University of Pennsylvania Technical Report MS-CIS-13-10

Closed Type Families with Overlapping Equations (Extended version)

Richard A. Eisenberg
University of Pennsylvania
eir@cis.upenn.edu

Dimitrios Vytiniotis
Simon Peyton Jones
Microsoft Research Cambridge
{dimitris,simonpj}@microsoft.com

Stephanie Weirich
University of Pennsylvania
sweirich@cis.upenn.edu

Abstract

Open, type-level functions are a recent innovation in Haskell that move Haskell towards the expressiveness of dependent types, while retaining the look and feel of a practical programming language. This paper shows how to increase expressiveness still further, by adding closed type functions whose equations may overlap, and may have non-linear patterns over an open type universe. Although practically useful and simple to implement, these features go *beyond* conventional dependent type theory in some respects, and have a subtle metatheory.

Categories and Subject Descriptors F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—type structure; D.3.3 [*Programming Languages*]: Language Constructs and Features; F.4.2 [*Mathematical Logic and Formal Languages*]: Grammars and Other Rewriting Systems—parallel rewriting systems

General Terms Design, Languages, Theory

Keywords Type families; Type-level computation; Haskell; System FC

1. Introduction

Type families are a relatively recent extension to Haskell that allows the programmer to express type-level computation (Chakravarty et al. 2005). For example, one can say

```
type family Elt (a :: ⋆) :: ⋆
type instance Elt ByteString = Word8
type instance Elt [b] = b
```

The first line declares the type family *Elt* and gives its kind; the second and third are two independent declarations that give two equations for *Elt*. Now the types (*Elt ByteString*) and *Word8* are considered equivalent by the type inference engine, and likewise (*Elt* [*Int*]) and *Int*. Type families have proved to be a popular feature in Haskell, dovetailing particularly nicely with Haskell's type classes. Type families are naturally *partial* and *open*. For example, there is no equation for *Elt Char* above, so *Elt Char* will never be equal to any other type. On the other hand, the author of a new library is free to add a new instance, such as this one:

```
type instance Elt (Set b) = b
```

However, not *all* type-level functions can be defined by open type families. An important example is the equality function, which determines whether two types can be shown equal at compile-time:[1]

[1] Here we use *datatype promotion*, allowing data types like *Bool*, and lists, to be used as kinds (Yorgey et al. 2012).

```
type family Equal a b :: Bool
type instance Equal a a = True    -- Instance (A)
type instance Equal a b = False   -- Instance (B)
```

The programmer intends these equations to be read top-to-bottom, like a term-level function definition in Haskell. However, because GHC's current type families are open, they must be defined by independent, un-ordered **type instance** equations. The two equations overlap, so they are rightly rejected lest they be used to deduce unsound type equalities. For example, we could reduce the type *Equal Int Int* to both *True* and *False*, since both patterns match.

Yet equality *is* a well-defined function, and a useful one too, as we discuss in Section 2. To fix this omission we introduce *closed type families* with ordered equations, thus:

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

Now all the equations for the type family are given together, and can be read top-to-bottom. However, behind this simple idea lie a number of complexities. In this paper we describe these pitfalls and their sometimes non-obvious solutions. We make the following contributions:

- We introduce closed type families with overlapping equations, and show how they can readily express programs that were previously inexpressible or required indirect encodings (Section 2).

- Our system supports *non-linear left-hand sides*, such as that for Equal above, where the variable a is repeated in the first equation. It also supports *coincident overlap*, which allows some lightweight theorem-proving capability to be incorporated in the definitional equality of types (Section 3.4).
- We give the subtle rules that govern type family simplification, including those that determine when a pattern *cannot* be matched by a type (Section 3).
- We describe a typed core language that includes both open and closed type families (Section 4), and prove that it is type-safe, assuming that type families terminate (Section 5). We do that by establishing a *consistency* property of the type equations induced by type families.
- We identify the complications for consistency that arise from non-terminating type families and we expose a subtle oversight in GHC's current rules for open type families in Section 6.
- We have implemented closed type families in GHC as well as a number of case studies, such as the units package, an extensible framework for dimensional analysis, presented in Appendix A. Closed type families are available now in GHC 7.8.

In short, the programmer sees a simple, intuitive language feature, but the design space (and its metatheory) is subtle. Although type families resemble the type-level computation and "large eliminations" found in full-spectrum dependently-typed languages like Coq and Agda, there are important semantic and practical differences. We discuss these in Section 8.

2. Closed type families

Haskell (in its implementation in GHC) has supported *type families* for several years. They were introduced to support *associated types*, a feature that Garcia et al.'s (2003) comparison between C++, Haskell, and ML, noted as a C++'s main superiority for generic programming.

Type families were designed to dovetail smoothly with type classes. For example, the type function[2] *Elt* above could be used to specify the element type in a container class:

```
class Container c where
  empty :: c
  member :: Elt c → c → Bool
instance Container [a] where ...
instance Container ByteString where ...
```

New instances for *Container* can be defined as new types are introduced, often in different modules, and correspondingly new equations for *Elt* must be added too. Hence *Elt* must be *open* (that is, can be extended in modules that import it), and *distributed* (can be scattered over many different modules). This contrasts with term-level functions where we are required to define the function all in one place.

The open, distributed nature of type families, typically associated with classes, requires strong restrictions on overlap to maintain soundness. Consider

```
type family F a b :: ⋆
type instance F Int a     = Bool
type instance F a   Bool = Char
```

Now consider the type (*F Int Bool*). Using the first equation, this type is equal to *Bool*, but using the second it is equal to *Char*. So if we are not careful, we could pass a *Bool* to a function expecting a *Char*, which would be embarrassing.

GHC therefore brutally insists that the left-hand sides of two **type instance** equations must not overlap (unify). (At least, unless the right-hand sides would then coincide; see Section 3.4.)

2.1 Closed families: the basic idea

As we saw in the Introduction, disallowing overlap means that useful, well-defined type-level functions, such as type level equality, cannot be expressed. Since openness is the root of the overlap problem, it can be solved by defining the equations for the type family *all in one place*. We call this a *closed type family* and define it using a **where** clause on the function's original declaration. The equations may overlap, and are matched top-to-bottom. For example:

```
type family And (a :: Bool) (b :: Bool) :: Bool where
  And True True = True
  And a    b    = False
```

Since the domain of *And* is closed and finite, it is natural to write all its equations in one place. Doing so directly expresses the fact that no further equations are expected.

Although we have used overlap in this example, one can always write functions over *finite* domains without overlap:

[2] We use "type family" and "type function" interchangeably.

```
type family And' (a :: Bool) (b :: Bool) :: Bool where
  And' True  True  = True
  And' False True  = False
  And' True  False = False
  And' False False = False
```

Nevertheless, overlap is convenient for the programmer, mirrors what happens at the term level, avoids a polynomial blowup in program size, and is more efficient (for the type checker) to execute. Furthermore, when defined over an *open* kind, such as $\star$, closed type families allow a programmer to express relationships (such as inequality of types—see Section 2.4) that are otherwise out of reach.

2.2 Non-linear patterns

Let us return to our equality function, which can now be defined thus:

```
type family Equal (a :: ⋆) (b :: ⋆) :: Bool where
  Equal a a = True
  Equal a b = False
```

This declaration introduces the type function *Equal*, gives its kind and, in the **where** clause, specifies all its equations. The first equation has a non-linear pattern, in which *a* is repeated, and it overlaps with the second equation. If the domain were finite we could avoid both features by writing out all the equations exhaustively, but new types can be introduced at any time, so we cannot do that here.

The issue becomes even clearer when we use *kind polymorphism* (Yorgey et al. 2012), thus:

```
type family Equal (a :: κ) (b :: κ) :: Bool where
  Equal a a = True
  Equal a b = False
```

For example, (*Equal Maybe List*) should evaluate to *False*. It may seem unusual to define a function to compute equality even over types of *function kind* $(\star \to \star)$. After all, there is no construct that can compare functions at the term level.

At the type level, however, the type checker decides equality at function kinds all the time! In the world of Haskell types there exist no anonymous type-level functions, nor can type families appear partially applied, so this equality test—which checks for *definitional* equality, in type theory jargon—is straightforward. All *Equal* does is reify the (non-extensional) equality test of the type checker.

In fact, Haskell programmers are used to this kind of equality matching on types; for example, even in Haskell 98 one can write

```
instance Num a ⇒ Num (T a a) where ...
```

Because the type inference engine already supports decidable equality, it is very straightforward to implement non-linear patterns for type functions as well as type classes. Non-linear patterns are convenient for the programmer, expected by Haskell users, and add useful expressiveness. They do make the metatheory much harder, as we shall see, but that is a problem that has to be solved only once.

2.3 Type structure matching

In our experience, most cases where closed type families with overlapping equations are useful involve a variation on type equality. However, sometimes we would like to determine whether a type matches a specific top-level structure.

For example, we might want to look at a function type of the form *Int* → (*Bool* → *Char*) → *Int* → *Bool* and determine that this is a function of three arguments.

```
data Nat = Zero | Succ Nat
type family CountArgs (f :: ⋆) :: Nat where
   CountArgs (a → b) = Succ (CountArgs b)
   CountArgs result    = Zero
```

Because the equations are tried in order, any function type will trigger the first equation and any ground non-function type (that is, a type that is not a type variable or an arrow type) will trigger the second. Thus, the type family effectively counts the number of parameters a function requires.

When might this be useful? We have used this type family to write a variable-arity *zipWith* function that infers the correct arity, assuming that the result type is not a function type. Other approaches that we are aware of (Fridlender and Indrika 2000; McBride 2002; Weirich and Casinghino 2010) require some encoding of the desired arity to be passed explicitly. A full presentation of the variable-arity *zipWith* is presented in Appendix B. To achieve the same functionality in a typical dependently typed language like

Agda or Coq, we must pattern-match over some inductive universe of codes that can be interpreted into types.

2.4 Observing inequality

Type families such as *Equal* allow programmers to observe when types *do not* match. In other words, *Equal Int Bool* automatically reduces to *False*, via the second equation. With open type families, we could only add a *finite number* of reductions of un-equal types to *False*.

However, the ability to observe inequality is extremely useful for expressing failure in compile-time search algorithms. This search could be a simple linear search, such as finding an element in a list. Such search underlies the `HList` library and its encoding of heterogeneous lists and extensible records (Kiselyov et al. 2004). It also supports Swierstra's solution to the expression problem via extensible datatypes (Swierstra 2008). Both of these proposals use the extension `-XOverlappingInstances` to implement a compile-time equality function.[3]

Type families can directly encode more sophisticated search algorithms than linear list searching, including those requiring backtracking, simply by writing a functional program. For example, the following closed type family determines whether a given element is present in a tree.

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
type family TMember (e :: κ) (set :: Tree κ) :: Bool where
  TMember e (Leaf x)             = Equal e x
  TMember e (Branch left right) =
    Or (TMember e left) (TMember e right)
```

Implementing this search using overlapping type classes, which do not support backtracking, requires an intricate encoding with explicit stack manipulation.

2.5 Summary

Type-level computation is a powerful idea: it allows a programmer to express application-specific compile-time reasoning in the type system. Closed type families fill in a missing piece in the design space, making type families more expressive, convenient, and more uniform with term-level functional programming.

τ, σ	Types
ρ	Type patterns (no type families)
F	Type families
Ω	Substitutions from type variables to types

Figure 1. Grammar of Haskell metavariables

3. Simplifying closed family applications

We have shown in the previous sections how type family reduction can be used to equate types. For example, a function requiring an argument of type *T True* can take an argument of type *T* (*And True True*), because the latter reduces to the former.

Because the definition of type equality is determined by type family reduction, the static semantics must precisely define what reductions are allowed to occur. That definition turns out to be quite subtle, so this section develops an increasingly refined notion of type family reduction, motivated by a series of examples. The presentation gives a number of definitions, using the vocabulary of Figure 1, but we eschew full formality until Section 4. We use the term *"target"* to designate the type-function application that we are trying to simplify. We say that a type τ_1 *"simplifies"* or *"reduces"* to another type τ_2 if we can rewrite the τ_1 to τ_2 using a (potentially empty) sequence of left-to-right applications of type family equations. We also use the notation $\tau_1 \rightsquigarrow \tau_2$ to denote exactly one application of a type family equation and $\tau_1 \rightsquigarrow^* \tau_2$ to denote an arbitrary number of reductions. Type equality is defined to be roughly the reflexive, symmetric, transitive, congruent closure of type reduction; details are in Section 4.3.

We frequently refer to the example in the introduction, repeated below, with the variables renamed to aid in understanding:

```
type family Equal (a :: κ) (b :: κ) :: Bool where
  Equal a a = True    -- Eqn (A)
  Equal b c = False   -- Eqn (B)
```

3.1 No functions on the LHS

If we wish to simplify *Equal Int Int*, equation (A) of the definition matches, so we can safely "fire" the equation (A) to simplify the application to *True*.

Even here we must take a little care. What happens if try this?

```
type family F (a :: Bool) where
  F False       = False
  F True        = True
  F (Equal x y) = True
```

Then F ($Equal$ Int $Bool$) superficially appears to match only the third equation. But of course, if we simplify the argument of F in the target, it would become F $False$, which matches the first equation.

The solution here is quite standard: in type family definitions (both open and closed) we do not allow functions in the argument types on the LHS. In terms of Figure 1, the LHS of a function axiom must be a *pattern* ρ. This is directly analogous to allowing only constructor patterns in term-level function definitions, and is already required for Haskell's existing open type families.

We then propose the following first attempt at a reduction strategy:

Candidate Rule 1 (Closed type family simplification)**.** *An equation for a closed type family F can be used to simplify a target $(F\ \overline{\tau})$ if (a) the target matches the LHS of the equation, and (b) no*

[3] This extension allows class instances, but not type family instances, to overlap. If the type inference engine chooses the wrong class instance, a program may have incoherent behavior, but it is believed that type safety is not compromised. See Morris and Jones (2010) for relevant discussion.

LHS of an earlier *equation for F matches the target.*

The formal definition of matching follows:

Definition 1 (Matching)**.** *A pattern ρ* matches *a type τ, written* $\mathsf{match}(\rho, \tau)$*, when there is a well-kinded substitution Ω such that $\Omega(\rho) = \tau$. The domain of Ω must be a subset of the set of free variables of the pattern ρ.*

3.2 Avoiding premature matches with apartness

Suppose we want to simplify *Equal Bool d*. Equation (A) above fails to match, but (B) matches with a substitution $\Omega = [b \mapsto Bool, c \mapsto d]$. But it would be a mistake to simplify *Equal Bool d* to *False*. Consider the following code:

```
type family FunIf (b :: Bool) :: ⋆ where
  FunIf True = Int → Int
  FunIf False = ()
bad :: d → FunIf (Equal Bool d)
bad _ = ()
segFault :: Int
segFault = bad True 5
```

If we do simplify the type *Equal Bool d* to *False* then we can show that *bad* is well typed, since *FunIf False* is (). But then *segFault* calls *bad* with *d* instantiated to *Bool*. So *segFault* expects *bad True* to return a result of type *FunIf* (*Equal Bool Bool*), which reduces to *Int* → *Int*, so the call in *segFault* type-checks too. Result: we apply () as a function to 5, and crash.

The error, of course, is that we wrongly simplified the type (*Equal Bool d*) to *False*; wrongly because the choice of which equation to match depends on how *d* is instantiated. While the target ($\mathsf{\mathit{Equal\ Bool\ d}}$) does not match the earlier equation, *there is a substitution* for d that causes it to match the earlier equation. Our Candidate Rule 1 is insufficient to ensure type soundness. We need a stronger notion of *apartness* between a (target) type and a pattern, which we write as $\mathsf{apart}(\rho, \tau)$ in what follows.

Candidate Rule 2 (Closed type family simplification)**.** *An equation for a closed type family* F *can be used to simplify a target* $(F\ \overline{\tau})$ *if (a) the target matches the LHS of the equation, and (b) every LHS* $\overline{\rho}$ *of an* earlier *equation for* F *is apart from the target; that is,* $\mathsf{apart}(\overline{\rho}, \overline{\tau})$.

As a notational convention, $\mathsf{apart}(\overline{\rho}, \overline{\tau})$ considers the lists $\overline{\rho}$ and $\overline{\tau}$ as tuples of types; the apartness check does *not* go element-by-element. We similarly treat uses of match and unify (defined shortly) when applied to lists.

To rule out our counterexample to type soundness, apartness must at the very least satisfy the following property:

Property 2 (Apartness through substitution)**.** *If* $\mathsf{apart}(\rho, \tau)$ *then there exists no* Ω *such that* $\mathsf{match}(\rho, \Omega(\tau))$.

An appealing implementation of $\mathsf{apart}(\rho, \tau)$ that satisfies Property 2 is to check that the target τ and the pattern ρ are *not unifiable*, under the following definition:

Definition 3 (Unification)**.** *A type* τ_1 unifies *with a type* τ_2 *when there is a well-kinded substitution* Ω *such that* $\Omega(\tau_1) = \Omega(\tau_2)$. *We write* $\mathsf{unify}(\tau_1, \tau_2) = \Omega$ *for the most general such unifier if it exists.*[4]

[4] For instance, the implementation of unify can be the standard first-order unification algorithm of Robinson.

However this test is not sufficient for type soundness. Consider the type $\mathsf{\mathit{Equal\ Int\ (G\ Bool)}}$, where $\mathsf{\mathit{G}}$ is a type family. This type does not match equation (A), nor does it *unify* with (A), but it *does* match (B). So according to our rule, we can use (B) to simplify $\mathsf{\mathit{Equal\ Int\ (G\ Bool)}}$ to $\mathsf{\mathit{False}}$. But, if $\mathsf{\mathit{G}}$ were a type function with equation

$$\textbf{type instance}\ \mathsf{\mathit{G\ Bool}} = \mathsf{\mathit{Int}}$$

then we could use this equation to rewrite the type to $\mathsf{\mathit{Equal\ Int\ Int}}$, which patently *does* match (A) and simplifies to $\mathsf{\mathit{True}}$!

In our check of previous equations of a closed family, we wish to ensure that no previous equation can *ever* apply to a given application. Simply checking for unification of a previous pattern and the target is not enough. To rule out this counterexample we need yet another property from the $\mathsf{apart}(\rho, \tau)$ check, which ensures that the target cannot match a pattern of an earlier equation through arbitrary reduction too.

Property 4 (Apartness through reduction)**.** *If* $\mathsf{apart}(\rho, \tau)$*, then for any* τ' *such that* $\tau \leadsto^* \tau'$*:* $\neg\mathsf{match}(\rho, \tau')$.

3.3 A definition of apartness

We have so far sketched *necessary* properties that the apartness check must satisfy—otherwise, our type system surely is not sound. We have also described why a simple unification-based test does not meet these conditions, but we have not yet given a concrete definition of this check.

Note that we cannot use Property 4 to *define* $\mathsf{apart}(\rho, \tau)$ because it would not be well founded. We need $\mathsf{apart}(\rho, \tau)$ to define how type families should reduce, but Property 4 itself refers to type family reduction. Furthermore, even if this were acceptable, it seems hard to implement. We have to ensure that, for any substitution, no reducts of a target can possibly match a pattern; there can be exponentially many reducts in the size of the type and the substitution.

Hence we seek a conservative but cheap test. Let us consider again why unification is not sufficient. In the example from the previous section, we showed that type $\mathit{Equal}\ \mathit{Int}\ (G\ \mathit{Bool})$ does not match equation (A), nor does it *unify* with (A). However, $\mathit{Equal}\ \mathit{Int}\ (G\ \mathit{Bool})$ can simplify to $\mathit{Equal}\ \mathit{Int}\ \mathit{Int}$ and now equation (A) does match the reduct.

To take the behavior of type families into account, we first *flatten* any type family applications in the arguments of the target (i.e., the types $\overline{\tau}$ in a target $F\,\overline{\tau}$) to fresh variables. Only then do we check that the new target is not unifiable with the pattern. This captures the notion that a type family can potentially reduce to any type—anything more refined would require advance knowledge of all type families, impossible in a modular system. In our example, we must check $\mathsf{apart}((a, a), (\mathit{Int}, G\ \mathit{Bool}))$ when trying to use the second equation of Equal to simplify $\mathit{Equal}\ \mathit{Int}\ (G\ \mathit{Bool})$. We first flatten $(\mathit{Int}, G\ \mathit{Bool})$ into (Int, x) (for some fresh variable x). Then we check whether (a, a) cannot be unified with (Int, x). We quickly discover that these types *can* be unified. Thus, (a, a) and $(\mathit{Int}, G\ \mathit{Bool})$ are *not* apart and simplifying $\mathit{Equal}\ \mathit{Int}\ (G\ \mathit{Bool})$ to False is prohibited.

What if two type family applications in the target type are syntactically identical? Consider the type family F below:

```
type family F a b where
  F Int Bool = Char
  F a   a    = Bool
```

Should the type $F\ (G\ Int)\ (G\ Int)$ be apart from the left-hand-side $F\ Int\ Bool$? If we flatten to two distinct type variables then it is not apart; if we flatten using a common type variable then it becomes apart. How can we choose if flattening should preserve sharing or not? Let us consider the type $F\ b\ b$, which matches

the second equation. It is definitely apart from $F\ Int\ Bool$ and can indeed be simplified by the second equation. What happens, though, if we substitute $G\ Int$ for b in $F\ b\ b$? If flattening did not take sharing into account, $(G\ Int, G\ Int)$ would *not* be apart from $(Int, Bool)$, and $F\ (G\ Int)\ (G\ Int)$ wouldn't reduce. Hence, the ability to simplify would not be stable under substitution. This, in turn, threatens the preservation theorem.

Thus, we must identify repeated type family applications and flatten these to the *same* variable. In this way, $F\ (G\ Int)\ (G\ Int)$ is flattened to $F\ x\ x$ (never $F\ x\ y$), will be apart from the first equation, and will be able to simplify to $Bool$, as desired.

With these considerations in mind, we can now give our implementation of the apartness check:

Definition 5 (Flattening). *To flatten a type τ into τ', written $\tau' = \mathsf{flatten}(\tau)$, process the type τ in a top-down fashion, replacing every type family application with a type variable. Two or more syntactically identical type family applications are flattened to the* *same variable; distinct type family applications are flattened to distinct fresh variables.*

Definition 6 (Apartness)**.** *To test for* $\mathsf{apart}(\rho, \tau)$*, let* $\tau' = \mathsf{flatten}(\tau)$ *and check* $\mathsf{unify}(\rho, \tau')$*. If this unification fails, then* ρ *and* τ *are* apart*. More succinctly:* $\mathsf{apart}(\rho, \tau) = \neg\mathsf{unify}(\rho, \mathsf{flatten}(\tau))$.

We can show that this definition does indeed satisfy the identified necessary properties from Section 3.2. In Section 5.1 we will also identify the *sufficient* conditions for type soundness for *any* possible type-safe implementation of apartness, show that these conditions imply the properties identified in the previous section (a useful sanity check!) and prove that the definition of apartness that we just proposed meets these sufficient conditions.

3.4 Allowing more reductions with compatibility

Checking for apartness in previous equations might be unnecessarily restrictive. Consider this code, which uses the function And from Section 2.1:

```
f  :: T a → T b → T (And a b)
tt :: T True

g :: T a → T a
g x = f x tt
```

Will the definition of g type-check? Alas no: the call $(f\ x\ tt)$ returns a result of type $T\ (\mathit{And}\ a\ \mathit{True})$, and that matches neither of the equations for And. Perhaps we can fix this by adding an equation to the definition of And, thus:

```
type family And (a :: Bool) (b :: Bool) :: Bool where
  And True True = True    -- (1)
  And a    True = a       -- (2)
```

```
    And a    b    = False   -- (3)
```

But that does not work either: the target (*And a True*) matches (2) *but is not apart from (1)*, so (2) cannot fire. And yet we would *like* to be able to simplify (*And a True*) to *a*, as Eqn (2) suggests. Why should this be sound? Because anything that matches both (1) and (2) will reduce to *True* using either equation. We say that the two equations *coincide* on these arguments. When such a coincidence happens, the apartness check is not needed.

We can easily formalize this intuition. Let us say that two equations are *compatible* when any type that matches both left-hand sides would be rewritten by both equations to the same result, eliminating non-convergent critical pairs in the induced rewriting system:

Definition 7 (Compatibility)**.** *Two type-family equations p and q are* compatible *iff* $\Omega_1(lhs_p) = \Omega_2(lhs_q)$ *implies* $\Omega_1(rhs_p) = \Omega_2(rhs_q)$.

For example, (1) and (2) are compatible because a type, such as *And True True*, would be rewritten by both to the same type, namely *True*. It is easy to test for compatibility:

Definition 8 (Compatibility implementation)**.** *The test for compatibility, written* $\mathsf{compat}(p, q)$*, checks that* $\mathsf{unify}(lhs_p, lhs_q) = \Omega$ *implies* $\Omega(rhs_p) = \Omega(rhs_q)$*. If* $\mathsf{unify}(lhs_p, lhs_q)$ *fails,* $\mathsf{compat}(p, q)$ *holds vacuously.*

The proof that $\mathsf{compat}(p, q)$ implies that p and q are compatible appears in Appendix G and is straightforward. We can now state our final simplification rule for closed type families:

Rule 9 (Closed type family simplification)**.** *An equation q of a closed type family can be used to simplify a target application $F\,\overline{\tau}$ if the following conditions hold:*

1. *The target $\overline{\tau}$ matches the type pattern lhs_q.*
2. *For each earlier equation p, either $\mathsf{compat}(p, q)$ or $\mathsf{apart}(lhs_p, \overline{\tau})$.*

For example, we can fire equation (2) on a target that is not apart from (1), because (1) and (2) are compatible. We show that Rule 9 is sufficient for establishing type soundness in Section 5.

Through this use of compatibility, we allow for a limited form of theorem proving within a closed type family definition. The fact that equation (2) is compatible with (1) essentially means that the rewrite rule for (2) is admissible given that for (1). By being able to write such equations in the closed type family definition, we can expand Haskell's definitional equality to relate more types.

3.5 Optimized matching

In our original Candidate Rule 2 above, when simplifying a target $F\,\overline{\tau}$ with an equation q, we are obliged to check $\mathsf{apart}(lhs_p, \overline{\tau})$, *for every earlier equation p*. But much of this checking is wasted duplication. For example, consider

```
type family F a where
  F Int  = Char   -- (1)
  F Bool = Bool   -- (2)
  F x    = Int    -- (3)
```

If a target matches (2) there is really no point in checking its apartness from (1), because *anything* that matches (2) will be apart from (1). We need only check that the target is apart from any preceding equations that could possibly match the same target.

Happily, this intuition is already embodied in our new simplification Rule 9. This rule checks $\mathsf{compat}(p, q) \vee \mathsf{apart}(lhs_p, \overline{\tau})$ for each preceding equation p. But we can *precompute* $\mathsf{compat}(p, q)$ (since it is independent of the target), and in the simplification rule we need check apartness only for the pre-computed list of earlier incompatible equations. In our example, equations (1) and (2) are vacuously compatible, since their left-hand sides do not unify, and hence no type can match both. Thus, there is no need to check for apartness from (1) of a target matching (2).

3.6 Compatibility for open families

As discussed in the introduction, **type instance** declarations for open type families must not overlap. With our definition of compatibility, however, we can treat open and closed families more uniformly by insisting that any two instances of the same open type family are compatible:

Definition 10 (Open type family overlap check)**.** *Every pair of equations p and q for an open type family F must satisfy* $\mathsf{compat}(p, q)$.

Notice that this definition also allows for coincident right-hand sides (as in the case for closed type families, Section 3.4). For example, these declarations are legal:

```
type family Coincide a b
type instance Coincide Int b      = Int
type instance Coincide a   Bool = a
```

These equations overlap, but in the region of overlap they always produce the same result, and so they should be allowed. (GHC already allowed this prior to our extensions.)

3.7 Type inference for closed type families

Given the difficulty of type inference for open type families (Chakravarty et al. 2005; Schrijvers et al. 2008), how do we deal with closed ones? Thankfully, this turns out to be remarkably easy: we simply use Rule 9 to simplify closed families in exactly the same stage of type inference that we would simplify an open one. The implementation in GHC is accordingly quite straightforward.

Despite the ease of implementation, there are perhaps complex new possibilities opened by the use of closed families—these are explored in Section 7.6.

4. System µFC: formalizing the problem

Thus far we have argued informally. In this section we formalize our design and show that it satisfies the usual desirable properties of type preservation and progress, assuming termination of type family reduction. It is too hard to formulate these proofs for all of Haskell, so instead we formalize µFC, a small, explicitly-typed lambda calculus. This is more than a theoretical exercise: GHC really does elaborate all of Haskell into System FC (Sulzmann et al. 2007a; Weirich et al. 2013), of which µFC is a large subset that omits some details of FC—such as kind polymorphism (Yorgey et al. 2012)—that are irrelevant here.

4.1 System µFC

System µFC is an extension of System F, including kinds and explicit equality coercions. Its syntax is presented in Figure 2. This syntax is very similar to recent treatments of System FC (Weirich et al. 2013). We omit from the presentation the choice of ground types and their constructors and destructors, as they are irrelevant for our purposes.

There are a few points to note about type families, all visible in Figure 2. A type family has a particular arity, and always appears saturated in types. That explains the first-order notation $F(\overline{\kappa}){:}\kappa'$ in ground contexts Σ, and $F(\overline{\tau})$ in types.

A closed type family appears in μFC as a kind signature $F(\overline{\kappa}){:}\kappa'$, and a single *axiom* $C{:}\Psi$, both in the top-level ground context Σ. The "type" Ψ of the axiom is a list of equations, each of form $[\overline{\alpha{:}\kappa}].\ F(\overline{\tau}) \sim \sigma$, just as we have seen before except that the quantification is explicit. For example, the axiom for Equal (restricted for simplicity to kind $\star$) looks like this:

$$\begin{array}{rr} \mathsf{axiomEq} : & [\alpha{:}\star].(\mathsf{Equal}\,\alpha\,\alpha) \sim \mathsf{True} \quad ; \\ & [\alpha{:}\star, \beta{:}\star].(\mathsf{Equal}\,\alpha\,\beta) \sim \mathsf{False} \end{array}$$

Although our notation for lists does not make it apparent, we restrict the form of the equations to require that F refers to only one type family—that is, there are no independent F_i. We use subscripts on metavariables to denote which equation they refer to, and we refer to the types $\overline{\rho_i}$ as the *type patterns* of the i'th equation. We assume that the variables $\overline{\alpha}$ bound in each equation are distinct from the variables bound in other equations.

An open type family appears as a kind signature and zero or more separate axioms, each with one equation.

4.2 Static semantics

Typing in μFC is given by the judgments in Figure 3. Most of the rules are uninteresting and are thus presented in Appendix C. The typing rules for expressions are entirely straightforward. The only

Expressions:

$$\begin{array}{lcll}
e & ::= & x \mid \lambda x{:}\tau.e \mid e_1\ e_2 \mid \Lambda\alpha{:}\kappa.e \mid e\,\tau & \\
 & \mid & e \triangleright \gamma & \text{Cast} \\
 & \mid & \ldots & \text{Constructors and destructors of datatypes}
\end{array}$$

Types:

$$\begin{array}{lcll}
\tau,\sigma, & ::= & \alpha \mid \tau_1 \to \tau_2 \mid \forall\,\alpha{:}\kappa.\tau & \\
\psi,\upsilon & \mid & \tau_1\,\tau_2 & \text{Application} \\
 & \mid & F(\overline{\tau}) & \text{Saturated type family} \\
 & \mid & H & \text{Datatype, such as } \mathit{Int}
\end{array}$$

ρ denotes a type pattern (with no type families)

$$\begin{array}{lcll}
\kappa & ::= & \star \mid \kappa_1 \to \kappa_2 & \text{Kinds}
\end{array}$$

Propositions:

$$\begin{array}{lcll}
\phi & ::= & \tau_1 \sim \tau_2 & \text{Equality propositions} \\
\Phi & ::= & [\overline{\alpha{:}\kappa}].\ F(\overline{\rho}) \sim \sigma & \text{Axiom equations} \\
\Psi & ::= & \overline{\Phi} & \text{List of axiom eqns. (axiom types)}
\end{array}$$

Coercions:

$$\begin{array}{lcll}
\gamma,\eta & ::= & \gamma_1 \to \gamma_2 \mid \forall\,\alpha{:}\kappa.\gamma \mid \gamma_1\,\gamma_2 \mid F(\overline{\gamma}) & \\
 & \mid & \langle\tau\rangle & \text{Reflexivity} \\
 & \mid & \mathbf{sym}\,\gamma & \text{Symmetry} \\
 & \mid & \gamma_1 \mathbin{;} \gamma_2 & \text{Transitivity} \\
 & \mid & \mathbf{left}\,\gamma & \text{Left decomposition} \\
 & \mid & \mathbf{right}\,\gamma & \text{Right decomposition} \\
 & \mid & C[i]\,\overline{\tau} & \text{Axiom application}
\end{array}$$

Contexts:

Ground: $\Sigma ::= \cdot \mid \Sigma, H{:}\overline{\kappa} \to \star \mid \Sigma, F(\overline{\kappa}){:}\kappa' \mid \Sigma, C{:}\Psi$

Variables: $\Delta ::= \cdot \mid \Delta, x{:}\tau \mid \Delta, \alpha{:}\kappa$

Combined: $\Gamma ::= \Sigma; \Delta$

Substitutions: $\Omega ::= [\overline{\alpha \mapsto \tau}]$

Figure 2. The grammar of System μFC

$\Gamma \vdash_{\mathsf{tm}} e : \tau$	Expression typing
$\Gamma \vdash_{\mathsf{ty}} \tau : \kappa$	Type kinding
$\Gamma \vdash_{\mathsf{co}} \gamma : \phi$	Coercion typing
$\vdash_{\mathsf{gnd}} \Sigma$	Ground context validity
$\Sigma \vdash_{\mathsf{var}} \Delta$	Variables context validity
$\vdash_{\mathsf{ctx}} \Gamma$	Context validity

Figure 3. Typing judgments for System μFC

noteworthy rule is the one for casting, which gives the raison d'être for coercions:

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \sim \tau_2 \qquad \Gamma \vdash_{\mathsf{tm}} e : \tau_1}{\Gamma \vdash_{\mathsf{tm}} e \triangleright \gamma : \tau_2} \quad \textsc{Tm\_Cast}$$

Here, we see that a cast by a coercion changes the type of an expression. This is what we mean by saying that a coercion witnesses the equality of two types—if there is a coercion between τ_1 and τ_2, then any expression of type τ_1 can be cast into one of type τ_2.

The rules for deriving the kind of a type are straightforward and are omitted from this presentation.

4.3 Coercions and axiom application

Coercions are less familiar, so we present the coercion typing rules in full, in Figure 4. The first four rules say that equality is *congruent*—that is, types can be considered equal when they are formed of components that are considered equal. The following three rules assert that coercibility is a proper equivalence relation. The CO_LEFT and CO_RIGHT rules assert that we can decompose complex equal-

$\boxed{\Gamma \vdash_{\mathsf{co}} \gamma : \phi}$ Coercion typing

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{co}} \gamma_1 : \tau_1 \sim \tau_1' \qquad \Gamma \vdash_{\mathsf{co}} \gamma_2 : \tau_2 \sim \tau_2' \\ \Gamma \vdash_{\mathsf{ty}} \tau_1 \to \tau_2 : \star\end{array}}{\Gamma \vdash_{\mathsf{co}} \gamma_1 \to \gamma_2 : (\tau_1 \to \tau_2) \sim (\tau_1' \to \tau_2')} \quad \textsc{Co\_Arrow}$$

$$\frac{\Gamma, \alpha{:}\kappa \vdash_{\mathsf{co}} \gamma : \tau_1 \sim \tau_2 \qquad \Gamma \vdash_{\mathsf{ty}} \forall\, \alpha{:}\kappa.\tau_1 : \star}{\Gamma \vdash_{\mathsf{co}} \forall\, \alpha{:}\kappa.\gamma : (\forall\, \alpha{:}\kappa.\tau_1) \sim (\forall\, \alpha{:}\kappa.\tau_2)} \quad \textsc{Co\_Forall}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{co}} \gamma_1 : \tau_1 \sim \sigma_1 \qquad \Gamma \vdash_{\mathsf{co}} \gamma_2 : \tau_2 \sim \sigma_2 \\ \Gamma \vdash_{\mathsf{ty}} \tau_1\, \tau_2 : \kappa\end{array}}{\Gamma \vdash_{\mathsf{co}} \gamma_1\, \gamma_2 : (\tau_1\, \tau_2) \sim (\sigma_1\, \sigma_2)} \quad \textsc{Co\_App}$$

$$\frac{\begin{array}{c}\overline{\Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \sim \tau_2} \\ \Gamma \vdash_{\mathsf{ty}} F(\overline{\tau_1}) : \kappa\end{array}}{\Gamma \vdash_{\mathsf{co}} F(\overline{\gamma}) : F(\overline{\tau_1}) \sim F(\overline{\tau_2})} \quad \textsc{Co\_TyFam}$$

$$\frac{\Gamma \vdash_{\mathsf{ty}} \tau : \kappa}{\Gamma \vdash_{\mathsf{co}} \langle \tau \rangle : \tau \sim \tau} \quad \textsc{Co\_Refl}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \sim \tau_2}{\Gamma \vdash_{\mathsf{co}} \mathbf{sym}\, \gamma : \tau_2 \sim \tau_1} \quad \textsc{Co\_Sym}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma_1 : \tau_1 \sim \tau_2 \qquad \Gamma \vdash_{\mathsf{co}} \gamma_2 : \tau_2 \sim \tau_3}{\Gamma \vdash_{\mathsf{co}} \gamma_1 \fatsemi \gamma_2 : \tau_1 \sim \tau_3} \quad \textsc{Co\_Trans}$$

$$\frac{\begin{array}{l} \Gamma \vdash_{\mathsf{co}} \gamma : \tau_1\, \tau_2 \sim \sigma_1\, \sigma_2 \\ \Gamma \vdash_{\mathsf{ty}} \tau_1 : \kappa \qquad \Gamma \vdash_{\mathsf{ty}} \sigma_1 : \kappa \end{array}}{\Gamma \vdash_{\mathsf{co}} \mathbf{left}\, \gamma : \tau_1 \sim \sigma_1} \quad \textsc{Co\_Left}$$

$$\frac{\begin{array}{l} \Gamma \vdash_{\mathsf{co}} \gamma : \tau_1\, \tau_2 \sim \sigma_1\, \sigma_2 \\ \Gamma \vdash_{\mathsf{ty}} \tau_2 : \kappa \qquad \Gamma \vdash_{\mathsf{ty}} \sigma_2 : \kappa \end{array}}{\Gamma \vdash_{\mathsf{co}} \mathbf{right}\, \gamma : \tau_2 \sim \sigma_2} \quad \textsc{Co\_Right}$$

$$\frac{\begin{array}{l} C{:}\Psi \in \Sigma \qquad \Psi = \overline{[\overline{\alpha{:}\kappa}].\ F(\overline{\rho}) \sim \upsilon} \\ \overline{\Sigma; \Delta \vdash_{\mathsf{ty}} \tau : \kappa_i} \qquad \vdash_{\mathsf{ctx}} \Sigma; \Delta \\ \forall\, j < i, \mathsf{no\_conflict}(\Psi, i, \overline{\tau}, j) \end{array}}{\Sigma; \Delta \vdash_{\mathsf{co}} C[i]\ \overline{\tau} : F(\overline{\rho_i[\overline{\tau/\alpha_i}]}) \sim \upsilon_i[\overline{\tau/\alpha_i}]} \quad \textsc{Co\_Axiom}$$

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

$$\frac{\Psi = \overline{[\overline{\alpha{:}\kappa}].\ F(\overline{\rho}) \sim \upsilon} \qquad \mathsf{apart}(\overline{\rho_j}, \overline{\rho_i[\overline{\tau/\alpha_i}]})}{\mathsf{no\_conflict}(\Psi, i, \overline{\tau}, j)} \quad \textsc{NC\_Apart}$$

$$\frac{\mathsf{compat}(\Psi[i], \Psi[j])}{\mathsf{no\_conflict}(\Psi, i, \overline{\tau}, j)} \quad \textsc{NC\_Compatible}$$

$\boxed{\mathsf{compat}(\Phi_1, \Phi_2)}$ Equation compatibility

$$\frac{\begin{array}{l}\Phi_1 = [\overline{\alpha_1{:}\kappa_1}].\ F(\overline{\rho_1}) \sim \upsilon_1 \\ \Phi_2 = [\overline{\alpha_2{:}\kappa_2}].\ F(\overline{\rho_2}) \sim \upsilon_2 \\ \mathsf{unify}(\overline{\rho_1}, \overline{\rho_2}) = \Omega \\ \Omega(\upsilon_1) = \Omega(\upsilon_2)\end{array}}{\mathsf{compat}(\Phi_1, \Phi_2)}\ \textsc{Compat\_Coincident}$$

$$\frac{\begin{array}{l}\Phi_1 = [\overline{\alpha_1{:}\kappa_1}].\ F(\overline{\rho_1}) \sim \upsilon_1 \\ \Phi_2 = [\overline{\alpha_2{:}\kappa_2}].\ F(\overline{\rho_2}) \sim \upsilon_2 \\ \mathsf{unify}(\overline{\rho_1}, \overline{\rho_2}) \text{ fails}\end{array}}{\mathsf{compat}(\Phi_1, \Phi_2)}\ \textsc{Compat\_Distinct}$$

Figure 4. Coercion formation rules

ities to simpler ones. These formation rules are incomplete with respect to some unspecified notion of *semantic* equality—that is, we can imagine writing down two types that we "know" are equal, but for which no coercion is derivable. For example, there is no way to use induction over a data structure to prove equality. However, recall that these coercions must all be inferred from a source program, and it is unclear how we would reliably infer inductive coercions.

The last rule of coercion formation, Co_Axiom, is the one that we are most interested in. The coercion $C[i]\ \overline{\tau}$ witnesses the equality obtained by instantiating the i'th equation of axiom C with the types $\overline{\tau}$. For example,

$$\mathsf{axiomEq}[0]\ \mathsf{Int} : \mathsf{Equal}\ \mathsf{Int}\ \mathsf{Int} \sim \mathsf{True}$$

This says that if we pick the first equation of *axiomEq* (we index from 0), and instantiate it at *Int*, we have a witness for *Equal Int Int* $\sim$ *True*.

Notice that the coercion $C[i]\,\overline{\tau}$ specifies exactly which equation is picked (the i'th one); μFC is a fully-explicit language. However, the typing rules for μFC must reject unsound coercions like

$$\mathsf{axiomEq}[1]\ \mathsf{Int}\ \mathsf{Int} : \mathsf{Equal}\ \mathsf{Int}\ \mathsf{Int} \sim \mathsf{False}$$

and that is expressed by rule CO_AXIOM. The premises of the rule check to ensure that $\Sigma; \Delta$ is a valid context and that all the types $\overline{\tau}$ are of appropriate kinds to be applied in the i'th equation. The last premise implements Rule 9 (Section 3.4), by checking no_conflict for each preceding equation j. The no_conflict judgment simply checks that *either* (NC_COMPATIBLE) the i'th and j'th equation for C are compatible, *or* (NC_APART) that the target is apart from the LHS of the j'th equation, just as in Rule 9.

In NC_COMPATIBLE, note that the compat judgment does not take the types $\overline{\tau}$: compatibility is a property of equations, and is independent of the specific arguments at an application site. The two rules for compat are exactly equivalent to Definition 8.

These judgments refer to algorithms apart and unify. We assume a correct implementation of unify and propose sufficient properties of apart in Section 5.1. We then show that our chosen algorithm for apart (Definition 6) satisfies these properties.

As a final note, the rules do not check the closed type family axioms for exhaustiveness. A type-family application that matches no axiom simply does not reduce. Adding an exhaustiveness check based on the kind of the arguments of the type family might be a useful, but orthogonal, feature.

5. Metatheory

A summary of the structure of the type safety proof, highlighting the parts that are considered in this paper, is in Figure 5. Our main goals are to prove (i) the substitution lemma of types into coercions (Section 5.2), and (ii) a consistency property that ensures we never equate two types such as *Int* and *Bool* (Section 5.3). The substitution and consistency lemmas lead to the preservation and progress theorems respectively, which together ensure type safety. We omit the operational semantics of μFC as well as the other lemmas in the main proofs of preservation and progress, because these are all direct adaptations from previous work (Weirich et al. 2011; Sulzmann et al. 2007a).

We stress that, as Figure 5 indicates, we have proved type safety only for *terminating* type families. What exactly does that mean? We formally define the rewrite relation, now written $\Sigma \vdash \cdot \rightsquigarrow \cdot$ to explicit mention the set of axioms, with the following rule:

$$\frac{\begin{array}{lll} C{:}\Psi \in \Sigma & \Psi = \overline{[\overline{\alpha{:}\kappa}].\, F(\overline{\rho}) \sim \upsilon} & \\ \vdash_{\mathsf{gnd}} \Sigma & \overline{\tau} = \rho_i[\overline{\psi/\alpha_i}] & \tau' = \upsilon_i[\overline{\psi/\alpha_i}] \\ \multicolumn{3}{l}{\forall j < i, \mathsf{no\_conflict}(\Psi, i, \overline{\psi}, j)} \end{array}}{\Sigma \vdash \mathcal{C}[F(\overline{\tau})] \rightsquigarrow \mathcal{C}[\tau']} \quad \textsc{Red}$$

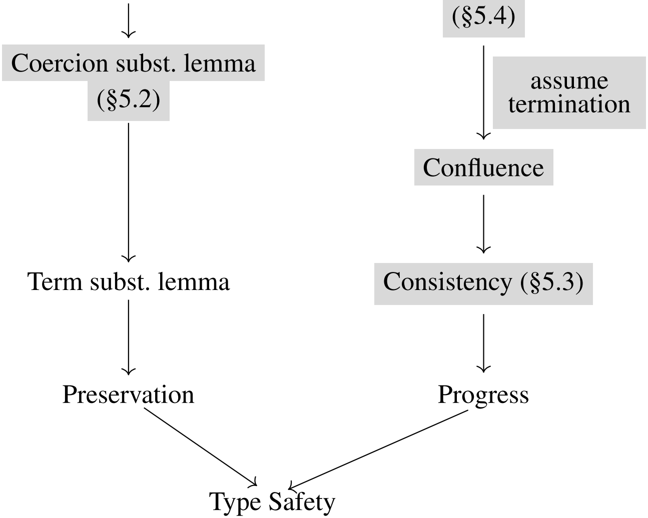


Figure 5. Structure of type safety proof. The arrows represent implications. The nodes highlighted in gray are the parts considered in the present work.

In the conclusion of this rule, $\mathcal{C}[\cdot]$ denotes a type context with exactly one hole. Its use in the rule means that a type family can simplify *anywhere* within a type. Note that the no_conflict premise of this rule is identical to that of the CO_AXIOM rule. By “terminating type families” we mean that the $\Sigma \vdash \cdot \rightsquigarrow \cdot$ relation cannot have infinite chains. We discuss non-terminating type families in Section 6.

As a notational convention, we extend the relation to lists of types by using $\Sigma \vdash \overline{\tau_1} \rightsquigarrow \overline{\tau_2}$ to mean that exactly one of the types in $\overline{\tau_1}$ steps to the corresponding type in $\overline{\tau_2}$; in all other positions $\overline{\tau_1}$ and $\overline{\tau_2}$ are identical.

5.1 Preliminaries: properties of unification and apartness

In order to prove properties about $\mathsf{no\_conflict}$, we must assume the correctness of the unification algorithm:

Property 11 (unify correct)**.** *If there exists a substitution Ω such that $\Omega(\overline{\sigma}) = \Omega(\overline{\tau})$, then $\mathsf{unify}(\overline{\sigma}, \overline{\tau})$ succeeds. If $\mathsf{unify}(\overline{\sigma}, \overline{\tau}) = \Omega$ then Ω is a most general unifier of $\overline{\sigma}$ and $\overline{\tau}$.*

In Section 3.2, we gave some *necessary* properties of apart, namely Properties 2 and 4. To prove type soundness we need *sufficient* properties, such as the following three. *Any* implementation of apart that has these three properties would lead to type safety. We prove (in Appendix F) that the given algorithm for apart (Definition 6) satisfies these properties. Due to flattening in the definition of apart, this proof is non-trivial. As a sanity check, we also prove that the sufficient properties imply the necessary ones of Section 3.2.

Property 12 (Apartness is stable under type substitution)**.** *If $\mathsf{apart}(\overline{\rho}, \overline{\tau})$, then for all substitutions Ω, $\mathsf{apart}(\overline{\rho}, \Omega(\overline{\tau}))$.*

Property 13 (No unifiers for apart types)**.** *If $\mathsf{apart}(\overline{\rho}, \overline{\tau})$, then there exists no substitution Ω such that $\Omega(\overline{\rho}) = \Omega(\overline{\tau})$.*

The final property of the apartness check is the most complex. It ensures that, if an equation can fire for a given target and that target steps, then it is possible to simplify the reduct even further so that the same equation can fire on the final reduct.

Property 14 (Apartness can be regained after reduction)**.** *If* $\overline{\tau} = \Omega(\overline{\rho})$ *and* $\Sigma \vdash \overline{\tau} \rightsquigarrow \overline{\tau'}$*, then there exists a* $\overline{\tau''}$ *such that*

1. $\Sigma \vdash \overline{\tau'} \rightsquigarrow^{*} \overline{\tau''}$,
2. $\overline{\tau''} = \Omega'(\overline{\rho})$ *for some* Ω'*, and*
3. *for every* $\overline{\rho'}$ *such that* $\mathsf{apart}(\overline{\rho'}, \overline{\tau})$*:* $\mathsf{apart}(\overline{\rho'}, \overline{\tau''})$.

Here is an example of Property 14 in action. Consider the following type families F and G:

```
type family F a where
  F (Int, Bool) = Char   -- (A)
  F (a,   a)    = Bool   -- (B)
type family G x where G Int = Double
```

Suppose that our target is $F\ (G\ \mathit{Int}, G\ \mathit{Int})$, and that our particular implementation of apart allows equation (B) to fire; that is, $\mathsf{apart}((\mathit{Int}, \mathit{Bool}), (G\ \mathit{Int}, G\ \mathit{Int}))$. Now, suppose that instead of firing (B) we chose to reduce the first $G\ \mathit{Int}$ argument to Double. The new target is now $F\ (\mathit{Double}, G\ \mathit{Int})$. Now (B) cannot fire, because the new target simply does not match (B) any more. Property 14 ensures that there exist further reductions on the new target that make (B) firable again—in this case, stepping the second $G\ \mathit{Int}$ to Double does the job. Conditions (2) and (3) of Property 14 formalize the notion “make (B) firable again”.

5.2 Type substitution in coercions

System μFC enjoys a standard term substitution lemma. This lemma is required to prove the preservation theorem. As shown in Figure 5, the term substitution lemma depends on the substitution lemma for coercions. We consider only the case of interest here, that of substitution in the rule Co_Axiom.

Lemma 15 (Co_Axiom Substitution)**.** *If* $\Sigma; \Delta, \beta{:}\kappa, \Delta' \vdash_{\mathsf{co}} C[i]\,\overline{\tau} : F(\overline{\rho_i[\overline{\tau/\alpha_i}]}) \sim \upsilon_i[\overline{\tau/\alpha_i}]$ *and* $\Sigma; \Delta \vdash_{\mathsf{ty}} \sigma : \kappa$*, then* $\Sigma; \Delta, \Delta'[\sigma/\beta] \vdash_{\mathsf{co}} C[i]\ \overline{\tau[\sigma/\beta]} : F(\overline{\rho_i[\overline{\tau/\alpha_i}][\sigma/\beta]}) \sim \upsilon_i[\overline{\tau/\alpha_i}][\sigma/\beta]$.

The proof of this lemma, presented in Appendix D, proceeds by case analysis on the no_conflict judgment. It requires the use of the (standard) type substitution lemma and Property 12, but is otherwise unremarkable.

5.3 Consistency

As discussed at the beginning of this section, to establish progress we must show *consistency*. Consistency ensures that we can never deduce equalities between distinct *value types*, denoted with ξ:

$$\xi \quad ::= \quad H\,\overline{\tau} \mid \tau_1 \to \tau_2 \mid \forall\,\alpha{:}\kappa.\tau$$

For example, Int, Bool, and $\forall\,\alpha{:}\star.\alpha \to \alpha$ are all value types. A set of axioms is consistent if we cannot deduce bogus equalities like $\mathit{Int} \sim \mathit{Bool}$ or $\mathit{Int} \sim \forall\,\alpha{:}\star.\alpha \to \alpha$:

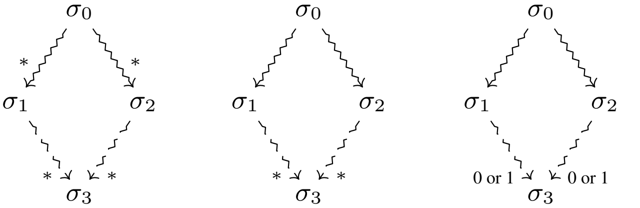
Definition 16 (Consistent contexts)**.** *A ground context* Σ *is* consistent *if, for all coercions* γ *such that* $\Sigma; \cdot \vdash_{\mathsf{co}} \gamma : \xi_1 \sim \xi_2$*:*

1. *if* $\xi_1 = H\,\overline{\tau_1}$*, then* $\xi_2 = H\,\overline{\tau_2}$*,*
2. *if* $\xi_1 = \tau_1 \to \tau_1'$*, then* $\xi_2 = \tau_2 \to \tau_2'$*, and*
3. *if* $\xi_1 = \forall\,\alpha{:}\kappa.\tau_1$*, then* $\xi_2 = \forall\,\beta{:}\kappa.\tau_2$*.*

How can we check whether an axiom set is consistent? It is extremely hard to do so in general, so instead, following previous work (Weirich et al. 2011), we place syntactic restrictions on the axioms that conservatively guarantee consistency. A set of axioms that pass this check are said to be **Good**. We then prove the consistency lemma:

Lemma 17 (Consistency)**.** *If* $\mathbf{Good}\,\Sigma$*, then* Σ *is consistent.*

Following previous proofs, we show that if $\mathbf{Good}\,\Sigma$ and $\Sigma; \cdot \vdash_{\mathsf{co}} \gamma : \sigma_1 \sim \sigma_2$, then σ_1 and σ_2 have a common reduct in the $\rightsquigarrow$ relation. Because the simplification relation preserves type constructors on the heads of types, we may conclude that Σ is consistent.



(a) Confluence (b) Local confluence (c) Local diamond

Figure 6. Graphical representation of confluence properties. A solid line is a universally quantified input, and a dashed line is an existentially quantified output.

However, one of the cases in this argument is transitivity: the joinability relation must be transitive. That is, if τ_1 and τ_2 have a common reduct σ_1, and if τ_2 and τ_3 have a common reduct σ_2, then τ_1 and τ_3 must have a common reduct (they are *joinable*). To show transitivity of joinability, we must show confluence of the rewrite relation, in order to find the common reduct of σ_1 and σ_2 (which share τ_2 as an ancestor).

Our approach to this problem is to show *local confluence* (see Figure 6) and then use Newman's Lemma (1942) to get full confluence. Newman's Lemma requires that the rewrite system is terminating—this is where the assumption of termination is used.

The full, detailed proof appears in Appendix E.

5.4 Good contexts

What sort of checks should be in our syntactic conditions, $\mathbf{Good}$? We would like $\mathbf{Good}$ to be a small set of common-sense conditions for a type reduction system, such as the following:

Definition 18 ($\mathbf{Good}$ contexts)**.** *We have* $\mathbf{Good}\,\Sigma$ *whenever the following four conditions hold:*

1. *For all* $C{:}\Psi \in \Sigma$*:* Ψ *is of the form* $\overline{[\overline{\alpha{:}\kappa}].\ F(\overline{\rho}) \sim \upsilon}$ *where all of the* F_i *are the same type family* F *and all of the type patterns* $\overline{\rho_i}$ *do not mention any type families.*
2. *For all* $C{:}\Psi \in \Sigma$ *and equations* $[\overline{\alpha{:}\kappa}].\ F(\overline{\rho}) \sim \upsilon$ *in* Ψ*: the variables* $\overline{\alpha}$ *all appear free at least once in* $\overline{\rho}$*.*
3. *For all* $C{:}\Psi \in \Sigma$*: if* Ψ *defines an axiom over a type family* F *and has multiple equations, then no other axiom* $C'{:}\Psi' \in \Sigma$ *defines an axiom over* F*. That is, all type families with ordered equations are closed.*
4. *For all* $C_1{:}\Phi_1 \in \Sigma$ *and* $C_2{:}\Phi_2 \in \Sigma$ *(each with only one*

equation), $\mathsf{compat}(\Phi_1, \Phi_2)$*. That is, among open type families, the patterns of distinct equations do not overlap.*

The clauses of the definition of $\mathbf{Good}$ are straightforward syntactic checks. In fact, these conditions are exactly what GHC checks for when compiling type family instances. This definition of $\mathbf{Good}$ leads to the proof of Lemma 39, as described above.

6. Non-terminating type families

By default GHC checks every type family for termination, to guarantee that the type checker will never loop. Any such check is necessarily conservative; indeed, GHC rejects the *TMember* function of Section 2.4 (Schrijvers et al. 2008). Although GHC's test could readily be improved, any conservative check limits expressiveness or convenience, so GHC allows the programmer to disable

```
type instance A = C A
type instance C x = D x (C x)
type instance D x x = Int
```

$$\begin{array}{ll} (1) & A \leadsto C\ A \leadsto D\ A\ (C\ A) \leadsto D\ (C\ A)\ (C\ A) \leadsto Int \\ (2) & A \leadsto C\ A \leadsto^{*}_{\text{by (1)}} C\ Int \end{array}$$

Int and *C Int* have no common reduct.

Figure 7. Counter-example to confluence

the check. This may make the type checker loop, but *it should not threaten soundness*.

However, the soundness result of Section 5 covers only terminating type families. Surprisingly (to us) non-termination really does lead to a soundness problem (Section 6.1). We propose a solution that (we believe) rules out this problem (Section 6.2), but explain why the main result of this paper is difficult to generalize to non-terminating type families, leaving an open problem for further work.

6.1 The problem with infinity

Consider this type family, adapted from Huet (1980):

```
type family D x where
  D ([b], b) = Bool
  D (c,   c) = Int
```

We wish to simplify the target $D\ (a, a)$. The type (a, a) matches the second pattern (c, c), but is it apart from the first pattern $([b], b)$? Definition 6 asserts that they *are* apart since they do not unify: unification fails with an occurs check error. Accordingly, Rule 9 would simplify $D\ (a, a)$ to Int. But consider the following definitions, where type family $Loop$ is a nullary (0-argument) type family:

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

If we instantiate a with $Loop$ we get $(Loop, Loop)$ which *can* simplify to $([Loop], Loop)$. The latter *does match* the pattern $([b], b)$, violating Property 4, a necessary condition for soundness.

So, in a non-terminating system our apartness check is unsound. Concretely, using our apartness implementation from Definition 6, we can equate types Int and Bool, thus:

$$\mathit{Int} \sim D\ (\mathit{Loop}, \mathit{Loop}) \sim D\ ([\mathit{Loop}], \mathit{Loop}) \sim \mathit{Bool}$$

Conclusion: we must not treat (a, a) as apart from the pattern $([b], b)$, *even though they do not unify*. In some ways this is not so surprising. In our earlier examples, apartness was based on an explicit contradiction ("a Bool cannot be an Int"), but here unification fails only because of an occurs check. As the Loop example shows, allowing non-terminating type-family definitions amounts to introducing infinite types, and if we were to allow infinite types, then (a, a) *does* unify with $([b], b)$!

6.2 Fixing the problem

The problem with the current apartness check is that finite unification fails too often. We need to replace the unification test in the definition of apartness with unification over *infinite types*:

Definition 19 (Infinite unification)**.** *Two types τ_1, τ_2 are infinitely unifiable, written* $\mathsf{unify}_\infty(\tau_1, \tau_2)$*, if there exists a substitution ω whose range may include* infinite *types, such that $\omega(\tau_1) = \omega(\tau_2)$.*

For example types (a, a) and $([b], b)$ are unifiable with a substitution $\omega = [a \mapsto [[[...]]], b \mapsto [[[...]]]]$. Efficient algorithms

to decide unification over infinite types (and compute most general unifiers) have existed for some time and are based on well-established theory (Huet 1976; Courcelle 1983). See Jaffar (1984) for such an algorithm, and Knight (1989) for a general survey.

We conjecture that replacing all uses of unify with unify$_\infty$ in our definitions guarantees soundness, even in the presence of non-terminating family equations. Alas, this conjecture turns out to be very hard to prove, and touches on open problems in the term-rewriting literature. For example, a rewrite system that has (a) infinite rewrite sequences and (b) non-left-linear patterns, *does not necessarily guarantee confluence*, even if its patterns do not overlap. Figure 7 gives an example, from Klop (1993).

Notice that replacing unify with unify$_\infty$ may change the reduction relation. For example, a target which is apart from a pattern with a unify-based apartness check may no longer be apart from the same pattern with the more conservative unify$_\infty$-based apartness check. Yet, type safety (for terminating axiom sets) is not compromised since Property 11 carries over to unification algorithms over infinite types (Huet 1976).

6.3 Ramifications for open families

We pause briefly to consider the implications for GHC's existing *open* type families. GHC allows the following definition for an open type family D':

```
type family D' x y
type instance D' [b] b = Bool
type instance D' c   c = Int
```

As described in Section 2, the **type instance** equations of an open type family are required to have non-overlapping left-hand sides, and GHC 7.6 believes that the two equations do not overlap because they do not unify. But, using certain flags, GHC also accepts the definition of *Loop*, and the target (D' *Loop Loop*) demonstrates that the combination is unsound precisely as described above.[5]

Happily, if the conjecture of Section 6.2 holds true, we can apply the same fix for open families as we did for closed families: simply use unify_∞ instead of unify when checking for overlap. Indeed, this is exactly how we have corrected this oversight in GHC 7.8.

7. Discussion and Future Work

The study of closed type families opens up a wide array of related issues. This section discusses some of the more interesting points we came across in our work.

7.1 Denotational techniques for consistency

We do not have a proof of consistency for a system with non-terminating, non-left-linear axioms (even when using unify_∞ instead of unify). We have seen that confluence is false, and hence cannot be used as a means to show consistency.

A possible alternative approach to proving consistency—side-stepping confluence—is via a denotational semantics for types. We would have to show that if we can build a coercion γ such that $\Gamma \vdash \gamma : \tau \sim \sigma$, then $[\![\tau]\!] = [\![\sigma]\!]$, for some interpretation of types into a semantic domain. The "obvious" domain for such a semantics, in the presence of non-terminating computations, is the domain that includes $\bot$ as well as finite and infinite trees. Typically in denotational semantics, recursive type families would be interpreted as the limit of approximations of continuous functions. However, the "obvious" interpretation of type families in this simple domain is not monotone. Consider this type family:

[5] Akio Takano has posted an example of how this can cause a program to

fail, at `http://ghc.haskell.org/trac/ghc/ticket/8162`.

type family $F\ a\ b$ **where**
  $F\ x\quad x \qquad\qquad\ = \mathit{Int}$
  $F\ [x]\ (\mathit{Maybe}\ x) = \mathit{Char}$

It is the case that $(\bot \sqsubseteq [\bot])$ and $(\bot \sqsubseteq \mathit{Maybe}\ \bot)$, but the semantic interpretation of F, call it f, should satisfy $f(\bot,\bot) = \mathit{Int}$ and $f([\bot], \mathit{Maybe}\ \bot) = \mathit{Char}$. Hence, monotonicity breaks. The lack of monotonicity means that limits of chains of approximations do not exist, and thus that interpretations of functions, such as f, are ill-defined.

An alternate definition would give $f(\bot,\bot) = \bot$, but then substitutivity breaks. Indeed, the proof theory can deduce that $F\ x\ x$ is equal to Int for *any* type x, even those that have denotation $\bot$.

Alternatively to these approaches, one might want to explore different domains to host the interpretation of types.

7.2 Conservativity of apartness

We note in Section 3.3 that our implementation of apartness is conservative. This conservativity is unavoidable—it is possible for open type families to have instances scattered across modules, and thus the apartness check cannot adequately simplify the types involved in every case. However, the current check considers *none* of the type family axioms available, even if one would inform the apartness check. For example, consider

type family $G\ a$ **where**
  $G\ \mathit{Int} = \mathit{Bool}$

$$G\ [a] = \mathit{Char}$$

and we wish to simplify target $\mathit{Equal}\ \mathit{Double}\ (G\ b)$. It is clear that an application of G can never simplify to Double, so we could imagine a more refined apartness check that could reduce this target to False. We leave the details of such a check to future work.

7.3 Conservativity of coincident overlap: partial knowledge

It is worth noting that the compatibility check (Definition 8) is somewhat conservative. For example, take the type family

$$\begin{array}{l} \textbf{type family } F\ a\ b \textbf{ where} \\ \quad F\ \mathit{Bool}\ c = \mathit{Int} \\ \quad F\ d\ \quad\ \ e = e \end{array}$$

Consider a target $F\ g\ \mathit{Int}$. The target matches the second equation, but not the first. But, the simplification rule does not allow us to fire the second equation—the two equations are not compatible, and the target is not apart from the first equation. Yet it clearly *would* be safe to fire the second equation in this case, because even if g turns out to be Bool, the first equation would give the same result.

It would, however, be easy to modify F to allow the desired simplification: just add a new second equation $F\ a\ \mathit{Int} = \mathit{Int}$. This new equation would be compatible with the first one and therefore would allow the simplification of $F\ g\ \mathit{Int}$.

7.4 Conservativity of coincident overlap: requiring syntactic equality

The compatibility check is conservative in a different dimension: it requires syntactic equality of the RHSs after substitution. Consider this tantalizing example:

```
type family Plus a b where
  Plus Zero     a        = a                 -- (A)
  Plus (Succ b) c        = Succ (Plus b c)   -- (B)
  Plus d        Zero     = d                 -- (C)
  Plus e        (Succ f) = Succ (Plus e f)   -- (D)
```

If this type family worked as one would naively expect, it would simplify an addition once *either* argument's top-level constructor were known. (In other dependently typed languages, definitions like this are not possible and require auxiliary lemmas to reduce when the second argument's structure only is known.) Alas, it does not work as well as we would hope. The problem is that not all the equations are compatible. Let's look at (B) and (C). To check if these are compatible, we unify $((\mathit{Succ}\ b), c)$ with (d, Zero) to get $[c \mapsto \mathit{Zero}, d \mapsto \mathit{Succ}\ b]$. The right-hand sides under this substitution are $\mathit{Succ}\ (\mathit{Plus}\ b\ \mathit{Zero})$ and $\mathit{Succ}\ b$. However, these are not syntactically identical, so equations (B) and (C) are not compatible, and a target such as $\mathit{Plus}\ g\ \mathit{Zero}$ is stuck.

Why not just allow reduction in the RHSs before checking for compatibility? Because doing so is not obviously well-founded! Reducing the $\mathit{Succ}\ (\mathit{Plus}\ b\ \mathit{Zero})$ type that occurred during the compatibility check above requires knowing that equations (B) and (C) are compatible, which is exactly what we're trying to establish. So, we require syntactic equality to support compatibility, and leave the more general check for future work.

7.5 Lack of inequality evidence

One drawback of closed type families is that they sometimes do not compose well with generalized algebraic datatypes (GADTs). Consider the following sensible-looking example:

```
data X a where
  XInt  :: X Int
  XBool :: X Bool
  XChar :: X Char

type family Collapse a where
  Collapse Int = Int
  Collapse x   = Char

collapse :: X a → X (Collapse a)
collapse XInt = XInt
collapse _    = XChar
```

The type function *Collapse* takes *Int* to itself and every other type to *Char*. Note the type of the term-level function *collapse*. Its implementation is to match *XInt*—the only constructor of *X* parameterized by *Int*—and return *XInt*; all other constructors become *XChar*. The structure of *collapse* exactly mimics that of *Collapse*. Yet, this code does not compile.

The problem is that the type system has no evidence that, in the second equation for *collapse*, the type variable *a* cannot be *Int*. So, when type-checking the right-hand side *XChar*, it is not type-safe to equate *Collapse a* with *Char*. The source of this problem is that the type system has no notion of *inequality*. If the **case** construct were enhanced to track inequality evidence and axiom application could consider such evidence, it is conceivable that the example above could be made to type-check. Such a notion of inequality has not yet been considered in depth, and we leave it as future work.

7.6 Type inference

The addition of closed type families to Haskell opens up new possibilities in type inference. By definition, the full behavior of a closed type family is known all at once. This closed-world assumption allows the type inference engine to perform more improvement on types than would otherwise be possible. Consider the following type family:

```
type family Inj a where
  Inj Int  = Bool
  Inj Bool = Char
  Inj Char = Double
```

Type inference can discover in this case that *Inj* is indeed an injective type function. When trying to solve a constraint of the form $\mathit{Inj}\ \mathit{Int} \sim \mathit{Inj}\ q$ the type inference engine can deduce that q *must be* equal to *Int* for the constraint to have a solution. By

contrast, if *Inj* were not identified as injective, we would be left with an unsolved constraint as in principle there could be multiple other types for q that could satisfy $\mathit{Inj}\ \mathit{Int} \sim \mathit{Inj}\ q$.

Along similar lines, we can imagine improving the connection between *Equal* and $(\sim)$. Currently, if a proof $a \sim b$ is available, type inference will replace all occurrences of a with b, after which $\mathit{Equal}\ a\ b$ will reduce to *True*. However, the other direction does not work: if the inference engine knows $\mathit{Equal}\ a\ b \sim \mathit{True}$, it will not deduce $a \sim b$. Given the closed definition of *Equal*, though, it seems possible to enhance the inference engine to be able to go both ways.

These deductions are not currently implemented, but remain as compelling future work.

8. Related work

8.1 Previous work on System FC

The proof of type soundness presented in this paper depends heavily on previous work for System FC, first presented by Sulzmann et al. (2007a). That work proves consistency only for terminating type families, as we do here.

In a non-terminating system, local confluence does not imply confluence. Therefore, previous work (Weirich et al. 2011) showed confluence of the rewrite system induced by the (potentially non-terminating) axiom set by establishing a *local diamond* property (see Figure 6). However, the proof took a shortcut: the requirements for good contexts effectively limited all axioms to be left-linear. The local diamond proof relies on the fact that, in a system with linear patterns, matching is preserved under reduction. For instance, consider these axioms:

```
type instance F a b = H a
type instance G Int = Bool
```

The type $F\ (G\ Int)\ (G\ Int)$ matches the equation for F and can potentially simplify to $F\ (G\ Int)\ Bool$ or to $F\ Bool\ (G\ Int)$ or even to $F\ Bool\ Bool$. But, in all cases the reduct *also* matches the very same pattern for F, allowing local diamond property to be true.[6]

What is necessary to support a local diamond property in a system with *closed* type families, still restricted to linear patterns?

We need this property: If $F\ \overline{\tau}$ can reduce by some equation q, and $\overline{\tau} \rightsquigarrow \overline{\tau'}$, then $F\ \overline{\tau'}$ can reduce by that same equation q. With only open families, this property means that matching must be preserved by reduction. With closed families, however, both matching and *apartness* must be preserved by reduction. Consider the definition for *F'* below (where *H* is some other type family):

```
type family F' a b where
  F' Int Bool = Char
  F' a   b    = H a
```

We know that *F'* (*G Int*) (*G Int*) matches the second equation and is apart (Definition 6) from the first equation. The reduct *F'* (*G Int*) *Bool* also matches the second equation but is *not* apart from the first equation. Hence, *F'* (*G Int*) *Bool* cannot simplify by either equation for *F'*, and the local diamond property does not hold. Put simply, our apartness implementation is not preserved by reduction.

In a terminating system, we are able to get away with the *weaker* Property 14 for apart (where apartness is not directly preserved under reduction), which our implementation does satisfy. We have designed an implementation of apart which *is* provably stable under reduction, but it is more conservative and less intuitive for programmers. Given that this alternative definition of apart brought a proof of type safety only for potentially non-terminating but *linear* patterns (prohibiting our canonical example *Equal*), and

[6] Actually, under *parallel* reduction; see (Weirich et al. 2011).

that it often led to stuck targets where a reduction naively seemed possible, we have dismissed it as being impractical. We thus seek out a proof of type safety in the presence of non-terminating, non-left-linear axiom sets.

8.2 Type families vs. functional dependencies

Functional dependencies (Jones 2000) (further formalized by Sulzmann et al. (2007b)) allow a programmer to specify a dependency between two or more parameters of a type class. For example, Kiselyov et al. (2004) use this class for their type-level equality function:[7]

```
class HEq x y (b :: Bool) | x y → b
instance HEq x x True
instance (b ∼ False) ⇒ HEq x y b
```

The annotation $x\ y \rightarrow b$ in the class header declares a functional dependency from x and y to b. In other words, given x and y, we can always find b.

Functional dependencies have no analogue in GHC's internal language, System FC; indeed they predate it. Rather, functional dependencies simply add extra unification constraints that guide type inference. This can lead to very compact and convenient code, especially when there are multiple class parameters and bi-directional functional dependencies. However, functional dependencies do not generate coercions witnessing the equality between two types. Hence they interact poorly with GADTs and, more generally, with local type equalities. For example, consider the following:

```
class Same a b | a → b
instance Same Int Int
```

```
data T a where
  T1 :: T Int
  T2 :: T a
data S a where
  MkS :: Same a b ⇒ b → S a

f :: T a → S a → Int
f T1 (MkS b) = b
f T2 s       = 3
```

In the *T1* branch of *f* we know that *a* is *Int*, and hence (via the functional dependency and the *Same Int Int* instance declaration) the existentially-quantified *b* must also be *Int*, and the definition should type-check. But GHC rejects *f*, because it cannot produce a well-typed FC term equivalent to it. Could we fix this, by producing evidence in System FC for functional dependencies? Yes; indeed, one can regard functional dependencies as a convenient syntactic sugar for a program using type families. For example we could translate the example like this:

```
class F a ∼ b ⇒ Same a b where
  type F a
instance Same Int Int where
  type F Int = Int
```

Now the (unchanged) definition of *f* type-checks.

A stylistic difference is that functional dependencies and type classes encourage *logic* programming in the type system, whereas type families encourage *functional* programming.

[7] Available from `http://okmij.org/ftp/Haskell/types.html#`

`HList`.

8.3 Controlling overlap

Morris and Jones (2010) introduce *instance chains*, which obviate the need for overlapping instances by introducing a syntax for ordered overlap among instances. Their ideas are quite similar to the ones we present here, with a careful check to make sure that one instance is impossible before moving onto the next. However, the proof burden for their work is lower than ours—a flaw in instance selection may lead to incoherent behavior (e.g., different instances selected for the same code in different modules), but it cannot violate type safety. This is because class instances are compiled solely into term-level constructs (dictionaries), not type-level constructs. In particular, no equalities between different types are created as part of instance compilation.

8.4 Full-spectrum dependently typed languages

Type families resemble the type-level computation supported by dependently typed languages. Languages such as Coq (Coq development team 2004) and Agda (Norell 2007) allow ordinary functions to return *types*. As in Haskell, type equality in these languages is defined to include β-reduction of function application and ι-reduction of pattern matching.

However, there are several significant differences between these type-level functions and type families. The first is that Coq and Agda do not allow the elimination of their equivalents of kind $\star$. There is no way to write a Coq/Agda function analogous to the closed type family below, which returns *True* for function types and *False* otherwise.

```
type family IsArrow (a :: ⋆) :: Bool where
  IsArrow (a → b) = True
  IsArrow a        = False
```

Instead, pattern matching is only available for *inductive* datatypes. The consistency of these languages prohibits the elimination of non-inductive types such as ⋆ (or *Set*, *Prop*, and *Type*).

Furthermore, pattern matching in Coq and Agda does not support non-linear patterns. As we discussed above, non-linear patterns allow computation to observe whether two types are equal. However, the equational theory of full spectrum languages is much more expressive than that of Haskell. Because these languages allow unsaturated functions in types, it must define when two functions are equal. This comparison is intensional, and allowing computation to observe intensional equality is somewhat suspicious. However, in Haskell, where all type functions must always appear saturated, this issue does not arise.

Due to the lack of non-linear patterns, Coq and Agda programmers must define individual functions for every type that supports decidable equality. (Coq provides a tactic—`decide equality`—to automate this definition.) Furthermore, these definitions do not immediately imply that equality is reflexive; this result must be proved separately and manually applied. In contrast, the closed type family *Equal a a* immediately reduces to *True*.

Similarly, functions in Coq and Agda do not support coincident overlap at definition time. Again, these identities can be proven as lemmas, but must be manually applied.

8.5 Other functional programming languages

Is our work on closed type families translatable to other functional programming languages with rich type-level programming? We think so. Though the presentation in this paper is tied closely to Haskell, we believe that the notion of apartness would be quite similar (if not the same) in another programming language. Accordingly, the analysis of Section 3 would carry over without much change. The one caveat is that, as mentioned above, non-linear pattern matching depends on the saturation of all type-level functions. If this criterion is met, however, we believe that other languages could adopt the surface syntax and behavior of closed type families as presented here without much change.

9. Conclusions

Closed type families improve the usability of type-level computation, and make programming at the type level more reminiscent of ordinary term-level programming. At the same time, closed families allow for the definition of manifestly-reflexive, decidable equality on types of any kind. They allow automatic reductions of types with free variables and allow the user to specify multiple, potentially overlapping but coherent reduction strategies (such as the equations for the *And* example).

On the theoretical side, the question of consistency for non-terminating non-left-linear rewrite systems is an interesting research problem in its own right, quite independent of Haskell or type families, and we offer it as a challenge problem to the reader.

Acknowledgments

We particularly thank Conor McBride, Joxan Jaffar, and Stefan Kahrs for helping us navigate the literature on term rewriting, and on unification over infinite types. Stefan Kahrs provided the counter-example to confluence. Thanks also to José Pedro Magalhães for detailed and helpful feedback on the paper.

This material is partly supported by the National Science Foundation under Grant Nos. 1116620 and 1319880.

References

M. Chakravarty, G. Keller, and S. Peyton Jones. Associated type synonyms. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'05)*, Tallinn, Estonia, 2005.

Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. URL `http://coq.inria.fr`. Version 8.0.

B. Courcelle. Fundamental properties of infinite trees. *Theoretical computer science*, 25(2):95–169, 1983.

D. Fridlender and M. Indrika. Functional pearl: Do we need dependent types? *Journal of functional programming*, 10(4):409–415, 2000.

R. Garcia, J. Jarvi, A. Lumsdaine, J. G. Siek, and J. Willcock. A comparative study of language support for generic programming. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, OOPSLA '03, pages 115–134, New York, NY, USA, 2003. ACM. ISBN 1-58113-712-5. . URL `http://doi.acm.org/10.1145/949305.949317`.

G. Huet. *Résolution d'équations dans les langages d'ordre* $1, 2, \ldots, \omega$. PhD thesis, Université de Paris VII, 1976.

G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *J. ACM*, 27(4):797–821, Oct. 1980. ISSN 0004-5411.

. URL http://doi.acm.org/10.1145/322217.322230.

J. Jaffar. Efficient unification over infinite terms. *New Generation Computing*, 2(3):207–219, 1984. ISSN 0288-3635. . URL http://dx.doi.org/10.1007/BF03037057.

M. P. Jones. Type classes with functional dependencies. In G. Smolka, editor, *ESOP*, volume 1782 of *Lecture Notes in Computer Science*, pages 230–244. Springer, 2000. ISBN 3-540-67262-1.

O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Proc. 2004 ACM SIGPLAN Workshop on Haskell*, Haskell '04, pages 96–107. ACM, 2004.

J. Klop. Term rewriting systems. In *Handbook of logic in computer science (vol. 2)*, pages 1–116. Oxford University Press, Inc., 1993.

K. Knight. Unification: a multidisciplinary survey. *ACM Comput. Surv.*, 21 (1):93–124, Mar. 1989. ISSN 0360-0300. . URL http://doi.acm.org/10.1145/62029.62030.

C. McBride. Faking it: Simulating dependent types in Haskell. *J. Funct. Program.*, 12(5):375–392, July 2002.

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, pages 375–386, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3. . URL http://doi.acm.org/10.1145/1863543.1863596.

M. H. A. Newman. On theories with a combinatorial definition of "equivalence". *Annals of Mathematics*, 43(2):pp. 223–243, 1942. ISSN 0003486X. URL http://www.jstor.org/stable/1968867.

U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

T. Schrijvers, S. Peyton Jones, M. Chakravarty, and M. Sulzmann. Type checking with open type functions. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*, ICFP '08, pages 51–62, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-919-7. . URL `http://doi.acm.org/10.1145/1411204.1411215`.

M. Sulzmann, M. M. T. Chakravarty, S. Peyton Jones, and K. Donnelly. System F with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, TLDI '07, pages 53–66, New York, NY, USA, 2007a. ACM.

M. Sulzmann, G. Duck, S. Peyton Jones, and P. Stuckey. Understanding functional dependencies via constraint handling rules. *Journal of Functional Programming*, 17:83–130, Jan. 2007b.

W. Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4):423–436, July 2008. ISSN 0956-7968. . URL `http://dx.doi.org/10.1017/S0956796808006758`.

S. Weirich and C. Casinghino. Arity-generic datatype-generic programming. In *Proceedings of the 4th ACM SIGPLAN workshop on Programming languages meets program verification*, PLPV '10, pages 15–26, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-890-2. . URL `http://doi.acm.org/10.1145/1707790.1707799`.

S. Weirich, D. Vytiniotis, S. Peyton Jones, and S. Zdancewic. Generative type abstraction and type-level computation. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 227–240, New York, NY, USA, 2011. ACM.

S. Weirich, J. Hsu, and R. A. Eisenberg. Towards dependently typed Haskell: System FC with kind equality. In *Proceedings of the 18th ACM*

SIGPLAN International Conference on Functional Programming, ICFP '13, Boston, MA, USA, New York, NY, USA, 2013. ACM. To appear.

B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving Haskell a promotion. In *Proc. 8th ACM SIGPLAN workshop on Types in Language Design and Implementation*, TLDI '12, pages 53–66. ACM, 2012.

A. Description of units package

Using closed type families, we have written a library units,[8] for strongly-typed dimensional analysis. For example, we want to write functions like this:

$$\begin{array}{l} \mathit{curPos} :: \mathit{Pos} \to \mathit{Velocity} \to \mathit{Acceleration} \to \mathit{Time} \to \mathit{Pos} \\ \mathit{curPos}\ x_0\ v\ a\ t = x_0 \mathbin{.+} (v \mathbin{.*} t) \mathbin{.+} (0.5 \mathbin{*.} a \mathbin{.*} (t \mathbin{.\hat{}} \mathit{pTwo})) \end{array}$$

The above code works with our library and type-checks. However, if we were to make an expression that does not respect physical units (say, by forgetting the t in the $v \mathbin{.*} t$), we get a type error at compile time. For that particular case, the error says `Couldn't match type 'Meter' with 'Second'`, rather helpfully.

Importantly, this library is fully extensible. There are no wired-in units, except for *Scalar*. This way, users can apply the library to situations beyond just physics. For example, it might be sensible to have *HPixel* and *VPixel* units when writing a drawing program, to

[8] `cabal install units`; you will need GHC 7.8.

make sure that you don't ever add a height with a width.

In order to support extensibility, new units are represented by new datatypes, in kind ⋆. For example, here are the definitions for two units:

```
data Meter = Meters
instance Unit Meter where
  type BaseUnit Meter = Canonical
data Foot = Feet
instance Unit Foot where
  type BaseUnit Foot = Meter
  conversionRatio _ = 0.3048
type Pos = MkDim Meter
```

It is the library's extensibility that requires closed type families. It needs to reason about type-level structures without being able to enumerate all the possibilities, and without requiring the user to be well-versed in type families. There are two independent ways that closed type families are required in the design of the library: manipulating dimension specifications as type-level sets (which is similar to the example in Section 2.4) and managing the hierarchies of inter-convertible units.

Building a hierarchy of units with a distinguished root In the definition of *Meter* and *Foot* above, we also defined their relationship. The code says that *Meter* is a canonical unit—that is, it is not defined in terms of something else. On the other hand, *Foot* is defined in terms of *Meter*, so that we can write code like

```
height :: Double
height = (1.8 % Meters) # Feet
```

and *height* will have the value 5.9055. Of course, we could convert feet to meters simply by reversing the statement.

Say a library has been written on top of units that defines several different length measurements, such as *Meters*, *Feet*, and *LightYears*. Now, a user of that library realizes that she needs to define *Inches*. She would like to define inches in terms of *Feet*, because she knows that conversion ratio. But, she doesn't know which of the existing length units is the canonical one. Part of the design principle behind the units library is that she does not need to know—she can define *Inches* in terms of any of the available length units.

With this design in hand, we still need a way to compute the conversion from our internal representation of a length—which will be in *Meters*, the canonical unit—to *Inches*. We can see that the declared units form a tree, rooted at *Meters*, and each new unit refers to its *BaseUnit*, or parent in the tree. To find the right conversion ratio, we simply have to walk up the tree from the desired unit, multiplying all of the conversion ratios together.

But, how to implement this in Haskell? Recall that this tree is a tree of types, which are erased at runtime. We should use a class *Unit* that defines the conversion ratios, and we can have an associated type *BaseUnit* the defines a unit's parent in the tree. We introduce an empty type *Canonical* to serve as a canonical unit's (i.e., *Meter*'s) parent, or *BaseUnit*. Then, we can (seemingly) implement the conversion ratio calculation straightforwardly:

```
class (Unit (BaseUnit u)) ⇒ Unit u where
  type BaseUnit u :: ⋆
  conversionRatio :: u → Double
       -- ratio from u to u's parent

  canonicalConvRatio :: u → Double
       -- ratio from u to canonical unit,
      -- with default implementation
  canonicalConvRatio u
```

```
      = (conversionRatio u) ∗
        (canonicalConvRatio (⊥ :: BaseUnit u))
```

(The instance for the *Canonical* type breaks the recursion in *canonicalConvRatio* by overriding the default definition.)

There is a major problem with *Unit* as defined here—it has a superclass cycle. The header states that every *Unit*'s *BaseUnit* must also be a *Unit*, which is clearly ill-founded. Yet, this idea is sensible, because we need to be able to call *canonicalConvRatio* on a *BaseUnit*. What to do?

The full answer would take up too much space to describe (and is available if you download the units package), but it boils down to this:

```
type family CheckCanonical (unit :: ⋆) :: Bool where
  CheckCanonical Canonical = True
  CheckCanonical unit      = False
```

Using *CheckCanonical*, we can define a conditional constraint, essentially saying that every non-canonical unit must have a unit as its parent. This breaks the type-level recursion and brings us back onto solid footing.

It is never wise to say that an alternate encoding is *impossible* in Haskell, but we were unable to find another one that works smoothly and presents a very easy interface to users.

B. zipWith with inferred arity

Using the *CountArgs* closed type family from Section 2.3, we can define a variable-arity *zipWith* function that infers the correct arity from its first argument.

We first need a definition of the natural numbers. This definition will only be used as a promoted data*kind*.

$$\textbf{data}\ \mathit{Nat} = \mathit{Zero} \mid \mathit{Succ}\ \mathit{Nat}$$

In our description, we will abbreviate these unary numbers with ordinary decimals.

What will the type of our final *zipWith* be? It will first take a function and then several lists. The types of these lists is determined by the type of the function passed in. For example, suppose our function f has type $\mathit{Int} \rightarrow \mathit{Bool} \rightarrow \mathit{Double}$, then the type of *zipWith* should be $(\mathit{Int} \rightarrow \mathit{Bool} \rightarrow \mathit{Double}) \rightarrow [\mathit{Int}] \rightarrow [\mathit{Bool}] \rightarrow [\mathit{Double}]$. Thus, we wish to take the type of the function and apply the list type constructor $[\,]$ to each component of it.

Before we write the code for this operation, we pause to note an ambiguity in this definition. Both of the following are sensible concrete types for a *zipWith* over the function f:

$$\begin{aligned}
\mathit{zipWith} &:: (\mathit{Int} \rightarrow \mathit{Bool} \rightarrow \mathit{Double}) \\
&\quad \rightarrow [\mathit{Int}] \rightarrow [\mathit{Bool} \rightarrow \mathit{Double}] \\
\mathit{zipWith} &:: (\mathit{Int} \rightarrow \mathit{Bool} \rightarrow \mathit{Double}) \\
&\quad \rightarrow [\mathit{Int}] \rightarrow [\mathit{Bool}] \rightarrow [\mathit{Double}]
\end{aligned}$$

The first of these is essentially *map*; the second is the classic function *zipWith* that expects two lists. Thus, we must pass in the desired number of parameters to apply the list type constructor to.

(The inferred arity comes in later.) The function to apply these list constructors is named *Listify*:

```
type family Listify (n :: Nat) arrows where
  Listify Zero     a        = [a]
  Listify (Succ n) (a → b) = [a] → Listify n b
```

We now need to create some runtime evidence of our choice for the number of arguments. This will be used to control the runtime operation of *zipWith*—after all, our function must have both the correct behavior and the correct type. We use a GADT *NumArgs* that plays two roles: it controls the runtime behavior as just described, and it also is used as evidence to the type checker that the number argument to *Listify* is appropriate. After all, we do not want to call *Listify* 2 (*Int* → *Bool*), as that would be stuck. By pattern-matching on the *NumArgs* GADT, we get enough information to allow *Listify* to fully reduce.

```
data NumArgs :: Nat → ⋆ → ⋆ where
  NAZero ::                  NumArgs Zero     a
  NASucc :: NumArgs n b → NumArgs (Succ n) (a → b)
```

We now write the runtime workhorse *listApply*, with the following type:

```
listApply :: NumArgs n a → [a] → Listify n a
```

The first argument is the encoding of the number of arguments to the function. The second argument is a *list* of functions to apply to corresponding elements of the lists passed in after the second argument. Why do we need a list of functions? Consider evaluating *zipWith* (+) [1, 2] [3, 4], where we recur not only on the elements in the list, but on the number of arguments. After processing the first list, we have to be able to apply different functions to each of the elements of the second list. To wit, we need to apply the functions [(1+), (2+)] to corresponding elements in the list [3, 4]. (Here, we are using Haskell's "section" notation for partially-applied operators.)

Here is the definition of *listApply*:

```
listApply NAZero       fs = fs
listApply (NASucc na) fs =
  λargs → listApply na (apply fs args)
  where apply :: [a → b] → [a] → [b]
        apply (f : fs)  (x : xs) = (f x : apply fs xs)
        apply _         _        = []
```

It first pattern-matches on its first argument. In the *NAZero* case, the list of functions passed in has 0 arguments, so we just return them. In the *NASucc* case, we process one more argument (*args*), apply the list of functions *fs* respectively to the elements of *args*, and then recur. Note how the GADT pattern-matching is essential for this to type-check— the type checker gets just enough information for *Listify* to reduce enough so that the second case can expect one more argument than the first case.

Inferring arity As explained in Section 2.3, here is the closed type family that counts the number of arguments in a function type:

```
type family CountArgs (f :: ⋆) :: Nat where
  CountArgs (a → b) = Succ (CountArgs b)
  CountArgs result = Zero
```

We still need to connect this type-level function with the term-level GADT *NumArgs*. We use Haskell's method for reflecting type-level decisions on the term-level, type classes. The following definition essentially repeats the definition of *NumArgs*, but because this is a definition for a class, the instance is inferred rather than given explicitly:

```
class CNumArgs (numArgs :: Nat) (arrows :: ⋆) where
  getNA :: NumArgs numArgs arrows
instance CNumArgs Zero a where
  getNA = NAZero
instance CNumArgs n b ⇒
         CNumArgs (Succ n) (a → b) where
  getNA = NASucc getNA
```

Note that the instances do *not* overlap; they are distinguished by their first parameter.

It is now straightforward to give the final definition of *zipWith*, using the extension `-XScopedTypeVariables` to give the body of *zipWith* access to the type variable *f*:

```
zipWith :: ∀ f. CNumArgs (CountArgs f) f
          ⇒ f → Listify (CountArgs f) f
zipWith fun
   = listApply (getNA :: NumArgs (CountArgs f) f) (repeat fun)
```

The standard Haskell function *repeat* creates an infinite list of its one argument.

The following examples show that *zipWith* indeed infers the arity:

$$\begin{array}{l}
\mathit{example}_1 = \mathit{zipWith}\ (\wedge)\ [\mathit{False}, \mathit{True}, \mathit{False}]\ [\mathit{True}, \mathit{True}, \mathit{False}] \\
\mathit{example}_2 = \mathit{zipWith}\ ((+) :: \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Int})\ [1, 2, 3]\ [4, 5, 6]
\end{array}$$

$$\begin{array}{l}
\mathit{concat} :: \mathit{Int} \rightarrow \mathit{Char} \rightarrow \mathit{Double} \rightarrow \mathit{String} \\
\mathit{concat}\ a\ b\ c = (\mathit{show}\ a) +\!\!+ (\mathit{show}\ b) +\!\!+ (\mathit{show}\ c) \\
\mathit{example}_3 = \mathit{zipWith}\ \mathit{concat}\ [1, 2, 3]\ [\texttt{'a'}, \texttt{'b'}, \texttt{'c'}] \\
\qquad\qquad\qquad\qquad\qquad\quad [3.14, 2.1728, 1.01001]
\end{array}$$

In $\mathit{example}_2$, we must specify the concrete instantiation of $(+)$. In Haskell, built-in numerical operations are generalized over a type class Num. In this case, the operator $(+)$ has the type $\mathit{Num}\ a \Rightarrow a \rightarrow a \rightarrow a$. Because it is theoretically possible (but deeply strange!) for a to be instantiated with a function type, using $(+)$ without an explicit type will not work—there is no way to infer an unambiguous arity. Specifically, $\mathit{CountArgs}$ gets stuck. $\mathit{CountArgs}\ (a \rightarrow a \rightarrow a)$ simplifies to $\mathit{Succ}\ (\mathit{Succ}\ (\mathit{CountArgs}\ a))$ but can go no further; $\mathit{CountArgs}\ a$ will not simplify to Zero, because a is not apart from $b \rightarrow c$.

C. Typing judgments for System μFC

$\boxed{\vdash_{\mathsf{gnd}} \Sigma}$ Ground context validity

$$\frac{}{\vdash_{\mathsf{gnd}} \cdot} \quad \textsc{Gnd\_Empty}$$

$$\frac{\vdash_{\mathsf{gnd}} \Sigma \qquad H \mathrel{\#} \Sigma}{\vdash_{\mathsf{gnd}} \Sigma, H{:}\overline{\kappa} \rightarrow \star} \quad \textsc{Gnd\_Ground}$$

$$\frac{\vdash_{\mathsf{gnd}} \Sigma \qquad F \mathrel{\#} \Sigma}{\vdash_{\mathsf{gnd}} \Sigma, F(\overline{\kappa}){:}\kappa'} \quad \textsc{Gnd\_TyFam}$$

$$\begin{array}{c} F(\overline{\kappa}){:}\kappa' \in \Sigma \\ \hline\hline \Sigma; \overline{\alpha{:}\kappa} \vdash_{\mathsf{ty}} \rho : \kappa \end{array} \qquad C \mathrel{\#} \Sigma$$

$$\frac{\overline{\Sigma; \overline{\alpha{:}\kappa} \vdash_{\mathsf{ty}} \upsilon : \kappa'} \qquad \vdash_{\mathsf{gnd}} \Sigma}{\vdash_{\mathsf{gnd}} \Sigma, C{:}\overline{[\overline{\alpha{:}\kappa}].\ F(\overline{\rho}) \sim \upsilon}} \quad \textsc{Gnd\_Axiom}$$

$\boxed{\Sigma \vdash_{\mathsf{var}} \Delta}$ Variables context validity

$$\frac{\vdash_{\mathsf{gnd}} \Sigma}{\Sigma \vdash_{\mathsf{var}} \cdot} \quad \textsc{Var\_Empty}$$

$$\frac{\Sigma; \Delta \vdash_{\mathsf{ty}} \tau : \kappa \qquad x \# \Delta}{\Sigma \vdash_{\mathsf{var}} \Delta, x{:}\tau} \quad \textsc{Var\_TermVar}$$

$$\frac{\Sigma \vdash_{\mathsf{var}} \Delta \qquad \alpha \# \Delta}{\Sigma \vdash_{\mathsf{var}} \Delta, \alpha{:}\kappa} \quad \textsc{Var\_TypeVar}$$

$\boxed{\vdash_{\mathsf{ctx}} \Gamma}$ Context validity

$$\frac{\Sigma \vdash_{\mathsf{var}} \Delta}{\vdash_{\mathsf{ctx}} \Sigma; \Delta} \quad \textsc{Ctx\_Valid}$$

$\boxed{\Gamma \vdash_{\mathsf{tm}} e : \tau}$ Expression typing

$$\frac{x{:}\tau \ \in \ \Delta \qquad \vdash_{\mathsf{ctx}} \Sigma; \Delta}{\Sigma; \Delta \vdash_{\mathsf{tm}} x : \tau} \quad \textsc{Tm\_Var}$$

$$\frac{\Gamma, x{:}\tau_1 \vdash_{\mathsf{tm}} e : \tau_2}{\Gamma \vdash_{\mathsf{tm}} \lambda x{:}\tau_1 . e : \tau_1 \to \tau_2} \quad \textsc{Tm\_Abs}$$

$$\frac{\Gamma \vdash_{\mathsf{tm}} e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash_{\mathsf{tm}} e_2 : \tau_1}{\Gamma \vdash_{\mathsf{tm}} e_1\ e_2 : \tau_2} \quad \textsc{Tm\_App}$$

$$\frac{\Gamma, \alpha{:}\kappa \vdash_{\mathsf{tm}} e : \tau}{\Gamma \vdash_{\mathsf{tm}} \Lambda\alpha{:}\kappa.e : \forall\, \alpha{:}\kappa.\tau} \quad \textsc{Tm\_TyAbs}$$

$$\frac{\Gamma \vdash_{\mathsf{tm}} e : \forall\, \alpha{:}\kappa.\tau_2 \qquad \Gamma \vdash_{\mathsf{ty}} \tau_1 : \kappa}{\Gamma \vdash_{\mathsf{tm}} e\, \tau_1 : \tau_2[\tau_1/\alpha]} \quad \textsc{Tm\_TyApp}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \sim \tau_2 \qquad \Gamma \vdash_{\mathsf{tm}} e : \tau_1}{\Gamma \vdash_{\mathsf{tm}} e \rhd \gamma : \tau_2} \quad \textsc{Tm\_Cast}$$

$\boxed{\Gamma \vdash_{\mathsf{ty}} \tau : \kappa}$ Type kinding

$$\frac{\alpha{:}\kappa \in \Delta \qquad \vdash_{\mathsf{ctx}} \Sigma; \Delta}{\Sigma; \Delta \vdash_{\mathsf{ty}} \alpha : \kappa} \quad \textsc{Ty\_Var}$$

$$\frac{\begin{array}{c} F(\overline{\kappa}){:}\kappa' \in \Sigma \\ \overline{\Sigma; \Delta \vdash_{\mathsf{ty}} \tau : \kappa} \end{array} \qquad \vdash_{\mathsf{ctx}} \Sigma; \Delta}{\Sigma; \Delta \vdash_{\mathsf{ty}} F(\overline{\tau}) : \kappa'} \quad \textsc{Ty\_TyFam}$$

$$\frac{H{:}\overline{\kappa} \to \star \in \Sigma \qquad \vdash_{\mathsf{ctx}} \Sigma; \Delta}{\Sigma; \Delta \vdash_{\mathsf{ty}} H : \overline{\kappa} \to \star} \quad \textsc{Ty\_Ground}$$

$$\frac{\Gamma \vdash_{\mathsf{ty}} \tau_1 : \star \qquad \Gamma \vdash_{\mathsf{ty}} \tau_2 : \star}{\Gamma \vdash_{\mathsf{ty}} \tau_1 \to \tau_2 : \star} \quad \textsc{Ty\_Arrow}$$

$$\frac{\Gamma, \alpha{:}\kappa \vdash_{\mathsf{ty}} \tau : \star}{\Gamma \vdash_{\mathsf{ty}} \forall\, \alpha{:}\kappa.\tau : \star} \quad \textsc{Ty\_Forall}$$

$$\frac{\Gamma \vdash_{\mathsf{ty}} \tau_1 : \kappa_1 \to \kappa_2 \qquad \Gamma \vdash_{\mathsf{ty}} \tau_2 : \kappa_1}{\Gamma \vdash_{\mathsf{ty}} \tau_1\, \tau_2 : \kappa_2} \quad \textsc{Ty\_App}$$

D. Proof of substitution lemma

The kinding judgment for types, the proposition validity judgment, and the context validity judgments are all mutually recursive. They all support a standard substitution lemma, which we do not prove here:

Lemma 20 (Type substitution)**.** *Assume* $\Gamma \vdash_{\mathsf{ty}} \sigma : \kappa$*. Then, the following are true:*

1. *If* $\Gamma, \alpha{:}\kappa, \Delta \vdash_{\mathsf{ty}} \tau : \kappa$*, then* $\Gamma, \Delta[\sigma/\alpha] \vdash_{\mathsf{ty}} \tau[\sigma/\alpha] : \kappa$*.*
2. *If* $\vdash_{\mathsf{ctx}} \Gamma, \alpha{:}\kappa, \Delta$*, then* $\vdash_{\mathsf{ctx}} \Gamma, \Delta[\sigma/\alpha]$*.*
3. *If* $\Gamma, \alpha{:}\kappa, \Delta \vdash_{\mathsf{prop}} \phi\ \mathsf{ok}$*, then* $\Gamma, \Delta[\sigma/\alpha] \vdash_{\mathsf{prop}} \phi[\sigma/\alpha]\ \mathsf{ok}$*.*

Lemma (CO_AXIOM Substitution [Lemma 15])**.** *If* $\Sigma; \Delta, \beta{:}\kappa, \Delta' \vdash_{\mathsf{co}} C[i]\ \overline{\tau} : F(\overline{\rho_i[\overline{\tau/\alpha_i}]}) \sim \upsilon_i[\overline{\tau/\alpha_i}]$ *and* $\Sigma; \Delta \vdash_{\mathsf{ty}} \sigma : \kappa$*, then* $\Sigma; \Delta, \Delta'[\sigma/\beta] \vdash_{\mathsf{co}} C[i]\ \overline{\tau[\sigma/\beta]} : F(\overline{\rho_i[\overline{\tau/\alpha_i}][\sigma/\beta]}) \sim \upsilon_i[\overline{\tau/\alpha_i}][\sigma/\beta]$*.*

Proof. We invert $\Sigma; \Delta, \beta{:}\kappa, \Delta' \vdash_{\mathsf{co}} C[i]\ \overline{\tau} : F(\overline{\rho_i[\overline{\tau/\alpha_i}]}) \sim \upsilon_i[\overline{\tau/\alpha_i}]$ to get the following:

- $C{:}\Psi \in \Sigma$
- $\Psi = \overline{[\overline{\alpha{:}\kappa}].\ F(\overline{\rho}) \sim \upsilon}$
- $\overline{\Sigma; \Delta, \beta{:}\kappa, \Delta' \vdash_{\mathsf{ty}} \tau : \kappa_i}$
- $\vdash_{\mathsf{ctx}} \Sigma; \Delta, \beta{:}\kappa, \Delta'$
- $\forall j < i, \mathsf{no\_conflict}(\Psi, i, \overline{\tau}, j)$

Lemma 20 gives $\overline{\Sigma; \Delta, \Delta'[\sigma/\beta] \vdash_{\mathsf{ty}} \tau[\sigma/\beta] : \kappa_i}$ and $\vdash_{\mathsf{ctx}} \Sigma; \Delta, \Delta'[\sigma/\beta]$. Let $\phi = F(\overline{\rho}) \sim \upsilon$. It now remains only to show that $\forall j < i, \mathsf{no\_conflict}(\Psi, i, \overline{\tau[\sigma/\beta]}, j)$ and $\phi[\overline{\tau/\alpha_i}][\sigma/\beta] = \phi[\overline{\tau[\sigma/\beta]/\alpha_i}]$, and then we can use CO_AXIOM to get the desired result.

$$\frac{\begin{array}{l} C{:}\Psi \in \Sigma \qquad \Psi = \overline{[\overline{\alpha{:}\kappa}].\ F(\overline{\rho}) \sim \upsilon} \\ \vdash_{\mathsf{gnd}} \Sigma \qquad \overline{\tau} = \rho_i[\overline{\psi/\alpha_i}] \qquad \tau' = \upsilon_i[\overline{\psi/\alpha_i}] \\ \forall\, j < i, \mathsf{no\_conflict}(\Psi, i, \overline{\psi}, j) \end{array}}{\Sigma \vdash \mathcal{C}[F(\overline{\tau})] \rightsquigarrow \mathcal{C}[\tau']}\ \textsc{Red}$$

Figure 8. The type rewriting rule

The second fact above is immediate from the fact that the variable β must not be free in ϕ, invoking the Barendregt variable convention and noting that β is introduced separately from any of the variables in scope in ϕ.

Thus, we must only show $\forall\, j < i, \mathsf{no\_conflict}(\Psi, i, \overline{\tau[\sigma/\beta]}, j)$. Thus, given $j < i$ (and knowing $\mathsf{no\_conflict}(\Psi, i, \overline{\tau}, j)$), we must show $\mathsf{no\_conflict}(\Psi, i, \overline{\tau[\sigma/\beta]}, j)$. We proceed by case analysis on $\mathsf{no\_conflict}(\Psi, i, \overline{\tau}, j)$:

Case NC_APART: We must show only that $\mathsf{apart}(\overline{\rho_j}, \overline{\rho_i[\overline{\tau[\sigma/\beta]/\alpha_i}]})$, assuming $\mathsf{apart}(\overline{\rho_j}, \rho_i[\overline{\tau/\alpha_i}])$. The result is immediate after invoking Property 12, with $\Omega = \overline{\beta \mapsto \sigma}$ and noting that β cannot be free in $\overline{\rho_i}$.

Case NC_COMPATIBLE: We note that $\overline{\tau}$ appears nowhere else in the premises of this rule. Therefore, changing $\overline{\tau}$ has no effect, and we are done.

□

E. Proof of consistency

As described in Section 5.3, we use a rewrite relation, defined in Figure 8, show that it is complete with respect to $\Sigma; \Delta \vdash_{\mathsf{co}} \gamma : \tau_1 \sim \tau_2$, and then conclude that Σ must be consistent, as rewriting preserves non-type-family head forms.

Type contexts Throughout this proof, we use a notion of type contexts, or types with holes. The notation $\mathcal{C}[\cdot]$ denotes a type with exactly one hole in it. Similarly, $\mathcal{C}[\![\cdot]\!]$ denotes a type with any number of holes (possibly 0) in it. We generalize these definitions to lists, saying that $\mathbb{C}[\cdot]$ denotes a list of types with exactly one hole (in one specific type, not one hole per type) and that $\mathbb{C}[\![\cdot]\!]$ denotes a list of types with any number of holes.

E.1 Rewrite relation

The only form of reduction is type family simplification, using the same $\mathsf{no\_conflict}$ judgment that appears in the $\textsc{Co\_Axiom}$ rule. The use of $\mathcal{C}[\cdot]$ in the conclusion states that a type family application can reduce anywhere within the structure of a type. As $\mathcal{C}[\cdot]$ denotes a type context with exactly one hole, only one type family reduction happens in one step. Note that this rule is nondeterministic.

We use the notation $\Sigma \vdash \sigma_1 \rightsquigarrow^* \sigma_2$ to mean the reflexive, transitive closure of the relation $\Sigma \vdash \cdot \rightsquigarrow \cdot$. We write single-step joinability of σ_1 and σ_2 as $\Sigma \vdash \sigma_1 \Leftrightarrow \sigma_2$; this fact holds whenever there exists σ_3 such that $\Sigma \vdash \sigma_1 \rightsquigarrow \sigma_3$ and $\Sigma \vdash \sigma_2 \rightsquigarrow \sigma_3$, or $\Sigma \vdash \sigma_1 \rightsquigarrow \sigma_2$, or $\Sigma \vdash \sigma_2 \rightsquigarrow \sigma_1$, or $\sigma_1 = \sigma_2$. General joinability is written $\Sigma \vdash \sigma_1 \Leftrightarrow^* \sigma_2$; this fact holds whenever there exists σ_3 such that $\Sigma \vdash \sigma_1 \rightsquigarrow^* \sigma_3$ and $\Sigma \vdash \sigma_2 \rightsquigarrow^* \sigma_3$.

We generalize the relation to hold over lists of types, written $\Sigma \vdash \overline{\tau} \rightsquigarrow \overline{\sigma}$, to say that the list $\overline{\sigma}$ is identical to the list $\overline{\tau}$ except for one element which takes one step. We also say $\Sigma \vdash \overline{\tau} \rightsquigarrow^* \overline{\sigma}$, which is identical to $\overline{\Sigma \vdash \tau \rightsquigarrow^* \sigma}$.

Definition 21 (Confluence)**.** *Our rewrite system is confluent if, for all σ_0, σ_1, and σ_2 such that $\Sigma \vdash \sigma_0 \rightsquigarrow^* \sigma_1$ and $\Sigma \vdash \sigma_0 \rightsquigarrow^* \sigma_2$, $\Sigma \vdash \sigma_1 \Leftrightarrow^* \sigma_2$.*

In order to show the completeness of the rewrite relation for transitivity coercions, we need to show the transitivity of the joinability relation—that is, that $\Sigma \vdash \sigma_1 \Leftrightarrow^* \sigma_2$ and $\Sigma \vdash \sigma_2 \Leftrightarrow^* \sigma_3$ implies $\Sigma \vdash \sigma_1 \Leftrightarrow^* \sigma_3$. This fact requires confluence of the rewrite system.

E.2 Local confluence

Newman's lemma (Newman 1942) states that a terminating rewrite system is confluent if it is *locally confluent*.

Definition 22 (Local confluence)**.** *Our rewrite system is locally confluent if, for all σ_0, σ_1, and σ_2 such that $\Sigma \vdash \sigma_0 \rightsquigarrow \sigma_1$ and $\Sigma \vdash \sigma_0 \rightsquigarrow \sigma_2$, then $\Sigma \vdash \sigma_1 \Leftrightarrow^* \sigma_2$.*

A diagrammatic presentation of different confluence properties is in Figure 6.

Because we have assumed termination, we need only show local confluence to show confluence. As usual, we will need a small menagerie of supporting lemmas before we can get to the main proof.

Lemma 23 (Stability of choice of substitution of lists)**.** *If $\tau[\overline{\sigma/\alpha}] =$*

$\tau[\overline{\sigma'/\alpha}]$ *and all the* $\overline{\alpha}$ *are free in* τ*, then* $\overline{\sigma} = \overline{\sigma'}$*.*

Proof. By induction on the structure of τ:

Case $\tau = \alpha$**:** It must be that $\overline{\alpha} = \alpha$, a one-element list. Thus, we know that $\overline{\sigma} = \sigma$ and $\overline{\sigma'} = \sigma'$. The given equality reduces to $\sigma = \sigma'$, so we are done.

Case $\tau = \sigma_1 \to \sigma_2$**:** Divide the variables $\overline{\alpha}$ into three groups:

- $\overline{\beta_1}$ are the variables free in σ_1 but not free in σ_2,
- $\overline{\beta_2}$ are the variables free in σ_2 but not free in σ_1, and
- $\overline{\beta_3}$ are the variables free in both σ_1 and σ_2.

Divide $\overline{\sigma}$ and $\overline{\sigma'}$ accordingly. Then, we can use the induction hypothesis to get that $\overline{\sigma_1}, \overline{\sigma_3} = \overline{\sigma_1'}, \overline{\sigma_3'}$ and that $\overline{\sigma_2}, \overline{\sigma_3} = \overline{\sigma_2'}, \overline{\sigma_3'}$. Thus, we can conclude that $\overline{\sigma} = \overline{\sigma'}$ as desired.

Cases $\tau = \forall\, \alpha{:}\kappa.\upsilon$**,** $\tau = \upsilon_1\, \upsilon_2$**, and** $\tau = F(\overline{\upsilon})$**:** Similar.

Case $\tau = H$**:** The list of variables $\overline{\alpha}$ must be empty, as must be $\overline{\sigma}$ and $\overline{\sigma'}$, so we are done.

□

Lemma 24 (Stability of choice of substitution of lists in lists)**.** *If* $\overline{\tau[\overline{\sigma/\alpha}]} = \overline{\tau[\overline{\sigma'/\alpha}]}$ *and all the* $\overline{\alpha}$ *are free in* $\overline{\tau}$*, then* $\overline{\sigma} = \overline{\sigma'}$*.*

Proof. By induction on the length of $\overline{\tau}$, appealing to Lemma 23 and using logic as above to manage the free variables. □

Lemma 25 (One step/one hole context substitution)**.** *If* $\Sigma \vdash \tau \rightsquigarrow \tau'$*, then* $\Sigma \vdash \mathcal{C}[\tau] \rightsquigarrow \mathcal{C}[\tau']$*.*

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

Lemma 26 (One step/many holes context substitution)**.** *If* $\Sigma \vdash$

$\tau \rightsquigarrow \tau'$, then $\Sigma \vdash \mathcal{C}[\![\tau]\!] \rightsquigarrow^* \mathcal{C}[\![\tau']\!]$.

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

Lemma 27 (Multistep/many holes context substitution)**.** *If $\Sigma \vdash \tau \rightsquigarrow^* \tau'$, then $\Sigma \vdash \mathcal{C}[\![\tau]\!] \rightsquigarrow^* \mathcal{C}[\![\tau']\!]$.*

Proof. Straightforward induction on the length of the reduction $\Sigma \vdash \tau \rightsquigarrow^* \tau'$, appealing to Lemma 26. □

Lemma 28 (One step/one variable substitution)**.** *If $\Sigma \vdash \tau \rightsquigarrow \tau'$, then $\Sigma \vdash \sigma[\tau/\alpha] \rightsquigarrow^* \sigma[\tau'/\alpha]$.*

Proof. By Lemma 25. □

Lemma 29 (One step/list of variables substitution)**.** *If $\Sigma \vdash \overline{\tau} \rightsquigarrow \overline{\tau'}$, then $\Sigma \vdash \sigma[\overline{\tau/\alpha}] \rightsquigarrow^* \sigma[\overline{\tau'/\alpha}]$.*

Proof. Straightforward induction on the list $\overline{\tau}$, using Lemma 28. □

Lemma 30 (Multistep/list of variables substitution)**.** *If $\Sigma \vdash \overline{\tau} \rightsquigarrow^* \overline{\tau'}$, then $\Sigma \vdash \sigma[\overline{\tau/\alpha}] \rightsquigarrow^* \sigma[\overline{\tau'/\alpha}]$.*

Proof. Straightforward induction on the length of the reduction $\Sigma \vdash \overline{\tau} \rightsquigarrow^* \overline{\tau'}$, appealing to Lemma 29. □

Lemma 31 (One step linear type pattern anti-substitution)**.** *If $\overline{\alpha}$ is the set of free variables in linear pattern ρ and $\Sigma \vdash \rho[\overline{\sigma/\alpha}] \rightsquigarrow \rho[\overline{\sigma'/\alpha}]$, then $\Sigma \vdash \overline{\sigma} \rightsquigarrow \overline{\sigma'}$.*

Proof. By induction on the structure of ρ, where the linearity assumption is needed when dividing up the variables and combining the results when appealing to multiple induction hypotheses. □

Lemma 32 (Multistep linear type pattern anti-substitution)**.** *If $\overline{\alpha}$ is the set of free variables in linear pattern ρ and $\Sigma \vdash \rho[\overline{\sigma/\alpha}] \rightsquigarrow^* \rho[\overline{\sigma'/\alpha}]$, then $\Sigma \vdash \overline{\sigma} \rightsquigarrow^* \overline{\sigma'}$.*

Proof. By induction on the length of the reduction $\Sigma \vdash \rho[\overline{\sigma/\alpha}] \rightsquigarrow^* \rho[\overline{\sigma'/\alpha}]$, appealing to Lemma 31 in the inductive case and Lemma 23 in the base case. □

Lemma 33 (Multistep type pattern anti-substitution)**.** *If $\overline{\alpha}$ is the set of free variables in pattern ρ and $\Sigma \vdash \rho[\overline{\sigma/\alpha}] \rightsquigarrow^* \rho[\overline{\sigma'/\alpha}]$, then $\Sigma \vdash \overline{\sigma} \rightsquigarrow^* \overline{\sigma'}$.*

Proof. Let ρ' be the result of replacing all variables in ρ with fresh variables. Thus ρ' is a linearized version of ρ. Let the set of free variables in ρ' be $\overline{\alpha'}$. We can see that for some list of types $\overline{\psi}$, $\rho[\overline{\sigma/\alpha}] = \rho'[\overline{\psi/\alpha'}]$. (The list of types $\overline{\psi}$ is just like $\overline{\sigma}$ but with some repetitions to account for the linearization.) Similarly, we have $\rho[\overline{\sigma'/\alpha}] = \rho'[\overline{\psi'/\alpha'}]$. Thus, we know $\Sigma \vdash \rho'[\overline{\psi/\alpha'}] \rightsquigarrow^* \rho'[\overline{\psi'/\alpha'}]$. We then appeal to Lemma 32 to get $\Sigma \vdash \overline{\psi} \rightsquigarrow^* \overline{\psi'}$. Recall that this notation means that $\overline{\Sigma \vdash \psi \rightsquigarrow^* \psi'}$. Thus, we can conclude that $\overline{\Sigma \vdash \sigma \rightsquigarrow^* \sigma'}$ (because each ψ and ψ' has an equal σ or σ') and then $\Sigma \vdash \overline{\sigma} \rightsquigarrow^* \overline{\sigma'}$. □

Lemma 34 (Local confluence)**.** *If* **Good** Σ*, the rewrite relation $\Sigma \vdash \cdot \rightsquigarrow \cdot$ is locally confluent.*

Proof. We assume $\Sigma \vdash \sigma_0 \rightsquigarrow \sigma_1$ and $\Sigma \vdash \sigma_0 \rightsquigarrow \sigma_2$ and we must find σ_3 such that $\Sigma \vdash \sigma_1 \rightsquigarrow^* \sigma_3$ and $\Sigma \vdash \sigma_2 \rightsquigarrow^* \sigma_3$. We proceed by induction on the structure of σ_0.

Case $\sigma_0 = \tau_1 \to \tau_2$: Inverting $\Sigma \vdash \sigma_0 \rightsquigarrow \sigma_1$ and $\Sigma \vdash \sigma_0 \rightsquigarrow \sigma_2$

tells us that $(\tau_1 \to \tau_2) = \mathcal{C}_1[F_1(\overline{\psi_1})]$ and $(\tau_1 \to \tau_2) = \mathcal{C}_2[F_2(\overline{\psi_2})]$, with $\sigma_1 = \mathcal{C}_1[\psi_1']$ and $\sigma_2 = \mathcal{C}_2[\psi_2']$. We now do case analysis on $\mathcal{C}_1[\cdot]$ and $\mathcal{C}_2[\cdot]$:

Case $\mathcal{C}_1[\cdot] = \mathcal{C}_1'[\cdot] \to \tau_2$, $\mathcal{C}_2[\cdot] = \mathcal{C}_2'[\cdot] \to \tau_2$: Note that $\mathcal{C}_1'[F_1(\overline{\psi_1})] = \tau_1 = \mathcal{C}_2'[F_2(\overline{\psi_2})]$. Therefore, using the other conditions known from inverting the original steps from σ_0, we know that $\Sigma \vdash \tau_1 \rightsquigarrow \tau_{11}$ and $\Sigma \vdash \tau_1 \rightsquigarrow \tau_{12}$, where $\tau_{11} = \mathcal{C}_1'[\psi_1']$ and $\tau_{12} = \mathcal{C}_2'[\psi_2']$. Use the induction hypothesis to get τ_{13} such that $\Sigma \vdash \tau_{11} \rightsquigarrow^* \tau_{13}$ and $\Sigma \vdash \tau_{12} \rightsquigarrow^* \tau_{13}$. Then, by Lemma 27 to lift this result back to $\tau_1 \to \tau_2$, we are done, showing that $\sigma_3 = \tau_{13} \to \tau_2$.

Case $\mathcal{C}[\cdot] = \mathcal{C}_1'[\cdot] \to \tau_2$, $\mathcal{C}_2[\cdot] = \tau_1 \to \mathcal{C}_2'[\cdot]$: Let $\tau_1' = \mathcal{C}_1'[\psi_1']$ and $\tau_2' = \mathcal{C}_2'[\psi_2']$. Then, $\sigma_1 = \tau_1' \to \tau_2$ and $\sigma_2 = \tau_1 \to \tau_2'$ with $\Sigma \vdash \tau_1 \rightsquigarrow \tau_1'$ and $\Sigma \vdash \tau_2 \rightsquigarrow \tau_2'$. We let $\sigma_3 = \tau_1' \to \tau_2'$, and we are done.

Other cases: Similar to the cases above.

Case $\sigma_0 = \forall\, \alpha{:}\kappa.\tau$: Similar to the case for $\tau_1 \to \tau_2$.

Case $\sigma_0 = \tau_1\, \tau_2$: Similar to the case for $\tau_1 \to \tau_2$.

Case $\sigma_0 = F(\overline{\upsilon})$: Inverting $\Sigma \vdash \sigma_0 \rightsquigarrow \sigma_1$ and $\Sigma \vdash \sigma_0 \rightsquigarrow \sigma_2$ gives us $\sigma_0 = \mathcal{C}'[F'(\overline{\tau})]$ and $\sigma_0 = \mathcal{C}''[F''(\overline{\tau''})]$. If $\mathcal{C}'[\cdot] \neq \cdot$ and $\mathcal{C}''[\cdot] \neq \cdot$, then we are in a case similar to the case for $\tau_1 \to \tau_2$, and we simply use induction. Otherwise, we are left with three cases:

Case $\mathcal{C}'[\cdot] = F(\mathbb{C}[\cdot])$, $\mathcal{C}''[\cdot] = \cdot$: In this case, $\overline{\upsilon} = \mathbb{C}[F'(\overline{\tau})]$. Let τ' be the top-level reduct of $F'(\overline{\tau})$. Thus, $\sigma_1 = F(\mathbb{C}[\tau'])$. Let $\overline{\upsilon'} = \mathbb{C}[\tau']$.

We also know that $\Sigma \vdash F(\overline{\upsilon}) \rightsquigarrow \sigma_2$ by a top level reduction.

Inverting gives us the following:

- $C{:}\Psi \in \Sigma$
- $\Psi = \overline{[\overline{\alpha{:}\kappa}].\ F(\overline{\rho}) \sim \sigma'}$
- $\overline{\upsilon} = \overline{\rho_i[\overline{\psi/\alpha_i}]}$
- $\sigma_2 = \sigma'_i[\overline{\psi/\alpha_i}]$
- $\forall\, j < i, \mathsf{no\_conflict}(\Psi, i, \overline{\psi}, j)$

We want to find a common reduct of σ_2 (which might not be headed by F) and $F(\overline{\upsilon'})$. Thus, we must find a way to reduce $F(\overline{\upsilon'})$ at the top level. We now use Property 14 to get $\overline{\upsilon''}$ such that $\Sigma \vdash \overline{\upsilon'} \rightsquigarrow^* \overline{\upsilon''}$, $\overline{\upsilon''} = \Omega'(\overline{\rho_i})$ for some Ω', and for every $\overline{\rho'}$ such that $\mathsf{apart}(\overline{\rho'}, \overline{\upsilon})$, $\mathsf{apart}(\overline{\rho'}, \overline{\upsilon''})$.

Instead of reducing $F(\overline{\upsilon'})$ directly, we step $F(\overline{\upsilon'})$ to $F(\overline{\upsilon''})$ (getting $\Sigma \vdash F(\overline{\upsilon'}) \rightsquigarrow^* F(\overline{\upsilon''})$ from repeated application of Lemma 27) and then show that $F(\overline{\upsilon''})$ can reduce at the top level by the same equation at $F(\overline{\upsilon})$ reduced to form σ_2. Thus, we must prove that $\overline{\upsilon''} = \overline{\rho_i[\overline{\psi'/\alpha_i}]}$ (for some $\overline{\psi'}$) and that, for all $j < i$, $\mathsf{no\_conflict}(\Psi, i, \overline{\psi'}, j)$.

We know that $\overline{\upsilon''} = \Omega'(\overline{\rho_i})$. We also know that, by assumption, the free variables in $\overline{\upsilon''}$ are distinct from the free variables in $\overline{\rho_i}$. Thus, Ω' must map every free variable in $\overline{\rho_i}$ to some other type. Thus, we have $\overline{\upsilon''} = \overline{\rho_i[\overline{\psi'/\alpha_i}]}$ for the $\overline{\psi'}$ taken from the range of Ω'.

We then perform inversion on the known facts that, for all $j < i$, $\mathsf{no\_conflict}(\Psi, i, \overline{\psi}, j)$. We now fix j, and repeat this argument for all $j < i$:

Case NC_APART: We see that $\mathsf{apart}(\overline{\rho_j}, \overline{\rho_i[\overline{\psi/\alpha_i}]})$. From Property 14, we see that $\mathsf{apart}(\overline{\rho_j}, \overline{\rho_i[\overline{\psi'/\alpha_i}]})$ as desired.

Case NC_Compatible: The check $\mathsf{compat}(\Psi[i], \Psi[j])$ does not depend on the types $\overline{\psi}$ or $\overline{\psi'}$, and thus we are done.

Thus, $F(\overline{\upsilon''})$ reduces at the top level to $\sigma_3 = \sigma'_i[\overline{\psi'/\alpha_i}]$. It remains to show that σ_2 (which equals $\sigma'_i[\overline{\psi/\alpha_i}]$), the initial top-level reduct of $F(\overline{\upsilon})$ reduces to σ_3. We know that $\Sigma \vdash \overline{\rho_i[\overline{\psi/\alpha_i}]} \rightsquigarrow^* \overline{\rho_i[\overline{\psi'/\alpha_i}]}$. Thus, by repeated application of Lemma 33 (and appealing to clause 2 of **Good** to show that every $\psi \in \overline{\psi}$ is considered), we get $\Sigma \vdash \overline{\psi} \rightsquigarrow^* \overline{\psi'}$. By Lemma 30, we can conclude $\Sigma \vdash \sigma'_i[\overline{\psi/\alpha_i}] \rightsquigarrow^* \sigma'_i[\overline{\psi'/\alpha_i}]$, as desired.

Case $\mathcal{C}'[\cdot] = \cdot, \mathcal{C}''[\cdot] = F(\overline{\mathcal{C}''[\cdot]})$: Similar to the case above.

Case $\mathcal{C}'[\cdot] = \cdot, \mathcal{C}''[\cdot] = \cdot$: We will show a stronger property than local confluence in this case; we will show that if $\Sigma \vdash F(\overline{\tau}) \rightsquigarrow \upsilon_1$ and $\Sigma \vdash F(\overline{\tau}) \rightsquigarrow \upsilon_2$, both at the top level, then $\upsilon_1 = \upsilon_2$.

We invert both reductions to get the following facts, along with $\vdash_{\mathsf{gnd}} \Sigma$:

from $\Sigma \vdash F(\overline{\tau}) \leadsto \upsilon_1$	from $\Sigma \vdash F(\overline{\tau}) \leadsto \upsilon_2$
$C_1{:}\Psi_1 \in \Sigma$	$C_2{:}\Psi_2 \in \Sigma$
$\Psi_1 = \overline{[\overline{\alpha_1{:}\kappa_1}].\ F(\overline{\rho_1}) \sim \sigma'_1}$	$\Psi_2 = \overline{[\overline{\alpha_2{:}\kappa_2}].\ F(\overline{\rho_2}) \sim \sigma'_2}$
$\overline{\tau} = \rho_{1\,i}[\overline{\psi_1/\alpha_{1\,i}}]$	$\overline{\tau} = \rho_{2\,j}[\overline{\psi_2/\alpha_{2\,j}}]$
$\upsilon_1 = \sigma'_{1\,i}[\overline{\psi_1/\alpha_{1\,i}}]$	$\upsilon_2 = \sigma'_{2\,j}[\overline{\psi_2/\alpha_{2\,j}}]$
$\forall k < i,$ $\mathsf{no\_conflict}(\Psi_1, i, \overline{\psi_1}, k)$	$\forall k < j,$ $\mathsf{no\_conflict}(\Psi_2, j, \overline{\psi_2}, k)$

Thus, we must show that $\sigma'_{1\,i}[\overline{\psi_1/\alpha_{1\,i}}] = \sigma'_{2\,j}[\overline{\psi_2/\alpha_{2\,j}}]$. From clause 3 of **Good**, we see that either $i = j = 0$ (open family) or $C_1 = C_2$ (closed family). We will tackle these cases separately:

Open family: In this case, the axioms C_1 and C_2 have one equation each and thus we simply drop the i and j subscripts. Let Φ_1 and Φ_2 be the the equations of C_1 and C_2, respectively.

We know from the inversions that $\overline{\rho_1[\overline{\psi_1/\alpha_1}]} = \overline{\rho_2[\overline{\psi_2/\alpha_2}]}$. Let $\Omega_2 = [\overline{\alpha_1 \mapsto \psi_1}, \overline{\alpha_2 \mapsto \psi_2}]$. We can see that Ω_2 is a unifier of $\overline{\rho_1}$ and $\overline{\rho_2}$. Then, clause 4 of **Good** tells us that $\mathsf{compat}(\Phi_1, \Phi_2)$. Here, we have two cases:

Case COMPAT_COINCIDENT: We know that Ω is a most general unifier of $\overline{\rho_1}$ and $\overline{\rho_2}$ (appealing to Property 11) and $\Omega(\sigma'_1) = \Omega(\sigma'_2)$. Thus, there must be some Ω' such that $\Omega_2 = \Omega' \circ \Omega$.

We must show that $\sigma'_1[\overline{\psi_1/\alpha_1}] = \sigma'_2[\overline{\psi_2/\alpha_2}]$. This equation is equivalent to $\Omega_2(\sigma_1') = \Omega_2(\sigma_2')$, which in turn is $\Omega'(\Omega(\sigma_1')) = \Omega'(\Omega(\sigma_2'))$. But, we know that $\Omega(\sigma_1') = \Omega(\sigma_2')$, so we are done.

Case COMPAT_DISTINCT: We know that unify$(\overline{\rho_1}, \overline{\rho_2})$ fails. Yet, we have Ω_2 as a unifier of these types. Appealing to Property 40, we have a contradiction, and thus this case cannot happen.

Closed family: We know $C_1 = C_2$ and, by $\vdash_{\mathsf{gnd}} \Sigma$, there can be only one axiom of the same name in the context, so $\Psi_1 = \Psi_2$, and thus we can drop the 1 and 2 subscripts, except on the $\overline{\psi}$, which do not appear in the axiom types. Thus, we must show $\sigma_i'[\overline{\psi_1/\alpha_i}] = \sigma_j'[\overline{\psi_2/\alpha_j}]$.

Now, we must examine the indices i and j. If $i = j$, then we are done by an application of Lemma 24, using $\overline{\rho[\overline{\psi_1/\alpha}]} = \overline{\rho[\overline{\psi_2/\alpha}]}$ and clause 2 of **Good**. So, we assume, without loss of generality, that $i > j$. Inverting no_conflict$(\Psi, i, \overline{\psi_1}, j)$ leads us to three cases:

Case NC_APART: We see here that apart$(\overline{\rho_j}, \overline{\rho_i[\overline{\psi_1/\alpha_i}]})$. Yet, we know from the original inversions that $\overline{\rho_i[\overline{\psi_1/\alpha_i}]} = \overline{\rho_j[\overline{\psi_2/\alpha_j}]}$. The substitution $[\overline{\alpha_j \mapsto \psi_2}]$ is then a unifier of the two types that we know are apart, leading to a contradiction, appealing to Property 13. Thus, this case cannot happen.

Case NC_COMPATIBLE/COMPAT_COINCIDENT: Here, we know that Ω is a most general unifier (appealing to

Property 11) for $\overline{\rho_i}$ and $\overline{\rho_j}$ and that $\Omega(\sigma'_i) = \Omega(\sigma'_j)$. From the original inversions, we know $\overline{\rho_i[\overline{\psi_1/\alpha_i}]} = \overline{\rho_j[\overline{\psi_2/\alpha_j}]}$. Let $\Omega_2 = [\overline{\alpha_i \mapsto \psi_1}, \overline{\alpha_j \mapsto \psi_2}]$. We can say $\Omega_2 = \Omega' \circ \Omega$ for some Ω'. We can rewrite our goal as showing that $\Omega'(\Omega(\sigma'_i)) = \Omega'(\Omega(\sigma'_j))$. This is immediate from the fact that $\Omega(\sigma'_i) = \Omega(\sigma'_j)$, and so we are done.

Case NC_COMPATIBLE/COMPAT_DISTINCT: We know $\mathsf{unify}_\infty(\overline{\rho_i}, \overline{\rho_j})$ fails. Yet, we know from the original inversions that $\overline{\rho_i[\overline{\psi_1/\alpha_i}]} = \overline{\rho_j[\overline{\psi_2/\alpha_j}]}$. The substitution $[\overline{\alpha_i \mapsto \psi_1}, \overline{\alpha_j \mapsto \psi_2}]$ is then a unifier of $\overline{\rho_i}$ and $\overline{\rho_j}$, leading to a contradiction, appealing to Property 11.

□

Lemma 35 (Confluence of terminating systems)**.** *If* $\mathbf{Good}\,\Sigma$ *and* $\Sigma \vdash \cdot \rightsquigarrow \cdot$ *is a terminating rewrite relation, then it is confluent.*

Proof. By appealing to Newman's lemma (Newman 1942) and Lemma 34. □

E.3 From confluence to consistency

Lemma 36 (Transitivity)**.** *If* $\Sigma \vdash \cdot \rightsquigarrow \cdot$ *is a terminating rewrite* *relation,* $\mathbf{Good}\,\Sigma$*,* $\Sigma \vdash \tau_1 \Leftrightarrow^* \tau_2$ *and* $\Sigma \vdash \tau_2 \Leftrightarrow^* \tau_3$*, then* $\Sigma \vdash \tau_1 \Leftrightarrow^* \tau_3$*.*

Proof. By Lemma 35. □

Lemma 37 (Congruence)**.** *If* $\Sigma \vdash \tau_1 \Leftrightarrow^* \tau_2$*, then* $\Sigma \vdash \mathcal{C}[\![\tau_1]\!] \Leftrightarrow^* \mathcal{C}[\![\tau_2]\!]$*.*

Proof. By appealing to Lemma 27. □

Lemma 38 (Completeness)**.** *If* $\Sigma \vdash \cdot \leadsto \cdot$ *is a terminating rewrite relation,* $\mathbf{Good}\,\Sigma$ *and* $\Sigma; \Delta \vdash_{\mathsf{co}} \gamma : \sigma_1 \sim \sigma_2$*, then* $\Sigma \vdash \sigma_1 \Leftrightarrow^* \sigma_2$*.*

Proof. We proceed by induction on $\Sigma; \Delta \vdash_{\mathsf{co}} \gamma : \sigma_1 \sim \sigma_2$:

Cases Co_Arrow, Co_Forall, Co_App, and Co_TyFam: By the induction hypothesis, appealing to Lemma 37 and Lemma 36.

Cases Co_Refl, Co_Sym, and Co_Trans: From the fact that $\Sigma \vdash \cdot \Leftrightarrow^* \cdot$ is an equivalence relation, appealing to Lemma 36.

Cases Co_Left and Co_Right: The induction hypothesis gives us that $\Sigma \vdash \tau_1\,\tau_2 \Leftrightarrow^* \sigma_1\,\sigma_2$. We can see that any reduct of a type application must also be a type application. Thus, the common reduct must be $\upsilon_1\,\upsilon_2$ (for some υ_1 and υ_2) where υ_1 joins τ_1 and σ_1 and υ_2 joins τ_2 and σ_2. Thus, we are done.

Case Co_Axiom: From clause 1 of **Good** and the Co_Axiom rule, we know that $\sigma_1 = F(\overline{\upsilon_i[\rho/\alpha_i]})$ and that $\sigma_2 = \upsilon_i'[\overline{\rho/\alpha_i}]$. We conclude that $\Sigma \vdash \sigma_1 \leadsto \sigma_2$, as the premises of the rule Red are all given by the premises of the rule Co_Axiom.

□

Lemma 39 (Consistency)**.** *If $\Sigma \vdash \cdot \rightsquigarrow \cdot$ is a terminating rewrite relation and $\mathbf{Good}\,\Sigma$, then Σ is consistent.*

Proof. A consistent coercion equates two types with the same ground head forms. By Lemma 38, these two types must be joinable under the rewrite relation. Yet, the rewriting rule preserves all head forms except for type families. As type families are not ground head forms, we are done. □

F. Proof of properties of apart

This appendix includes the proofs that our concrete definition of apart, as given in Definition 6, satisfies the properties stated in Section 5.1. Then, we show that these properties, along with the assumption of termination, imply the high-level (sanity-check) properties from Section 3.2. It is well-founded to use our confluence result for these later proofs as those properties are not used anywhere in other proofs—they simply serve as a higher-level check on our formal results.

F.1 Proofs of Properties 12–14

We restate our implementation of apart:

Definition (Apartness [Definition 6])**.**
$\mathsf{apart}(\rho, \tau) = \neg\mathsf{unify}_\infty(\rho, \mathsf{flatten}(\tau))$

Recall that $\mathsf{flatten}$ (Definition 5) replaces all type family applications in a (finite) type with fresh variables, maximally preserving sharing. That is, flattening the same type family application twice in the same type (or list of types) converts both applications to the same fresh variable. In order for flatten to be a well-defined function, it must refer to a mapping from every possible type headed by a type family to fresh variables. This mapping is countably infinite, but we can assume, as usual, a countably infinite set of fresh variables. Furthermore, we assume that the set of variables in the range of this mapping is distinct from variables used elsewhere (particularly, in patterns). If this assumption is violated for some use of flatten, we simply rename the variables accordingly.

The above definition of flattening with respect to an infinite mapping of type families to variables, means that flattening commutes with type constructors. For example, $\mathsf{flatten}(\tau_1 \to \tau_2) = \mathsf{flatten}(\tau_1) \to \mathsf{flatten}(\tau_2)$.

For completeness, we also restate the correctness of unification, but now for unify_∞.

Property 40 (unify_∞ correct)**.** *If and only if there exists a substitution ω (whose range may include infinite types) such that $\omega(\overline{\sigma}) = \omega(\overline{\tau})$, then* $\mathsf{unify}_\infty(\overline{\sigma}, \overline{\tau})$ *succeeds, returning ω. Furthermore, ω is a most general unifier of $\overline{\sigma}$ and $\overline{\tau}$.*

Before getting to the properties themselves, we must prove some properties about flatten. First, we extend flatten to apply to substitutions and define an inverse operation:

Definition 41 (Flattening a substitution)**.** *If $\Omega = [\overline{\alpha \mapsto \tau}]$, we say* $\mathsf{flatten}(\Omega)$ *for* $[\overline{\alpha \mapsto \mathsf{flatten}(\tau)}]$*, where sharing is maximally preserved between the different types τ.*

Definition 42 (Inverse flattening)**.** *We let* $\mathsf{flatten}^{-1}$ *denote the inverse operation to flattening, implemented by doing a reverse lookup in the map from type family applications to variables.*

Note that $\mathsf{flatten}^{-1}$ is a substitution, infinite in extent, but ordinary in other respects. In particular, note that the elements in the range of $\mathsf{flatten}^{-1}$ are *finite*—that is, $\mathsf{flatten}^{-1}$ could be denoted by the metavariable Ω.

Lemma 43 (Flattened substitutions)**.** *For all type patterns ρ and substitutions Ω,* $\mathsf{flatten}(\Omega(\rho)) = (\mathsf{flatten}(\Omega))(\rho)$.

Proof. The pattern ρ contains no type families, so $\mathsf{flatten}$ does not affect the parts of ρ unchanged by the application of Ω. Because $\mathsf{flatten}$ preserves maximal sharing, it must be the case that applying a flattened substitution yields the same result as flattening an substituted pattern. This can be shown by straightforward induction on ρ. □

Lemma 44 (Flattening commutes with substitution)**.** *For all Ω, there exists an Ω' such that, for all τ,* $\mathsf{flatten}(\Omega(\tau)) = \Omega'(\mathsf{flatten}(\tau))$.

Proof. We can say that

$$\mathsf{flatten}(\Omega(\tau)) = \mathsf{flatten}(\Omega(\mathsf{flatten}^{-1}(\mathsf{flatten}(\tau))))$$

Because $\mathsf{flatten}^{-1}$ is a substitution, and appealing to Lemma 43 (noting that $\mathsf{flatten}(\tau)$ is a pattern), we can rewrite this as $\mathsf{flatten}(\Omega\circ \mathsf{flatten}^{-1})(\mathsf{flatten}(\tau))$. Thus, we let Ω' be the substitution $\mathsf{flatten}(\Omega\circ \mathsf{flatten}^{-1})$ and we are done. □

Lemma 45 (Flattening a list commutes with substitution)**.** *For* *all Ω, there exists an Ω' such that, for all $\overline{\tau}$,* $\mathsf{flatten}(\Omega(\overline{\tau})) = \Omega'(\mathsf{flatten}(\overline{\tau}))$.

Proof. By induction on the length of the list, appealing to Lemma 44. □

Property (Apartness is stable under type substitution [Property 12])**.** *If* $\mathsf{apart}(\overline{\rho}, \overline{\tau})$*, then for all substitutions* Ω*,* $\mathsf{apart}(\overline{\rho}, \Omega(\overline{\tau}))$.

Proof. Expanding definitions, we must show that

$$\neg\mathsf{unify}_\infty(\overline{\rho}, \mathsf{flatten}(\overline{\tau}))$$

implies

$$\neg\mathsf{unify}_\infty(\overline{\rho}, \mathsf{flatten}(\Omega(\overline{\tau}))).$$

We prove the contrapositive, that is, that $\mathsf{unify}_\infty(\overline{\rho}, \mathsf{flatten}(\Omega(\overline{\tau})))$ implies $\mathsf{unify}_\infty(\overline{\rho}, \mathsf{flatten}(\overline{\tau}))$. Thus, we have a substitution ω such that $\omega(\overline{\rho}) = \omega(\mathsf{flatten}(\Omega(\overline{\tau})))$ and must find a ω' such that $\omega'(\overline{\rho}) = \omega'(\mathsf{flatten}(\overline{\tau}))$.

By Lemma 45, we can say $\mathsf{flatten}(\Omega(\overline{\tau})) = \Omega'(\mathsf{flatten}(\overline{\tau}))$ for some Ω'. Then, choose $\omega' = \omega \circ \Omega'$. We can see that $\omega'(\overline{\rho}) = \omega'(\mathsf{flatten}(\overline{\tau}))$ (noting that the variables in $\overline{\rho}$ are fresh from those in $\mathsf{flatten}(\overline{\tau})$) as desired. □

Property (No unifiers for apart types [Property 13])**.** *If* $\mathsf{apart}(\overline{\rho}, \overline{\tau})$*, then there exists no substitution Ω such that $\Omega(\overline{\rho}) = \Omega(\overline{\tau})$.*

Proof. Expanding definitions, we must show that $\neg\mathsf{unify}(\overline{\rho}, \mathsf{flatten}(\overline{\tau}))$ implies $\neg\mathsf{unify}(\overline{\rho}, \overline{\tau})$. We will show the contrapositive. Thus, we assume Ω such that $\Omega(\overline{\rho}) = \Omega(\overline{\tau})$ and we must find Ω' such that $\Omega'(\overline{\rho}) = \Omega'(\mathsf{flatten}(\overline{\tau}))$.

Choose $\Omega' = \Omega \circ \mathsf{flatten}^{-1}$. Because the free variables in $\overline{\rho}$ are distinct from the variables in the domain of $\mathsf{flatten}^{-1}$, we have

$\Omega'(\overline{\rho}) = \Omega(\overline{\rho})$. We also have

$$\Omega'(\mathsf{flatten}(\overline{\tau})) = \Omega(\mathsf{flatten}^{-1}(\mathsf{flatten}(\overline{\tau}))) = \Omega(\overline{\tau})$$

and we are done. □

Property (Apartness can be regained after reduction [Property 14])**.** *If* $\overline{\tau} = \Omega(\overline{\rho})$ *and* $\Sigma \vdash \overline{\tau} \rightsquigarrow \overline{\tau'}$*, then there exists a* $\overline{\tau''}$ *such that*

1. $\Sigma \vdash \overline{\tau'} \rightsquigarrow^{*} \overline{\tau''}$,
2. $\overline{\tau''} = \Omega'(\overline{\rho})$ *for some* Ω'*, and*
3. *for every* $\overline{\rho'}$ *such that* $\mathsf{apart}(\overline{\rho'}, \overline{\tau})$*:* $\mathsf{apart}(\overline{\rho'}, \overline{\tau''})$.

Proof. We know that $\overline{\tau}$ matches some pattern $\overline{\rho}$ and that one element in $\overline{\tau}$ steps, forming $\overline{\tau'}$. Suppose that one element is τ_k. Thus, $\Sigma \vdash \tau_k \rightsquigarrow \tau'_k$. Inverting this step relation gives us that $\tau_k = \mathcal{C}[F(\overline{\upsilon})]$ and $\tau'_k = \mathcal{C}[\upsilon']$, where $F(\overline{\upsilon})$ reduces to υ' at the top level.

Define $\mathfrak{C}[\![\cdot]\!]$ to be the list of types $\overline{\tau}$ such that every occurrence of $F(\overline{\upsilon})$ is replaced by $\cdot$. Thus, $\mathfrak{C}[\![F(\overline{\upsilon})]\!] = \overline{\tau}$. We choose $\overline{\tau''}$ (from the statement of the property) to be $\mathfrak{C}[\![\upsilon']\!]$. We must show the following:

- $\Sigma \vdash \overline{\tau'} \rightsquigarrow^{*} \overline{\tau''}$: Straightforward application of the rule RED.
- $\overline{\tau''} = \Omega'(\overline{\rho})$ for some Ω': Because $\overline{\rho}$ cannot contain type families, it must be that Ω maps some variables to types containing $F(\overline{\upsilon})$. Choose Ω' to be Ω with all occurrences of $F(\overline{\upsilon})$ replaced by υ'. Because *all* occurrences of $F(\overline{\upsilon})$ in $\overline{\tau}$ have been replaced by υ', we can see that $\Omega'(\overline{\rho})$ must be $\overline{\tau''}$.
- For every $\overline{\rho'}$ such that $\mathsf{apart}(\overline{\rho'}, \overline{\tau})$, we have $\mathsf{apart}(\overline{\rho'}, \overline{\tau''})$: Assume we have $\overline{\rho'}$ such that $\mathsf{apart}(\overline{\rho'}, \overline{\tau})$. Unfolding definitions (and taking the contrapositive) gives us ω such that $\omega(\overline{\rho'}) = \omega(\mathsf{flatten}(\overline{\tau''}))$, and we must find ω' such that

$\omega'(\overline{\rho'}) = \omega'(\mathsf{flatten}(\overline{\tau}))$.

Let α be the variable mapped from $F(\overline{\upsilon})$. Thus, $\mathsf{flatten}(F(\overline{\upsilon})) = \alpha$. Let $\Omega_0 = [\alpha \mapsto \upsilon']$ and choose $\omega' = \omega \circ \Omega_0$. Noting that α does not appear in $\overline{\rho'}$, we see that $\omega'(\overline{\rho'}) = \omega(\overline{\rho'})$. Now, we must only show that $\omega'(\mathsf{flatten}(\overline{\tau})) = \omega(\mathsf{flatten}(\overline{\tau''}))$. By our choice of ω', we know $\omega'(\mathsf{flatten}(\overline{\tau})) = \omega(\Omega_0(\mathsf{flatten}(\overline{\tau})))$, thus we must show $\Omega_0(\mathsf{flatten}(\overline{\tau})) = \mathsf{flatten}(\overline{\tau''})$. By its definition, $\mathsf{flatten}$ takes all occurrences of $F(\overline{\upsilon})$ in $\overline{\tau}$ to α. Then, Ω_0 takes all of these occurrences of α to υ'. Since the only difference between $\overline{\tau}$ and $\overline{\tau''}$ is that all occurrences of $F(\overline{\upsilon})$ are replaced by υ', we can see that $\Omega_0(\mathsf{flatten}(\overline{\tau}))$ is indeed $\mathsf{flatten}(\overline{\tau''})$, and we are done.

□

F.2 Proofs of Properties 2 and 4

Property (Apartness through substitution [Property 2])**.** *If* $\mathsf{apart}(\rho, \tau)$ *then there exists no* Ω *such that* $\mathsf{match}(\rho, \Omega(\tau))$.

Proof. We shall prove by contradiction: assume Ω and Ω' such that $\Omega'(\rho) = \Omega(\tau)$. We can simplify a bit and combine these substitutions, because the free variables of ρ are distinct from those in τ; we can say $\Omega_0(\rho) = \Omega_0(\tau)$. Then, this is a contradiction, appealing to Property 13, and we are done. □

The next property (Property 4) requires an important auxiliary lemma.

Lemma 46 (Matching can be regained after reduction)**.** *If* **Good** Σ *and* $\Sigma \vdash \Omega(\rho) \rightsquigarrow^* \tau$ *then there exists an* Ω' *such that* $\Sigma \vdash \tau \rightsquigarrow^* \Omega'(\rho)$.

Proof. Throughout this proof, we will consider types as abstract syntax trees. We will use "type" and "tree" interchangeably.

Define the operation $\mathsf{linearize}$ to take a pattern and freshen all the type variables therein, thus producing a linear pattern. Our first step is to show that τ matches $\mathsf{linearize}(\rho)$. How does $\Omega(\rho)$ step to τ? It must be through a series of type family reductions. Because ρ does not mention type families, these type families must occur in $\Omega(\rho)$ *at or beneath* where variables appear in the tree ρ. Thus, as $\Omega(\rho)$ steps, the tree structure imposed by ρ does not change. However, it is possible that a type family application, say $F(\overline{v})$ steps in two different ways throughout the tree $\Omega(\rho)$ as $\Omega(\rho)$ is reducing. Thus, we can claim only that τ matches $\mathsf{linearize}(\rho)$, not ρ itself.

When comparing the trees τ and ρ, define a *mismatch* to be two locations in the respective trees where ρ has a repeated variable and τ has two different sub-trees. Count only those matches that involve the left-most occurrence of a variable in ρ. We proceed by induction on the number of mismatches between ρ and τ.

Base case: If there are no mismatches, then we know that ρ must match τ with a substitution Ω'. We are done.

Inductive case: Choose the left-most mismatch. Say that the repeated variable in ρ is α and the disagreeing types in τ are σ_1 and σ_2. We know that $\Sigma \vdash \Omega(\rho) \rightsquigarrow^* \tau$, and thus that $\Sigma \vdash \Omega(\alpha) \rightsquigarrow^* \sigma_1$ and $\Sigma \vdash \Omega(\alpha) \rightsquigarrow^* \sigma_2$. By confluence (Lemma 35), we know that there exists a σ_3 such that

$\Sigma \vdash \sigma_1 \rightsquigarrow^* \sigma_3$ and $\Sigma \vdash \sigma_2 \rightsquigarrow^* \sigma_3$. Let τ' be τ, except that both σ_1 and σ_2 in τ are replaced by σ_3 in τ'. We know that $\Sigma \vdash \tau \rightsquigarrow^* \tau'$ by congruence of the rewrite relation and thus that $\Sigma \vdash \Omega(\rho) \rightsquigarrow^* \tau'$. Thus, we can use the induction hypothesis to get Ω' such that $\Sigma \vdash \tau' \rightsquigarrow^* \Omega'(\rho)$. Then, by transitivity of $\Sigma \vdash \cdot \rightsquigarrow^* \cdot$, we are done.

□

2013/11/15

Lemma 47 (Matching normal forms)**.** *If* $\mathbf{Good}\,\Sigma$ *and* $\Sigma \vdash \Omega(\rho) \rightsquigarrow^* \upsilon$ *where* υ *is a normal form, then there exists* Ω' *such that* $\upsilon = \Omega'(\rho)$.

Proof. We apply Lemma 46 to see that there exists an Ω' such that $\Sigma \vdash \upsilon \rightsquigarrow^* \Omega'(\rho)$. But, we know that υ cannot step, and thus $\upsilon = \Omega'(\rho)$. □

Lemma 48 (Longest reduction)**.** *Suppose* $\mathbf{Good}\,\Sigma$. *For every type* τ *and its normal form* υ *(whose uniqueness is guaranteed by the combination of confluence and termination), there exists a number* n *such that all reductions from* τ *to* υ *are of length at most* n.

Proof. König's lemma states that every tree with infinitely many vertices, each having finite degree, has at least one infinite simple path. Here, we are considering trees of reductions, rooted at τ. We will use the contrapositive of König's lemma: that if every node in a tree has finite degree and all simple paths are finite, then there are finitely many vertices. For any type σ, there are finitely many types σ' such that $\Sigma \vdash \sigma \rightsquigarrow \sigma'$, because there are finitely many locations within σ that can be headed by a type family and finitely many equations that type family application might match. By termination, we know all simple paths in the tree of reductions are finite. Thus, the contrapositive of König's lemma tells us that the tree has a finite number of nodes. Thus, we can simply enumerate all paths from τ to υ to discover the one with the longest path. This path's length is our result n. □

Lemma 49 (Apartness and normal forms)**.** *If* $\mathsf{apart}(\rho, \tau)$ *and* $\Sigma \vdash \tau \rightsquigarrow^* \upsilon$ *where* υ *is a normal form, then* $\mathsf{apart}(\rho, \upsilon)$.

Proof. Let the longest path from τ to υ be of length n (Lemma 48). We perform induction on n.

Base case: Trivial.

Inductive case: We know that τ can step to some τ'; that is, $\Sigma \vdash \tau \rightsquigarrow \tau'$. We then appeal to Property 14 (choosing $\rho = \mathsf{flatten}(\tau)$, but the choice is irrelevant) to get τ'' such that $\Sigma \vdash \tau' \rightsquigarrow^* \tau''$ and $\mathsf{apart}(\rho, \tau'')$. By our assumption that n is the length of the longest path from τ to υ and the fact that $\Sigma \vdash \tau \rightsquigarrow^* \tau''$ by at least one step, we know that the longest path from τ'' to υ has length less than n. Thus, we can use the induction hypothesis, and we are done.

□

Lemma 50 (Apartness implies no match)**.** *If* $\mathsf{apart}(\rho, \tau)$*, then* $\neg\mathsf{match}(\rho, \tau)$.

Proof. We prove by contradiction. Assume Ω such that $\Omega(\rho) = \tau$. By the assumption that pattern variables are fresh, we can say $\Omega(\rho) = \Omega(\tau)$. Then, by Property 13, we have a contradiction. □

Property (Apartness through reduction and substitution [Property 4])**.** *If* $\mathsf{apart}(\rho, \tau)$*, then for any* τ' *such that* $\tau \leadsto^* \tau'$*:* $\neg\mathsf{match}(\rho, \tau')$.

Proof. Let υ be the unique normal form of τ. By Lemma 49, we know $\mathsf{apart}(\rho, \upsilon)$. By Lemma 50, $\neg\mathsf{match}(\rho, \upsilon)$. Note that the uniqueness of normal forms, we know $\Sigma \vdash \tau' \leadsto^* \upsilon$. By the contrapositive of Lemma 47, we see that $\neg\mathsf{match}(\rho, \tau')$ as desired. □

G. Proof of compatibility soundness

In this appendix, we show that the concrete implementation of compatibility (Definition 8) satisfies the definition of compatibility (Property 7). We use the implementation of compatibility included in our formal inference rules, as it separates Definition 8 into its two cases:

$\boxed{\mathsf{compat}(\Phi_1, \Phi_2)}$ Equation compatibility

$$\begin{array}{l}
\Phi_1 = [\overline{\alpha_1{:}\kappa_1}].\ F(\overline{\rho_1}) \sim \upsilon_1 \\
\Phi_2 = [\overline{\alpha_2{:}\kappa_2}].\ F(\overline{\rho_2}) \sim \upsilon_2 \\
\mathsf{unify}(\overline{\rho_1}, \overline{\rho_2}) = \Omega
\end{array}$$

$$\frac{\Omega(\upsilon_1) = \Omega(\upsilon_2)}{\mathsf{compat}(\Phi_1, \Phi_2)} \quad \textsc{Compat\_Coincident}$$

$$\frac{\begin{array}{l}\Phi_1 = [\overline{\alpha_1{:}\kappa_1}].\ F(\overline{\rho_1}) \sim \upsilon_1 \\ \Phi_2 = [\overline{\alpha_2{:}\kappa_2}].\ F(\overline{\rho_2}) \sim \upsilon_2 \\ \mathsf{unify}(\overline{\rho_1}, \overline{\rho_2}) \text{ fails}\end{array}}{\mathsf{compat}(\Phi_1, \Phi_2)} \quad \textsc{Compat\_Distinct}$$

We generalize Property 7 to work with unify_∞.

Property 51 (Compatibility (with infinite unification))**.** *Two type-family equations p and q are* compatible *iff $\omega_1(\mathit{lhs}_p) = \omega_2(\mathit{lhs}_q)$ implies $\omega_1(\mathit{rhs}_p) = \omega_2(\mathit{rhs}_q)$.*

Proof. For all type family equations Φ_1 and Φ_2, where $\Phi_1 = [\overline{\alpha_1{:}\kappa_1}].\ F(\overline{\rho_1}) \sim \upsilon_1$ and $\Phi_2 = [\overline{\alpha_2{:}\kappa_2}].\ F(\overline{\rho_2}) \sim \upsilon_2$, we must show that $\mathsf{compat}(\Phi_1, \Phi_2)$ implies that, for all ω_1 and ω_2 such that $\omega_1(\overline{\rho_1}) = \omega_2(\overline{\rho_2})$, it is the case that $\omega_1(\upsilon_1) = \omega_2(\upsilon_2)$.

We have two cases:

Case Compat_Coincident: Here, we know that $\omega(\overline{\rho_1}) = \omega(\overline{\rho_2})$ and, by Property 40, that ω is a most general unifier. We further know that $\omega(\upsilon_1) = \omega(\upsilon_2)$. By assumption, $\omega_1(\overline{\rho_1}) = \omega_2(\overline{\rho_2})$. By the assumption that all patterns in type families have distinct variables, we know that the domains of ω_1 and ω_2 are distinct. Thus, we can write $\omega' = \omega_1 \cup \omega_2$, and say $\omega'(\overline{\rho_1}) = \omega'(\overline{\rho_2})$. Similarly, we can say that we wish to show $\omega'(\upsilon_1) = \omega'(\upsilon_2)$. Because ω is a most general unifier, we can say that $\omega' = \omega'' \circ \omega$ for some ω''. Thus, we wish to show $\omega''(\omega(\upsilon_1)) = \omega''(\omega(\upsilon_2))$. But, we know that $\omega(\upsilon_1) = \omega(\upsilon_2)$ so we are done.

Case COMPAT_DISTINCT: Here, we know that there exists no ω such that $\omega(\overline{\rho_1}) = \omega(\overline{\rho_2})$. Yet, we have assumed that $\omega_1(\overline{\rho_1}) = \omega_2(\overline{\rho_2})$ and by an argument similar to the last case, we can combine ω_1 and ω_2 to ω'. This substitution ω is then a unifier, leading to a contradiction.

□