Draft, as of 28 Feb 2014

Safe Coercions (extended version)

Joachim Breitner
Karlsruhe Institute of Technology
breitner@kit.edu

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

Simon Peyton Jones
Microsoft Research
simonpj@microsoft.com

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

Abstract

Generative type abstractions – present in Haskell, OCaml, and other languages – are useful concepts to help prevent programmer errors. They serve to create new types that are distinct at compile time but share a run-time representation with some base type. We present a new mechanism that allows for zero-cost conversions between generative type abstractions and their representations, even when such types are deeply nested. We prove type safety in the presence of these conversions and have implemented our work into GHC.

1. Introduction

Modular languages support *generative type abstraction*, the ability for programmers to define application-specific types, and rely on the type system to distinguish between these new types and their underlying representation. Type abstraction is a powerful tool for programmers, enabling both flexibility (implementors can change representations) and security (implementors can maintain invariants about representations). Typed languages provide these mechanisms with zero run-time cost – there should be no performance penalty for creating abstractions – using mechanisms such as ML's module system [8] and Haskell's **newtype** declaration [7].

For example, in Haskell a programmer might create an abstract type for HTML data, representing them as Strings.

```
module Html( HTML, text, unMk, ... ) where
  newtype HTML = Mk String
  unMk :: HTML → String
  unMk (Mk s) = s
  text :: String → Maybe HTML
  text s = if valid s then Just (Mk s) else Nothing
```

Although String values use the same patterns of bits in memory as HTML values, the two types are distinct. That is, a String will not be accepted by a function expecting an HTML.

The constructor Mk converts a String to an HTML (see function text), while using Mk in a pattern converts in the other direction (see function unMk). Furthermore, by exporting the type HTML, but not its data constructor, module Html ensures that the type HTML is *abstract* – clients cannot make arbitrary strings into HTML – and thereby prevent cross-site scripting attacks.

Using **newtype** for abstraction in Haskell has always suffered from an embarrassing difficulty. Suppose in the module Html, the programmer wants to convert a *list* of HTML to a list of String:

```
concatH :: [HTML] → HTML
concatH hs = Mk (concat (map unMk hs))
```

To get the [String] to pass to concat we are forced to map unMk over the list. Operationally, this map is the identity function – the run-time representation of [String] is identical to [HTML] – *but it will carry a run-time cost nevertheless*. The optimiser in the Glasgow Haskell Compiler (GHC) is powerless to fix the problem, because it works over a *typed* intermediate language; the unMk function changes the type of its operand, and hence cannot be optimised away. What has become of the claim of zero-overhead abstraction?

In this paper we describe a robust, simple solution the problem, making the following contributions:

- We describe the design of *safe coercions* (Section 2), which introduces the function

  ```
  coerce :: Coercible a b ⇒ a → b
  ```

  and a new type class Coercible. This function performs a zero-cost conversion between two types a and b that have the same representation. The crucial question becomes *what instances of Coercible exist?* We give a simple (but non-obvious) strategy (Sections 2.1–2.2), expressed largely in the familiar language of Haskell type classes.
- The strategy depends critically on the concept of *roles* (Section 2.2), a key contribution of this work. Roles ensure soundness, but the new mechanism should *also* preserve abstraction and coherence; we explain the issues and how they can be addressed (Section 3). We also give a role inference algorithm in Section 5.
- The function coerce gives access to the *run-time* (representational) type equality between, say, HTML and String. How can we now be sure that we respect *compile-time*

(nominal) type equality? We make this question precise, and answer it, by formalising the new system in our core calculus, System FC (Section 4). The new calculus includes newtypes and type families, roles, multiple explicit notions of type equality, and coercions to witness equality proofs. We show that it is consistent by giving the usual proofs of preservation and progress.

- Our new approach also resolves a notorious and long-standing bug in GHC (#1496), which concerns the interaction of newtype coercions with type families (Section 7). While earlier work [14] was motivated by the same bug, it was too complicated to implement. Our new approach finds a sweet spot, offering a much simpler system in exchange for a minor loss of expressiveness (Section 8).
- We have implemented role inference and safe coercions in GHC (Section 6), and we show how the usual machinery of rewrite rules can be used to bring these improvements to existing, unmodified code. (Section 6.4)

As this work demonstrates, the interactions between type abstraction and advanced type system features, such as type families and GADTs, are subtle. The ability to create and enforce zero-cost type abstraction is not unique to Haskell – notably the ML module system also provides this capability, and more. As a result, OCaml developers are now grappling with similar difficulties. We discuss the connection between roles and OCaml's variance annotations (Section 8), as well as other related work.

2. The design and interface of Coercible

We begin by focusing exclusively on the programmer's-eye-view of safe coercions. We need no new syntax; rather, the programmer simply sees a new API, provided in just two declarations:

```
class Coercible a b
coerce :: Coercible a b ⇒ a → b
```

The typeclass Coercible is abstract. Its methods are not visible, and it is not possible to create manual instances of this class. Instead, as we shall see, instances are automatically generated by the compiler.

The key principle is this: *If two types s and t are related by Coercible s t, then s and t have bit-for-bit identical run-time representations*. Moreover, as you can see from the type of coerce, if Coercible s t holds then coerce can convert a value of type s to one of type t. And that's it!

The crucial question, to which we devote the rest of this section and the next, becomes this: exactly when does Coercible s t hold? To whet your appetite consider these declarations:

```
newtype Age      = MkAge Int
newtype AgeRange = MkAR (Int,Int)
newtype BigAge   = MkBig Age
```

Here are some coercions that hold, so that a single call to coerce suffices to convert between the two types:

- Coercible Int Age: we can coerce from Int to Age at zero cost; this is simply the MkAge constructor.
- Coercible Age Int: and the reverse (pattern matching on MkAge).
- Coercible [Age] [Int]: lifting the coercion over lists.
- Coercible (Either Int Age) (Either Int Int): lifting the coercion over Either.

GHC generates the following instances of Coercible:

1. **instance** Coercible a a
2. For every **newtype** NT x = MkNT (T x), the instances

   ```
   instance Coercible (T x) b ⇒ Coercible (NT x) b
   instance Coercible a (T x) ⇒ Coercible a (NT x)
   ```

   which are visible if and only if the constructor MkNT is in scope.
3. For every type constructor TC r p n, where
   - r stands for TC's parameters at role representational,
   - p for those at role phantom and
   - n for those at role nominal,

   the instance

   ```
   instance Coercible r1 r2 ⇒
          Coercible (TC r1 p1 n) (TC r2 p2 n)
   ```

Figure 1. Coercible instances

- Coercible (Either Int Age) (Either Age Int): this is more complicated, because first argument of Either must be coerced in one direction, and the second in the other.
- Coercible (Int → Age) (Age → Int): all this works over function arrows too.
- Coercible (Age, Age) AgeRange: we have to unwrap the pair of Ages and then wrap with MkAR.
- Coercible [BigAge] [Int]: two levels of coercion.

In the rest of this section we will describe how Coercible constraints are solved or, equivalently, which instances of Coercible exist. (See Figure 1 for a concise summary.)

2.1 Coercing newtypes

Since Coercible relates a newtype with its base type, we need Coercible instance declarations for every such newtype. The naive **instance** Coercible Int Age does not work well, for reasons explained in the box on page 3, so instead we generate *two* instances for each newtype:

```
instance Coercible a Int ⇒ Coercible a Age  — (A1)
instance Coercible Int b ⇒ Coercible Age b  — (A2)

instance Coercible a Age ⇒ Coercible a BigAge  — (B1)
instance Coercible Age b ⇒ Coercible BigAge b  — (B2)

instance Coercible a AgeRange ⇒ Coercible a (Int,Int)
instance Coercible AgeRange b ⇒ Coercible (Int,Int) b
```

Notice that each instance unwraps just one layer of the newtype, so we call them the "unwrapping instances".

If we now want to solve, say, a constraint Coercible s Age, for any type s, we can use (A1) to reduce it to the simpler goal Coercible s Int. A more complicated, two-layer coercion Coercible BigAge Int is readily reduced, in two such steps, to Coercible Int Int. All we need now is for GHC to have a built-in witness of reflexivity, expressing that any type has the same run-time representation as itself:

```
instance Coercible a a
```

This simple scheme allows coercions that involve arbitrary levels of wrapping or unwrapping, in either direction, with a single call to coerce. The solution path is not fully determined, but that does not matter. For example, here are two ways to solve Coercible BigAge Age:

```
      Coercible BigAge Age
  ⟶   Coercible BigAge Int     — By (A1)
  ⟶   Coercible Age Int        — By (B2)
  ⟶   Coercible Int Int        — By (A2)
  ⟶   solved                   — By reflexivity

      Coercible BigAge Age
  ⟶   Coercible Age Age        — By (B2)
  ⟶   solved                   — By reflexivity
```

Since Coercible constraints have no run-time behaviour (unlike normal type-class constraints), we have no concerns about incoherence; any solution will do.

The newtype-unwrapping instances (i.e., (2) in Figure 1) are available *only if the corresponding newtype data constructor* (Mk in our current example) *is in scope*; this is required to preserve abstraction, as we explain in Section 3.1.

2.2 Coercing under type constructors

As Figure 1 shows, as well as the unwrapping instances for a **newtype**, we also generate one instance for each type constructor, including data types, newtypes (see Section 3.2), the function type, and built-in data types like tuples. We call this instance the "lifting instance" for the type, because it lifts coercions through the type. The shape of the instance depends on the so-called *roles* of the type constructor. Each type parameter of a type constructor has a role, determined by the way in which the parameter is used in the definition of the type constructor. In practice, the roles of a declared datatype are determined by a role inference algorithm (Section 5) and can be modified by role annotations (Section 3.3).

Roles, a development of earlier work [14] (Section 8), are a new concept for the programmer. There are three possible roles, *representational*, *phantom* or *nominal*, which are discussed in the following subsections.

2.2.1 Coercing representational type parameters

The most common role is *representational*. It is the role that is assigned to the type parameters of ordinary data types like Maybe, the list type and Either. It is also the role of the arrow type constructor's parameters. The Coercible instances for these type constructors are:

```
instance Coercible a b ⇒ Coercible (Maybe a) (Maybe b)
instance Coercible a b ⇒ Coercible [a] [b]
instance (Coercible a1 b1, Coercible a2 b2)
        ⇒ Coercible (Either a1 a2) (Either b1 b2)
instance (Coercible a1 b1, Coercible a2 b2)
        ⇒ Coercible (a1 → a2) (b1 → b2)
```

These instances are just as you would expect: for example, the type Maybe t1 and Maybe t2 have the same run-time representation if and only if t1 and t2 have the same representation. Returning to the introduction, we can use these instances to write concatH very directly, thus:

```
concatH :: [HTML] → HTML
concatH hs = Mk (concat (coerce hs))
```

or even

```
concatH = coerce (concat :: [String] → String)
```

In the former case, the call to coerce gives rise to a constraint Coercible [HTML] [String], which gets simplified to Coercible HTML String using the instance for the list type.

Why a single instance is not enough

Why do we create two instances for every newtype, rather than just the single declaration

```
instance Coercible Int Age
```

to witness the fact that Int and Age have the same run-time representation?

That would indeed allow us to convert from Int to Age, using coerce, but what about the reverse direction? We then might need a second function

```
uncoerce :: Coercible a b ⇒ b → a
```

although it would be tiresome for the programmer to remember which one to call. Alternatively, perhaps GHC should generate *two* instances:

```
instance Coercible Int Age
instance Coercible Age Int
```

But how would we get from BigAge to Int? We could try this:

```
down :: BigAge → Int
down x = coerce (coerce x)
```

Our intent here is that each invocation of coerce unwraps one "layer" of newtype. But this is not good, because the type inference engine cannot figure out which type to use for the result of the inner coerce. To make the code typecheck we would have to add a type signature:

```
down :: BigAge → Int
down x = coerce (coerce x :: Age)
```

Not very nice. Moreover we would prefer to do all this with a *single* call to coerce, implying that Coercible BigAge Int must hold. That might make us consider adding the instance declaration

```
instance (Coercible a b, Coercible b c) ⇒ Coercible a c
```

to express the transitivity of Coercible. But now the problem of the un-specified intermediate type b re-appears, and cannot be solved with a type signature.

All of these problems are nicely solved using the instances in Figure 1.

Then the instance for the newtype HTML reduces it to Coercible String String, which is solved by the reflexive instance. In the latter case, we need an explicit type annotation so that the instance solver knows where to begin its search – it cannot solve a constraint Coercible ([[a]] → [a]) ([HTML] → HTML) without an instantiation for a.

2.2.2 Coercing phantom type parameters

A type parameter has a *phantom* role if it does not occur in the definition of the type, or if it does, then only as a phantom parameter of another type constructor. For example, these declarations

```
data Phantom b = Phantom
data NestedPhantom b = L [Phantom b] | SomethingElse
```

both have b at a phantom role.

When do the types Phantom t1 and Phantom t2 have the same run-time representation? Always! Therefore, we have the instances

```
instance Coercible (Phantom a) (Phantom b)
instance Coercible (NestedPhantom a) (NestedPhantom b)
```

and coerce can be used to change the phantom parameter arbitrarily.

2.2.3 Coercing nominal type parameters

In contrast, the *nominal* role induces the strictest preconditions for Coercible instances. This role is assigned to a parameter that possibly affects the run-time representation of a type, commonly because it is passed to a type function. For example, consider the following code

```
type family EncData a where
  EncData String = (ByteString, Encoding)
  EncData HTML = ByteString

data Encoding = ...
data EncText a = MkET (EncData a)
```

Even though we have Coercible HTML String, it would be wrong to derive the instance Coercible (EncText HTML) (EncText String), because these two types have quite different run-time representations! Therefore, there are no instances that change a nominal parameter of a type constructor.

All the parameters of a type or data *family* have nominal role, since they are potentially inspected by the type-family instances.

2.2.4 Coercing multiple type parameters

A type constructor can have multiple type parameters, each at a different role. In that case, an appropriate constraint for each type parameter is used:

```
data Params r p n = Con1 (Maybe r) | Con2 (EncData n)
```

yields the instance

```
instance Coercible r1 r2
         ⇒ Coercible (Params r1 p1 n) (Params r2 p2 n)
```

This instance expresses that the representational type parameters may change if there is a Coercible instance for them; the phantom type parameters may change arbitrarily; and the nominal type parameters must stay the same.

3. Abstraction and coherence

The purpose of the HTML type from the introduction is to prevent accidentally mixing up unescaped strings and HTML fragments. Rejecting programs that make this mistake is not a matter of type safety as traditionally construed, but rather of preserving a desired abstraction.

While the previous section described how the Coercible instance declarations ensure that uses of coerce are type-safe, this section discusses how we preserve two other properties: *abstraction* and *class coherence*.

3.1 Abstraction and unwrapping newtypes

The goal of coerce is to offer a zero-cost conversion between two types; *not* to enable users to write code that was previously impossible, which would risk betraying a programmer's intent of type abstraction. Thus, our general principle is this: *any use of coerce should be semantically equivalent to some legal hand-written code*. (There may be efficiency differences, however.) If this principle holds, we cannot violate any existing abstraction boundaries. Conversely, if coerce can do something that could not be done before, we need to consider the consequences carefully.

The unwrapping instances for a newtype give the programmer the same power as the newtype data constructor itself so, following the principle, we make those instances available[1] only if the data constructor is in scope (Section 2.1). For example, since the author of module Html did not export Mk, a client does not see the unwrapping instances for HTML, and the abstraction is preserved.

3.2 Lifting over abstractions

On many occasions, though, we want to contradict our principle. Consider

```
module BagLib( Bag, emptyBag, unionBags, ... ) where
  data Bag a = MkBag [a]
  ...etc...
```

The module does not export the MkBag data constructor, because we might later want to change the representation of Bag; it is an abstract data type. But we *do* want to be able to coerce from (Bag HTML) to (Bag String), using the lifting coercions of Section 2.2, *even though a client of BagLib could not write the code to do so* (lacking access to MkBag). So the lifting instances are made available regardless of the visibility of the data constructors. This also applies to the lifting instance for a **newtype**; just imagine that Bag was declared above with **newtype** instead of **data**.

3.3 Abstraction through role annotations

Although it is usually right to expose the lifting instance for data type, it is sometimes dead wrong. Consider the data type Map k v, which implements an efficient finite map from keys of type k to values of type v, using an internal representation based on a balanced tree, something like this:

```
data Map k v = Leaf | Node k v (Map k v) (Map k v)
```

It would be disastrous if the user were allowed to to coerce from (Map Age v) to (Map Int v), because a valid tree with regard to the ordering of Age might be a completely bogus when using the ordering of Int. On the other hand we certainly *do* want the ability to coerce Map k HTML to Map k String, just as in the previous section. However, in the declaration of Map the parameters k and v are used in exactly the same way, so no inference mechanism can guess that they should be treated differently by Coercible.

Thus motivated we allow the programmer to use a *role annotation* to specify role for each type parameter. For example:

```
type role Map nominal representational
```

Based on these declared roles, the rules of Section 2.2 will generate the desirable and useful instance declaration

```
instance Coercible a b ⇒ Coercible (Map k a) (Map k b)
```

that preserves the abstraction of Map.

The compiler ensures that role annotations cannot subvert the type system: if the annotation specifies an unsafe role, the compiler will reject the program.

[1] Instance lookup for Coercible uses a customised algorithm to support this behaviour.

3.4 Preserving class coherence

Another property of Haskell, independent of type-safety, is the coherence of type classes. There should only ever be one class instance for a particular class and type. We call this desirable property *coherence*. Without extra checks, Coercible could be used to create incoherence.

Consider this (non-Haskell98) data type, which reifies a Show instance as a value:

```
data HowToShow a where
  MkHTS :: Show a ⇒ HowToShow a

showH :: HowToShow a → a → String
showH MkHTS x = show x
```

Here showH pattern-matches on a HowToShow value, and uses the instance stored inside it to obtain the show method. If we are not careful, the following code would break the coherence of the Show type class:

```
instance Show HTML where
  show s = "HTML:" ++ show s

stringShow :: HowToShow String
stringShow = MkHTS
htmlShow :: HowToShow HTML
htmlShow = MkHTS
badShow :: HowToShow HTML
badShow = coerce stringShow

λ> showH stringShow "Hello"
"Hello"
λ> showH htmlShow (Mk "Hello")
"HTML:Hello"
λ> showH badShow (Mk "Hello")
"Hello"
```

In the final example we were applying show to a value of type HTML, but the Show instance for String (coerced to (Show HTML)) was used.

To avoid this confusion, the parameters of a type class are all assigned a *nominal* role. Accordingly, the parameter of HowToShow is also assigned a nominal role, preventing the coercion between (HowToShow HTML) and (HowToShow String).

4. Ensuring type safety: System FC with roles

Haskell is a large and complicated language. How do we know that the ideas sketched above in source-language terms are actually sound? What, precisely, do roles mean, and when precisely are two types equal? In this section we answer these questions for GHC's small, statically-typed intermediate language, GHC Core. Every Haskell program is translated into Core, and we can typecheck Core to reassure ourselves that the (large, complicated) front end accepts only good programs.

Core is an implementation of a calculus called System FC, itself an extension of the classical Girard/Reynolds System F. A full exposition of FC[2] is beyond the scope of this work, but it is well documented elsewhere (e.g. Yorgey et al. [15]).

[2] Several versions of System FC are described in published work. Some of these variants have had decorations to the FC name, such as FC_2 or $F_C^{\uparrow}$. We do not make these distinctions in the present work, referring instead to all of these systems – in fact, one evolving system – as "FC".

Metavariables:

x	term	α, β	type	c	coercion
C	axiom	D	datatype	N	newtype
F	type family	K	data constructor		

e	$::= \lambda c{:}\phi.e \mid e\,\gamma \mid e \rhd \gamma \mid \cdots$	terms
τ, σ	$::= \alpha \mid \tau_1\,\tau_2 \mid \forall\,\alpha{:}\kappa.\tau \mid H \mid F(\overline{\tau})$	types
κ	$::= \star \mid \kappa_1 \to \kappa_2$	kinds
H	$::= (\to) \mid (\Rightarrow) \mid (\sim^{\kappa}_{\rho}) \mid T$	type constants
T	$::= D \mid N$	algebraic datatypes
ϕ	$::= \tau \sim^{\kappa}_{\rho} \sigma$	proposition
γ, η	$::=$	coercions
	$\mid \langle\tau\rangle \mid \langle\tau,\sigma\rangle_{\mathsf{P}} \mid \mathbf{sym}\,\gamma \mid \gamma_1 \mathbin{;} \gamma_2$	equivalence
	$\mid H(\overline{\gamma}) \mid F(\overline{\gamma}) \mid \gamma_1\,\gamma_2 \mid \forall\alpha{:}\kappa.\gamma$	congruence
	$\mid c \mid C(\overline{\tau})$	assumptions
	$\mid \mathbf{nth}^i\,\gamma \mid \mathbf{left}\,\gamma \mid \mathbf{right}\,\gamma \mid \gamma@\tau$	decomposition
	$\mid \mathbf{sub}\,\gamma$	sub-roling
ρ	$::= \mathsf{N} \mid \mathsf{R} \mid \mathsf{P}$	roles
Γ	$::= \varnothing \mid \Gamma, \alpha{:}\kappa \mid \Gamma, c{:}\phi \mid \Gamma, x{:}\tau$	typing contexts
Ω	$::= \varnothing \mid \Omega, \alpha{:}\rho$	role contexts

Figure 2. An excerpt of the grammar of System FC

Figure 2 gives the syntax of System FC. The starting point is an entirely conventional lambda calculus in the style of System F. We therefore elide most of the syntax of terms e, giving the typing judgement for terms in Appendix C.2.Types τ are also conventional, except that we add (saturated) type-family applications $F(\overline{\tau})$, to reflect their addition to source Haskell [1, 2]. Types are classified by kinds κ in the usual way; the kinding judgement $\Gamma \vdash \tau : \kappa$ on types is conventional and appears in Appendix C.2. To avoid clutter we use only monomorphic kinds, but it is easy to add kind polymorphism along the lines of Yorgey et al. [15], and our implementation does so.

4.1 Roles and casts

FC's distinctive feature is a type-safe cast $(e \rhd \gamma)$ (Figure 2), which uses a *coercion* γ to cast a term from one type to another. A coercion γ is a witness or proof of the equality of two types. Coercions are classified by the judgement

$$\Gamma \vdash \gamma : \tau \sim^{\kappa}_{\rho} \sigma$$

given in Figure 3, and pronounced "in type environment Γ the coercion γ witnesses that the types τ and σ both have kind κ, and are equal at role ρ". The notion of being "equal at role ρ" is the new feature of this paper; it is a development of earlier work, as Section 8 describes. There are precisely three roles (Figure 2), written N, R, and P, with the following meaning:

Nominal equality, written $\sim_{\mathsf{N}}$, is the equality that the type checker reasons about. When a Haskell programmer says that two Haskell types are the "same", we mean that the types are nominally equal. Thus, we can say that Int $\sim_{\mathsf{N}}$ Int. Type families introduce new nominal equalities. So, if we have **type instance** F Int = Bool, then F Int $\sim_{\mathsf{N}}$ Bool.

Representational equality, written $\sim_{\mathsf{R}}$, holds between two types that share the same run-time representation. Because all types that are nominally equal also share the same representation, nominal equality is a subset of representational equality. Continuing the example from the introduction, $\mathsf{HTML} \sim_{\mathsf{R}} \mathsf{String}$.

Phantom equality, written $\sim_{\mathsf{P}}$, holds between any two types, whatsoever. It may seem odd that we produce and consume proofs of this "equality", but doing so keeps the system uniform and easier to reason about. The idea of phantom equality is new in this work, and it allows for zero-cost conversions among types with phantom parameters.

We can now give the typing judgement for type-safe cast:

$$\frac{\Gamma \vdash e : \tau_1 \qquad \Gamma \vdash \gamma : \tau_1 \sim_{\mathsf{R}} \tau_2}{\Gamma \vdash e \rhd \gamma : \tau_2}\ \textsc{Tm\_Cast}$$

The coercion γ must be a proof of *representational* equality, as witnessed by the R subscript to the result of the coercion typing premise. This makes good sense: we can treat an expression of one type τ_1 as an expression of some other type τ_2 if and only if those types share a representation.

4.2 Coercions

Coercions (Figure 2) and their typing rules (Figure 3) are the heart of System FC. The basic typing judgement for coercions is $\Gamma \vdash \gamma : \tau \sim_\rho^\kappa \sigma$. When this judgement holds, it is easy to prove that τ and σ must have the same kind κ. However, kinds are not very relevant to the focus of this work, and so we often omit the kind annotation in our presentation. It can always be recovered by using the (syntax-directed) kinding judgement on types.

We can understand the typing rules in Figure 3, by thinking about the equalities that they define.

4.2.1 Nominal implies representational

If we have a proof that two types are nominally equal, then they are certainly representationally equal. This intuition is expressed by the **sub** operator, and the rule Co_Sub.

4.2.2 Equality is an equivalence relation

Equality is an equivalence relation at all three roles. Symmetry (rule Co_Sym) and transitivity (Co_Trans) work for any role ρ. Reflexivity is more interesting: Co_Refl is a proof of nominal equality only. From this we can easily get representational reflexivity using **sub**. But what does "phantom" reflexivity mean? It is a proof term that any two types τ and σ are equal at role P, and we need a new coercion form to express that, written as $\langle \tau, \sigma \rangle_{\mathsf{P}}$ (rule Co_Phantom).

4.2.3 Axioms for equality

Each newtype declaration, and each type-family instance, gives rise to an FC *axiom*; newtypes give rise to representational axioms, and type-family instances give rise to nominal axioms.[3] For example, the declarations

```
newtype HTML = Mk String
type family F [a] = Maybe a
```

produce the axioms

$$C_1 : \mathsf{HTML} \sim_{\mathsf{R}} \mathsf{String}$$
$$C_2 : [\alpha{:}\star].\mathsf{F}\,([\alpha]) \sim_{\mathsf{N}} \mathsf{Maybe}\,\alpha$$

[3] For simplicity, we are restricting ourselves to *open* type families. Closed type families [4] are readily accommodated.

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

$$\frac{\Gamma \vdash \tau : \kappa}{\Gamma \vdash \langle \tau \rangle : \tau \sim_{\mathsf{N}} \tau}\ \textsc{Co\_Refl}$$

$$\frac{\Gamma \vdash \gamma : \sigma \sim_\rho \tau}{\Gamma \vdash \mathbf{sym}\,\gamma : \tau \sim_\rho \sigma}\ \textsc{Co\_Sym}$$

$$\frac{\Gamma \vdash \gamma_1 : \tau_1 \sim_\rho \tau_2 \qquad \Gamma \vdash \gamma_2 : \tau_2 \sim_\rho \tau_3}{\Gamma \vdash \gamma_1 \fatsemi \gamma_2 : \tau_1 \sim_\rho \tau_3}\ \textsc{Co\_Trans}$$

$$\frac{\overline{\Gamma \vdash \gamma : \tau \sim_\rho \sigma} \qquad \overline{\rho} \text{ is a prefix of } \mathit{roles}(H) \qquad \Gamma \vdash H\,\overline{\tau} : \kappa \qquad \Gamma \vdash H\,\overline{\sigma} : \kappa}{\Gamma \vdash H(\overline{\gamma}) : H\,\overline{\tau} \sim_{\mathsf{R}} H\,\overline{\sigma}}\ \textsc{Co\_TyConApp}$$

$$\frac{\overline{\Gamma \vdash \gamma : \tau \sim_{\mathsf{N}} \sigma} \qquad \Gamma \vdash F(\overline{\tau}) : \kappa \qquad \Gamma \vdash F(\overline{\sigma}) : \kappa}{\Gamma \vdash F(\overline{\gamma}) : F(\overline{\tau}) \sim_{\mathsf{N}} F(\overline{\sigma})}\ \textsc{Co\_TyFam}$$

$$\frac{\Gamma \vdash \gamma_1 : \tau_1 \sim_\rho \sigma_1 \qquad \Gamma \vdash \gamma_2 : \tau_2 \sim_{\mathsf{N}} \sigma_2 \qquad \Gamma \vdash \tau_1\,\tau_2 : \kappa \qquad \Gamma \vdash \sigma_1\,\sigma_2 : \kappa}{\Gamma \vdash \gamma_1\,\gamma_2 : \tau_1\,\tau_2 \sim_\rho \sigma_1\,\sigma_2}\ \textsc{Co\_App}$$

$$\frac{\Gamma, \alpha{:}\kappa \vdash \gamma : \tau \sim_\rho \sigma}{\Gamma \vdash \forall \alpha{:}\kappa.\gamma : \forall\,\alpha{:}\kappa.\tau \sim_\rho \forall\,\alpha{:}\kappa.\sigma}\ \textsc{Co\_ForAll}$$

$$\frac{\Gamma \vdash \tau : \kappa \qquad \Gamma \vdash \sigma : \kappa}{\Gamma \vdash \langle \tau, \sigma \rangle_{\mathsf{P}} : \tau \sim_{\mathsf{P}} \sigma}\ \textsc{Co\_Phantom}$$

$$\frac{c{:}\tau \sim_\rho \sigma \in \Gamma}{\Gamma \vdash c : \tau \sim_\rho \sigma}\ \textsc{Co\_Var}$$

$$\frac{C : [\overline{\alpha{:}\kappa}].\sigma_1 \sim_\rho \sigma_2 \qquad \overline{\Gamma \vdash \tau : \kappa}}{\Gamma \vdash C(\overline{\tau}) : \sigma_1[\overline{\tau/\alpha}] \sim_\rho \sigma_2[\overline{\tau/\alpha}]}\ \textsc{Co\_Axiom}$$

$$\frac{\Gamma \vdash \gamma : H\,\overline{\tau} \sim_{\mathsf{R}} H\,\overline{\sigma} \qquad \overline{\rho} \text{ is a prefix of } \mathit{roles}(H) \qquad H \text{ is not a } \mathbf{newtype}}{\Gamma \vdash \mathbf{nth}^i\,\gamma : \tau_i \sim_{\rho_i} \sigma_i}\ \textsc{Co\_Nth}$$

$$\frac{\Gamma \vdash \gamma : \tau_1\,\tau_2 \sim_{\mathsf{N}} \sigma_1\,\sigma_2 \qquad \Gamma \vdash \tau_1 : \kappa \qquad \Gamma \vdash \sigma_1 : \kappa}{\Gamma \vdash \mathbf{left}\,\gamma : \tau_1 \sim_{\mathsf{N}} \sigma_1}\ \textsc{Co\_Left}$$

$$\frac{\Gamma \vdash \gamma : \tau_1\,\tau_2 \sim_{\mathsf{N}} \sigma_1\,\sigma_2 \qquad \Gamma \vdash \tau_2 : \kappa \qquad \Gamma \vdash \sigma_2 : \kappa}{\Gamma \vdash \mathbf{right}\,\gamma : \tau_2 \sim_{\mathsf{N}} \sigma_2}\ \textsc{Co\_Right}$$

$$\frac{\Gamma \vdash \gamma : \forall\,\alpha{:}\kappa.\tau_1 \sim_\rho \forall\,\alpha{:}\kappa.\sigma_1 \qquad \Gamma \vdash \tau : \kappa}{\Gamma \vdash \gamma @ \tau : \tau_1[\tau/\alpha] \sim_\rho \sigma_1[\tau/\alpha]}\ \textsc{Co\_Inst}$$

$$\frac{\Gamma \vdash \gamma : \tau \sim_{\mathsf{N}} \sigma}{\Gamma \vdash \mathbf{sub}\,\gamma : \tau \sim_{\mathsf{R}} \sigma}\ \textsc{Co\_Sub}$$

Figure 3. Formation rules for coercions

Axiom C_1 states that HTML is *representationally* equal to String (since they are distinct types, but share a common representation), while C_2 states that $F([\sigma])$ is *nominally* equal to Maybe σ (meaning that the two are considered to be the same type by the type checker). In C_2, the notation "$[\alpha{:}\star]$." binds α in the types being equated. Uses of these axioms are governed by the rule CO_AXIOM. Axioms must always appear fully applied, and we assume that they live in a global context, separate from the local context Γ.

4.2.4 Equality can be abstracted

Just as one can abstract over types and values in System F, one can also abstract over equality proofs in FC. To this end, FC terms (Figure 2) include coercion abstraction $\lambda c{:}\phi.e$ and application $e\,\gamma$. These are the introduction and elimination forms for the coercion-abstraction arrow ($\Rightarrow$), just as ordinary value abstraction and application are the introduction and elimination forms for ordinary arrow ($\rightarrow$) (see Appendix C.2).

A coercion abstraction binds a coercion variable $c{:}\phi$. These variables can occur only in coercions; see the entirely conventional rule CO_VAR. Coercion variables can also be bound in the patterns of a **case** expression, which supports the implementation of generalised algebraic data types (GADTs).

4.2.5 Equality is congruent

Several rules witness that, ignoring roles, equality is *congruent* – for example, if $\sigma \sim_\rho \tau$ then Maybe $\sigma \sim_\rho$ Maybe τ. However, the roles in these rules deserve some study, as they are the key to understanding the whole system.

Congruence of type application Before diving into the rules themselves, it is helpful to consider some examples of how we want congruence and roles to interact. Let's consider the following definitions:

```
newtype HTML = Mk String

type family F a
type instance F String  = Int
type instance F HTML  = Bool

data T a = MkT (F a)
```

With these definitions in hand, what equalities should be derivable? (Recall the intuitive meanings of the different roles in Section 4.1.)

1. Should Maybe HTML $\sim_R$ Maybe String hold? Yes, it should. The type parameter to Maybe has a representational role, so it makes sense that two Maybes built out of representationally equal types should be representationally equal.
2. Should Maybe HTML $\sim_N$ Maybe String hold? Certainly not. These two types are entirely distinct to Haskell programmers and its type checker.
3. Should T HTML $\sim_R$ T String hold? Certainly not. We can see, by unfolding the definition for T, that the representations of the two types should be different.
4. Should α HTML $\sim_R$ α String hold, for some type variable α? It depends on the instantiation of α! If α becomes Maybe, then "yes"; if α becomes T, then "no". Since we may be abstracting over α, we do not know which of the two will happen, so we take the conservative stance and say that α HTML $\sim_R$ α String does *not* hold.

This last point is critical. The alternative is to express α's argument roles in its kind, but that leads to a much more complicated system; see related work in Section 8. A distinguishing feature of this paper is the substantial simplification we obtain by attributing roles only to the arguments to type constants (H, in the grammar), and not to abstracted type variables. We thereby lose a little expressiveness, but we have not found that to be a big problem in practice. See Section 8.1 for an example of an easily-fixed problem case.

To support both (1) and (4) requires two coercion forms and corresponding typing rules:

- The coercion form $H(\overline{\gamma})$ has an explicit type constant at its head. This form always proves a representational equality, and it requires input coercions of the roles designated by the roles of H's parameters (rule CO_TYCONAPP). The *roles* function gives the list of roles assigned to H's parameters, as explained in Section 2.2. We allow $\overline{\rho}$ to be a prefix of *roles*(H) to accommodate partially-applied type constants.
- The coercion form $\gamma_1\,\gamma_2$ does not have an explicit type constant, so we must use the conservative treatment of roles discussed above. Rule CO_APP therefore requires γ_2 to be a nominal coercion, though the role of γ_1 carries through to $\gamma_1\,\gamma_2$.

What if we wish to prove a nominal equality such as Maybe (F String) $\sim_N$ Maybe Int? We can't use the $H(\overline{\gamma})$ form, which proves only representational equality, but we can still use the $\gamma_1\,\gamma_2$ form. The leftmost coercion would just be $\langle$Maybe$\rangle$.

Congruence of type family application Rule CO_TYFAM proves the equality of two type-family applications. It requires nominal coercions among all the arguments. Why? Because type families can inspect their (type) arguments and branch on them. We would not want to be able to prove any equality between F String and F HTML.

Congruence of polymorphic types The rule CO_FORALL works for any role ρ; polymorphism and roles do not interact.

4.2.6 Equality can be decomposed

If we have a proof of Maybe $\sigma \sim_\rho$ Maybe τ, should we be able to get a proof of $\sigma \sim_\rho \tau$, by decomposing the equality? Yes, in this case, but we must be careful here as well.

Rule CO_NTH is almost an inverse to CO_TYCONAPP. The difference is that CO_NTH prohibits decomposing equalities among newtypes. Why? Because **nth** witnesses injectivity and newtypes are not injective! For example, consider these definitions:

```
data Phant a = MkPhant
newtype App a b = MkApp (a b)
```

Here, *roles*(App) = R, N. (The roles are inferred during compilation; see Section 5.) Yet, we can see the following chain of equalities:

$$\text{App Phant Int} \sim_R \text{Phant Int} \sim_R \text{Phant Bool} \sim_R \text{App Phant Bool}$$

By transitivity, we can derive a coercion γ witnessing

$$\text{App Phant Int} \sim_R \text{App Phant Bool}$$

If we could use $\mathbf{nth}^2$ on γ, we would get Int $\sim_N$ Bool: disaster! We eliminate this possibility by preventing **nth** on newtypes.

The rules CO_LEFT and CO_RIGHT are almost inverses to CO_APP. The difference is that both CO_LEFT and CO_RIGHT require and produce only nominal coercions. We need a new newtype to see why this must be so:

newtype EitherInt a = MkEI (Either a Int)

This definition yields an axiom showing that, for all a, EitherInt a $\sim_{\mathsf{R}}$ (Either a Int). Suppose we could apply **left** and **right** to coercions formed from this axiom. Using **left** would get us a proof of EitherInt $\sim_{\mathsf{R}}$ (Either a), which could then be used to show, say, (Either Char) $\sim_{\mathsf{R}}$ (Either Bool) and then (using **nth**) Char $\sim_{\mathsf{N}}$ Bool. Using **right** would get us a proof of a $\sim_{\mathsf{R}}$ Int, for *any* a. These are both clearly disastrous. So, we forbid using these coercion formers on representational coercions.[4]

Thankfully, polymorphism and roles play well together, and the CO_INST rule (inverse to CO_FORALL) shows quite straightforwardly that, if two polytypes are equal, then so are the instantiated types.

There is no decomposition form for type family applications: knowing that $F(\overline{\tau})$ is equal to $F(\overline{\sigma})$ tells us nothing whatsoever about the relationship between $\overline{\tau}$ and $\overline{\sigma}$.

4.3 Role attribution for type constants

In System FC we assume an unwritten global environment of top-level constants: data types, type families, axioms, and so on. For a data type H, for example, this environment will give kind of H, the types of H's data constructors, and the roles of H's parameters. Clearly this global environment must be internally consistent. For example, a data constructor K must return a value of type $D\,\overline{\tau}$ where D is a data type; K's type must be well-kinded, and that kind must be consistent with D's kind.

All of this is standard except for roles. It is essential that the roles of D's parameters, *roles*(D), are consistent with D's definition. For example, it would be utterly wrong for the global environment to claim that *roles*(Maybe) $= P$, because then we could prove that Maybe Int $\sim_{\mathsf{R}}$ Maybe Bool using CO_TYCONAPP.

We use the judgement $\overline{\rho} \models H$, to mean "$\overline{\rho}$ are suitable roles for the parameters of H", and in our proof of type safety, we assume that *roles*$(H) \models H$ for all H. The rules for this judgement and two auxiliary judgements appear in Figure 4.

Start with ROLES_NEWTYPE. Recall that a newtype declaration for N gives rise to an axiom $C : [\overline{\alpha{:}\kappa}].N\,\overline{\alpha} \sim_{\mathsf{R}} \sigma$. The rule says that roles $\overline{\rho}$ are acceptable for N if each parameter α_i is used in σ in a way consistent with ρ_i, expressed using the auxiliary judgement $\overline{\alpha{:}\rho} \vdash \sigma : \mathsf{R}$.

The key auxiliary judgement $\Omega \vdash \tau : \rho$ checks that the type variables in τ are used in a way consistent with their roles specified in Ω, when considered at role ρ. More precisely, if $\alpha{:}\rho' \in \Omega$ and if $\sigma_1 \sim_{\rho'} \sigma_2$ then $\tau[\sigma_1/\alpha] \sim_{\rho} \tau[\sigma_2/\alpha]$. Unlike in many typing judgements, the role ρ (as well as Ω) is an *input* to this judgement, not an output. With this in mind, the rules for the auxiliary judgement are straightforward. For example, RTY_TYFAM says that the argument types of a type family application are used at nominal role. The variable rule, RTY_VAR, allows a variable to be assigned a more restrictive role (via the sub-role judgement) than required, which is needed both for multiple occurrences of the same variable, and to account for role signatures. Note that rules RTY_TYCONAPP and RTY_APP overlap – this judgement is not syntax-directed.

[4] We note in passing that the forms **left** and **right** are present merely to increase expressivity. They are not needed anywhere in the metatheory to prove type soundness. Though originally part of FC, they were omitted in previous versions [14] and even in the implementation. Haskell users then found that some desirable program were no longer type-checking. Thus, these forms were re-introduced.

$\boxed{\overline{\rho} \models H}$ "$\overline{\rho}$ are appropriate roles for H."

$$\frac{\begin{array}{l}\forall\,\overline{\alpha},\overline{\beta},\overline{\sigma} \text{ s.t. } K : \forall\,\overline{\alpha{:}\kappa}.\forall\,\overline{\beta{:}\kappa'}.\overline{\phi} \Rightarrow \overline{\sigma} \to D\,\overline{\alpha} :\\ \quad \forall\,\tau \text{ s.t. } \tau \in \overline{\sigma} \lor \tau \in \overline{\phi} :\\ \quad\quad \overline{\alpha{:}\rho}, \overline{\beta{:}\mathsf{N}} \vdash \tau : \mathsf{R}\end{array}}{\overline{\rho} \models D}\ \textsc{Roles\_Data}$$

$$\frac{C : [\overline{\alpha{:}\kappa}].N\,\overline{\alpha} \sim_{\mathsf{R}} \sigma \qquad \overline{\alpha{:}\rho} \vdash \sigma : \mathsf{R}}{\overline{\rho} \models N}\ \textsc{Roles\_Newtype}$$

$$\frac{}{\mathsf{R},\mathsf{R} \models (\to)} \qquad \frac{}{\mathsf{R},\mathsf{R} \models (\Rightarrow)} \qquad \frac{}{\rho,\rho \models (\sim_{\rho})}$$

$\boxed{\Omega \vdash \tau : \rho}$ "Assuming Ω, τ can be used at role ρ."

$$\frac{\alpha{:}\rho' \in \Omega \qquad \rho' \leq \rho}{\Omega \vdash \alpha : \rho}\ \textsc{RTy\_Var}$$

$$\frac{\begin{array}{l}\overline{\rho} \text{ is a prefix of } \mathit{roles}(H)\\ \overline{\Omega \vdash \tau : \rho}\end{array}}{\Omega \vdash H\,\overline{\tau} : \mathsf{R}}\ \textsc{RTy\_TyConApp}$$

$$\frac{}{\Omega \vdash H : \mathsf{N}}\ \textsc{RTy\_TyCon}$$

$$\frac{\Omega \vdash \tau : \rho \qquad \Omega \vdash \sigma : \mathsf{N}}{\Omega \vdash \tau\,\sigma : \rho}\ \textsc{RTy\_App}$$

$$\frac{\Omega, \alpha{:}\mathsf{N} \vdash \tau : \rho}{\Omega \vdash \forall\,\alpha{:}\kappa.\tau : \rho}\ \textsc{RTy\_ForAll}$$

$$\frac{\overline{\Omega \vdash \tau : \mathsf{N}}}{\Omega \vdash F(\overline{\tau}) : \rho}\ \textsc{RTy\_TyFam}$$

$$\frac{}{\Omega \vdash \tau : \mathsf{P}}\ \textsc{RTy\_Phantom}$$

$\boxed{\rho_1 \leq \rho_2}$ "ρ_1 is a sub-role of ρ_2."

$$\frac{}{\mathsf{N} \leq \rho} \qquad \frac{}{\rho \leq \mathsf{P}} \qquad \frac{}{\rho \leq \rho}$$

Figure 4. Rules asserting a correct assignment of roles to datatypes

Returning to our original judgement $\overline{\rho} \models H$, ROLES_DATA deals with algebraic data types D, by checking roles in each of its data constructors K. The type of a constructor is parameterised by universal type variables $\overline{\alpha}$, existential type variables $\overline{\beta}$, coercions (with types $\overline{\phi}$), and term-level arguments (with types $\overline{\sigma}$). For each constructor, we must examine each proposition ϕ and each term-level argument type σ, checking to make sure that each is used at a representational role. Why check for a representational role specifically? Because *roles* is used in CO_TYCONAPP, which produces a representational coercion. In other words, we must make sure that each term-level argument appears at a representational role within the type of each constructor K for CO_TYCONAPP to be sound.

Finally $(\to)$ and $(\Rightarrow)$ have representational roles: functions care about representational equality but never branch on the nominal identity of a type. (For example, functions always treat HTML and String identically.) We also see that the roles of the arguments to an equality proposition match the role of the proposition. This fact comes from congruence of the respective equality relations.

4.4 Metatheory

The preceding discussion gave several non-obvious examples where admitting *too many* coercions would lead to unsoundness. However, we must have *enough* coercions to allow us to make progress when evaluating a program. (We do not have space to elaborate, but a key example is the use of **nth** in rule S_KPUSH, presented in Appendix C.3.) Happily, we can be confident that we have enough coercions, but not too many, because we prove the usual progress and preservation theorems for System FC. The structure of the proofs follows broadly that in previous work, such as Weirich et al. [14] or Yorgey et al. [15].

A key step in the proof of progress is to prove *consistency*; that is, that no coercion can exist between, say, Int and Bool. This is done by defining a non-deterministic, role-directed rewrite relation on types and showing that the rewrite system is confluent and preserves type constants (other than newtypes) appearing in the heads of types. We then prove that, if a coercion exists between two types τ_1 and τ_2, these two types both rewrite to a type σ. We conclude then that τ_1 and τ_2, if headed by a non-newtype type constant, must be headed by the same such constant.

Alas, the rewrite relation is *not* confluent! The non-linear patterns allowed in type families (that is, with a repeated variable on the left-hand side), combined with non-termination, break the confluence property (previous work gives full details [4]). However, losing confluence does not necessarily threaten consistency – it just threatens the particular proof technique we use. However, a more powerful proof appears to be an open problem in the term rewriting community.[5] For the purposes of our proof we dodge this difficulty by restricting type families to have only linear patterns, thus leading to confluence; consistency of the full system remains an open problem.

The full proof of type safety appears in the appendix; it exhibits no new proof techniques.

5. Role inference

In System FC we assume that, for every type constant H, the global enviroment specifies $roles(H)$, the roles of H's parameters. But where do these roles come from?

- Built-in type constructors like $(\to)$ have built-in roles (Figure 4).
- Type families (Section 2.2.3) and type classes (Section 3.4) have nominal roles for all parameters.
- For a **data** type or **newtype** T GHC *infers* the roles for T's type parameters, informed by any role annotations (Section 3.3).

The role inference algorithm is quite straightforward. At a high level, it simply starts with the role information of the built-in constants $(\to)$, $(\Rightarrow)$, and $(\sim_\rho)$, and propagates the roles until it finds a fixpoint. In the description of the algorithm, we assume a mutable environment; $roles(H)$ pulls a list of roles from this environment. Only after the algorithm is complete will $roles(H) \models H$ hold.

1. Populate $roles(T)$ (for all T) with user-supplied annotations; omitted role annotations default to phantom.
2. For every datatype D, every constructor for that datatype K, and every coercion type and term-level argument type σ to that constructor: run $\mathsf{walk}(D, \sigma)$.
3. For every newtype N with representation type σ, run $\mathsf{walk}(N, \sigma)$.
4. If the role of any parameter to any type constant changed in the previous steps, go to step 2.
5. For every T, check $roles(T)$ against a user-supplied annotation, if any. If these disagree, reject the program. Otherwise, $roles(T) \models T$ holds.

The procedure $\mathsf{walk}(T, \sigma)$ is defined as follows, matching from top to bottom:

```
$\mathsf{walk}(T, \alpha)$          := mark the $\alpha$ parameter to $T$ as R.
$\mathsf{walk}(T, H\,\overline{\tau})$       := let $\overline{\rho} = roles(H)$;
                      for every $i$, $0 < i \leq \mathsf{length}\ (\overline{\tau})$:
                         if $\rho_i = \mathsf{N}$, then
                            mark all variables free in $\tau_i$ as N;
                         else if $\rho_i = \mathsf{R}$, then $\mathsf{walk}(T, \tau_i)$.
$\mathsf{walk}(T, \tau_1\,\tau_2)$      := $\mathsf{walk}(T, \tau_1)$;
                      mark all variables free in $\tau_2$ as N.
$\mathsf{walk}(T, F(\overline{\tau}))$     := mark all variables free in the $\overline{\tau}$ as N.
$\mathsf{walk}(T, \forall\,\beta{:}\kappa.\tau)$ := $\mathsf{walk}(T, \tau)$.
```

When marking, we must follow these two rules:

1. If a variable to be marked does not appear as a type-level argument to the datatype T in question, ignore it.
2. Never allow a variable previously marked N to be marked R. If such a mark is requested, ignore it.

The first rule above deals with existential and local ($\forall$-bound) type variables, and the second one deals with the case where a variable is used both in a nominal and in a representational context. In this case, we wish the variable to be marked N, not P.

Theorem. *The role inference algorithm always terminates.*

Theorem (Role inference is sound)**.** *After running the role inference algorithm, $roles(H) \models H$ will hold for all H.*

Theorem (Role inference is optimal)**.** *After running the role inference algorithm, any loosening of roles (a change from ρ to ρ', where $\rho \leq \rho'$ and $\rho \neq \rho'$) would violate $roles(H) \models H$.*

Proofs of these theorems appear in Appendix I.

6. Implementing Coercible

We have described the source-language view of Coercible (Sections 2, 3), and System FC, the intermediate language into which the source language is elaborated (Section 4). In this section we link the two by describing how the source-language use of Coercible is translated into Core.

6.1 Coercible and coerce

When the compiler transforms Haskell to Core, type classes become ordinary types and typeclass constraints turn into ordinary value arguments [13]. In particular, type classes typically become simple product types with one field per method. The built-in type class Coercible is a bit different: it wraps the primitive witness of representational equality $\sim_\mathsf{R}$ in a datatype:

data Coercible a b = MkCoercible (a $\sim_\mathsf{R}$ b)

The definition of coerce, which is possible to give only in

[5] Specifically, we believe that a positive answer to open problem #79 of the Rewriting Techniques and Applications (RTA) conference would lead to a proof of consistency; see http://www.win.tue.nl/rtaloop/problems/79.html.

Core, pattern-matches on MkCoercible to get hold of the equality witness, and then uses Core's primitive cast operation:

```
coerce :: forall α β. Coercible α β → α → β
coerce = \α β (c :: Coercible α β) (x :: α). case c of
  MkCoercible eq → x ▷ eq
```

Since type applications are explicit in Core, coerce now takes four arguments: the types to cast from and to, the coercion witness, and finally the value to cast.

The data type Coercible serves to *box* the primitive, unboxed type $\sim_R$, just as Int serves to box the primitive, unboxed type Int#:

```
data Int = I# Int#
```

All boxed types are represented uniformly by a heap pointer. In GHC all constraints (such as Eq a or Coercible a b) are boxed, so that they can be treated uniformly, and even polymorphically [15]. In contrast, an unboxed type is represented by a non-pointer bit field, such as a 32 or 64-bit int in the case of Int# [9].

A witness of (unboxed) type $\sim_R$ carries no information: we never actually inspect an equality proof at run-time. So the type $\sim_R$ can be represented by a *zero-width* bit-field – that is, by nothing at all. This implementation trick, of boxing a zero-bit witness, is exactly analogous to the wrapping of boxed nominal equalities used to implement deferred type errors [12].

Since Coercible is a regular data type, you might worry about bogus programs like this, which uses recursion to contruct an unsound witness co whose value is bottom:

```
looksUnsound :: forall α β. α → β
looksUnsound = \α β x →
  let co :: Coercible α β = co in
  coerce α β co x
```

However, since coerce evaluates the Coercible argument (see the definition of coerce above), looksUnsound will simply diverge. Again, this follows the behaviour of deferred type errors [12].

In uses of coerce, the Coercible argument will be constructed from the instances which, as described below (Section 6.3), are guaranteed to be acyclic. The usual simplification machinery of GHC then ensures that these are inlined, causing the **case** to cancel with the MkCoercible constructor, leaving only the cast x ▷ eq, which is operationally free.

6.2 Instance generation and solving

The implementation must also solve Coercible constraints using the generated instances (Figure 1). The code for these instances, however, is not created when a datatype is defined. Instead, they are built on-demand by the type checker when a Coercible instance is to be solved. This approach has various benefits:

- It is simpler to control the use of instances that should not be used due lack of an imported constructor (see Section 3.1); and to be appropriately relaxed about incoherence (Section 2.1).
- It is straightforward to deal with newtypes with a *higher-rank* representation type, as we elaborate next.
- There is no need to compile, export, and link the instances, avoiding an increase in interface file size and compilation time.

Concerning the second point, consider this declaration, whose constructor uses a higher-rank type:

```
newtype Sel = MkSel (forall a. [a] → a)
```

Its newtype unwrapping instances take a form that is usually illegal, even with all GHC extensions enabled:

```
instance Coercible (forall a. [a] → a) b ⇒ Coercible Sel b
instance Coercible a (forall a. [a] → a) ⇒ Coercible a Sel
```

Moreover, **forall** is also a type constructor, so we need its lifting instance, something like:

```
instance (forall a. Coercible s t)
      ⇒ Coercible (forall a. s) (forall a. t)
```

This instance is also illegal, even with GHC's many extensions. By giving Coercible special treatment we can deal correctly with its higher-rank instances, without to specify and implement the general case.

6.3 Preventing circular reasoning and diverging instances

For most type classes, like Show, it is perfectly fine (and useful) to use a not-yet solved type class constraint to solve another, even though this can lead to cycles [6]. Consider the following code and execution:

```
newtype Fix a = MkFix (a (Fix a))
deriving instance Show (a (Fix a)) ⇒ Show (Fix a)

λ> show (MkFix (Just (MkFix (Just (MkFix Nothing)))))
"MkFix (Just (MkFix (Just (MkFix Nothing))))"
```

There are two Show instances at work: one for Show (Maybe a), which uses the instance of Show a; and one for Show (Fix a), which uses the the instance Show (a (Fix a)). Plugging them together to solve Show (Fix Maybe), we see that this instance calls, by way of Show (Maybe (Fix Maybe)), itself. Nevertheless, the result is perfectly well-behaved and indeed terminates.

But with Coercible, such circular reasoning would be problematic; we could then seemingly write the bogus function looksUnsoundH:

```
newtype Id a = MkId a
c1 :: a → Fix Id
c1 = coerce
c2 :: Fix Id → b
c2 = coerce
looksUnsoundH :: a → b
looksUnsoundH = c2 ∘ c1
```

With the usual constraint solving, this code would type check: to solve the constraint Coercible a (Fix Id), we need to solve Coercible a (Id (Fix Id)), which requires Coercible a (Fix Id). This is a constraint we already looked at, so the constraint solver would normally consider all required constraints solved and accept the program.

Fortunately, there is no soundness problem here. Circular constraint-solving leads to a recursive definition of the Coercible constraints, exactly like the (Core) looksUnsound in Section 6.1, and looksUnsoundH will diverge just like looksUnsound. Nevertheless, unlike normal type classes, a recursive definition of Coercible is *never* useful, so it is more helpful to reject it statically. GHC therefore uses a simple depth-bounding technique to spot and reject recursion of Coercible costraints.

6.4 Coercible and rewrite rules

What if a client of module Html writes this?

```
....( map unMk hs)...
```

She cannot use coerce because HTML is an abstract type, so the type system would (rightly) reject an attempt to use coerce (Section 3.1). However, since HTML is a newtype, one might hope that GHC's optimiser would transform (map unMk) to coerce. The optimiser must respect type soundness, but (by design) it does not respect abstraction boundaries: dissolving abstractions is one key to high performance.

The correctness of transforming (map unMk) to coerce depends on a theorem about map, which a compiler can hardly be expected to identify and prove all by itself. Fortunately GHC already comes with a mechanism that allows a library author to specify *rewrite rules* for their code [10]. The author takes the proof obligation that the rewrite is semantics-preserving, while GHC simply applies the rewrite whenever possible. In this case the programmer could write

```
{-# RULES "map/co" map coerce = coerce #-}
```

In our example, the programmer wrote (map unMk). Function unMk is, in module Html, implemented by coerce; via cross-module inlining (more dissolution of abstraction boundaries) unMk will be inlined, transforming the call to (map coerce), and that in turn fires the rewrite rule. Indeed even a nested call like map (map unMk) will also be transformed to a single call of coerce by this same process applied twice.

The bottom line is this: the author of a map-like function someMap can accompany someMap with a RULE, and thereby optimise calls of someMap that do nothing into a simple call to coerce.

Could we dispense with a user-visible coerce function altogether, instead using map-like functions and RULEs as above? No: doing so would replace the zero-cost guarantee with best-effort optimisation; it would burden the author of every map-like function with the obligation to write a suitable RULE; it would be much less convenient to use in deeply-nested cases; and there might simply *be* no suitable map-like function available.

7. Generalized Newtype Deriving done right

As mentioned before, **newtype** is a great tool to make programs more likely to be correct, by having the type checker enforce certain invariants or abstractions. But newtypes can also lead to tedious boilerplate. Continuing the example from the introduction, assume the programmer needs an instance of the typeclass Monoid for her type HTML. The underlying type String already comes with a suitable instance for Monoid. Nevertheless, she has to write quite a bit of code to convert that instance into one for HTML:

```
instance Monoid HTML where
  mempty = Mk mempty
  mappend (Mk a) (Mk b) = Mk (mappend a b)
```

Note that this definition is not only verbose, but also non-trivial, as invocations of Mk and unMk have to be put in the right places, possibly via some higher order functions like map – all just to say "just use the underlying instance"!

This task is greatly simplified with Coercible: Instead of wrapping and unwrapping arguments and results, she can directly coerce the method of the base type's instance itself:

```
instance Monoid HTML where
  mempty = coerce (mempty :: String)
  mappend = coerce (mappend :: String → String → String)
```

```
newtype Id1 a = MkId1 a
newtype Id2 a = MkId2 (Id1 a)
  deriving (UnsafeCast b)

type family Discern a b
type instance Discern (Id1 a) b = a
type instance Discern (Id2 a) b = b

class UnsafeCast to from where
  unsafe :: from → Discern from to

instance UnsafeCast b (Id1 a) where
  unsafe (MkId1 x) = x

unsafeCoerce :: a → b
unsafeCoerce x = unsafe (MkId2 (MkId1 x))
```

Figure 5. The above implementation of unsafeCoerce compiles (with appropriate flags) in GHC 7.6.3 but does not in GHC 7.8.1.

The code is pure boilerplate: apply coerce to the method, instantiated at the base type by a type signature. And because it is boilerplate, the compiler can do it for her; all she has to do is to declare which instances of the base type should be lifted to the new type by listing them in the **deriving** clause:

```
newtype HTML = Mk String deriving (Monoid)
```

This is not a new feature: GHC has provided this *Generalized Newtype Deriving* (GND) for many years. But, the implementation was "magic" – GND would produce code that a user could not write herself. Now, the feature can be explained easily and fully via coerce.

Furthermore, GND was previously unsound [14]! When combined with other extensions of GHC, such as type families [1, 2] or GADTs [3], GND could be exploited to completely break the type system: Figure 5 shows how this notorious bug can allow any type to be coerced to any other. The clause "**deriving** (UnsafeCast b)" is the bogus use of GND, and now will generate the instance

```
instance UnsafeCast b c ⇒ UnsafeCast b (Id2 c) where
  unsafe = coerce (unsafe :: c → Discern c b)
```

which will rightly be rejected because Discern's first parameter has a nominal role.

Similarly, it was possible to use GND to break invariants of abstract data types. As discussed in Section 3.1, this is now also prevented by the use of coerce.

8. Related work

Prior work (Weirich et al. [14], which we shall call WVPZ) discusses the relationship between roles in FC and languages with generativity and abstraction, type-indexed constructs, and universes in dependent type theory. We do not repeat that discussion here. Instead we use this section to clarify the relationship between this paper and WVPZ, as well as make connections to other systems.

8.1 Prior version of roles

The idea of *roles* was initially developed in WVPZ as a solution to the Generalized Newtype Deriving problem. That work introduces the equality relations $\sim_R$ and $\sim_N$ (called "type equality" and "code equality" resp. in WVPZ). However, the system presented in WVPZ was quite invasive: it required annotating every sub-tree of every kind with a role. The pervasiveness of the change is one of the reasons that the system was never implemented.

In this paper, we present a substantially simplified version of the roles system of WVPZ, requiring role information only on the parameters to datatypes. The key simplification is to "assume the worst" about higher-kinded parameters, by assuming that their arguments are all nominal.

In exchange we give up some expressiveness; specifically, we give up the ability to abstract over type constructors with non-nominal argument roles. This loss bites occasionally. One such example is this code from Edward Kmett's linear library which defines the type

```
newtype Point f a = P (f a)
```

and uses GND to coerce a class method of type f (f a) to Point f (Point f a). This coercion is potentially unsound when f's argument has nominal role, so the type system of this paper rejects it. However, the system in WVPZ could instead limit the instantiation of f to type constructors with representational parameters, allowing this use of GND. The workaround is easy, though: the library now uses a straightforward hand-written instance.We have not yet found an example where the loss of expressiveness is genuinely painful.

Surprisingly, our treatment of higher-kinded parameters as themselves taking nominal arguments actually *increases* expresssiveness compared to WVPZ in some places. In WVPZ a role is part of a type's kind, so a type expecting a higher-kinded argument (such as Monad) would also have to specify the roles expected by its argument. Therefore if Monad is applicable to Maybe, it would not also be applicable to a type T whose parameter has a nominal role. In the current work, however, there is no problem because Maybe and T have the same kind.

There are, of course, other minor differences between this system and WVPZ in keeping with the evolution of System FC. The main significant change, unrelated to roles, is the reintroduction of **left** and **right** coercions; see Section 4.2.6.

Finally, because this system has been implemented in GHC, this paper discusses more details related to compilation from source Haskell. In particular, the role inference algorithm of Section 5 is a new contribution of this work.

8.2 OCaml and variance annotations

The interactions between sub-typing, type abstraction, and various type system extensions such as GADTs and parameter constraints also appear in the OCaml language. In that context, *variance annotations* act like roles; they ensure that subtype coercions between compatible types are safe. For example, the type α `list` of immutable lists is covariant in the parameter α: if $\sigma \leq \tau$ then σ `list` $\leq \tau$ `list`. Variances form a lattice, with *invariant*, the most restrictive, at the bottom; *covariant* and *contravariant* incomparable; and *bivariant* at the top, allowing sub-typing in both directions. It is tempting to identify invariant with nominal and bivariant with phantom, but the exact connection is unclear. Scherer and Rémy [11] show that GADT parameters are not always invariant.

Exploration of the interactions between type abstraction, GADTs, and other features have recently revealed a soundness issue in OCaml[6] that has been confirmed to date back several years. Garrigue [5] discusses these issues. His proposed solution is to "assume that nothing is known about abstract types when they are used in parameter constraints and GADT return types" – akin to assigning nominal roles. However, this solution is too conservative, and in practice the OCaml 4.01 compiler relies on no fewer than *six* flags to describe the variance of type parameters. However, lacking anything equivalent to Core and its tractable metatheory, the OCaml developers cannot demonstrate the soundness of their solution in the way that we have done here.

What is clear, however, is that generative type abstraction interacts in interesting and non-trivial ways with type equality and sub-typing. Roles and type-safe coercion solve an immediate practical problem in Haskell, but we believe that the ideas have broader applicability in advanced type systems.

Acknowledgements Thanks to Antal Spector-Zabusky for his contributions to this version of System FC; to Edward Kmett for discussions on the design of Coercible and of roles; and to Dimitrios Vytiniotis for feedback on an earlier draft.

References

[1] M. M. T. Chakravarty, G. Keller, and S. L. Peyton Jones. Associated type synonyms. In *ICFP*, pages 241–253. ACM, 2005.

[2] M. M. T. Chakravarty, G. Keller, S. L. Peyton Jones, and S. Marlow. Associated types with class. In *POPL*, pages 1–13. ACM, 2005.

[3] J. Cheney and R. Hinze. First-class phantom types. Technical report, Cornell University, 2003.

[4] R. A. Eisenberg, D. Vytiniotis, S. Peyton Jones, and S. Weirich. Closed type families with overlapping equations. In *POPL*, pages 671–683. ACM, 2014.

[5] J. Garrigue. On variance, injectivity, and abstraction. OCaml Meeting, Boston., Sept. 2013.

[6] R. Lämmel and S. Peyton Jones. Scrap your boilerplate with class: Extensible generic functions. In *ICFP*, 2005.

[7] S. Marlow (editor). Haskell 2010 language report, 2010.

[8] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. 1997.

[9] S. Peyton Jones and J. Launchbury. Unboxed values as first class citizens. In R. Hughes, editor, *ACM Conference on Functional Programming and Computer Architecture (FPCA)*, volume 523 of *LNCS*, pages 636–666, Boston, 1991.

[10] S. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Haskell Workshop*, pages 203–233, 2001.

[11] G. Scherer and D. Rémy. GADTs meet subtyping. In *ESOP 2013 - 22nd European Symposium on Programming, Rome, Italy*, pages 554–573, 2013.

[12] D. Vytiniotis, S. Peyton Jones, and J. P. Magalhães. Equality proofs and deferred type errors: A compiler pearl. In *ICFP*, pages 341–352. ACM, 2012.

[13] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *POPL*, pages 60–76. ACM Press, 1989.

[14] S. Weirich, D. Vytiniotis, S. L. Peyton Jones, and S. Zdancewic. Generative type abstraction and type-level computation. In *POPL*, pages 227–240. ACM, 2011.

[15] B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving Haskell a promotion. In *TLDI*, pages 53–66, New York, NY, USA, 2012. ACM.

A. Further discussion

There are many aspects of roles and Coercible that may be of interest, especially to those considering adopting a similar mechanism in another programming language. We collect these observations here.

[6] http://caml.inria.fr/mantis/view.php?id=5985

A.1 Conservativity of roles

Roles are coarse-grained The system we describe has exactly three roles. However, by having only three roles, we have created a rather coarse-grained classification system. For example, consider the following definitions:

```
data Bar a = MkBar (F a)
type instance F Int  = Char
type instance F Bool = Char
type instance F [a]  = Double
```

Is it safe to coerce a Bar Int to a Bar Bool. Unraveling definitions, we see that the answer is "yes". What about coercing Bar Int to Bar [Double]? Clearly, "no". GHC assigns a nominal role to the parameter of Bar, but this choice of role eliminates the possibility of the Bar Int to Bar Bool coercion. If, instead, we had a *lattice* of roles, keyed by type families whose equality must be respected, we might be able to allow more safe coercions. We could similarly imagine a lattice keyed by classes whose instance definitions are to be respected; with such a lattice, we could allow the coercion of Map Int v to Map Age v precisely when Int's and Age's Ord instance correspond.

Equality does not propagate roles What role should be assigned to a parameter with an equality constraint involving a phantom? According to the rules in our formalism, such a parameter would get a nominal role. This case has come up in practice. Consider the following type from Edward Kmett's lens library:

```
data Magma i t b a where
  MagmaAp   :: Magma i (x → y) b a
    → Magma i x b a → Magma i y b a
  MagmaPure :: x → Magma i x b a
  MagmaFmap :: (x → y)
    → Magma i x b a → Magma i y b a
  Magma :: i → a → Magma i b b a
```

Role inference gives these roles, respectively: representational, nominal, nominal, representational. Close inspection of the type definition shows us that the third parameter, b, is almost a phantom – it is never used outside phantom contexts except in one place: the return type of the Magma constructor. There, we see that the second and third parameters must be equal. Another way to write this last constructor is Magma :: (t ~ b) ⇒ i → a → Magma i t b a. Also, note that the second parameter, t, is used representationally everywhere but in that same spot.

What this all leads to is the fact that Magma i x x a has the same run-time representation as Magma i y y a whenever x has the same representation as y. Yet, the role mechanism is not expressive enough to prove this.

A.2 Type inference

Inferring polymorphic coercions Recall the code

```
concatH :: [HTML] → HTML
concatH = coerce (concat :: [String] → String)
```

from the introduction. It certainly is neat to coerce a whole function this way, but having to give two type annotations is slightly annoying. But just

```
concatH :: [HTML] → HTML
concatH = coerce concat
```

does not work: concat has the polymorphic type [[a]] → [a], and the type inference has no idea how to instantiate a. In this instance, it is quite obvious what the programmer wants, and it may be feasible to make the compiler smart enough to see that.

Explicit type application A different approach to the concatH problem immediately above is to be able to provide an explicit type argument to the concat function. With hypothetical syntax, we would want to write

```
concatH = coerce (concat @String)
```

The use of @ above denotes an explicitly-passed type parameter. With this new syntax, we are freed from the burden of instantiating all of concat's type to specify a single type parameter. This desire for explicit type application came up numerous times in our experimentation with coerce, particularly in the implementation of GND.

A.3 Parametricity

There seems to be some relationship between roles, parametricity, and categorical structures. For example, consider the class

```
class Functor f where
  fmap :: (a → b) → f a → f b
```

with the functor law, stating that fmap id should be identically id. We conjecture that it is impossible to write a lawful instance of Functor over a type whose one parameter would be inferred to have a nominal role. This observation stems from the fact that fmap must be parametric in a and b, and thus cannot take any action which observes any nominal equality dealing with those type parameters. We have not proved this conjecture, however.

A.4 Eta-reduction

Consider the definition

```
newtype MyMaybe a = MkMM (Maybe a)
```

What is the axiom created by this declaration? Naively, we could expect an axiom $C : [\alpha{:}\star].\mathsf{MyMaybe}\,\alpha \sim_{\mathsf{R}} \mathsf{Maybe}\,\alpha$. Now, suppose we are using the monad transformer ListT. As usual for a monad transformer, ListT has kind $(* \to *) \to * \to *$. That is, it takes a monad as its first argument and transforms it into a monad, now enhanced with nondeterminism. Let's say we have a value m of type ListT Maybe Int. Is m coercible into ListT MyMaybe Int? We would like it to be – these would have the same representation. But, with the axiom C above, such a coercion would be impossible. There is no way to extract MyMaybe $\sim_{\mathsf{R}}$ Maybe from C.

Instead, we *η-reduce* axioms. Accordingly, the declaration above would yield C_2 : MyMaybe $\sim_{\mathsf{R}}$ Maybe. This axiom can easily be used to show that ListT Maybe Int $\sim_{\mathsf{R}}$ ListT MyMaybe Int.

This η-reduction is why the Co_App rule works over representational coercions as well as nominal ones. An earlier version of the rule consumed and produced only nominal coercions; with that rule, it was impossible to derive MyMaybe Int $\sim_{\mathsf{R}}$ Maybe Int from C_2.

A.5 Extending role to families

A.5.1 Roles on type and data families

In GHC 7.8, all type and data family parameters have nominal roles. This stands to good reason, as a type or data family can pattern-match on its parameters. For example:

```
type family TF a
type instance TF Int = Double
type instance TF Age = Char
```

Clearly, TF Int is not representationally equal to TF Age.

Yet, it would be sensible to extend the idea of roles to type and data families. A family with a non-nominal parameter would need extra checks on its instance declarations, to make sure that they are compatible with the choice of roles. For example:

```
type role If nominal representational representational
type family If True (a :: Bool) b c
type instance If True  b c = b
type instance If False b c = c
```

The above definition, though not accepted in GHC 7.8, is perfectly type safe. Note that a representational parameter must not be matched on and must not be used in a nominal context on the right-hand side. The only barrier to implementing this is the extra complexity for the GHC maintainers and the extra complexity in the language. If a compelling use case for this comes up, we will likely add the feature.

A.5.2 Roles on data family instances

Roles on data families follow the same arguments as above. However, we can identify a separate issue involving roles on data family instances, which are, of course, datatypes. For example:

```
data family DF a
data instance DF (b, Int) = MkDF (Maybe b)
```

Data family instances are internally desugared into something resembling a type family instance and a fresh datatype declaration, somewhat like this:[7]

```
type family DF a
type instance DF (b, Int) = DFPairIntInstance b
data DFPairIntInstance b = MkDF (Maybe b)
```

Here, it is apparent that b can be assigned a representational role, even while we require a nominal role for a.

Role inference for data family instances is not currently implemented. Instead, all type variables in a data family instance are assigned nominal roles. Why? Essentially because there is no way of writing a role annotation for data family instances. Without the ability to write role annotations, library writers would be unable to enforce abstraction on these, and so it is safer just to default these (somewhat uncommon) parameters to have nominal roles.

If you wish to request roles on either type/data families or on data family instances, you can comment on GHC bug #8177 here: https://ghc.haskell.org/trac/ghc/ticket/8177

B. Design decisions

Here, we collect some of the design decisions that we made while formulating roles concretely into GHC.

B.1 Concrete syntax

It turns out that designing the concrete syntax of role annotations was non-trivial. We identified several desired traits of the syntax:

1. Role annotations must be optional. Otherwise, all existing code would be broken.
2. Role annotations should be succinct.
3. Role annotations will be a relatively obscure feature, and therefore should be searchable should a user come across one.
4. Code with role annotations should compile with older versions of GHC. This eases migration to GHC 7.8.
5. Role annotations should not be specified in a pragma; pragmas are meant to be reserved for implementation details (e.g., optimising), and roles are a type system feature.
6. Role annotations should be easy to refactor as a datatype evolves.
7. Code is read much more often than it is written; favour readability over concision.

We will use Map as a running example to demonstrate the various alternatives we considered for the syntax. Note that all options satisfy desire (1).

1. Standalone role annotations:

   ```
   type role Map nominal representational
   data Map k v = ...
   ```

   This is, of course, our final answer to the concrete syntax question. It satisfies (3), (5), and (7), at the cost of some others. In particular, this choice is not backward-compatible. A role annotation fails to parse in earlier versions of GHC. However, all is not lost, because GHC supports C preprocessor directives, and library authors can selectively include role annotations using preprocessor directives. The fact that the annotations are standalone means they can be grouped under one set of directives instead of sprinkled throughout the source file. Note that this syntax is very easy to search for, and the written-out nature of the roles makes them readable, if not so concise.

2. Inline, abbreviated role annotations:

   ```
   data Map k@N v@R = ...
   ```

   This version satisfies (2), (5), (6). It is not backwards-compatible, and the fact that the role annotations are inline means that each role-annotated definition would need its own preprocessor directives. Furthermore, now the type definition itself must be repeated in various places, so refactoring becomes more burdensome. This version is also not readable by non-experts and is nearly impossible to search for if a user is confused.

3. Inline pragmas:

   ```
   data Map {-# ROLE nominal #-} k
            {-# ROLE representational #-} v = ...
   ```

   This version satisfies (3), (4), (6), (7), but it clearly uses a pragma. We felt that, in several years, we would regret this decision. Backwards-compatibility would no longer be an issue, and we would be stuck with a pragma syntax for a core language feature.

4. Custom class constraints:

   ```
   data Map k v = ...
   instance NextParamNominal Map
   instance NextParamRepresentational (Map k)
   ```

[7] Type inference is somewhat different between type families, which are not necessarily injective, and data families, which are. Along similar lines, data families can appear unsaturated, while type families cannot. This desugaring does not change these facts.

This version satisfies (3), (4), (5), (7). But, there is a mismatch between the class instance mechanism and role annotations. In particular, declarations such as these would make no sense if orphaned (that is, if put in a separate module from the parent datatype declaration). Furthermore, what would it mean if one of these classes were used as a constraint on a function, like so:

```
fun :: (NextParamRepresentational f, Functor f) ⇒ ...
```

If we allowed the syntax, we would also have to implement the full complication of the roles system developed by Weirich et al. [14]. We didn't think the full system was necessary, so we would have to restrict the NextParamXXX classes to only instance declarations. In the end, there seemed to be too much of a mismatch to make this viable.

5. Custom Coercible instances:

```
data Map k v = ...
instance Coercible v1 v2 ⇒
  Coercible (Map k v1) (Map k v2)
```

This version satisfies (3), (4), (5), and (7). The roles are implicitly discovered by the usages of the variables in the declaration. This form has the advantage that it makes the Coercible instances obvious. However, it has many of the drawbacks of the previous version, with NextParamXXX. One important advantage this has over the previous version is that it doesn't lead to the full implementation of the system in Weirich et al. [14] – implication constraints are not allowed, so there is no way of writing the declaration above as a constraint. However, what would it mean if this annotation were omitted? It would seem strange that a Coercible instance would be automatically supplied if none were written. Also, what would it mean if the declaration used variables in a way inconsistent with roles? In the end, we felt that this version was in a bit of an uncanny valley: it's rather close to a "normal" instance declaration, but with some unexpected features and consequences. We thought it would lead to more confusion than other versions.

Do we staunchly defend our choice of the **type role** ... syntax? No, but we were unable to come up with a better one that might stand the test of time.

B.2 Default roles

What should the default roles of a datatype be? As described in Section 5, datatype roles default to phantom. However, this choice has a real consequence: if a library author wishes to make a fully-abstract datatype, a role annotation is often necessary to preserve the abstraction. Otherwise, depending on the actual datatype definition, the roles of the datatype parameters may be more permissive than nominal, and users can use coercions perhaps to subvert invariants. Indeed, this is the case with the oft-used Map example.

Because of the presence of Generalized Newtype Deriving in earlier versions of GHC, this lack of abstraction is not new – any specific use of coerce (that does not involve phantom roles) can be converted into a use of the legacy GND feature. However, with roles, this problem can now be contained via role annotations.

This all begs the question: what to do with legacy code without role annotations? If we default this code to use nominal roles, then it means that coerce (and, transitively, GND) will work only on those datatypes that explicitly support it. This choice favours abstraction over usability. The reason we chose to default to phantom, instead, is that a considerable amount of existing code *does* use GND. Much of that code would cease to compile if roles defaulted to nominal. We felt that the effect of this change would be too great on the user community, so we made the more conservative choice and chose to favour usability over abstraction.

B.3 Safe Haskell

One of the consequences of the unsoundness of earlier versions of GND is that the feature was prohibited from the Safe Haskell subset. However, even after roles were implemented and GND written in terms of coerce, the feature *still* did not meet the Safe Haskell criteria. At issue were both abstraction and coherence:

Abstraction We describe in Section 3.2 that we allow coercions to happen even on datatypes for which the constructors are not available, such as Map. However, this violates Safe Haskell's promise that no abstraction barrier is broken through. To rectify this problem, GHC uses a more stringent check when satisfying a Coercible constraint when compiling in Safe mode: all constructors of all datatypes to be coerced under must be visible. This means, essentially, traversing the entire tree of datatype definitions, making sure all constructors of all datatypes, recursively, are available. With this check in place, we can be sure not to break any abstraction boundaries.

Coherence Haskell classes are compiled into regular datatypes in GHC Core. Accordingly, classes have roles assigned to their parameters. However, we tend to think of, for example, Ord String to be quite independent from Ord HTML. Thus, as discussed in Section 3.4, we default all class roles to be nominal.

However, it's possible that a user wishes to override this default. As an interesting example, it is quite sensible that Coercible's roles should be representational! This fact can be seen in the fact that the roles of the $(\sim_{\mathsf{R}})$ operator are representational. Accordingly, we allow role annotations for classes, even though roles other than nominal can lead quickly to incoherence.

Safe Haskell claims to enforce class coherence. Thus, it is important that Safe Haskell restricts role annotations on classes. This is done by requiring the extension IncoherentInstances (which is disallowed in Safe Haskell) to have a nontrivial role annotation on a class.

C. System FC, in full

Throughout this entire proof of type safety, any omitted proof is by (perhaps mutual) straightforward induction on the relevant derivations.

As usual, all definitions and proofs are only up to α-equivalence. If there is a name clash, assume a variable renaming to a fresh variable.

We assume the regularity of typing judgements throughout the proof. That is, if $\Gamma \vdash \tau : \kappa$, $\Gamma \vdash \gamma : \phi$, or $\Gamma \vdash e : \tau$, we can conclude $\vdash \Gamma$. Accordingly, whenever proving one of the judgements above, we must also prove $\vdash \Gamma$. In practice, tracking these context consistency judgements presents no problems and is elided throughout.

C.1 The remainder of the grammar

$$
\begin{array}{llll}
\Phi & ::= & \overline{[\alpha{:}\kappa]}.\tau \sim_\rho \sigma & \text{axiom types} \\
e & ::= & & \text{expressions} \\
 & | & v & \quad\text{value} \\
 & | & x & \quad\text{variable} \\
 & | & e_1\, e_2 & \quad\text{application} \\
 & | & e\,\tau & \quad\text{type application} \\
 & | & e\,\gamma & \quad\text{coercion application} \\
 & | & \mathbf{case}_\tau\ e\ \mathbf{of}\ \overline{alt} & \quad\text{pattern match} \\
 & | & e \triangleright \gamma & \quad\text{cast} \\
 & | & \mathbf{contra}\,\gamma\,\tau & \quad\text{absurdity} \\
v & ::= & & \text{expression values} \\
 & | & \lambda x{:}\tau.e & \quad\text{value abstraction} \\
 & | & \Lambda\alpha{:}\kappa.e & \quad\text{type abstraction} \\
 & | & \lambda c{:}\phi.e & \quad\text{coercion abstraction} \\
 & | & K\,\overline{\tau}\,\overline{\gamma}\,\overline{e} & \quad\text{applied data constructor} \\
alt & ::= & K\,\overline{\alpha}\,\overline{c}\,\overline{x} \to e & \text{alternative in pattern match} \\
\psi & ::= & & \text{value types} \\
 & | & D & \quad\text{datatype (}\textit{not}\ \textbf{newtype}\text{s!)} \\
 & | & (\to) & \quad\text{arrow} \\
 & | & (\Rightarrow) & \quad\text{prop. arrow} \\
 & | & (\sim^\kappa_\rho) & \quad\text{equality} \\
 & | & \forall\,\alpha{:}\kappa.\tau & \quad\text{polymorphism} \\
 & | & \psi\,\tau & \quad\text{application}
\end{array}
$$

C.2 Typing judgements

Note that the statement, for example, $\alpha\,\#\,\Gamma$ means that the variable α is fresh in the context Γ.

$\boxed{\vdash \Gamma}$ Context validity

$$\frac{}{\vdash \varnothing}\ \textsc{Ctx\_Empty}$$

$$\frac{\alpha\,\#\,\Gamma}{\vdash \Gamma, \alpha{:}\kappa}\ \textsc{Ctx\_TyVar}$$

$$\frac{\Gamma \vdash \tau \sim_\rho \sigma : \star \qquad c\,\#\,\Gamma}{\vdash \Gamma, c{:}\phi}\ \textsc{Ctx\_CoVar}$$

$$\frac{\Gamma \vdash \tau : \star \qquad x\,\#\,\Gamma}{\vdash \Gamma, x{:}\tau}\ \textsc{Ctx\_Var}$$

$\boxed{\Gamma \vdash \tau : \kappa}$ Type kinding

$$\frac{\alpha{:}\kappa \in \Gamma}{\Gamma \vdash \alpha : \kappa}\ \textsc{Ty\_Var}$$

$$\frac{\Gamma \vdash \tau_1 : \kappa_1 \to \kappa_2 \\ \Gamma \vdash \tau_2 : \kappa_1}{\Gamma \vdash \tau_1\,\tau_2 : \kappa_2}\ \textsc{Ty\_App}$$

$$\frac{T : \kappa}{\Gamma \vdash T : \kappa}\ \textsc{Ty\_ADT}$$

$$\frac{}{\Gamma \vdash (\to) : \star \to \star \to \star}\ \textsc{Ty\_Arrow}$$

$$\frac{}{\Gamma \vdash (\Rightarrow) : \star \to \star \to \star}\ \textsc{Ty\_PropArrow}$$

$$\frac{}{\Gamma \vdash (\sim^\kappa_\rho) : \kappa \to \kappa \to \star}\ \textsc{Ty\_Equality}$$

$$\frac{\Gamma, \alpha{:}\kappa \vdash \tau : \star}{\Gamma \vdash \forall\,\alpha{:}\kappa.\tau : \star}\ \textsc{Ty\_ForAll}$$

$$\frac{F : \overline{[\alpha{:}\kappa']}.\kappa \\ \overline{\Gamma \vdash \tau : \kappa'}}{\Gamma \vdash F(\overline{\tau}) : \kappa}\ \textsc{Ty\_TyFun}$$

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

$$\frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau}\ \textsc{Tm\_Var}$$

$$\frac{\Gamma, x{:}\tau \vdash e : \sigma}{\Gamma \vdash \lambda x{:}\tau.e : \tau \to \sigma}\ \textsc{Tm\_Abs}$$

$$\frac{\Gamma \vdash e_1 : \tau \to \sigma \\ \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\,e_2 : \sigma}\ \textsc{Tm\_App}$$

$$\frac{\Gamma, \alpha{:}\kappa \vdash e : \tau}{\Gamma \vdash \Lambda\alpha{:}\kappa.e : \forall\,\alpha{:}\kappa.\tau}\ \textsc{Tm\_TAbs}$$

$$\frac{\Gamma \vdash e : \forall\,\alpha{:}\kappa.\sigma \\ \Gamma \vdash \tau : \kappa}{\Gamma \vdash e\,\tau : \sigma[\tau/\alpha]}\ \textsc{Tm\_TApp}$$

$$\frac{\Gamma, c{:}\sigma_1 \sim_\rho \sigma_2 \vdash e : \tau}{\Gamma \vdash \lambda c{:}\sigma_1 \sim_\rho \sigma_2.e : \phi \Rightarrow \tau}\ \textsc{Tm\_CAbs}$$

$$\frac{\Gamma \vdash e : (\sigma_1 \sim_\rho \sigma_2) \Rightarrow \tau \\ \Gamma \vdash \gamma : \sigma_1 \sim_\rho \sigma_2}{\Gamma \vdash e\,\gamma : \tau}\ \textsc{Tm\_CApp}$$

$$\frac{K : \tau}{\Gamma \vdash K : \tau}\ \textsc{Tm\_DataCon}$$

$$\frac{\begin{array}{l}\Gamma \vdash e : D\,\overline{\sigma} \\ \Gamma \vdash \tau : \star \\ \forall\,alt_i \text{ s.t. } alt_i \in \overline{alt} : \\ \quad alt_i = K_i\,\overline{\alpha_i}\,\overline{c_i}\,\overline{x_i} \to e_i \\ \quad K_i : \forall\,\overline{\alpha'_i{:}\kappa_i}.\forall\,\overline{\beta'_i{:}\kappa'_i}.\overline{\phi}_i \Rightarrow \overline{\tau}_i \to D\,\overline{\alpha'_i} \\ \quad \Gamma, \overline{\alpha_i{:}\kappa'_i}, (\overline{c_i{:}\phi_i}, \overline{x_i{:}\tau_i})[\overline{\sigma/\alpha'_i}][\overline{\alpha_i/\beta'_i}] \vdash e_i : \tau \\ \overline{alt} \text{ is exhaustive}\end{array}}{\Gamma \vdash \mathbf{case}_\tau\ e\ \mathbf{of}\ \overline{alt} : \tau}\ \textsc{Tm\_Case}$$

$$\frac{\Gamma \vdash e : \tau_1 \\ \Gamma \vdash \gamma : \tau_1 \sim_{\mathsf{R}} \tau_2}{\Gamma \vdash e \triangleright \gamma : \tau_2}\ \textsc{Tm\_Cast}$$

$$\frac{\varnothing \vdash \gamma : H_1 \sim_{\mathsf{N}} H_2 \qquad H_1 \neq H_2 \\ \Gamma \vdash \tau : \star}{\Gamma \vdash \mathbf{contra}\,\gamma\,\tau : \tau}\ \textsc{Tm\_Contra}$$

C.3 Small-step operational semantics

$\boxed{e_1 \longrightarrow e_2}$ Small-step operational semantics

$$\frac{}{(\lambda x{:}\tau.e_1)\,e_2 \longrightarrow e_1[e_2/x]}\ \textsc{S\_Beta}$$

$$\frac{}{(\Lambda\alpha{:}\kappa.e)\,\tau \longrightarrow e[\tau/\alpha]}\ \textsc{S\_TBeta}$$

$$\frac{}{(\lambda c{:}\phi.e)\,\gamma \longrightarrow e[\gamma/c]}\ \textsc{S\_CBeta}$$

$$\frac{alt_i = K\,\overline{\alpha}\,\overline{c}\,\overline{x} \to e'}{\mathbf{case}_{\tau_0}\ K\,\overline{\tau}\,\overline{\sigma}\,\overline{\gamma}\,\overline{e}\ \mathbf{of}\ \overline{alt} \longrightarrow e'[\overline{\sigma/\alpha}][\overline{\gamma/c}][\overline{e/x}]}\ \textsc{S\_Iota}$$

$$\frac{e_1 \longrightarrow e'_1}{e_1\,e_2 \longrightarrow e'_1\,e_2}\ \textsc{S\_App\_Cong}$$

$$\frac{e \longrightarrow e'}{e\,\tau \longrightarrow e'\,\tau}\ \textsc{S\_TApp\_Cong}$$

$$\frac{e \longrightarrow e'}{e\,\gamma \longrightarrow e'\,\gamma}\ \textsc{S\_CApp\_Cong}$$

$$\frac{e \longrightarrow e'}{\mathbf{case}_\tau\ e\ \mathbf{of}\ \overline{alt} \longrightarrow \mathbf{case}_\tau\ e'\ \mathbf{of}\ \overline{alt}}\ \text{S\_CASE\_CONG}$$

$$\frac{e \longrightarrow e'}{e \triangleright \gamma \longrightarrow e' \triangleright \gamma}\ \text{S\_CAST\_CONG}$$

$$\frac{\begin{array}{l}\eta_1 = \mathbf{nth}^1\ \gamma \qquad \eta_2 = \mathbf{nth}^2\ \gamma \\ e' = e[x' \triangleright \mathbf{sym}\,\eta/x] \triangleright \eta_2 \\ \varnothing \vdash \gamma : (\tau \to \sigma) \sim_{\mathsf{R}} (\tau' \to \sigma')\end{array}}{(\lambda x{:}\tau.e) \triangleright \gamma \longrightarrow \lambda x'{:}\tau'.e'}\ \text{S\_PUSH}$$

$$\frac{e' = e \triangleright \gamma@\alpha}{(\Lambda\alpha{:}\kappa.e) \triangleright \gamma \longrightarrow \Lambda\alpha{:}\kappa.e'}\ \text{S\_TPUSH}$$

$$\frac{\begin{array}{l}\eta_1 = \mathbf{nth}^1\ (\mathbf{nth}^1\ \gamma) \\ \eta_2 = \mathbf{nth}^2\ (\mathbf{nth}^1\ \gamma) \\ e' = e[\eta_1 ⨾ c' ⨾ \mathbf{sym}\,\eta_2/c] \triangleright \mathbf{nth}^2\ \gamma \\ \varnothing \vdash \gamma : ((\sigma_1 \sim_\rho \sigma_2) \Rightarrow \tau) \sim_{\mathsf{R}} ((\sigma_3 \sim_\rho \sigma_4) \Rightarrow \tau')\end{array}}{(\lambda c{:}(\sigma_1 \sim_\rho \sigma_2).e) \triangleright \gamma \longrightarrow \lambda c'{:}(\sigma_3 \sim_\rho \sigma_4).e'}\ \text{S\_CPUSH}$$

$$\frac{\begin{array}{l}\varnothing \vdash \eta : D\,\overline{\tau} \sim_{\mathsf{R}} D\,\overline{\tau}' \\ K : \forall\,\overline{\alpha{:}\kappa}.\forall\,\overline{\beta{:}\kappa'}.\overline{(\sigma' \sim_\rho \sigma'')} \Rightarrow \overline{\tau}'' \to D\,\overline{\alpha} \\ \overline{\varnothing \vdash \gamma : (\sigma' \sim_\rho \sigma'')[\overline{\tau/\alpha}][\overline{\sigma/\beta}]} \\ \overline{\gamma' = \mathbf{sym}\,(\sigma'[\overline{\mathbf{nth}\,\eta/\alpha}]_\rho) ⨾ \gamma ⨾ \sigma''[\overline{\mathbf{nth}\,\eta/\alpha}]_\rho} \\ \overline{e' = e \triangleright \tau''[\overline{\mathbf{nth}\,\eta/\alpha}]_{\mathsf{R}}}\end{array}}{(K\,\overline{\tau}\,\overline{\sigma}\,\overline{\gamma}\,\overline{e}) \triangleright \eta \longrightarrow K\,\overline{\tau}'\,\overline{\sigma}\,\overline{\gamma}'\,\overline{e}'}\ \text{S\_KPUSH}$$

D. Global context well-formedness

We assume throughout the paper and this appendix that the global context is well formed. Here, we explain precisely what can appear in the global context and what restrictions there are:

1. The global context may contain $C : [\overline{\alpha{:}\kappa}].\tau \sim_\rho \sigma$:

   There are two forms of axiom, for which different rules apply:

   (a) Newtype axioms: All of the following must hold.

      i. $\tau = N\,\overline{\alpha}$

      ii. $\rho = \mathsf{R}$

      iii. There must not be two axioms mentioning the same newtype N.

      iv. The length of *roles*(N) must match the arity of the axiom C.

   (b) Type family axioms: All of the following must hold.

      i. $\tau = F(\overline{\tau}')$

      ii. $\rho = \mathsf{N}$

      iii. The types $\overline{\tau}'$ must not mention type families.

      iv. Each $\beta \in \overline{\alpha}$ must appear exactly once in the list $\overline{\tau}'$.

   Regardless of the form of axiom, the following must hold:

   (c) $\overline{\alpha{:}\kappa} \vdash \tau : \kappa_0$

   (d) $\overline{\alpha{:}\kappa} \vdash \sigma : \kappa_0$

   (e) Consider two axioms $C_1 : [\overline{\alpha{:}\kappa}].\tau_1 \sim_\rho \sigma_1$ and $C_2 : [\overline{\beta{:}\kappa'}].\tau_2 \sim_\rho \sigma_2$ (where variables are renamed so that $\overline{\alpha} \cap \overline{\beta} = \varnothing$). Then, if there exists some θ with $\theta(\tau_1) = \theta(\tau_2)$, it must be that $\theta(\sigma_1) = \theta(\sigma_2)$.

2. The global context may contain $T : \kappa$.

3. The global context may contain $K : \tau$:

   (a) $\tau = \forall\,\overline{\alpha{:}\kappa}.\forall\,\overline{\beta{:}\kappa'}.\overline{\phi} \Rightarrow \overline{\sigma} \to D\,\overline{\alpha}$

   (b) $\varnothing \vdash \tau : \star$

4. The global context may contain $F : [\overline{\alpha{:}\kappa}].\kappa_0$.

5. For all H, *roles*$(H) \models H$.

E. Properties of roles

Lemma 1 (Permutation of role checking). *If $\Omega \vdash \tau : \rho$ and Ω' is a permutation of Ω, then $\Omega' \vdash \tau : \rho$.*

Lemma 2 (Weakening of role checking). *If $\Omega \vdash \tau : \rho$, then $\Omega, \alpha{:}\rho' \vdash \tau : \rho$.*

Lemma 3 (Strengthening of role checking). *If $\Omega, \alpha{:}\rho' \vdash \tau : \rho$ and α does not appear free in τ, then $\Omega \vdash \tau : \rho$.*

Lemma 4 (Nominal roles are infectious). *Let $\overline{\alpha}$ be the free variables in σ. We have $\Omega \vdash \sigma : \mathsf{N}$ if and only if every $\alpha_i \in \overline{\alpha}$ must be at role N in Ω.*

Lemma 5 (Sub-roling). *If $\Omega \vdash \tau : \rho$ and $\rho \leq \rho'$, then $\Omega \vdash \tau : \rho'$.*

F. Structural properties

F.1 Weakening

Let *bnd* be a metavariable for a context binding. That is:

$$\begin{array}{rcl} bnd & ::= & \alpha{:}\kappa \\ & \mid & c{:}\phi \\ & \mid & x{:}\tau \end{array}$$

Lemma 6 (Type kinding weakening). *If $\Gamma, \Gamma' \vdash \tau : \kappa$ and $\vdash \Gamma, bnd, \Gamma'$, then $\Gamma, bnd, \Gamma' \vdash \tau : \kappa$.*

Lemma 7 (Coercion typing weakening). *If $\Gamma, \Gamma' \vdash \gamma : \phi$ and $\vdash \Gamma, bnd, \Gamma'$, then $\Gamma, bnd, \Gamma' \vdash \gamma : \phi$.*

Lemma 8 (Term typing weakening). *If $\Gamma, \Gamma' \vdash e : \tau$ and $\vdash \Gamma, bnd, \Gamma'$, then $\Gamma, bnd, \Gamma' \vdash e : \tau$.*

F.2 Substitution

Lemma 9 (Type variable substitution). *Suppose $\Gamma \vdash \sigma : \kappa_1$. Then:*

1. *If $\vdash \Gamma, \alpha{:}\kappa_1, \Gamma'$, then $\vdash \Gamma, \Gamma'[\sigma/\alpha]$;*
2. *If $\Gamma, \alpha{:}\kappa_1, \Gamma' \vdash \tau : \kappa_2$, then $\Gamma, \Gamma'[\sigma/\alpha] \vdash \tau[\sigma/\alpha] : \kappa_2$.*

Lemma 10 (Type variable substitution in coercions). *If $\Gamma, \alpha{:}\kappa, \Gamma' \vdash \gamma : \phi$ and $\Gamma \vdash \sigma : \kappa$, then $\Gamma, \Gamma'[\sigma/\alpha] \vdash \gamma[\sigma/\alpha] : \phi[\sigma/\alpha]$.*

Lemma 11 (Type variable substitution in terms). *If $\Gamma, \alpha{:}\kappa, \Gamma' \vdash e : \tau$ and $\Gamma \vdash \sigma : \kappa$, then $\Gamma, \Gamma'[\sigma/\alpha] \vdash e[\sigma/\alpha] : \tau[\sigma/\alpha]$.*

Lemma 12 (Coercion strengthening).

1. *If $\vdash \Gamma, c{:}\phi, \Gamma'$, then $\vdash \Gamma, \Gamma'$;*
2. *If $\Gamma, c{:}\phi, \Gamma' \vdash \tau : \kappa$, then $\Gamma, \Gamma' \vdash \tau : \kappa$.*

Lemma 13 (Coercion substitution). *If $\Gamma, c{:}\phi_1, \Gamma' \vdash \gamma : \phi_2$ and $\Gamma \vdash \eta : \phi_1$, then $\Gamma, \Gamma' \vdash \gamma[\eta/c] : \phi_2$.*

Lemma 14 (Coercion substitution in terms). *If $\Gamma, c{:}\phi, \Gamma' \vdash e : \tau$ and $\Gamma \vdash \eta : \phi$, then $\Gamma, \Gamma' \vdash e[\eta/c] : \tau$.*

Lemma 15 (Term strengthening).

1. *If $\vdash \Gamma, x{:}\tau, \Gamma'$, then $\vdash \Gamma, \Gamma'$;*
2. *If $\Gamma, x{:}\tau, \Gamma' \vdash \sigma : \kappa$, then $\Gamma, \Gamma' \vdash \sigma : \kappa$.*

Lemma 16 (Term strengthening in coercions). *If $\Gamma, x{:}\tau, \Gamma' \vdash \gamma : \phi$, then $\Gamma, \Gamma' \vdash \gamma : \phi$.*

Lemma 17 (Term substitution). *If $\Gamma, x{:}\sigma, \Gamma' \vdash e : \tau$ and $\Gamma \vdash e' : \sigma$, then $\Gamma, \Gamma' \vdash e[e'/x] : \tau$.*

F.3 Classifier regularity

Lemma 18 (Coercion typing regularity). *If $\Gamma \vdash \gamma : \tau \sim_\rho \sigma$, then $\Gamma \vdash \tau \sim_\rho \sigma : \star$.*

Lemma 19 (Coercion homogeneity). *If $\Gamma \vdash \gamma : \tau \sim_\rho \sigma$, then $\Gamma \vdash \tau : \kappa$ and $\Gamma \vdash \sigma : \kappa$.*

Proof. Direct from Lemma 18. □

Lemma 20 (Term typing regularity). *If $\Gamma \vdash e : \tau$, then $\Gamma \vdash \tau : \star$.*

G. Preservation

G.1 Lifting

Lifting is defined by the following algorithm, with patterns to be tried in order from top to bottom.

$$\begin{array}{lll}
\tau[\overline{\gamma/\beta}]_{\mathsf{P}} & = \langle \tau[\overline{\sigma/\beta}], \tau[\overline{\sigma'/\beta}]\rangle_{\mathsf{P}} & (\overline{\Gamma \vdash \gamma : \sigma \sim_\rho \sigma'}) \\
\alpha[\overline{\gamma/\beta}]_\rho & = \gamma_i & (\alpha = \beta_i \wedge \Gamma \vdash \gamma_i : \sigma \sim_\rho \sigma') \\
\alpha[\overline{\gamma/\beta}]_{\mathsf{R}} & = \mathbf{sub}\,\gamma_i & (\alpha = \beta_i) \\
\alpha[\overline{\gamma/\beta}]_{\mathsf{N}} & = \langle\alpha\rangle & (\alpha \notin \overline{\beta}) \\
\alpha[\overline{\gamma/\beta}]_{\mathsf{R}} & = \mathbf{sub}\,\langle\alpha\rangle & (\alpha \notin \overline{\beta}) \\
(H\,\overline{\tau})[\overline{\gamma/\beta}]_{\mathsf{R}} & = H(\overline{\tau[\overline{\gamma/\beta}]_\rho}) & (\overline{\rho} \text{ is a prefix of } \mathit{roles}(H)) \\
H[\overline{\gamma/\beta}]_{\mathsf{N}} & = \langle H\rangle & \\
(\tau_1\,\tau_2)[\overline{\gamma/\beta}]_\rho & = \tau_1[\overline{\gamma/\beta}]_\rho\,\tau_2[\overline{\gamma/\beta}]_{\mathsf{N}} & \\
(\forall\,\alpha{:}\kappa.\tau)[\overline{\gamma/\beta}]_\rho & = \forall\,\alpha{:}\kappa.\tau[\overline{\gamma/\beta}]_\rho & \\
(F(\overline{\tau}))[\overline{\gamma/\beta}]_{\mathsf{N}} & = F(\overline{\tau[\overline{\gamma/\beta}]_{\mathsf{N}}}) & \\
(F(\overline{\tau}))[\overline{\gamma/\beta}]_{\mathsf{R}} & = \mathbf{sub}\,F(\overline{\tau[\overline{\gamma/\beta}]_{\mathsf{N}}}) &
\end{array}$$

Lemma 21 (Lifting). *If:*

1. $\Gamma \vdash \gamma : H\,\overline{\tau} \sim_{\mathsf{R}} H\,\overline{\sigma}$;
2. $\overline{\Gamma \vdash \tau : \kappa}$;
3. $\overline{\Gamma \vdash \sigma : \kappa}$;
4. *H is not a **newtype**;*
5. $\Omega \vdash \sigma_0 : \rho_0$, *where $\overline{\beta'}$ is the type variables in Γ, Γ'*
   $\Omega = \overline{\beta'{:}\mathsf{N}}, \overline{\beta : \mathit{roles}(H)}$;
6. $\Gamma, \overline{\beta{:}\kappa}, \Gamma' \vdash \sigma_0 : \kappa'$; *and*
7. *Γ' contains only type variable bindings.*

then:

$$\Gamma, \Gamma' \vdash \sigma_0[\overline{\mathbf{nth}\,\gamma/\beta}]_{\rho_0} : \sigma_0[\overline{\tau/\beta}] \sim_{\rho_0} \sigma_0[\overline{\sigma/\beta}]$$

Proof. First, because Γ' contains only type variable bindings, then a type variable substitution has no effect on Γ' (which can contain only *kinds*).

If $\rho_0 = \mathsf{P}$, then the first equation of the algorithm matches, and we have $\sigma_0[\overline{\mathbf{nth}\,\gamma/\beta}]_{\mathsf{P}} = \langle\sigma_0[\overline{\tau/\beta}], \sigma_0[\overline{\sigma/\beta}]\rangle_{\mathsf{P}}$, and we are done, applying Lemma 9.

So, we assume now that $\rho_0 \neq \mathsf{P}$.

Let $\overline{\rho} = \mathit{roles}(H)$.

We proceed by induction on the derivation of $\Gamma, \overline{\beta{:}\kappa}, \Gamma' \vdash \sigma_0 : \kappa'$. Each case concludes by the application of the appropriate substitution lemma(s).

Case Ty_Var: We know $\sigma_0 = \alpha$.

Case ($\alpha = \beta_i$):

Case ($\rho_0 = \rho_i$): Here, we have $\sigma_0[\overline{\mathbf{nth}\,\gamma/\beta}]_{\rho_0} = \mathbf{nth}^i\,\gamma$, $\sigma_0[\overline{\tau/\beta}] = \tau_i$, and $\sigma_0[\overline{\sigma/\beta}] = \sigma_i$. Thus, we are done, by Co_Nth.

Case ($\rho_0 = \mathsf{R}, \rho_i = \mathsf{N}$): Similar to the last case, fixing the roles with a use of **sub**.

Case ($\rho_0 = \mathsf{N}, \rho_i \neq \mathsf{N}$): This case is impossible. We know $\Omega \vdash \alpha : \mathsf{N}$. By inversion then, we know $\alpha{:}\mathsf{N} \in \Omega$. Yet, we know that ρ_i is the ith role in $\mathit{roles}(H)$, and by the definition of Ω, $\alpha{:}\rho_i \in \Omega$. This contradicts $\rho_i \neq \mathsf{N}$, and we are done.

Case ($\alpha \notin \overline{\beta}$):

Case ($\rho_0 = \mathsf{N}$): Here, $\sigma_0[\overline{\mathbf{nth}\,\gamma/\beta}]_{\mathsf{N}} = \langle\sigma_0\rangle$, $\sigma_0[\overline{\tau/\beta}] = \sigma_0$, and $\sigma_0[\overline{\sigma/\beta}] = \sigma_0$, so we are done, by Co_Refl.

Case ($\rho_0 = \mathsf{R}$): Similar to last case, fixing the output role with **sub**.

Case Ty_App:

Case ($\sigma_0 = H'\,\overline{\sigma'}, \rho_0 = \mathsf{R}$): Here $(H'\,\overline{\sigma'})[\overline{\mathbf{nth}\,\gamma/\beta}]_{\mathsf{R}} = H'(\overline{\sigma'[\overline{\mathbf{nth}\,\gamma/\beta}]_{\rho'}})$, where $\overline{\rho'}$ is a prefix of $\mathit{roles}(H')$. Let $\eta = H'(\overline{\sigma'[\overline{\mathbf{nth}\,\gamma/\beta}]_{\rho'}})$. Then, we must show $\Gamma, \Gamma' \vdash \eta : H'\,\overline{\sigma'}[\overline{\tau/\beta}] \sim_{\mathsf{R}} H'\,\overline{\sigma'}[\overline{\sigma/\beta}]$. We will use Co_TyConApp. We must show

$$\overline{\Gamma, \Gamma' \vdash \sigma'[\overline{\mathbf{nth}\,\gamma/\beta}]_{\rho'} : \sigma'[\overline{\tau/\beta}] \sim_{\rho'} \sigma'[\overline{\sigma/\beta}]}.$$

We do this by induction, for each $\sigma'_i \in \overline{\sigma'}$. All of the premises of the lifting lemma are satisfied automatically, except for premise 5. Fix i. We must show $\Omega \vdash \sigma'_i : \rho'_i$. We know $\Omega \vdash H'\,\overline{\sigma'} : \mathsf{R}$. This can be proved by either RTy_TyConApp or RTy_App. If it is by the former, we are done by inversion. If it is by the latter, then we know $\Omega \vdash \sigma'_i : \mathsf{N}$. We apply Lemma 5, and we are done.

Other applications: Apply the induction hypothesis. Premise 5 of the lifting lemma is satisfied by correspondence between RTy_App and Co_App.

Case Ty_ADT:

Case ($\rho_0 = \mathsf{N}$): Here $H[\overline{\mathbf{nth}\,\gamma/\beta}]_{\mathsf{N}} = \langle H\rangle$, and we are done by Co_Refl.

Case ($\rho_0 = \mathsf{R}$): Here $H[\overline{\mathbf{nth}\,\gamma/\beta}]_{\mathsf{R}} = H(\varnothing)$ and we are done by Co_TyConApp.

Cases Ty_Arrow, Ty_Equality: Similar to Ty_ADT.

Case Ty_ForAll: By the induction hypothesis. Note that the roles in RTy_ForAll and Co_ForAll match up, and that the new binding in RTy_ForAll is given a nominal role, echoed in the definition of Ω in this lemma's premises.

Case Ty_TyFun: By the induction hypothesis, once again noting the correspondence between RTy_TyFam and Co_TyFam.

□

G.2 Preservation

Theorem 22 (Preservation). *If $\Gamma \vdash e : \tau$ and $e \longrightarrow e'$, then $\Gamma \vdash e' : \tau$.*

Proof. By induction on the derivation of $e \longrightarrow e'$.

Beta rules: By substitution.

Case S_Iota: We know $\Gamma \vdash \mathbf{case}_{\tau_0}\,K\,\overline{\tau}\,\overline{\sigma}\,\overline{\gamma}\,\overline{e}\;\mathbf{of}\;\overline{alt} : \tau_0$, where $alt_i = K\,\overline{\alpha}\,\overline{c}\,\overline{x} \to e'$. We must show $\Gamma \vdash e'[\overline{\sigma/\alpha}][\overline{\gamma/c}][\overline{e/x}]$:

τ_0. By inversion on TM_CASE, we see

$$\Gamma \vdash K\,\overline{\tau}\,\overline{\sigma}\,\overline{\gamma}\,\overline{e} : D\,\overline{\tau}$$
$$K : \forall\,\overline{\alpha':\kappa}.\forall\,\overline{\beta':\kappa'}.\overline{\phi} \Rightarrow \overline{\tau}' \to D\,\overline{\alpha'}$$
$$\Gamma, \overline{\alpha{:}\kappa'}, \overline{c{:}\phi[\overline{\tau/\alpha'}][\overline{\alpha/\beta'}]}, \overline{x{:}\tau'[\overline{\tau/\alpha'}][\overline{\alpha/\beta'}]} \vdash e' : \tau_0$$

We also know that $\Gamma \vdash \tau_0 : \star$, which implies that none of the variables $\overline{\alpha}$ are mentioned in τ_0. We can do induction on the length of $\overline{\tau}$ to see that

$$\Gamma \vdash K\,\overline{\tau} : \forall\,\overline{\beta':\kappa'}.\overline{\phi}[\overline{\tau/\alpha'}] \Rightarrow \overline{\tau}'[\overline{\tau/\alpha'}] \to D\,\overline{\alpha'}[\overline{\tau/\alpha'}]$$

This simplifies to

$$\Gamma \vdash K\,\overline{\tau} : \forall\,\overline{\beta':\kappa'}.\overline{\phi}[\overline{\tau/\alpha'}] \Rightarrow \overline{\tau}'[\overline{\tau/\alpha'}] \to D\,\overline{\tau}$$

Now, we do induction on the length of $\overline{\sigma}$ to see that

$$\Gamma \vdash K\,\overline{\tau}\,\overline{\sigma} : \overline{\phi}[\overline{\tau/\alpha'}][\overline{\sigma/\beta'}] \Rightarrow \overline{\tau}'[\overline{\tau/\alpha'}][\overline{\sigma/\beta'}] \to D\,\overline{\tau}$$

and

$$\overline{\Gamma \vdash \sigma : \kappa'}$$

We can then use repeated application of the type variable substitution lemma to get

$$\Gamma, \overline{c{:}\phi[\overline{\tau/\alpha'}][\overline{\sigma/\beta'}]}, \overline{x{:}\tau'[\overline{\tau/\alpha'}][\overline{\sigma/\beta'}]} \vdash e'[\overline{\sigma/\alpha}] : \tau_0$$

using the following facts

$$\tau_0[\overline{\sigma/\alpha}] = \tau_0$$
$$\phi[\overline{\tau/\alpha'}][\overline{\alpha/\beta'}][\overline{\sigma/\alpha}] = \phi[\overline{\tau/\alpha'}][\overline{\sigma/\beta'}]$$
$$\tau'[\overline{\tau/\alpha'}][\overline{\alpha/\beta'}][\overline{\sigma/\alpha}] = \tau'[\overline{\tau/\alpha'}][\overline{\sigma/\beta'}]$$

So, we have

$$\Gamma, \overline{c{:}\phi[\overline{\tau/\alpha'}][\overline{\sigma/\beta'}]}, \overline{x{:}\tau'[\overline{\tau/\alpha'}][\overline{\sigma/\beta'}]} \vdash e'[\overline{\sigma/\alpha}] : \tau_0$$

Starting from the type of $K\,\overline{\tau}\,\overline{\sigma}$, we do induction on the length of $\overline{\gamma}$ to get

$$\Gamma \vdash K\,\overline{\tau}\,\overline{\sigma}\,\overline{\gamma} : \overline{\tau}'[\overline{\tau/\alpha'}][\overline{\sigma/\beta'}] \to D\,\overline{\tau}$$

and

$$\overline{\Gamma \vdash \gamma : \phi[\overline{\tau/\alpha'}][\overline{\sigma/\beta'}]}$$

Thus, we can use the coercion variable substitution lemma to get

$$\Gamma, \overline{x{:}\tau'[\overline{\tau/\alpha'}][\overline{\sigma/\beta'}]} \vdash e'[\overline{\sigma/\alpha}][\overline{\gamma/c}] : \tau_0$$

Finally we use analogous reasoning for term arguments $\overline{e}$ to conclude

$$\Gamma \vdash e'[\overline{\sigma/\alpha}][\overline{\gamma/c}][\overline{e/x}] : \tau_0$$

as desired.

Congruence rules: By induction.

Case S_PUSH: We adopt the variables names from the statement of the rule:

$$\frac{\begin{array}{l} \eta_1 = \mathbf{nth}^1\,\gamma \qquad\qquad \eta_2 = \mathbf{nth}^2\,\gamma \\ e' = e[x' \triangleright \mathbf{sym}\,\eta/x] \triangleright \eta_2 \\ \varnothing \vdash \gamma : (\tau \to \sigma) \sim_{\mathsf{R}} (\tau' \to \sigma') \end{array}}{(\lambda x{:}\tau.e) \triangleright \gamma \longrightarrow \lambda x'{:}\tau'.e'}\ \text{S\_PUSH}$$

We know that $\Gamma \vdash (\lambda x{:}\tau.e) \triangleright \gamma : \tau' \to \sigma'$. Inversion tells us $\Gamma, x{:}\tau \vdash e : \sigma$ and $\Gamma \vdash \gamma : (\tau \to \sigma) \sim_{\mathsf{R}} (\tau' \to \sigma')$. We see that $\Gamma \vdash \eta : \tau \sim_{\mathsf{R}} \tau'$ and $\Gamma, x'{:}\tau' \vdash x' \triangleright \mathbf{sym}\,\eta : \tau$. Then, we use weakening to get $\Gamma, x'{:}\tau', x{:}\tau \vdash e : \sigma$ and substitution to get $\Gamma, x'{:}\tau' \vdash e[x' \triangleright \mathbf{sym}\,\eta/x] : \sigma$. Thus, $\Gamma, x'{:}\tau' \vdash e' : \sigma'$. Accordingly, $\Gamma \vdash (\lambda x'{:}\tau'.e') : \tau' \to \sigma'$ as desired.

Case S_TPUSH: We know that $\Gamma \vdash (\Lambda\alpha{:}\kappa.e) \triangleright \gamma : \tau$. Inversion tells us $\Gamma, \alpha{:}\kappa \vdash e : \sigma$ and $\Gamma \vdash \gamma : \forall\,\alpha{:}\kappa.\sigma \sim_{\mathsf{R}} \forall\,\alpha{:}\kappa.\sigma'$, where $\tau = \sigma'[\tau_1/\alpha]$. Let $e' = e \triangleright \gamma@\alpha$. Then, we can see that $\Gamma, \alpha{:}\kappa \vdash \gamma@\alpha : \sigma \sim_{\mathsf{R}} \sigma'$ and thus that $\Gamma, \alpha{:}\kappa \vdash e' : \sigma'$ and $\Gamma \vdash \Lambda\alpha{:}\kappa.e' : \tau$ as desired.

Case S_CPUSH: We adopt the variable names from the statement of the rule:

$$\frac{\begin{array}{l} \eta_1 = \mathbf{nth}^1\,(\mathbf{nth}^1\,\gamma) \\ \eta_2 = \mathbf{nth}^2\,(\mathbf{nth}^1\,\gamma) \\ e' = e[\eta_1 \fatsemi c' \fatsemi \mathbf{sym}\,\eta_2/c] \triangleright \mathbf{nth}^2\,\gamma \\ \varnothing \vdash \gamma : ((\sigma_1 \sim_\rho \sigma_2) \Rightarrow \tau) \sim_{\mathsf{R}} ((\sigma_3 \sim_\rho \sigma_4) \Rightarrow \tau') \end{array}}{(\lambda c{:}(\sigma_1 \sim_\rho \sigma_2).e) \triangleright \gamma \longrightarrow \lambda c'{:}(\sigma_3 \sim_\rho \sigma_4).e'}\ \text{S\_CPUSH}$$

We know that $\Gamma \vdash \lambda c{:}(\sigma_1 \sim_\rho \sigma_2).e \triangleright \gamma : (\sigma_3 \sim_\rho \sigma_4) \Rightarrow \tau'$ and that $\varnothing \vdash \gamma : ((\sigma_1 \sim_\rho \sigma_2) \Rightarrow \tau) \sim_{\mathsf{R}} ((\sigma_3 \sim_\rho \sigma_4) \Rightarrow \tau')$. Inversion on the first typing judgement tells us $\Gamma, c{:}(\sigma_1 \sim_\rho \sigma_2) \vdash e : \tau$. We can see that $\Gamma \vdash \eta_1 : \sigma_1 \sim_\rho \sigma_3$ and $\Gamma \vdash \eta_2 : \sigma_2 \sim_\rho \sigma_4$. We further see that $\Gamma, c'{:}(\sigma_3 \sim_\rho \sigma_4) \vdash \eta_1 \fatsemi c' \fatsemi \mathbf{sym}\,\eta_2 : \sigma_1 \sim_\rho \sigma_2$. Now, we use weakening to get $\Gamma, c'{:}(\sigma_3 \sim_\rho \sigma_4), c{:}(\sigma_1 \sim_\rho \sigma_2) \vdash e : \tau$ and substitution to get $\Gamma, c'{:}(\sigma_3 \sim_\rho \sigma_4) \vdash e[\eta_1 \fatsemi c' \fatsemi \mathbf{sym}\,\eta_2/c] : \tau$. Thus, $\Gamma, c'{:}(\sigma_3 \sim_\rho \sigma_4) \vdash e' : \tau'$ and then $\Gamma \vdash \lambda c'{:}(\sigma_3 \sim_\rho \sigma_4).e' : (\sigma_3 \sim_\rho \sigma_4) \Rightarrow \tau'$, as desired.

Case S_KPUSH: We adopt the variable names from the statement of the rule:

$$\frac{\begin{array}{l} \varnothing \vdash \eta : D\,\overline{\tau} \sim_{\mathsf{R}} D\,\overline{\tau}' \\ K : \forall\,\overline{\alpha{:}\kappa}.\forall\,\overline{\beta{:}\kappa'}.\overline{(\sigma' \sim_\rho \sigma'')} \Rightarrow \overline{\tau}'' \to D\,\overline{\alpha} \\ \overline{\varnothing \vdash \gamma : (\sigma' \sim_\rho \sigma'')[\overline{\tau/\alpha}][\overline{\sigma/\beta}]} \\ \overline{\gamma' = \mathbf{sym}\,(\sigma'[\overline{\mathbf{nth}\,\eta/\alpha}]_\rho) \fatsemi \gamma \fatsemi \sigma''[\overline{\mathbf{nth}\,\eta/\alpha}]_\rho} \\ \overline{e' = e \triangleright \tau''[\overline{\mathbf{nth}\,\eta/\alpha}]_{\mathsf{R}}} \end{array}}{(K\,\overline{\tau}\,\overline{\sigma}\,\overline{\gamma}\,\overline{e}) \triangleright \eta \longrightarrow K\,\overline{\tau}'\,\overline{\sigma}\,\overline{\gamma}'\,\overline{e}'}\ \text{S\_KPUSH}$$

Inversion gives us the premises of this rule. We also know $\Gamma \vdash (K\,\overline{\tau}\,\overline{\sigma}\,\overline{\gamma}\,\overline{e}) \triangleright \eta : D\,\overline{\tau}'$. We must show $\Gamma \vdash (K\,\overline{\tau}'\,\overline{\sigma}\,\overline{\gamma}'\,\overline{e}') : D\,\overline{\tau}'$.

Let $\overline{\phi} = \overline{(\sigma' \sim_\rho \sigma'')}$. From repeated inversion (and induction on the length of $\overline{\tau}$), we can derive

$$\overline{\Gamma \vdash \tau : \kappa}$$

Then, from homogeneity of coercions (Lemma 19) (and more induction on $\overline{\tau}'$), we see that

$$\overline{\Gamma \vdash \tau' : \kappa}$$

Putting this together, we get

$$\Gamma \vdash K\,\overline{\tau}' : (\forall\,\overline{\beta{:}\kappa'}.\overline{\phi} \Rightarrow \overline{\tau}'' \to D\,\overline{\alpha})[\overline{\tau'/\alpha}]$$

or

$$\Gamma \vdash K\,\overline{\tau}' : \forall\,\overline{\beta{:}\kappa'}.\overline{\phi}[\overline{\tau'/\alpha}] \Rightarrow \overline{\tau}''[\overline{\tau'/\alpha}] \to D\,\overline{\tau}'$$

Taking $K\,\overline{\tau}\,\overline{\sigma}\,\overline{\gamma}\,\overline{e}$ apart further (and induction on $\overline{\sigma}$) tells us

$$\overline{\Gamma \vdash \sigma : \kappa'}$$

and thus that

$$\Gamma \vdash K\,\overline{\tau}'\,\overline{\sigma} : \overline{\phi}[\overline{\tau'/\alpha}][\overline{\sigma/\beta}] \Rightarrow \overline{\tau}''[\overline{\tau'/\alpha}][\overline{\sigma/\beta}] \to D\,\overline{\tau}'[\overline{\sigma/\beta}]$$

But, from $\overline{\Gamma \vdash \tau' : \kappa}$, we see that $\overline{\beta}$ do not appear in $\overline{\tau}'$. So, we have

$$\Gamma \vdash K\,\overline{\tau}'\,\overline{\sigma} : \overline{\phi}[\overline{\tau'/\alpha}][\overline{\sigma/\beta}] \Rightarrow \overline{\tau}''[\overline{\tau'/\alpha}][\overline{\sigma/\beta}] \to D\,\overline{\tau}'$$

Using techniques similar to that for $\overline{\tau}$ and $\overline{\sigma}$, we can derive the following:

$$\overline{\Gamma \vdash \gamma : \phi[\overline{\tau/\alpha}][\overline{\sigma/\beta}]}$$
$$\overline{\Gamma \vdash e : \tau''[\overline{\tau/\alpha}][\overline{\sigma/\beta}]}$$

We need to conclude the following:

$$\overline{\Gamma \vdash \gamma' : \phi[\overline{\tau'/\alpha}][\overline{\sigma/\beta}]}$$
$$\overline{\Gamma \vdash e' : \tau''[\overline{\tau'/\alpha}][\overline{\sigma/\beta}]}$$

We wish to use the lifting lemma (Lemma 21) to get types for $\overline{\sigma'[\overline{\mathbf{nth}\,\eta/\alpha}]_\rho}$ and $\overline{\sigma''[\overline{\mathbf{nth}\,\eta/\alpha}]_\rho}$. So, we must first establish the premises of the lifting lemma.

1. $\Gamma \vdash \eta : D\,\overline{\tau} \sim_{\mathsf{R}} D\,\overline{\tau'}$, from the inversion on S_KPUSH (and weakening to change the context);
2. $\overline{\Gamma \vdash \tau : \kappa}$, as above;
3. $\overline{\Gamma \vdash \tau' : \kappa}$, as above;
4. D is not a **newtype**: by choice of metavariable.
5. $\overline{\Omega \vdash \sigma' : \rho}$ and $\overline{\Omega \vdash \sigma'' : \rho}$: Here, $\Omega = \overline{\beta'}{:}\mathsf{N}, \overline{\alpha} : \mathit{roles}(D)$ where $\overline{\beta'}$ are the type variables bound in Γ, along with the existential variables $\overline{\beta}$. (That is, the Γ' in the statement of the lifting lemma is $\overline{\beta{:}\kappa'}$.) By ROLES_DATA, we can see that $\overline{\Omega \vdash (\sigma' \sim_\rho \sigma'')} : \mathsf{R}$. This can be established by either RTY_TYCONAPP or by RTY_APP. In the former case, we get the desired outcome by looking at ROLES_EQUALITY. In the latter case, we see that $\Omega \vdash \sigma'_i : \mathsf{N}$ or $\Omega \vdash \sigma''_i : \mathsf{N}$ and then use role subsumption (Lemma 5).
6. $\Gamma, \overline{\alpha{:}\kappa}, \overline{\beta{:}\kappa'} \vdash \sigma' : \kappa''$ and the same for σ'': This comes from the well-formedness of the global context, including the type of K.
7. $\overline{\beta{:}\kappa'}$ must contain only type variable bindings: It sure does.

Now, we can conclude

$$\overline{\Gamma, \overline{\beta{:}\kappa'} \vdash \sigma'[\overline{\mathbf{nth}\,\eta/\alpha}]_\rho : \sigma'[\overline{\tau/\alpha}] \sim_\rho \sigma'[\overline{\tau'/\alpha}]}$$
$$\overline{\Gamma, \overline{\beta{:}\kappa'} \vdash \sigma''[\overline{\mathbf{nth}\,\eta/\alpha}]_\rho : \sigma''[\overline{\tau/\alpha}] \sim_\rho \sigma''[\overline{\tau'/\alpha}]}$$

We then do type variable substitution to get

$$\overline{\Gamma \vdash \sigma'[\overline{\mathbf{nth}\,\eta/\alpha}]_\rho[\overline{\sigma/\beta}] : \sigma'[\overline{\tau/\alpha}][\overline{\sigma/\beta}] \sim_\rho \sigma'[\overline{\tau'/\alpha}][\overline{\sigma/\beta}]}$$
$$\overline{\Gamma \vdash \sigma''[\overline{\mathbf{nth}\,\eta/\alpha}]_\rho[\overline{\sigma/\beta}] : \sigma''[\overline{\tau/\alpha}][\overline{\sigma/\beta}] \sim_\rho \sigma''[\overline{\tau'/\alpha}][\overline{\sigma/\beta}]}$$

Now, by CO_TRANS, we can conclude

$$\overline{\Gamma \vdash \gamma' : \phi[\overline{\tau'/\alpha}][\overline{\sigma/\beta}]}$$

as desired.
To type the $\overline{e'}$, we need to apply the lifting lemma once again, this time to $\overline{\tau''[\overline{\mathbf{nth}\,\eta/\alpha}]_{\mathsf{R}}}$. Much of our work at establishing premises carries over, except for these:

5. $\overline{\Omega \vdash \tau''} : \mathsf{R}$ (with Ω as above): This comes directly from the premises of ROLES_DATA, noting that $\overline{\tau''}$ appears in as an argument type to K.
6. $\overline{\Gamma, \overline{\alpha{:}\kappa}, \overline{\beta{:}\kappa'} \vdash \tau'' : \kappa''}$: This comes from the well-formedness of the global context, including the type of K.

We then apply the lifting lemma to conclude that

$$\overline{\Gamma, \overline{\beta{:}\kappa'} \vdash \tau''[\overline{\mathbf{nth}\,\gamma/\alpha}]_{\mathsf{R}} : \tau''[\overline{\tau/\alpha}] \sim_{\mathsf{R}} \tau''[\overline{\tau'/\alpha}]}$$

We use type variable substitution to get

$$\overline{\Gamma \vdash \tau''[\overline{\mathbf{nth}\,\gamma/\alpha}]_{\mathsf{R}}[\overline{\sigma/\beta}] : \tau''[\overline{\tau/\alpha}][\overline{\sigma/\beta}] \sim_{\mathsf{R}} \tau''[\overline{\tau'/\alpha}][\overline{\sigma/\beta}]}$$

We can then conclude

$$\overline{\Gamma \vdash e' : \tau''[\overline{\tau'/\alpha}][\overline{\sigma/\beta}]}$$

as desired.
Putting this all together, we see that $\Gamma \vdash K\,\overline{\tau'}\,\overline{\sigma}\,\overline{\gamma'}\,\overline{e'} : D\,\overline{\tau'}$ as originally desired, and we are done.

□

H. Progress

H.1 Consistency

Definition 23 (Type consistency). *Two types τ_1 and τ_2 are* consistent *if, whenever they are both value types:*

1. *If $\tau_1 = H\,\overline{\sigma}$, then $\tau_2 = H\,\overline{\sigma'}$;*
2. *If $\tau_1 = \forall\,\alpha{:}\kappa.\sigma$ then $\tau_2 = \forall\,\alpha{:}\kappa.\sigma'$.*

Note that if either τ_1 or τ_2 is *not* a value type, then they are vacuously consistent. Also, recall that a type headed by a **newtype** is not a value type.

Definition 24 (Context consistency). *The global context is* consistent *if, whenever $\varnothing \vdash \gamma : \tau_1 \sim_{\mathsf{R}} \tau_2$, τ_1 and τ_2 are consistent.*

In order to prove consistency, we define a type reduction relation $\tau \rightsquigarrow_\rho \sigma$, show that the relation preserves value type heads, and then show that any well-typed coercion corresponds to a path in the rewrite relation.

Here is the type rewrite relation:

$\boxed{\tau \rightsquigarrow_\rho \sigma}$ Type reduction

$$\frac{}{\tau \rightsquigarrow_\rho \tau}\ \textsc{Red\_Refl}$$

$$\frac{\tau_1 \rightsquigarrow_\rho \sigma_1 \qquad \tau_2 \rightsquigarrow_{\mathsf{N}} \sigma_2}{\tau_1\,\tau_2 \rightsquigarrow_\rho \sigma_1\,\sigma_2}\ \textsc{Red\_App}$$

$$\frac{\overline{\tau \rightsquigarrow_\rho \sigma} \qquad \overline{\rho} \text{ is a prefix of } \mathit{roles}(H)}{H\,\overline{\tau} \rightsquigarrow_{\mathsf{R}} H\,\overline{\sigma}}\ \textsc{Red\_TyConApp}$$

$$\frac{\tau \rightsquigarrow_\rho \sigma}{\forall\,\alpha{:}\kappa.\tau \rightsquigarrow_\rho \forall\,\alpha{:}\kappa.\sigma}\ \textsc{Red\_ForAll}$$

$$\frac{\overline{\tau \rightsquigarrow_{\mathsf{N}} \sigma}}{F(\overline{\tau}) \rightsquigarrow_\rho F(\overline{\sigma})}\ \textsc{Red\_TyFam}$$

$$\frac{C : [\overline{\alpha{:}\kappa}].\tau_1 \sim_\rho \tau_2 \qquad \rho \leq \rho'}{\tau_1[\overline{\sigma/\alpha}] \rightsquigarrow_{\rho'} \tau_2[\overline{\sigma/\alpha}]}\ \textsc{Red\_Axiom}$$

$$\frac{}{\tau \rightsquigarrow_{\mathsf{P}} \sigma}\ \textsc{Red\_Phantom}$$

Lemma 25 (Simple rewrite substitution). *If $\tau_1 \rightsquigarrow_\rho \tau_2$, then $\tau_1[\sigma/\alpha] \rightsquigarrow_\rho \tau_2[\sigma/\alpha]$.*

Proof. By straightforward induction, noting that axioms have no free variables. □

Lemma 26 (Rewrite substitution). *Let $\overline{\alpha}$ be the free variables in a type σ. If $\overline{\alpha{:}\rho} \vdash \sigma : \mathsf{R}$:*

1. *If $\overline{\tau \rightsquigarrow_\rho \tau'}$, then $\sigma[\overline{\tau/\alpha}] \rightsquigarrow_{\mathsf{R}} \sigma[\overline{\tau'/\alpha}]$;*
2. *If $\overline{\tau \rightsquigarrow_{\mathsf{N}} \tau'}$, then $\sigma[\overline{\tau/\alpha}] \rightsquigarrow_{\mathsf{N}} \sigma[\overline{\tau'/\alpha}]$.*

Proof. Let $\Omega = \overline{\alpha{:}\rho}$. Proceed by induction on the structure of σ.

Case $\sigma = \alpha$: There is thus only one free variable, α in σ. The one role ρ is R. For clause (1), we know $\tau \rightsquigarrow_{\mathsf{R}} \tau'$, so we are done. For clause (2), we know $\tau \rightsquigarrow_{\mathsf{N}} \tau'$, so we are done.

Case $\sigma = \sigma_1\,\sigma_2$:

Case (σ can be written as $H\,\overline{\sigma}$): Here, we assume that the length of $\overline{\sigma}$ is at most the length of *roles*(H). If this is not the case, fall through to the "otherwise" case.

Clause (1): We know $\overline{\tau \rightsquigarrow_\rho \tau'}$. We must show that $H\,\overline{\sigma}[\overline{\tau/\alpha}] \rightsquigarrow_{\mathsf{R}} H\,\overline{\sigma}[\overline{\tau'/\alpha}]$. We will use RED_TYCONAPP. Let $\overline{\rho}'$ be a prefix of *roles*(H) of the same length as $\overline{\sigma}$. We must show $\overline{\sigma[\overline{\tau/\alpha}] \rightsquigarrow_{\rho'} \sigma[\overline{\tau'/\alpha}]}$.

Fix i. We will show that $\sigma_i[\overline{\tau/\alpha}] \rightsquigarrow_{\rho'_i} \sigma_i[\overline{\tau'/\alpha}]$.

Case ($\rho'_i = \mathsf{N}$): In order to use the induction hypothesis, we must show that for every j such that α_j appears free in σ_i, $\rho_j = \mathsf{N}$. To use Lemma 4, we must establish that $\Omega \vdash \sigma_i : \mathsf{N}$. We can get this by inversion on $\Omega \vdash H\,\overline{\sigma} : \mathsf{R}$ – whether by RTY_TYCONAPP or by RTY_APP, we get $\Omega \vdash \sigma_i : \mathsf{N}$. So, we can use the induction hypothesis and we are done.

Case ($\rho'_i = \mathsf{R}$): Inverting $\overline{\alpha{:}\rho} \vdash H\,\overline{\sigma} : \mathsf{R}$ gives us two possibilities:

Case RTY_TYCONAPP: Here, we see $\overline{\Omega \vdash \sigma : \rho'}$, and thus, that $\Omega \vdash \sigma_i : \mathsf{R}$ (because $\rho'_i = \mathsf{R}$). We can then use the induction hypothesis (and using Lemma 3 to make the contexts line up) and we are done.

Case RTY_APP: We invert repeatedly, and we either get $\Omega \vdash \sigma_i : \mathsf{N}$ or $\Omega \vdash \sigma_i : \rho'_i$, depending on whether we hit a RTY_TYCONAPP during the inversions. In the second case, we proceed as above (the RTY_TYCONAPP case). In the first case, we use Lemma 5 to conclude $\Omega \vdash \sigma_i : \mathsf{R}$ and use the induction hypothesis.

Case ($\rho'_i = \mathsf{P}$): We are done by RED_PHANTOM.

Clause (2): We know that $\overline{\tau \rightsquigarrow_{\mathsf{N}} \tau'}$. We must show that $H\,\overline{\sigma}[\overline{\tau/\alpha}] \rightsquigarrow_{\mathsf{N}} H\,\overline{\sigma}[\overline{\tau'/\alpha}]$. It is easier to consider the original type σ just as $\sigma_1\,\sigma_2$, not as $H\,\overline{\sigma}$; fall through to the next case.

Otherwise:

Clause (1): We know $\overline{\tau \rightsquigarrow_\rho \tau'}$ and need to show that $(\sigma_1\,\sigma_2)[\overline{\tau/\alpha}] \rightsquigarrow_{\mathsf{R}} (\sigma_1\,\sigma_2)[\overline{\tau'/\alpha}]$. The fact $\Omega \vdash \sigma_1\,\sigma_2 : \mathsf{R}$ must be by RTY_APP. So, we can conclude $\Omega \vdash \sigma_1 : \mathsf{R}$ and $\Omega \vdash \sigma_2 : \mathsf{N}$. Then, we can use the induction hypothesis to get $\sigma_1[\overline{\tau/\alpha}] \rightsquigarrow_{\mathsf{R}} \sigma_1[\overline{\tau'/\alpha}]$. To use the induction hypothesis for σ_2, we must first establish that, for every j such that α_j appears free in σ_2, $\tau_j \rightsquigarrow_{\mathsf{N}} \tau'_j$. Lemma 4 provides exactly this information, so we get $\sigma_2[\overline{\tau/\alpha}] \rightsquigarrow_{\mathsf{N}} \sigma_2[\overline{\tau'/\alpha}]$. We are done by RED_APP.

Clause (2): We know $\overline{\tau \rightsquigarrow_{\mathsf{N}} \tau'}$ and need to show that $(\sigma_1\,\sigma_2)[\overline{\tau/\alpha}] \rightsquigarrow_{\mathsf{N}} (\sigma_1\,\sigma_2)[\overline{\tau'/\alpha}]$. We simply use induction to get:

$$\sigma_1[\overline{\tau/\alpha}] \rightsquigarrow_{\mathsf{N}} \sigma_1[\overline{\tau'/\alpha}]$$
$$\sigma_2[\overline{\tau/\alpha}] \rightsquigarrow_{\mathsf{N}} \sigma_2[\overline{\tau'/\alpha}]$$

We are done by RED_APP.

Case $\sigma = H$: We are done by RED_REFL.

Case $\sigma = \forall\,\beta{:}\kappa.\sigma'$: We assume that we have renamed variables so that $\beta \notin \overline{\alpha}$. We see that inverting $\Omega \vdash \forall\,\beta{:}\kappa.\sigma' : \mathsf{R}$ gives us $\Omega, \beta{:}\mathsf{N} \vdash \sigma' : \mathsf{R}$, where $\overline{\alpha}, \beta$ are the free variables in σ'. We can then use the induction hypothesis and we are done by RED_FORALL.

Case $\sigma = F(\overline{\sigma})$: Inversion on $\Omega \vdash F(\overline{\sigma}) : \mathsf{R}$ gives us $\overline{\Omega \vdash \sigma : \mathsf{N}}$. We can then apply Lemma 4 to see that $\overline{\rho = \mathsf{N}}$. We then use the induction hypothesis repeatedly to get

$$\overline{\sigma[\overline{\tau/\alpha}] \rightsquigarrow_{\mathsf{N}} \sigma[\overline{\tau'/\alpha}]}$$

We are now done by RED_TYFAM.

□

Lemma 27 (Sub-roling in the rewrite relation)**.** *If $\tau_1 \rightsquigarrow_{\mathsf{N}} \tau_2$, then $\tau_1 \rightsquigarrow_\rho \tau_2$.*

Proof. By straightforward induction on $\tau_1 \rightsquigarrow_{\mathsf{N}} \tau_2$. □

Lemma 28 (RED_APP/RED_TYCONAPP)**.** *If $H\,\overline{\tau}\,\tau' \rightsquigarrow_{\mathsf{R}} H\,\overline{\sigma}\,\sigma'$ by* RED_APP, *the length of $\overline{\tau}$ is less than the length of roles(H), then $H\,\overline{\tau}\,\tau' \rightsquigarrow_{\mathsf{R}} H\,\overline{\sigma}\,\sigma'$ also by* RED_TYCONAPP.

Proof. Fix H. We then proceed by induction on the length of $\overline{\tau}$.

Base case ($H\,\tau' \rightsquigarrow_{\mathsf{R}} H\,\sigma'$): The premises of RED_APP give us $H \rightsquigarrow_{\mathsf{R}} H$ and $\tau' \rightsquigarrow_{\mathsf{N}} \sigma'$. Regardless of *roles*$(H)$, we can use the sub-roling lemma (Lemma 27) to show $\tau' \rightsquigarrow_\rho \sigma'$ and we are done. (In the case where *roles*(H) is empty, an assumption is violated, and we are done anyway.)

Inductive case: Our inductive hypothesis says: if $H\,\overline{\tau} \rightsquigarrow_{\mathsf{R}} H\,\overline{\sigma}$ and $\tau' \rightsquigarrow_{\mathsf{N}} \sigma'$ (and the length of *roles*(H) is sufficient), then $\overline{\tau \rightsquigarrow_\rho \sigma}$ and $\tau' \rightsquigarrow_{\rho_i} \sigma'$, where i = (length of $\overline{\tau}$) + 1. We must show that, if $H\,\overline{\tau}\,\tau' \rightsquigarrow_{\mathsf{R}} H\,\overline{\sigma}\,\sigma'$ and $\tau'' \rightsquigarrow_{\mathsf{N}} \sigma''$ (and the length of *roles*(H) is sufficient) then $\overline{\tau \rightsquigarrow_\rho \sigma}$, $\tau' \rightsquigarrow_{\rho_i} \sigma'$, and $\tau'' \rightsquigarrow_{\rho_j} \sigma''$ (where $j = i + 1$).

Inverting $H\,\overline{\tau}\,\tau' \rightsquigarrow_{\mathsf{R}} H\,\overline{\sigma}\,\sigma'$ gives us several possibilities:

Case RED_REFL: We get $\overline{\tau \rightsquigarrow_\rho \sigma}$ and $\tau' \rightsquigarrow_{\rho_i} \sigma'$ by RED_REFL. We get $\tau'' \rightsquigarrow_{\rho_j} \sigma''$ by Lemma 27.

Case RED_APP: We get our first two desiderata from use of the induction hypothesis and our last from Lemma 27.

Case RED_TYCONAPP: Our first two desiderata come from the premises of RED_TYCONAPP, and the last one comes from Lemma 27.

Case RED_AXIOM: This case is impossible, because there can be only one newtype axiom for a newtype, and its arity is greater than (length of $\overline{\tau}$) + 1.

□

Lemma 29 (Pattern)**.** *Let $\overline{\alpha}$ be the free variables in a a type τ. We require that each variable α is mentioned exactly once in τ and that no type families appear in τ. Then, if, for some $\overline{\sigma}$, $\tau[\overline{\sigma/\alpha}] \rightsquigarrow_{\mathsf{N}} \tau'$, then there exist $\overline{\sigma}'$ such that $\tau' = \tau[\overline{\sigma'/\alpha}]$ and $\overline{\sigma \rightsquigarrow_{\mathsf{N}} \sigma'}$.*

Proof. We proceed by induction on the structure of τ.

Case $\tau = \alpha$: There is just one free variable (α), and thus just one type σ. We have $\sigma \rightsquigarrow_{\mathsf{N}} \tau'$. Let $\sigma' = \tau'$ and we are done.

Case $\tau = \tau_1\,\tau_2$**:** Partition the free variables into a list $\overline{\beta_1}$ that appear in τ_1 and $\overline{\beta_2}$ that appear in τ_2. This partition must be possible by assumption. Similarly, partition $\overline{\sigma}$ into $\overline{\sigma}_1$ and $\overline{\sigma}_2$. We can see that $\tau_1[\overline{\sigma_1/\beta_1}]\,\tau_2[\overline{\sigma_2/\beta_2}] \rightsquigarrow_{\mathsf{N}} \tau'$. Thus must be by RED_APP (noting that all newtype axioms are at role R). Thus, $\tau' = \tau_1'\,\tau_2'$ and $\tau_1[\overline{\sigma_1/\beta_1}] \rightsquigarrow_{\mathsf{N}} \tau_1'$ and $\tau_2[\overline{\sigma_2/\beta_2}] \rightsquigarrow_{\mathsf{N}} \tau_2'$. We then use the induction hypothesis to get $\overline{\sigma}_1'$ and $\overline{\sigma}_2'$ such that $\tau_1' = \tau_1[\overline{\sigma_1'/\beta_1}]$ and $\tau_2' = \tau_2[\overline{\sigma_2'/\beta_2}]$. We conclude that $\overline{\sigma}'$ is the combination of $\overline{\sigma}_1'$ and $\overline{\sigma}_2'$, undoing the partition done earlier.

Case $\tau = H$**:** Trivial.

Case $\tau = \forall\,\beta{:}\kappa.\tau_0$**:** We first note that, according to the definition of $\overline{\alpha}$, $\beta \notin \overline{\alpha}$. We wish to use the induction hypothesis, but we must be careful because τ_0 may mention β multiple times. So, we linearise τ_0 into τ_0', replacing every occurrence of β with fresh variables $\overline{\beta'}$. (Note that $\overline{\beta'}$ can be empty.) We know that $(\forall\,\beta{:}\kappa.\tau_0)[\overline{\sigma/\alpha}] \rightsquigarrow_{\mathsf{N}} \tau'$. We note that $(\forall\,\beta{:}\kappa.\tau_0)[\overline{\sigma/\alpha}] = \forall\,\beta{:}\kappa.(\tau_0[\overline{\sigma/\alpha}]) = \forall\,\beta{:}\kappa.(\tau_0'[\overline{\sigma/\alpha}][\overline{\beta/\beta'}])$. (We have abused notation somewhat in the second substitution. There is only one β; it is substituted for every variable in $\overline{\beta'}$.) Let $\overline{\sigma}''$ be $\overline{\sigma}$ appended with the right number of copies of β. Let $\overline{\alpha'}$ be $\overline{\alpha}$ appended with $\overline{\beta'}$. Then, we can say $\forall\,\beta{:}\kappa.(\tau_0'[\overline{\sigma''/\alpha'}]) \rightsquigarrow_{\mathsf{N}} \tau'$. We invert to get that $\tau' = \forall\,\beta{:}\kappa.\tau''$ and $\tau_0'[\overline{\sigma''/\alpha'}] \rightsquigarrow_{\mathsf{N}} \tau''$. We can now use the induction hypothesis to get $\overline{\sigma}'''$ such that $\tau' = \tau[\overline{\sigma'''/\alpha'}]$ and $\overline{\sigma'' \rightsquigarrow_{\mathsf{N}} \sigma'''}$. But, we can see that, β steps only to itself. Thus, the last entries in $\overline{\sigma}'''$ must be the same list of βs that $\overline{\sigma}''$ has. We let σ' be the prefix of $\overline{\sigma}'''$ without the βs, and we are done.

Case $\tau = F(\overline{\tau})$**:** Impossible, by assumption.

□

Lemma 30 (Patterns). *Let $\overline{\alpha}$ be the free variables in a list of types $\overline{\tau}$. Assume each variable α is mentioned exactly once in $\overline{\tau}$ and that no type families appear in $\overline{\tau}$. If, for some $\overline{\sigma}$, $\overline{\tau[\overline{\sigma/\alpha}] \rightsquigarrow_{\mathsf{N}} \tau'}$, then there exist $\overline{\sigma}'$ such that $\overline{\tau' = \tau[\overline{\sigma'/\alpha}]}$ and $\overline{\sigma \rightsquigarrow_{\mathsf{N}} \sigma'}$.*

Proof. By induction on the length of $\overline{\tau}$.

Base case: Trivial.

Inductive case: We partition and recombine variables as in the $\tau_1\,\tau_2$ case in the previous proof and proceed by induction.

□

Lemma 31 (Local diamond). *If $\tau \rightsquigarrow_\rho \sigma_1$ and $\tau \rightsquigarrow_\rho \sigma_2$, then there exists σ_3 such that $\sigma_1 \rightsquigarrow_\rho \sigma_3$ and $\sigma_2 \rightsquigarrow_\rho \sigma_3$.*

Proof. If $\rho = \mathsf{P}$, then the result is trivial, by RED_PHANTOM. So, we assume $\rho \neq \mathsf{P}$.

If $\sigma_1 = \tau$ or $\sigma_2 = \tau$, the result is trivial. So, we assume that neither reduction is by RED_REFL.

By induction on the structure of τ:

Case $\tau = \alpha$**:** We note that the left-hand side of an axiom can never be a bare variable, and so the only possibility of stepping is by RED_REFL. We are done.

Case $\tau = \tau_1\,\tau_2$**:** Suppose $\rho = \mathsf{N}$. All axioms at nominal role have a type family application on their left-hand side, so RED_AXIOM cannot apply. Thus, only RED_APP can be used, and we are done by induction.

Now, we can assume $\rho = \mathsf{R}$. If $\tau_1\,\tau_2$ cannot be rewritten as $H\,\overline{\tau}$ (for some H and some $\overline{\tau}$), then the only applicable rule is RED_APP (noting that relevant axiom left-hand sides can indeed be written as $H\,\overline{\tau}$) and we are done by induction.

So, we now rewrite τ as $H\,\overline{\tau}_0$. There are six possible choices of the two reductions, among RED_APP, RED_TYCONAPP, and RED_AXIOM. We handle each case separately:

Case RED_APP/RED_APP: We are done by induction.

Case RED_APP/RED_TYCONAPP: We apply Lemma 28 and finish by induction.

Case RED_APP/RED_AXIOM: Rewrite $\sigma_1 = \sigma_{11}\,\sigma_{12}$. We know then that $\tau_1 \rightsquigarrow_{\mathsf{R}} \sigma_{11}$ and $\tau_2 \rightsquigarrow_{\mathsf{N}} \sigma_{12}$. (Recall that $\tau_1\,\tau_2 = \tau = H\,\overline{\tau}_0$.) We also know that $H\,\overline{\tau}_0 \rightsquigarrow_{\mathsf{R}} \sigma_2$ by a newtype axiom $C : [\overline{\alpha{:}\kappa}].H\,\overline{\alpha} \sim_{\mathsf{R}} \sigma_0$, where $\sigma_2 = \sigma_0[\overline{\tau_0/\alpha}]$.

By induction we can discover that σ_{11} has the form $H\,\overline{\sigma}$ – we know that τ_1 cannot reduce by RED_AXIOM because the well-formedness of the global context says that newtype axioms are unique, and the axiom used on τ has a higher arity than any axiom that could be used on τ_1. Thus, $\sigma_1 = H\,\overline{\sigma}\,\sigma_{12}$. The same axiom C applies here. Let $\overline{\sigma}' = \overline{\sigma}, \sigma_{12}$. So, we can step σ_1 to $\sigma_3 = \sigma_0[\overline{\sigma'/\alpha}]$ by RED_AXIOM.

Now, we must show $\sigma_2 \rightsquigarrow_{\mathsf{R}} \sigma_3$. We wish to apply the rewrite-substitution lemma (Lemma 26). We must show that $\overline{\tau_0 \rightsquigarrow_\rho \sigma'}$, where $\overline{\alpha{:}\rho} \vdash \sigma_0 : \mathsf{R}$. This last fact is exactly what appears in the premise to ROLES_NEWTYPE (which, in turn, is guaranteed by the well-formedness of the global context). Now, we know $\tau = H\,\overline{\tau}_0$ and $\sigma_1 = H\,\overline{\sigma}'$, and that $\tau \rightsquigarrow_{\mathsf{R}} \sigma_1$ by RED_APP. We also know that an axiom is applicable to τ. Thus, the length of $\overline{\tau}$ must be the length of *roles*(H), by context well-formedness. So, we can use Lemma 28 to get $\overline{\tau_0 \rightsquigarrow_\rho \sigma'}$, as desired. We then apply Lemma 26 to conclude $\sigma_2 \rightsquigarrow_{\mathsf{R}} \sigma_3$, and we are done.

Case RED_TYCONAPP/RED_TYCONAPP: We are done by induction.

Case RED_TYCONAPP/RED_AXIOM: We see that $\sigma_1 = H\,\overline{\sigma}'$ where $\overline{\rho}$ is a prefix of *roles*(H) and $\overline{\tau_0 \rightsquigarrow_\rho \sigma'}$. We also see that $C : [\overline{\alpha{:}\kappa}].H\,\overline{\alpha} \sim_{\mathsf{R}} \sigma_0$ and that $\sigma_2 = \sigma_0[\overline{\tau_0/\alpha}]$.

Let $\sigma_3 = \sigma_0[\overline{\sigma'/\alpha}]$. We can see that $\sigma_1 \rightsquigarrow_{\mathsf{R}} \sigma_3$ by RED_AXIOM. And, by Lemma 26 (the rewrite-substitution lemma), we see that $\sigma_2 \rightsquigarrow_{\mathsf{R}} \sigma_3$. So, we are done.

Case RED_AXIOM/RED_AXIOM: Consider the possibility that the two reductions are by different axioms. This would violate context well-formedness, so it is impossible. Thus, we can assume that the axiom used in both reductions is the same: $C : [\overline{\alpha{:}\kappa}].H\,\overline{\alpha} \sim_{\mathsf{R}} \sigma_0$. The only way that σ_1 and σ_2 can be different is if the types substituted in the rule conclusion ($\overline{\sigma}$) are different in the two different reductions. Suppose then that we have $\overline{\sigma}$ and $\overline{\sigma}'$ so that $\sigma_1 = \sigma_0[\overline{\sigma/\alpha}]$ and $\sigma_2 = \sigma_0[\overline{\sigma'/\alpha}]$. It must be that $\tau = H\,\overline{\sigma}$ and that $\tau = H\,\overline{\sigma}'$. But, this tells us that $\overline{\sigma} = \overline{\sigma}'$ and thus that $\sigma_1 = \sigma_2$. We are done.

Case $\tau = H$**:** The only non-trivial step H can make is by RED_AXIOM. However, given that only one axiom for a newtype can exist, both steps must step to the same type, so we are done.

Case $\tau = \forall\,\alpha{:}\kappa.\tau'$**:** We are done by induction.

Case $\tau = F(\overline{\tau})$**:** Here, two rules may apply. We handle the different possibilities separately:

Case Red_TyFam/Red_TyFam: We are done by induction.

Case Red_TyFam/Red_Axiom: Here, we know that $\sigma_1 = F(\overline{\sigma})$ where $\overline{\tau \rightsquigarrow_{\mathsf{N}} \sigma}$, and that $\sigma_2 = \sigma_0[\overline{\sigma'/\alpha}]$ where $C : [\overline{\alpha{:}\kappa}].F(\overline{\tau'}) \sim_{\mathsf{N}} \sigma_0$ and $\overline{\tau} = \overline{\tau'}[\overline{\sigma'/\alpha}]$.
We wish to use Red_Axiom to reduce $F(\overline{\sigma})$. We apply Lemma 30 to get $\overline{\sigma''}$ such that $\overline{\sigma} = \overline{\tau'}[\overline{\sigma''/\alpha}]$ and $\overline{\sigma' \rightsquigarrow_{\mathsf{N}} \sigma''}$. We then use Red_Axiom to get $\sigma_1 \rightsquigarrow_{\mathsf{N}} \sigma_3$, where $\sigma_3 = \sigma_0[\overline{\sigma''/\alpha}]$. Now, we must show that $\sigma_2 \rightsquigarrow_{\mathsf{N}} \sigma_3$. This comes directly from Lemma 26, and we are done.

Case Red_Axiom/Red_Axiom:
We have $C_1 : [\overline{\alpha{:}\kappa}].F(\overline{\tau}_1) \sim_{\mathsf{N}} \sigma'_1$ and $C_2 : [\overline{\beta{:}\kappa'}].F(\overline{\tau}_2) \sim_{\mathsf{N}} \sigma'_2$. We also know that $\tau = F(\overline{\tau}_1)[\overline{\sigma'/\alpha}]$ and $\tau = F(\overline{\tau}_2)[\overline{\sigma''/\beta}]$. Thus, $F(\overline{\tau}_1)[\overline{\sigma'/\alpha}] = F(\overline{\tau}_2)[\overline{\sigma''/\beta}]$. Thus, $[\overline{\sigma'}, \overline{\sigma''}/\overline{\alpha}, \overline{\beta}]$ is a unifier for $F(\overline{\tau}_1)$ and $F(\overline{\tau}_2)$. Thus, by context well-formedness, we have $\sigma'_1[\overline{\sigma'/\alpha}] = \sigma'_2[\overline{\sigma''/\beta}]$. But, $\sigma_1 = \sigma'_1[\overline{\sigma'/\alpha}]$ and $\sigma_2 = \sigma'_2[\overline{\sigma''/\beta}]$, and so $\sigma_1 = \sigma_2$ and we are done.

□

Let the notation $\tau_1 \Leftrightarrow_\rho \tau_2$ mean that there exists a σ such that $\tau_1 \rightsquigarrow^*_\rho \sigma$ and $\tau_2 \rightsquigarrow^*_\rho \sigma$.

Lemma 32 (Confluence)**.** *The rewrite relation $\rightsquigarrow_\rho$ is confluent. That is, if $\tau \rightsquigarrow^*_\rho \sigma_1$ and $\tau \rightsquigarrow^*_\rho \sigma_2$, then $\sigma_1 \Leftrightarrow_\rho \sigma_2$.*

Proof. Confluence is a consequence of the local diamond property, Lemma 31. □

Lemma 33 (Stepping preserves value type heads)**.** *If τ_1 is a value type and $\tau_1 \rightsquigarrow_{\mathsf{R}} \tau_2$, then τ_2 has the same head as τ_1.*

Proof. By induction, noting that the left-hand side of well-formed axioms are never value types. □

Lemma 34 (Rewrite relation consistency)**.** *If $\tau_1 \Leftrightarrow_{\mathsf{R}} \tau_2$, then τ_1 and τ_2 are consistent.*

Proof. If either τ_1 or τ_2 is not a value type, then we are trivially done. So, we assume τ_1 and τ_2 are value types. By assumption, there exists σ such that $\tau_1 \rightsquigarrow^*_{\mathsf{R}} \sigma$ and $\tau_2 \rightsquigarrow^*_{\mathsf{R}} \sigma$. By induction over the length of these reductions and the use of Lemma 33, we can see that σ must have the same head as both τ_1 and τ_2. Thus, τ_1 and τ_2 have the same head, and are thus consistent. □

Lemma 35 (Completeness of the rewrite relation)**.** *If $\varnothing \vdash \gamma : \tau_1 \sim_\rho \tau_2$, then $\tau_1 \Leftrightarrow_\rho \tau_2$.*

Proof. By induction on $\varnothing \vdash \gamma : \tau_1 \sim_\rho \tau_2$.

Case Co_Refl: Trivial, as $\Leftrightarrow_\rho$ is manifestly reflexive.

Case Co_Sym: By induction, as $\Leftrightarrow_\rho$ is manifestly symmetric.

Case Co_Trans: We adopt the variable names in the statement of the rule:

$$\frac{\Gamma \vdash \gamma_1 : \tau_1 \sim_\rho \tau_2 \qquad \Gamma \vdash \gamma_2 : \tau_2 \sim_\rho \tau_3}{\Gamma \vdash \gamma_1 \fatsemi \gamma_2 : \tau_1 \sim_\rho \tau_3} \text{ Co\_Trans}$$

By induction, we know $\tau_1 \Leftrightarrow_\rho \tau_2$ and $\tau_2 \Leftrightarrow_\rho \tau_3$. Thus, we must find σ_{13} such that $\tau_1 \rightsquigarrow^*_\rho \sigma_{13}$ and $\tau_3 \rightsquigarrow^*_\rho \sigma_{13}$. Note that there must be σ_{12} with $\tau_1 \rightsquigarrow^*_\rho \sigma_{12}$ and $\tau_2 \rightsquigarrow^*_\rho \sigma_{12}$, and there must be σ_{23} with $\tau_2 \rightsquigarrow^*_\rho \sigma_{23}$ and $\tau_3 \rightsquigarrow^*_\rho \sigma_{23}$. Thus, we can use Lemma 32 (confluence) to find a σ_{13} such that $\sigma_{12} \rightsquigarrow^*_\rho \sigma_{13}$ and $\sigma_{23} \rightsquigarrow^*_\rho \sigma_{13}$. By transitivity of $\rightsquigarrow^*_\rho$, we are done.

Case Co_TyConApp: We know by induction that $\overline{\tau \Leftrightarrow_\rho \sigma}$. Let the list of common reducts be $\overline{\tau}'$. We can see that $H\,\overline{\tau} \rightsquigarrow^*_{\mathsf{R}} H\,\overline{\tau}'$ by repeated use of Red_TyConApp, and similarly for $H\,\overline{\sigma} \rightsquigarrow^*_{\mathsf{R}} H\,\overline{\tau}'$. Thus $H\,\overline{\tau}'$ is our common reduct and we are done.

Case Co_TyFam: We are done by induction and repeated use of Red_TyFam.

Case Co_App: We are done by induction and repeated use of Red_App.

Case Co_ForAll: We are done by induction and repeated use of Red_ForAll.

Case Co_Phantom: We are done by Red_Phantom.

Case Co_Var: Not possible, as the context is empty.

Case Co_Axiom: We are done by Red_Axiom.

Case Co_Nth: We adopt the variable names in the rule:

$$\frac{\Gamma \vdash \gamma : H\,\overline{\tau} \sim_{\mathsf{R}} H\,\overline{\sigma} \qquad \overline{\rho} \text{ is a prefix of } roles(H) \qquad H \text{ is not a } \textbf{newtype}}{\Gamma \vdash \textbf{nth}^i\,\gamma : \tau_i \sim_{\rho_i} \sigma_i} \text{ Co\_Nth}$$

We know by induction that $H\,\overline{\tau} \Leftrightarrow_{\mathsf{R}} H\,\overline{\sigma}$. In other words, there exists some τ_0 such that $H\,\overline{\tau} \rightsquigarrow^*_{\mathsf{R}} \tau_0$ and $H\,\overline{\sigma} \rightsquigarrow^*_{\mathsf{R}} \tau_0$. We can see by induction on the number of steps in the derivation (and a nested induction in the Red_App case) that τ_0 must have the form $H\,\overline{\tau}'$ for some $\overline{\tau}'$. In particular, note that no axioms can apply because H is not a newtype. Thus, each step is from either Red_App or from Red_TyConApp. However, by Lemma 28, we can consider just the Red_TyConApp case. This says that $\tau_i \rightsquigarrow^*_{\rho_i} \tau'_i$ and $\sigma_i \rightsquigarrow^*_{\rho_i} \tau'_i$, as desired, so we are done.

Case Co_Left: We adopt the variable names from the rule:

$$\frac{\Gamma \vdash \gamma : \tau_1\,\tau_2 \sim_{\mathsf{N}} \sigma_1\,\sigma_2 \qquad \Gamma \vdash \tau_1 : \kappa \qquad \Gamma \vdash \sigma_1 : \kappa}{\Gamma \vdash \textbf{left}\,\gamma : \tau_1 \sim_{\mathsf{N}} \sigma_1} \text{ Co\_Left}$$

We know by induction that $\tau_1\,\tau_2 \Leftrightarrow_{\mathsf{N}} \sigma_1\,\sigma_2$. The steps to reach the common reduct must all be Red_App, because newtype axioms are all at role R. Thus, the common reduct must be $\tau'_1\,\tau'_2$ where $\tau_1 \rightsquigarrow^*_{\mathsf{N}} \tau'_1$, and $\sigma_1 \rightsquigarrow^*_{\mathsf{N}} \tau'_1$, so we are done.

Case Co_Right: Similar to previous case.

Case Co_Inst: We adopt the variable names from the rule:

$$\frac{\Gamma \vdash \gamma : \forall\,\alpha{:}\kappa.\tau_1 \sim_\rho \forall\,\alpha{:}\kappa.\sigma_1 \qquad \Gamma \vdash \tau : \kappa}{\Gamma \vdash \gamma@\tau : \tau_1[\tau/\alpha] \sim_\rho \sigma_1[\tau/\alpha]} \text{ Co\_Inst}$$

We know by induction that $\forall\,\alpha{:}\kappa.\tau_1 \Leftrightarrow_\rho \forall\,\alpha{:}\kappa.\sigma_1$. We can easily see by inspection of the rewrite relation that the common reduct must have the form $\forall\,\alpha{:}\kappa.\tau_0$ for some τ_0. We can also see by a straightforward induction that $\tau_1 \rightsquigarrow^*_\rho \tau_0$ and $\sigma_1 \rightsquigarrow^*_\rho \tau_0$. We must show that $\tau_1[\tau/\alpha] \rightsquigarrow^*_\rho \tau_0[\tau/\alpha]$ and $\sigma_1[\tau/\alpha] \rightsquigarrow^*_\rho \tau_0[\tau/\alpha]$. These facts come from an induction over the lengths of the derivations and the use of the simple rewrite substitution lemma, Lemma 25.

Case Co_Sub: We adopt the variable names in the rule:

$$\frac{\Gamma \vdash \gamma : \tau \sim_{\mathsf{N}} \sigma}{\Gamma \vdash \textbf{sub}\,\gamma : \tau \sim_{\mathsf{R}} \sigma} \text{ Co\_Sub}$$

We know that $\tau \Leftrightarrow_{\mathsf{N}} \sigma$ and we need $\tau \Leftrightarrow_{\mathsf{R}} \sigma$. This follows by induction over the lengths of the reduction and the use of Lemma 27.

□

Lemma 36 (Consistency). *The global context is consistent.*

Proof. Take a γ such that $\varnothing \vdash \gamma : \tau_1 \sim_{\mathsf{R}} \tau_2$. By the completeness of the rewrite relation (Lemma 35), we see that $\tau_1 \Leftrightarrow_{\mathsf{R}} \tau_2$. But, the rewrite relation consistency lemma (Lemma 34) tells us that τ_1 and τ_2 are consistent. Thus, the context admits only consistent coercions and is itself consistent. □

H.2 Progress

Lemma 37 (Canonical forms).

1. *If $\varnothing \vdash v : \tau_1 \to \tau_2$, then v is either $\lambda x{:}\tau.e'$ or $K\,\overline{\tau}\,\overline{\gamma}\,\overline{e}$.*
2. *If $\varnothing \vdash v : \forall\,\alpha{:}\kappa.\tau$, then v is either $\Lambda\alpha{:}\kappa.e'$ or $K\,\overline{\tau}$.*
3. *If $\varnothing \vdash v : \phi \Rightarrow \tau$, then v is either $\lambda c{:}\phi.e'$ or $K\,\overline{\tau}\,\overline{\gamma}$.*
4. *If $\varnothing \vdash v : D\,\overline{\sigma}$, then v is $K\,\overline{\tau}\,\overline{\gamma}\,\overline{e}$.*

Theorem 38 (Progress). *If $\varnothing \vdash e : \tau$, then either e is a value or $e \longrightarrow e'$ for some e'.*

Proof. We proceed by induction on the typing judgement $\varnothing \vdash e : \tau$.

Case TM_VAR: Cannot happen in an empty context.
Abstraction forms: Trivial.
Case TM_APP: We know $e = e_1\,e_2$. By induction, we know that e_1 is either a value or steps to e_1'. If e_1 steps, we are done by S_APP_CONG. If e_1 is a value, the canonical forms lemma now gives us several cases:
Case $e_1 = \lambda x{:}\tau.e_3$**:** We are done by S_BETA.
Case $e_1 = K\,\overline{\tau}\,\overline{\gamma}\,\overline{e}$**:** Then, $e_1\,e_2$ is a value.
Case TM_TAPP: Similar to previous case.
Case TM_CAPP: Similar to previous case.
Case TM_DATACON: e is a value.
Case TM_CASE: We adopt the variable names from the rule:

$$\frac{\begin{array}{l}\Gamma \vdash e : D\,\overline{\sigma}\\ \Gamma \vdash \tau : \star\\ \forall\, alt_i \text{ s.t. } alt_i \in \overline{alt} :\\ \quad alt_i = K_i\,\overline{\alpha_i}\,\overline{c_i}\,\overline{x_i} \to e_i\\ \quad K_i : \forall\,\overline{\alpha_i'{:}\kappa_i}.\forall\,\overline{\beta_i'{:}\kappa_i'}.\overline{\phi_i} \Rightarrow \overline{\tau_i} \to D\,\overline{\alpha_i'}\\ \quad \Gamma, \overline{\alpha_i{:}\kappa_i'}, (\overline{c_i{:}\phi_i}, \overline{x_i{:}\tau_i})[\overline{\sigma/\alpha_i'}][\overline{\alpha_i/\beta_i'}] \vdash e_i : \tau\\ \overline{alt} \text{ is exhaustive}\end{array}}{\Gamma \vdash \mathbf{case}_\tau\ e\ \mathbf{of}\ \overline{alt} : \tau}\ \textsc{Tm\_Case}$$

We know by induction that e is a value or $e \longrightarrow e'$ for some e'. If e steps, then we are done by S_CASE_CONG. Thus, we assume that e is a value. By the canonical forms lemma, we see that $e = K\,\overline{\tau}\,\overline{\gamma}\,\overline{e}$. Thus, S_IOTA applies, noting that the match must be exhaustive.
Case TM_CAST: We adopt the variable names from the rule:

$$\frac{\begin{array}{l}\Gamma \vdash e : \tau_1\\ \Gamma \vdash \gamma : \tau_1 \sim_{\mathsf{R}} \tau_2\end{array}}{\Gamma \vdash e \triangleright \gamma : \tau_2}\ \textsc{Tm\_Cast}$$

By induction, we know that e is a value or $e \longrightarrow e'$.
If e steps, we are done by S_CAST_CONG.
If e is a value, we have four cases:
Case $e = \lambda x{:}\tau.e_1$**:** We wish to use S_PUSH, but we must establish that $\varnothing \vdash \gamma : (\tau \to \sigma) \sim_{\mathsf{R}} (\tau' \to \sigma')$. We know from the premises of TM_CAST that $\varnothing \vdash \gamma : \tau_1 \sim_{\mathsf{R}} \tau_2$, so must simply show that $\tau_1 = \tau \to \sigma$ and $\tau_2 = \tau' \to \sigma'$. The first fact comes from the fact $\varnothing \vdash e : \tau_1$ and that e is a λ-term. The second comes from consistency (Lemma 36), to show that τ_2 is headed by $(\to)$, and by homogeneity (Lemma 19) to show that it is fully applied.
Case $e = \Lambda\alpha{:}\kappa.e_1$**:** We are done by S_TPUSH.
Case $e = \lambda c{:}\phi.e_1$**:** Similar to the $\lambda x{:}\tau.e_1$ case, but using S_CPUSH.
Case $e = K\,\overline{\tau}\,\overline{\gamma}\,\overline{e}$**:** We use S_KPUSH, once again using consistency and homogeneity to show the premises of that rule (along with substitution to type the $\overline{\gamma}$).
Case TM_CONTRA: We adopt the variable names from the rule:

$$\frac{\varnothing \vdash \gamma : H_1 \sim_{\mathsf{N}} H_2 \qquad H_1 \neq H_2 \qquad \Gamma \vdash \tau : \star}{\Gamma \vdash \mathbf{contra}\,\gamma\,\tau : \tau}\ \textsc{Tm\_Contra}$$

By completeness of the rewrite relation (Lemma 35), we know that $H_1 \Leftrightarrow_{\mathsf{N}} H_2$. But, if $H \rightsquigarrow_{\mathsf{N}} H'$, then $H = H'$ (by induction on $H \rightsquigarrow_{\mathsf{N}} H'$, noting that all newtype axioms are at role R). So $H_1 = H_2$, contradicting a premise to this rule. Thus, this case cannot happen.

□

I. Role inference

Lemma 39 (Walking). *Let $\overline{\alpha}$ be the parameters to some type constant T. For some type σ, let $\overline{\beta}$ be the free variables in σ that are not in $\overline{\alpha}$. Let $\overline{\rho}$ be a list of roles of the same length as $\overline{\alpha}$. Let $\Omega = \overline{\alpha{:}\rho}, \overline{\beta{:}\mathsf{N}}$.*

If $\mathsf{walk}(T, \sigma)$ makes no change to the role of any of the $\overline{\alpha}$, then $\Omega \vdash \sigma : \mathsf{R}$.

Proof. By induction on the structure of σ:

Case $\sigma = \alpha'$**:** By assumption, it must be that $\alpha'{:}\mathsf{R} \in \Omega$ or $\alpha'{:}\mathsf{N} \in \Omega$. In either case, we can derive $\Omega \vdash \alpha' : \mathsf{R}$, so we are done.
Case $\sigma = \sigma_1\,\sigma_2$**:** We check if σ can also be written as $H'\,\overline{\tau}$.
Case $\sigma = H'\,\overline{\tau}$**:** Let $\overline{\rho}' = roles(H')$. In order to conclude $\Omega \vdash H'\,\overline{\tau} : \mathsf{R}$, we will show that $\overline{\Omega \vdash \tau : \rho'}$. Fix i; we will show $\Omega \vdash \tau_i : \rho_i'$. Here, we have three cases:
Case $\rho_i' = \mathsf{N}$**:** By assumption, it must be that all the free variables in τ_i are assigned to N in Ω. Thus, by Lemma 4, we have $\Omega \vdash \tau_i : \mathsf{N}$ and we are done.
Case $\rho_i' = \mathsf{R}$**:** By assumption, it must be that $\mathsf{walk}(T, \tau_i)$ makes no change. We then use the induction hypothesis to say that $\Omega \vdash \tau_i : \mathsf{R}$, and we are done.
Case $\rho_i' = \mathsf{P}$**:** We are done by RTY_PHANTOM.
Other applications: We wish to use RTY_APP. Thus, we must show that $\Omega \vdash \sigma_1 : \mathsf{R}$ and $\Omega \vdash \sigma_2 : \mathsf{N}$. For the former, we see that $\mathsf{walk}(T, \sigma_1)$ must make no change, and we are done by induction. For the latter, we see that all the free variables in σ_2 must be assigned to N, and we are done by Lemma 4.
Case $\sigma = H$**:** We are done by immediate application of RTY_TYCONAPP.
Case $\sigma = \forall\,\alpha'{:}\kappa.\sigma_1$**:** We are done by induction, noting that in RTY_FORALL, α' gets assigned role N when checking σ_1.

This matches our expectations that the type variables $\overline{\beta}$ are at role N in the inductive hypothesis.

Case $\sigma = F(\overline{\tau})$**:** Repeated use of Lemma 4 tells us that $\overline{\Omega \vdash \tau : \mathsf{N}}$. We are done by RTY_TYFAM.

□

Theorem 40. *The role inference algorithm always terminates.*

Proof. First, we observe that the walk procedure always terminates, as it is structurally recursive.

For the algorithm to loop in step 4, a role assigned to a variable must have changed. Yet, there are a finite number of such variables, and each variable may be updated only at most twice (from P to R and from R to N). Thus, at some point no more updates will happen and the algorithm will terminate. □

Theorem 41 (Role inference is sound)**.** *After running the role inference algorithm, roles$(H) \models H$ will hold for all H.*

Proof. We handle the datatype case first. Fix a D. We will show that $roles(D) \models D$. Because the role inference algorithm has terminated, we know that $\mathsf{walk}(D, \sigma)$ has caused no change for every σ that appears as a coercion type or term-level argument type in a constructor for D. Choose a constructor K, such that

$$K : \forall\, \overline{\alpha{:}\kappa}.\forall\, \overline{\beta{:}\kappa'}.\overline{\phi} \Rightarrow \overline{\sigma} \to D\,\overline{\alpha}$$

Let $\overline{\rho} = roles(D)$ and $\Omega = \overline{\alpha{:}\rho}, \overline{\beta{:}\mathsf{N}}$. We have satisfied the premises of the walking lemma (Lemma 39), and thus we can conclude that $\Omega \vdash \sigma : \mathsf{R}$. We have shown $roles(D) \models D$ by ROLES_DATA.

The newtype case is similar, using the right-hand side of the newtype definition in place of σ. □

Lemma 42 (Stumbling)**.** *Let $\overline{\alpha}$ be the parameters to some type constant T. For some type σ, let $\overline{\beta}$ be the free variables in σ that are not in $\overline{\alpha}$. Let $\overline{\rho}$ be a list of roles of the same length as $\overline{\alpha}$. Let $\Omega = \overline{\alpha{:}\rho}, \overline{\beta{:}\mathsf{N}}$.*

If $\mathsf{walk}(T, \sigma)$ *were modified to skip one of its attempts to mark a variable, then it is not possible to conclude* $\Omega \vdash \sigma : \mathsf{R}$.

Proof. By induction on the structure of σ:

Case $\sigma = \alpha'$**:** If that mark were not done, then Ω would contain $\alpha'{:}\mathsf{P}$; this clearly violates $\Omega \vdash \alpha' : \mathsf{R}$.

Case $\sigma = \sigma_1\,\sigma_2$**:** We check if σ can also be written as $H'\,\overline{\tau}$.

Case $\sigma = H'\,\overline{\tau}$**:** Let $\overline{\rho}' = roles(H')$. Fix i.

Case $\rho'_i = \mathsf{N}$**:** If we do not mark every free variable in τ_i as N, then it would be impossible to conclude $\Omega \vdash \tau_i : \mathsf{N}$, by Lemma 4. Thus, we would not be able to conclude $\Omega \vdash H'\,\overline{\tau} : \mathsf{R}$ by RTY_TYCONAPP. What about by RTY_APP? This, too, would require $\Omega \vdash \tau_i : \mathsf{N}$, which we are unable to do.

Case $\rho'_i = \mathsf{R}$**:** By induction, it is not possible to conclude $\Omega \vdash \tau_i : \mathsf{R}$, and thus impossible to use RTY_TYCONAPP. What about RTY_APP? This would require $\Omega \vdash \tau_i : \mathsf{N}$, which is not possible via the contrapositive of Lemma 5.

Case $\rho'_i = \mathsf{P}$**:** There is no marking to be done here, so the assumption that walk is modified is false.

Other applications: Suppose the skipped marking were in the recursive call. Then, by induction, it is not possible to conclude $\Omega \vdash \sigma_1 : \mathsf{R}$. Thus, it is not possible to conclude $\Omega \vdash \sigma_1\,\sigma_2 : \mathsf{R}$ by RTY_APP.

Now, suppose the skipped marking is when marking all free variables in σ_2 as N. In this case, we know that $\Omega \vdash \sigma_2 : \mathsf{N}$ is impossible (by Lemma 4) and thus we cannot use RTY_APP.

Case $\sigma = H$**:** No mark was skipped, so the assumption that walk is modified is false.

Case $\sigma = \forall\, \alpha'{:}\kappa.\sigma_1$**:** We are done by induction, noting that in RTY_FORALL, α' gets assigned role N when checking σ_1. This matches our expectations that the type variables $\overline{\beta}$ are at role N in the inductive hypothesis.

Case $\sigma = F(\overline{\tau})$**:** If one of the variables free in the $\overline{\tau}$ were not marked as N, then it would be impossible to conclude $\Omega \vdash \tau_i : \mathsf{N}$ for that τ_i (by Lemma 4. Thus, we would be unable to use RTY_TYFAM.

□

Theorem 43 (Role inference is optimal)**.** *After running the role inference algorithm, any loosening of roles (a change from ρ to ρ', where $\rho \leq \rho'$ and $\rho \neq \rho'$) would violate roles$(H) \models H$.*

Proof. Every time the role inference algorithm changes an assigned role from ρ' to ρ, it is the case that $\rho \leq \rho'$ and $\rho \neq \rho'$. Thus, all we must show is that every change the algorithm makes is necessary – that is, not making the change would then violate $roles(H) \models H$.

Role inference runs only on algebraic datatypes, so we need only concern ourselves with Ts, not general Hs. In both the datatype and newtype cases, showing $roles(T) \models T$ requires showing $\Omega \vdash \sigma : \mathsf{R}$, where $\Omega = \overline{\alpha{:}\rho}, \overline{\beta{:}\mathsf{N}}$ and $\overline{\alpha}$ are the parameters to T and $\overline{\beta}$ are the remaining free variables of σ. (In the newtype case, $\overline{\beta}$ is empty.) The list of roles $\overline{\rho}$ is $roles(T)$. So, we must show that skipping any change in the $\mathsf{walk}(T, \sigma)$ algorithm means that $\Omega \vdash \sigma : \mathsf{R}$ would not be derivable. This is precisely what Lemma 42 shows and so we are done. □