau/\alpha]$ ctx.

Proof. Straightforward mutual induction, with the usual reasoning in the variable case, appealing to Lemma A.11 to show that a variable bound in Γ cannot be affected by the substitution and to Lemma A.9 to extend the contexts. □

Lemma A.16 (Type substitution in coercions). *Assume* $\Gamma \vdash \tau$ type.

(1) If $\Gamma, \alpha, \Gamma' \vdash \gamma : \phi$, *then* $\Gamma, \Gamma'[\tau/\alpha] \vdash \gamma[\tau/\alpha] : \phi[\tau/\alpha]$.
(2) If $\Gamma, \alpha, \Gamma' \vdash \overline{q} : \overline{\chi}$, *then* $\Gamma, \Gamma'[\tau/\alpha] \vdash \overline{q}[\tau/\alpha] : \overline{\chi}[\tau/\alpha]$.

Proof. By mutual induction, appealing to Lemma A.15, Lemma A.14, and Lemma A.11. □

Lemma A.17 (Type substitution). *If* $\Gamma, \alpha, \Gamma' \vdash e : \sigma$ *and* $\Gamma \vdash \tau$ type, *then* $\Gamma, \Gamma'[\tau/\alpha] \vdash e[\tau/\alpha] : \sigma[\tau/\alpha]$.

Proof. By induction, appealing to Lemma A.15, Lemma A.16, and Lemma A.11. □

Lemma A.18 (Coercion substitution in coercions). *Assume* $\Gamma \vdash \gamma' : \phi'$.

(1) If $\Gamma, c{:}\phi', \Gamma' \vdash \gamma : \phi$, *then* $\Gamma, \Gamma' \vdash \gamma[\gamma'/c] : \phi$.
(2) If $\Gamma, c{:}\phi', \Gamma' \vdash \overline{q} : \overline{\chi}$, *then* $\Gamma, \Gamma' \vdash \overline{q}[\gamma'/c] : \overline{\chi}$.

Proof. By mutual induction, with the usual reasoning in the variable case, appealing to Lemma A.9 and Lemma A.10. □

Lemma A.19 (Coercion substitution). *If* $\Gamma, c{:}\phi, \Gamma' \vdash e : \tau$ *and* $\Gamma \vdash \gamma : \phi$, *then* $\Gamma, \Gamma' \vdash e[\gamma/c] : \tau$.

Proof. By induction, appealing to Lemma A.18, Lemma A.11, and Lemma A.10. □

Lemma A.20 (Resolution substitution). *If* $\Gamma, \alpha, c{:}F\,\overline{\tau} \sim \alpha \vdash e : \sigma$ *and* $\Gamma \vdash q : \chi$, *then* $\Gamma \vdash e[q/\chi] : \sigma[q/\chi]$.

Proof. Corollary of Lemma A.17 and Lemma A.19. □

Lemma A.21 (Substitution). *If* $\Gamma, x{:}\tau', \Gamma' \vdash e : \tau$ *and* $\Gamma \vdash e' : \tau'$, *then* $\Gamma, \Gamma' \vdash e[e'/x] : \tau$.

Proof. By induction, with the usual reasoning in the variable case, appealing to Lemma A.9 and Lemma A.10. □

A.7 Regularity II

LEMMA A.22 (CONTEXT TYPES). *If $x{:}\tau \in \Gamma$ and $\vdash \Gamma$ ctx, then $\Gamma \vdash \tau$ type.*

PROOF. By induction on the structure of Γ, using Lemma A.9 at the end to fix the context in the conclusion. □

LEMMA A.23 (CONTEXT PROPS). *If $c{:}\phi \in \Gamma$ and $\vdash \Gamma$ ctx, then $\Gamma \vdash \phi$ prop.*

PROOF. Similar to previous proof. □

ASSUMPTION A.24 (CONSTANT TYPES). *If $K : H$, then $\emptyset \vdash H$ type.*

LEMMA 6.5 (TYPE REGULARITY). *If $\Gamma \vdash e : \tau$, then $\Gamma \vdash \tau$ type.*

PROOF. By induction on the derivation of $\Gamma \vdash e : \tau$, appealing to Lemma A.22, Assumption A.24, Lemma A.15 (in the E_TAPP case), Lemma A.23 (in the E_CLAM case), and Lemma A.10 (in the E_ASSUME case). Note that we need the $\Gamma \vdash \tau_2$ type premise in the E_CAST case. □

A.8 Preservation

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

Remark. Note that this theorem requires an *empty* context, in contrast to many statements of type preservation. This choice is necessary in order to support the S_RESOLVE rule and its use of $\Downarrow$, available only in an empty context. (See Property 6.1.) If we wanted a statement of type preservation that worked in non-empty contexts, we would need to make $\Downarrow$ partial and then assert in the premise to S_RESOLVE that it succeeds.

PROOF. By induction on the derivation of $\emptyset \vdash e : \tau$.

Case E_VAR: Impossible.
Case E_CONST: Impossible.
Case E_LAM: Impossible.
Case E_APP: We have several cases, depending on how e has stepped:

Case S_APP: By induction.
Case S_BETA: By Lemma A.21.
Case S_PUSH: We adopt the metavariable names from the rule:

$$\dfrac{\begin{array}{rcl} v & = & \lambda x:\tau.e_0 \\ \gamma_1 & = & \mathbf{sym}\,(\mathbf{nth}_0\;\gamma) \\ \gamma_2 & = & \mathbf{nth}_1\;\gamma \end{array}}{(v \triangleright \gamma)\;e \longrightarrow v\,(e \triangleright \gamma_1) \triangleright \gamma_2}\;\textsc{S\_Push}$$

Inversion tells us the following (for some τ', σ, and σ'):

- $\emptyset \vdash \gamma : (\tau \to \tau') \sim (\sigma \to \sigma')$
- $\emptyset \vdash v : \tau \to \tau'$
- $\emptyset \vdash e : \sigma$
- $\emptyset \vdash (v \triangleright \gamma)\;e : \sigma'$
- $\emptyset \vdash \sigma$ type
- $\emptyset \vdash \sigma'$ type

Lemma 6.5 and inversion then gives us:

- $\emptyset \vdash \tau$ type
- $\emptyset \vdash \tau'$ type

We now show that γ_1 and γ_2 are well typed:

$\emptyset \vdash \gamma_1 : \sigma \sim \tau$**:** By C_NTHARROW and C_SYM.

$\emptyset \vdash \gamma_2 : \tau' \sim \sigma'$: By C_NthArrow.

Thus:

- $\emptyset \vdash e \triangleright \gamma_1 : \tau$ (by E_Cast)
- $\emptyset \vdash v\,(e \triangleright \gamma_1) : \tau'$ (by E_App)
- $\emptyset \vdash v\,(e \triangleright \gamma_1) \triangleright \gamma_2 : \sigma'$ (by E_Cast)

The final derivation is what we seek, and thus we are done.

Case E_TLam: Impossible.

Case E_TApp: We have several cases, depending on how e has stepped:

Case S_TApp: By induction.

Case S_TBeta: By Lemma A.17.

Case S_TPush: We adopt the metavariable names from the rule:

$$\frac{\begin{array}{rcl} v & = & \Lambda\alpha.e \\ \gamma' & = & \gamma@\tau \end{array}}{(v \triangleright \gamma)\,\tau \longrightarrow v\,\tau \triangleright \gamma'}\ \textsc{S\_TPush}$$

Inversion tells us the following (for some σ and σ'):

- $\emptyset \vdash \gamma : (\forall\,\alpha.\sigma) \sim (\forall\,\alpha.\sigma')$
- $\emptyset \vdash v : \forall\,\alpha.\sigma$
- $\emptyset \vdash (v \triangleright \gamma)\,\tau : \sigma'[\tau/\alpha]$
- $\emptyset \vdash \forall\,\alpha.\sigma'$ type
- $\alpha \vdash \sigma'$ type
- $\emptyset \vdash \tau$ type

We see that $\emptyset \vdash \gamma' : \sigma[\tau/\alpha] \sim \sigma'[\tau/\alpha]$ by C_Inst. Thus:

- $\emptyset \vdash v\,\tau : \sigma[\tau/\alpha]$ (by E_TApp)
- $\emptyset \vdash \sigma'[\tau/\alpha]$ type (by Lemma A.15)
- $\emptyset \vdash v\,\tau \triangleright \gamma' : \sigma'[\tau/\alpha]$

The final derivation is what we seek, and thus we are done.

Case E_CLam: Impossible.

Case E_CApp: We have several cases, depending on how e has stepped:

Case S_CApp: By induction.

Case S_CBeta: By Lemma A.19.

Case S_CPush: We adopt the metavariable names from the rule:

$$\frac{\begin{array}{rcl} v & = & \lambda c : \phi.e_0 \\ \eta_0 & = & \mathbf{nth}_0\ \eta \\ \eta_1 & = & \mathbf{sym}\,(\mathbf{nth}_1\ \eta) \\ \eta_2 & = & \mathbf{nth}_2\ \eta \end{array}}{(v \triangleright \eta)\,\gamma \longrightarrow v\,(\eta_0 \mathbin{⨾} \gamma \mathbin{⨾} \eta_1) \triangleright \eta_2}\ \textsc{S\_CPush}$$

Let $\phi = \tau_0 \sim \tau_1$. Inversion tells us the following (for some τ_2, σ_0, σ_1, and σ_2):

- $\emptyset \vdash v : \tau_0 \sim \tau_1 \Rightarrow \tau_2$
- $\emptyset \vdash \eta : (\tau_0 \sim \tau_1 \Rightarrow \tau_2) \sim (\sigma_0 \sim \sigma_1 \Rightarrow \sigma_2)$
- $\emptyset \vdash \gamma : \sigma_0 \sim \sigma_1$
- $\emptyset \vdash (v \triangleright \eta)\,\gamma : \sigma_2$
- $\emptyset \vdash \sigma_2$ type

We can now deduce:

- $\emptyset \vdash \eta_0 : \tau_0 \sim \sigma_0$ (by C_NthQual)

- $\emptyset \vdash \eta_1 : \sigma_1 \sim \tau_1$ (by C_NthQual and C_Sym)
- $\emptyset \vdash \eta_2 : \tau_2 \sim \sigma_2$ (by C_NthQual)
- $\emptyset \vdash \eta_0 \mathbin{⨾} \gamma \mathbin{⨾} \eta_1 : \tau_0 \sim \tau_1$ (by C_Trans)
- $\emptyset \vdash v\,(\eta_0 \mathbin{⨾} \gamma \mathbin{⨾} \eta_1) : \tau_2$ (by E_CApp)
- $\emptyset \vdash v\,(\eta_0 \mathbin{⨾} \gamma \mathbin{⨾} \eta_1) \triangleright \eta_2 : \sigma_2$ (by E_Cast)

The final derivation is what we seek, and thus we are done.

Case E_Cast: We have two possibilities:

Case S_Cast: By induction.

Case S_Trans: We adopt the metavariable names from the rule:

$$\frac{}{(v \triangleright \gamma_1) \triangleright \gamma_2 \longrightarrow v \triangleright (\gamma_1 \mathbin{⨾} \gamma_2)}\ \text{S\_Trans}$$

Inversion tells us the following (for some τ_1, τ_2, and τ_3):

- $\emptyset \vdash v : \tau_1$
- $\emptyset \vdash \gamma_1 : \tau_1 \sim \tau_2$
- $\emptyset \vdash v \triangleright \gamma_1 : \tau_2$
- $\emptyset \vdash \gamma_2 : \tau_2 \sim \tau_3$
- $\emptyset \vdash (v \triangleright \gamma_1) \triangleright \gamma_2 : \tau_3$
- $\emptyset \vdash \tau_3$ type

We can thus deduce:

- $\emptyset \vdash \gamma_1 \mathbin{⨾} \gamma_2 : \tau_1 \sim \tau_3$ (by C_Trans)
- $\emptyset \vdash v \triangleright (\gamma_1 \mathbin{⨾} \gamma_2) : \tau_3$ (by E_Cast)

The final derivation is what we seek, and thus we are done.

Case E_Assume: We must step by S_Resolve.

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

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

We can assume the premises of E_Assume. We thus invoke Property 6.1 to see that $\emptyset \vdash q : \chi$. We are done by Lemma A.20.

□

A.9 Consistency

Lemma A.25 (Top-level reduction). *If $F\,\overline{\tau} \rightsquigarrow_{\top} \sigma_1$ and $F\,\overline{\tau} \rightsquigarrow_{\top} \sigma_2$, then $\sigma_1 \;=\; \sigma_2$.*

Proof. We proceed by induction on the sum of the sizes of the derivations $F\,\overline{\tau} \rightsquigarrow_{\top} \sigma_1$ and $F\,\overline{\tau} \rightsquigarrow_{\top} \sigma_2$. We may thus assume the induction hypothesis for any top-level reduction that appears in the premises of either $F\,\overline{\tau} \rightsquigarrow_{\top} \sigma_1$ or $F\,\overline{\tau} \rightsquigarrow_{\top} \sigma_2$.

From Assumption A.5, we see that every type family F is either open with potentially multiple, single-equations axioms or closed with at most one, potentially many-equationed axiom. We handle these cases separately:

Open family: Let $\xi_1 : E_1 \in \Sigma$ and $\xi_2 : E_2 \in \Sigma$ be the two axioms from the two reductions, respectively, and let $E_1 \;=\; \forall\,\overline{\alpha}_1\,\overline{\chi}_1.F\,\overline{\sigma}_1 \sim \sigma_{01}$ and $E_2 \;=\; \forall\,\overline{\alpha}_2\,\overline{\chi}_2.F\,\overline{\sigma}_2 \sim \sigma_{02}$. Furthermore, we know $\overline{\sigma}_1[\overline{\rho}_1/\overline{\alpha}_1] = \overline{\tau} = \overline{\sigma}_2[\overline{\rho}_2/\overline{\alpha}_2]$ (for the $\overline{\rho}_1$ and $\overline{\rho}_2$ learned by inversion). We know (by Assumption A.5), that compat(E_1, E_2). We now have two cases, depending on how compat(E_1, E_2) has been established:

Case Co_Coinc:

$$\frac{\begin{array}{l} E_1 \;=\; \forall\,\overline{\alpha}_1\,\overline{\chi}_1.F\,\overline{\tau}_1 \sim \tau_{01} \\ E_2 \;=\; \forall\,\overline{\alpha}_2\,\overline{\chi}_2.F\,\overline{\tau}_2 \sim \tau_{02} \\ \mathsf{unify}(\overline{\tau}_1;\,\overline{\tau}_2) \;=\; \mathsf{Just}\,\theta \\ \tau_{01}[\theta \circ \mathit{subst}(\overline{\chi}_1)] \;=\; \tau_{02}[\theta \circ \mathit{subst}(\overline{\chi}_2)] \end{array}}{\mathsf{compat}(E_1, E_2)}\;\textsc{Co\_Coinc}$$

Consider the substitution $\theta \;=\; \overline{\rho}_1/\overline{\alpha}_1, \overline{\rho}_2/\overline{\alpha}_2$. This is a unifier of $\overline{\sigma}_1$ and $\overline{\sigma}_2$. Thus, by Property A.6, $\mathsf{unify}(\overline{\sigma}_1;\,\overline{\sigma}_2) \;=\; \mathsf{Just}\,\theta_0$ where $\theta \;=\; \theta' \circ \theta_0$. We see above that $\sigma_{01}[\theta_0 \circ \mathit{subst}(\overline{\chi}_1)] \;=\; \sigma_{02}[\theta_0 \circ \mathit{subst}(\overline{\chi}_2)]$ and thus $\sigma_{01}[\theta \circ \mathit{subst}(\overline{\chi}_1)] \;=\; \sigma_{02}[\theta \circ \mathit{subst}(\overline{\chi}_2)]$. Because the free variables in σ_{01} and σ_{02} are distinct, we can simplify to $\sigma_{01}[\overline{\rho}_1/\overline{\alpha}_1 \circ \mathit{subst}(\overline{\chi}_1)] \;=\; \sigma_{02}[\overline{\rho}_2/\overline{\alpha}_2 \circ \mathit{subst}(\overline{\chi}_2)]$. Call that type ρ_0.

Let's consider the shapes of ρ_0 and σ_1, one of the top-level reducts of $F\,\overline{\tau}$. The former is $\sigma_{01}[\overline{\rho}_1/\overline{\alpha}_1 \circ \mathit{subst}(\overline{\chi}_1)]$ and the latter is $\sigma_{01}[\overline{\rho}_1/\overline{\alpha}_1, \overline{\rho}_1'/\mathit{tvs}(\overline{\chi}_1)]$. Thus, the only difference between ρ_0 and σ_1 is the choice for the instantiation of the $\mathit{tvs}(\overline{\chi}_1)$—$\rho_0$ replaces these with type family applications, and σ_1 replaces them with proper (type-family-free) types. However, note that in the premises to RTop, the choice of these types (the $\overline{\rho}'$ in the rule) is determined by the type family applications, using a $\rightsquigarrow_{\mathsf{T}}$ reduction. These reductions are in a premise to a rule we are performing induction on, and therefore we may assume that the type family application uniquely determines the reduct. Thus there exists precisely one σ_1 that corresponds to the ρ_0, and accordingly σ_2 must be that same σ_1. We are done with this case.

Case Co_Distinct: In this case, there is no unifier between $\overline{\sigma}_1$ and $\overline{\sigma}_2$, a contradiction with Property A.6.

Closed family: In this case, we have the possibility that $F\,\overline{\tau}$ is reducible by more than one equation of the single applicable axiom ξ with equations $\overline{E_n \;=\; \forall\,\overline{\alpha}_n\,\overline{\chi}_n.F\,\overline{\sigma}_n \sim \sigma_{0\,n}}$. Number the two equations i and j. If $i \;=\; j$, we are done.[9] We thus assume, without loss of generality, that $j < i$. We see as a premise to RTop that $\mathsf{no\_conflict}(\overline{E}, i, \overline{\rho}, j)$, where $\overline{\tau} \;=\; \overline{\sigma}_i[\overline{\rho}/\overline{\alpha}_i]$. Thus, either the two equations are apart (NC_Apart) or they are compatible. We'll handle these cases separately:

Case NC_Apart: In this case, we know that $\mathsf{apart}(\overline{\sigma}_j;\,\overline{\sigma}_i[\overline{\rho}/\overline{\alpha}_i])$—that is, $\mathsf{apart}(\overline{\sigma}_j;\,\overline{\tau})$. By Property A.6, we can conclude that RTop cannot apply at equation j, a contradiction.

Case NC_Compatible: Here, equations E_i and E_j are compatible; follow the logic used in the open-type-family case.

□

This is very closely based on the similar proof by Eisenberg et al. [5].

Definition A.26 (Type contexts).

(1) Let $C[\cdot]$ be a type with exactly one hole.
(2) Let $\mathbb{C}[\cdot]$ be a list of types with exactly one hole (in the whole list).
(3) Let $C[\![\cdot]\!]$ be a type with any number of holes.
(4) Let $\mathbb{C}[\![\cdot]\!]$ be a list of types with any number of holes.

Lemma A.27 (One step/many holes context substitution). *If $\tau \rightsquigarrow \tau'$, then $C[\![\tau]\!] \rightsquigarrow^{*} C[\![\tau']\!]$.*

Proof. Straightforward induction on the structure of $C[\![\cdot]\!]$. □

Lemma A.28 (Multistep/many holes context substitution). *If $\tau \rightsquigarrow^{*} \tau'$, then $C[\![\tau]\!] \rightsquigarrow^{*} C[\![\tau']\!]$.*

Proof. Straightforward induction on the length of the reduction $\tau \rightsquigarrow^{*} \tau'$, appealing to Lemma A.27. □

[9] We still need to be sure that the evaluation assumptions are satisfied by the same types when running the reduction twice, but we can get this fact by a similar argument as given in the open type family case.

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

Proof. Proceed by induction on the structure of τ_0.

Case $\tau_0 = \sigma_1 \to \sigma_2$: Inversion on Red tells us $C_1[F_1\,\overline{\rho}_1] = \sigma_1 \to \sigma_2 = C_2[F_2\,\overline{\rho}_2]$, where $\tau_1 = C_1[\rho'_1]$ and $\tau_2 = C_2[\rho'_2]$. Proceed by case analysis on $C_1[\cdot]$ and $C_2[\cdot]$:

Case $C_1[\cdot] = C'_1[\cdot] \to \sigma_2, C_2[\cdot] = C'_2[\cdot] \to \sigma_2$: (We know the right-hand types to $(\to)$ must match from $C_1[F_1\,\overline{\rho}_1] = C_2[F_2\,\overline{\rho}_2]$.) Let $\sigma_3 = C'_1[\rho'_1]$ and $\sigma_4 = C'_2[\rho'_2]$. Noting that C appears only in the conclusion of Red, and not in any premise, we can see that $\sigma_1 \rightsquigarrow \sigma_3$ and $\sigma_1 \rightsquigarrow \sigma_4$. The induction hypothesis thus gives us σ_5 such that $\sigma_3 \rightsquigarrow^* \sigma_5 \leftsquigarrow^* \sigma_4$. We then say that τ_3, our common reduct, is $\sigma_5 \to \sigma_2$, appealing to Lemma A.28.

Case $C_1[\cdot] = C'_1[\cdot] \to \sigma_2, C_2[\cdot] = \sigma_1 \to C'_2[\cdot]$: Let $\tau_3 = C'_1[\rho'_1] \to C'_2[\rho'_2]$. Because we have $\sigma_1 = C'_1[F\,\overline{\rho}]$ and $\sigma_2 = C'_2[F\,\overline{\rho}]$, we can see that $(C'_1[\rho'_1] \to \sigma_2) \rightsquigarrow \tau_3 \leftsquigarrow (\sigma_1 \to C'_2[\rho'_2])$ as desired.

Other cases: Similar to the two previous cases.

Case $\tau_0 = H\,\overline{\sigma}$: Similar to previous case.

Case $\tau_0 = \alpha$: Impossible.

Case $\tau_0 = \forall\,\alpha.\sigma$: Similar to first sub-case of the $\tau_0 = \sigma_1 \to \sigma_2$ case.

Case $\tau_0 = (\sigma_1 \sim \sigma_2) \Rightarrow \sigma_3$: Similar to $\tau_0 = \sigma_1 \to \sigma_2$ case.

Case $\tau_0 = F\,\overline{\sigma}$: We have $C_1[F_1\,\overline{\rho}_1] = F\,\overline{\sigma} = C_2[F_2\,\overline{\rho}_2]$, where $\tau_1 = C_1[\rho'_1]$ and $\tau_2 = C_2[\rho'_2]$. We have several cases, depending on the structure of $C_1[\cdot]$ and $C_2[\cdot]$:

Case $C_1[\cdot] \neq \cdot, C_2[\cdot] \neq \cdot$: The top-level function F is not involved in the reductions. Proceed similarly to the $\tau_0 = \sigma_1 \to \sigma_2$ case.

Case $C_1[\cdot] = F\,\mathbb{C}'_1[\cdot], C_2[\cdot] = \cdot$: This case cannot happen. Examine the Red rule:

$$\frac{F\,\overline{\tau} \rightsquigarrow_{\top} \tau'}{C[F\,\overline{\tau}] \rightsquigarrow C[\tau']}\ \textsc{Red}$$

The $\overline{\tau}$—that is, the arguments to the function F—equal $\overline{\sigma}[\overline{\rho}/\overline{\alpha}]$. But $\overline{\sigma}$ are type-family free (by Assumption A.4 and D_Axiom) and the $\overline{\rho}$ are type-family free (by $\overline{\emptyset \vdash \rho_k \text{ type}}^{k}$). Thus, $\overline{\tau}$ must also be type-family free, meaning that there is no way τ_0 could have stepped to τ_1.

Case $C_1[\cdot] = \cdot, C_2[\cdot] = F\,\mathbb{C}'_2[\cdot]$: Similar to previous case.

Case $C_1[\cdot] = \cdot, C_2[\cdot] = \cdot$: In this case, we have two top-level reductions. Lemma A.25 gives us something stronger than what we seek: that $\tau_1 = \tau_2$ in this case. We are done.

□

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

Proof. Examine RTop:

$$\frac{\begin{array}{l}
\xi : \overline{E} \in \Sigma \qquad E_i = \forall\,\overline{\alpha}\,\overline{\chi}.F\,\overline{\sigma} \sim \sigma_0 \\
\overline{\tau} = \overline{\sigma}[\overline{\rho}/\overline{\alpha}] \qquad \tau' = \sigma_0[\overline{\rho}/\overline{\alpha}, \overline{\rho}'/tvs(\overline{\chi})] \\
\overline{\emptyset \vdash \rho_k \text{ type}}^{k} \qquad \forall\,j < i, \text{no\_conflict}(\overline{E}, i, \overline{\rho}, j) \\
\forall\,n : \\
\qquad \chi_n = (\alpha'|c' : F'\,\overline{\sigma}' \sim \alpha') \\
\qquad \emptyset \vdash \rho'_n \text{ type} \\
\qquad \theta_n = \overline{\rho}/\overline{\alpha}, \overline{\rho'_m/tv(\chi_m)}^{m\in 1..n-1} \\
\qquad F'\,\overline{\sigma}'[\theta_n] \rightsquigarrow_{\top} \rho'_n
\end{array}}{F\,\overline{\tau} \rightsquigarrow_{\top} \tau'}\ \textsc{RTop}$$

We see that the reduct type τ' equals $\sigma_0[\overline{\rho}/\overline{\alpha}, \overline{\rho}'/tvs(\overline{\chi})]$. By Assumption A.4, σ_0 has no type families. By $\overline{\emptyset \vdash \rho_k \text{ type}}^{k}$, we see that $\overline{\rho}$ has no type families. By $\overline{\emptyset \vdash \rho'_n \text{ type}}^{n}$, we see that $\overline{\rho}'$ has no type families. Thus τ' can mention no type families.

Yet, in Red,

$$\frac{F\,\overline{\tau} \rightsquigarrow_{\top} \tau'}{C[F\,\overline{\tau}] \rightsquigarrow C[\tau']}\ \textsc{Red}$$

we replace a type family application $F\,\overline{\tau}$ with τ'. We have thus replaced a type family application with a type that contains no type families. Accordingly, in every use of Red, the reduct must have exactly one fewer type family application than the redex. Given that types are an inductively defined structure, we cannot have an infinite number of type family applications in a type.

We have thus identified a decreasing measure: the number of type family applications in a type. Accordingly, all reduction chains terminate. □

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

Proof. By Lemma 6.9, Lemma 6.8, and Newman's Lemma. □

Lemma A.29 (Rigid reduction). *If $C[\tau_1] \rightsquigarrow^* \tau_0 \leftsquigarrow^* C[\tau_2]$, such that the path to the hole in $C[\cdot]$ does not go through any type family arguments, then there exists τ_3 such that $\tau_1 \rightsquigarrow^* \tau_3 \leftsquigarrow^* \tau_2$.*

Proof. Straightforward induction on the structure of $C[\cdot]$. □

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

Proof. By induction on the derivation of $\emptyset \vdash \gamma : \tau_1 \sim \tau_2$.

Case C_Refl: Trivial.
Case C_Sym: By induction hypothesis.
Case C_Trans: By Lemma 6.10.
Case C_App: By Lemma A.28.
Case C_Fun: By Lemma A.28.
Case C_Fam: By Lemma A.28.
Case C_Forall: By Lemma A.28.
Case C_Qual: By Lemma A.28.
Case C_Nth: By Lemma A.29.
Case C_NthArrow: By Lemma A.29.
Case C_NthQual: By Lemma A.29.
Case C_Inst: By Lemma A.29 and substitution.
Case C_Axiom:

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

The left-hand type steps to the right-hand type by Red.

Case C_Var: Impossible in an empty context.

□

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

Proof. Direct from the definition of $\rightsquigarrow$, Red. □

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* $\tau_1 = \tau_2$.

Proof. Lemma 6.6 tells us that there exists τ_3 such that $\tau_1 \rightsquigarrow^* \tau_3 \leftsquigarrow^* \tau_2$. But Lemma 6.7 tells us that τ_1 and τ_2 do not reduce. We must conclude that $\tau_1 = \tau_3 = \tau_2$, and we are done. □

A.10 Progress

Lemma A.30 (Canonical forms).

(1) If $\emptyset \vdash v : \tau_1 \rightarrow \tau_2$, then $v = \lambda x : \tau_1 . e$ for some e.
(2) If $\emptyset \vdash v : \forall \alpha . \tau$, then $v = \Lambda \alpha . e$ for some e.
(3) If $\emptyset \vdash v : \phi \Rightarrow \tau$, then $v = \lambda c : \phi . e$ for some e.

Proof. By case analysis on the typing derivation. □

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'.*

Proof. By induction on the derivation of $\emptyset \vdash e : \tau$.

Case E_Var: Impossible.
Case E_Const: Trivial.
Case E_Lam: Trivial.
Case E_App: We know that $e = e_1\, e_2$. A use of the induction hypothesis on e_1 gives us three possibilities:

Case $e_1 = v_1$: Lemma A.30 tells us that $e_1 = \lambda x : \tau_0 . e_0$ and we are thus done by E_Beta.

Case $e_1 = v_1 \triangleright \gamma$: Inversion tells us that $\emptyset \vdash \gamma : \tau_0 \sim (\sigma_1 \rightarrow \sigma_2)$, with $\Gamma \vdash (\sigma_1 \rightarrow \sigma_2)$ type. Lemma 6.5 (on $\emptyset \vdash v_1 : \tau_0$) tells us that $\Gamma \vdash \tau_0$ type. Lemma 6.4 then proves that $\tau_0 = \sigma_1 \rightarrow \sigma_2$. We can thus use Lemma A.30 to get $e_1 = \lambda x : \sigma_1 . e_0$ and we can step by S_Push.

Case $e_1 \longrightarrow e_1'$: We are done by S_App.

Case E_TLam: Trivial.
Case E_TApp: Similar to E_App case, using S_TBeta and S_TPush.
Case E_CLam: Trivial.
Case E_CApp: Similar to E_App case, using S_CBeta and S_CPush.
Case E_Cast: We know that $e = e_0 \triangleright \gamma$. A use of the induction hypothesis on e_0 gives us three possibilities:

Case $e_0 = v_0$: Trivial.

Case $e_0 = v_0 \triangleright \gamma_0$: We are done by S_Trans.

Case $e_0 \longrightarrow e_0'$: We are done by S_Cast.

Case E_Assume: We are done by S_Resolve, noting that the assumed derivation must exist by Property 6.1.

□