Under consideration for publication in J. Functional Programming

Safe Zero-cost Coercions for Haskell

JOACHIM BREITNER
Karlsruhe Institute of Technology
(*e-mail:* breitner@kit.edu)

RICHARD A. EISENBERG
University of Pennsylvania
(*e-mail:* eir@cis.upenn.edu)

SIMON PEYTON JONES
Microsoft Research
(*e-mail:* simonpj@microsoft.com)

STEPHANIE WEIRICH
University of Pennsylvania
(*e-mail:* 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 in 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 representations. 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 (Milner *et al.*, 1997) and Haskell's **newtype** declaration (Marlow (editor), 2010).

For example, a Haskell programmer might create an abstract type for HTML data, representing them as Strings (Figure 1). 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 data constructor MkHTML converts a String to an HTML (see function text), while using MkHTML in a pattern converts in the other direction (see function unHTML). By exporting

```
module Html( HTML, text, unHTML, ... ) where

newtype HTML = MkHTML String

unHTML :: HTML → String
unHTML (MkHTML s) = s

text :: String → HTML
text s = MkHTML (escapeSpecialCharacters s)
```

Figure 1. An abstraction for HTML values

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**s for abstraction in Haskell has always suffered from an embarrassing difficulty. Suppose that in the module Html, the programmer wants to break HTML data into a list of lines, using the standard Haskell library function lines :: String → [String]:

```
linesH :: HTML → [HTML]
linesH h = map MkHTML (lines (unHTML h))
```

To get the resulting [HTML] we are forced to map MkHTML 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 MkHTML constructor changes the type of its operand, and hence cannot be optimised away. There is nothing that the programmer can do to prevent this run-time cost. What has become of the claim of zero-overhead abstraction?

In this paper we describe a robust, simple mechanism that programmers can use to solve this 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 the new constraint Coercible. This function performs a zero-cost conversion between two types a and b that have the same representation. The crucial question becomes *for which types is the Coercible constraint satisfiable?* We describe how the constraint can be formed and used in Section 2.
- We formalise Coercible by translation into GHC's intermediate language System FC, augmented with the concept of *roles* (Section 2.2), adapted from prior work (Weirich *et al.*, 2011). One new contribution of this work is a simplification of the roles idea; we formalise this simpler system and give the usual proofs of preservation and progress in Section 4.

- Adding safe coercions to the source language raises new issues for abstract types, and for the coherence of type elaboration. We articulate the issues, and introduce *role annotations* to solve them (Section 3).
- It would be too onerous to insist on programmer-supplied role annotations for every type, so we give a *role inference algorithm* in Section 4.5.
- The precise algorithm used to simplify and solve Coercible constraints is subtle. It appears in Section 5.
- To support our claim of practical utility, we have implemented the whole scheme in GHC (Section 6), and evaluated it against thousands of Haskell libraries (Section 6.5). Our work also finally resolves a notorious and long-standing bug in GHC (#1496), which concerns the interaction of newtype coercions with type families (Section 6.1).

We build on earlier work on roles (Weirich *et al.*, 2011), which offered a very expressive, but very complicated, system of roles. In this paper we find a sweet spot offering a considerably simpler system in exchange for a minor loss of expressiveness. This article is a revised and expanded version of our ICFP'14 paper (Breitner *et al.*, 2014a), and describes the implementation as it has been refined since the original publication (Section 5).

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 7), 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. His entry point to the story is the function

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

that allows him to convert values between two types a and b, given that the compiler can determine that they are coercible. This relation is expressed with the new constraint Coercible a b, which shares the same syntax as type class constraints.

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 with no runtime cost. And that's it!

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

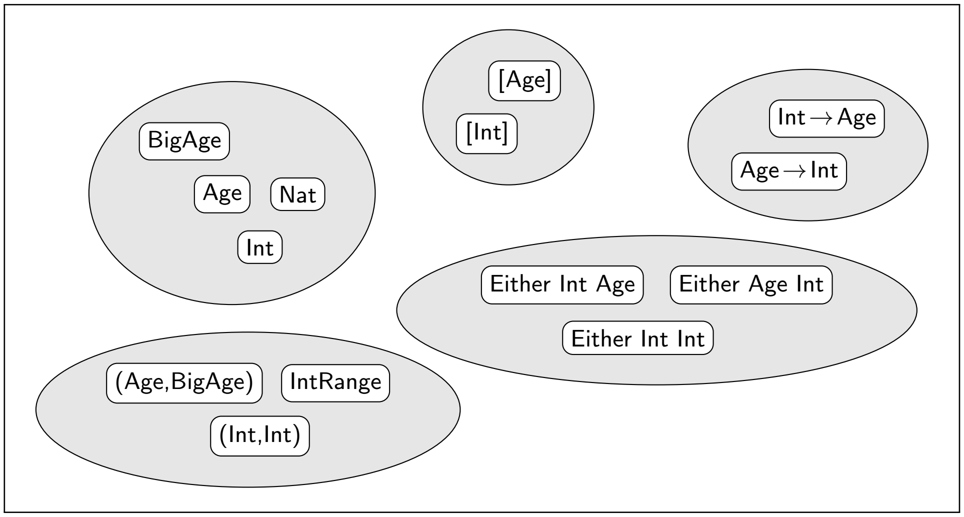

Figure 2. Coercible relates types with identical run-time representation

```
newtype Age      = MkAge Int
newtype BigAge   = MkBig Age
newtype Nat      = MkNat Int
newtype IntRange = MkIR (Int,Int)
```

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, as this corresponds to simply using the MkAge constructor,
- Coercible Age Int: and the reverse, as if we were pattern matching on MkAge.
- Coercible BigAge Int: We can unwrap two steps at once,
- Coercible BigAge Nat: and coerce between different newtypes that happen to have the same representation.
- Coercible [Age] [Int]: We can lift coercions over lists,
- Coercible (Either Int Age) (Either Int Int): and over Either.
- Coercible (Either Int Age) (Either Age Int): It also works if the 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, BigAge) IntRange: And even quite complex coercions like this are handled with one call to coerce.

Figure 2 visualises these coercions and shows that Coercible is constructed to be an equivalence relation: It partitions all Haskell types into equivalence classes, so that in each such class, every type has the same run-time representation, and one can convert between any two types that are in the same group, in either direction, with a single call to coerce; but not between types of different groups.

The rest of this section describes the basic rules for determining when one type is Coercible to another; see Figure 3 for a concise summary. The algorithm used in GHC to actually solve Coercible constraints is described in detail in Section 5.

The most important rules that GHC uses to solve Coercible constraints are as follows (the full rules are given in Figure 5):

(1) The *unwrapping rule*:

- ▶ For every **newtype** NT = MkNT t, we have Coercible t NT if and only if the constructor MkNT is in scope.

(2) The *lifting rule*:

- ▶ For every type constructor TC r p n, where
  - — r stands for TC's parameters at a representational role,
  - — p for those at a phantom role and
  - — n for those at a nominal role,

  if Coercible r1 r2, then Coercible (TC r1 p1 n) (TC r2 p2 n).

(3) Coercible is an equivalence relation:

- ▶ The *reflexivity rule*: Coercible a a.
- ▶ The *symmetry rule*: If Coercible a b then Coercible b a.
- ▶ The *transitivity rule*: If Coercible a b and Coercible b c then Coercible a c.

Figure 3. Coercible formation rules (pragmatic summary)

2.1 *Coercing newtypes*

We expect Coercible to relate a newtype with its base type; this is the most obvious rule for Coercible. In our example, this solves the following constraints:

- Coercible Int Age
- Coercible Age BigAge
- Coercible (Int,Int) IntRange

Notice that each of these rules unwraps just one layer of the newtype, so we call them the *unwrapping rules*.

The newtype-unwrapping rules (i.e., (1) in Figure 3) are available *only if the corresponding newtype data constructor* (e.g. MkNT) *is in scope*; this is required to preserve abstraction, as we explain in Section 3.1.

2.2 *Type constructors and roles*

As Figure 3 shows, as well as the unwrapping rules for a **newtype**, we also have one rule for each type constructor, including data types, newtypes , the function type, and built-in data types like tuples. We call this the *lifting rule* for the type, because it lifts coercions through the type.

The shape of the lifting rule depends on the so-called *roles* of the type constructor's parameters. 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 (Section 4.3). In practice, the roles of a declared data type are determined by a role inference algorithm (Section 4.5) and can be modified by role annotations (Section 3.1). Once defined, the roles of a type constructor are the same in every scope, regardless of whether the concrete definition of that type is available in that scope.

Roles, a development of earlier work (Weirich *et al.*, 2011), are a new concept for the programmer. In the following subsections, we discuss how the three possible roles, *representational*, *phantom* and *nominal*, ensure that lifting rules do not violate type safety by allowing coercions between types with different run-time representations.

2.3 Coercing representational type parameters

The most common role is *representational*. It is the role that is assigned to the type parameters of ordinary newtypes and data types like Maybe, the list type and Either. The Coercible rule for these type constructors are:

- If Coercible a b then Coercible (Maybe a) (Maybe b).
- If Coercible a b then Coercible [a] [b].
- If Coercible a1 b1 and Coercible a2 b2 then Coercible (Either a1 a2) (Either b1 b2).

These rules 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.

Most primitive type constructors also have representational roles for their arguments. For example, the domain and co-domain of arrow types are representational, giving rise to the following Coercible rule:

- If Coercible a1 b1 and Coercible a2 b2 then Coercible (a1 → a2) (b1 → b2).

As another example, the type IORef has a representational parameter, so expressions of type IORef Int can be converted to type IORef Age for zero cost (and outside of the IO monad).

Returning to the introduction, we can now write linesH very directly, thus:

```
linesH :: HTML → [HTML]
linesH = coerce lines
```

In this case, the call to coerce gives rise to a constraint Coercible (String → [String]) (HTML → [HTML]), which gets simplified to Coercible String HTML using the lifting rules for arrow and list types, and then solved by the unwrapping rule for the newtype HTML.

2.4 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 = MkNP [Phantom b] | SomethingElse
```

both have parameter 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 rules

- ▶ Coercible (Phantom a) (Phantom b)
- ▶ Coercible (NestedPhantom a) (NestedPhantom b)

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

2.5 *Coercing nominal type parameters*

In contrast, the *nominal* role induces the strictest preconditions for Coercible rules. 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.[1] 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 allow the constraint Coercible (EncText HTML) (EncText String), because these two types have quite different run-time representations! Therefore, there are no rules that change a nominal parameter of a type constructor.

All parameters of a type or data *family* have nominal role, because they could be inspected by the type family instances. For similar reasons, the non-uniform parameters to GADTs are also required to be nominal. Type classes also use nominal role for their type parameters; see Section 3.2.

2.6 *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)
```

Here r has representational role, n has nominal, while p is phantom. Hence, following (2) in Figure 3, we get:

- ▶ If Coercible r1 r2 then Coercible (Params r1 p1 n) (Params r2 p2 n).

[1] We use the terms “type function” and “type family” (the Haskell-specific term) interchangeably.

2.7 Inverting the lifting rule

For a data type constructor such as Maybe, there is only one rule that concludes that Coercible (Maybe a) (Maybe b), namely the lifting rule. As that rule has the assumption that Coercible a b holds, we can invert that rule and we can conclude Coercible a b from Coercible (Maybe a) (Maybe b). This is the *decomposition rule* (elided from Figure 3), and can be used for any parameter of a non-**newtype** type constructor, e.g.:

- If Coercible (Maybe a) (Maybe b) then Coercible a b.
- If Coercible [a] [b] then Coercible a b.
- If Coercible (Either a1 a2) (Either b1 b2) then Coercible a1 b1.
- If Coercible (Either a1 a2) (Either b1 b2) then Coercible a2 b2.
- If Coercible (a1 → a2) (b1 → b2) then Coercible a1 b1.
- If Coercible (a1 → a2) (b1 → b2) then Coercible a2 b2.

The general rule follows:

- ▶ Suppose non-**newtype** T has parameters with roles **representational**, **phantom**, and **nominal**, respectively. If Coercible (T r1 p1 n1) (T r2 p2 n2), then Coercible r1 r2 and n1 ∼ n2, where ∼ is Haskell's notation for type equality.

Although type constructors created with **newtype** also have lifting rules , Coercible constraints between them also could have been created using the unwrapping rule, so the argument above does not hold. And indeed, if we assume such a decomposition rule, we could derive invalid Coercible constraints. Consider the following code, where the programmer explicitly uses a role annotation (see Section 3.1) to set the role of the argument to **representational**:

```
type role TaggedInt representational
newtype TaggedInt a = MkTI Int
```

Using the unwrapping rule, together with transitivity and symmetry, we can conclude that Coercible (TaggedInt Bool) (TaggedInt Char) holds, so if we had a decomposition rule, we would now have Coercible Bool Char.

2.8 Supporting higher order polymorphism

So far, we have only seen Coercible applied to types of kind $*$, but that is not sufficient to support all coercions that we might want. For example, consider a monad transformer such as

```
data MaybeT m a = MaybeT (m (Maybe a))
```

and a newtype that wraps another monad, e.g.

```
newtype MyIO a = MyIO (IO a)
```

It is reasonable to expect that Coercible (MaybeT MyIO a) (MaybeT IO a) can be derived. Using the lifting rule for MaybeT, this requires Coercible MyIO IO to hold.

Therefore, for a **newtype** declaration as the one above, GHC will η-reduce the unwrapping rule to say Coercible IO MyIO instead of Coercible (IO a) (MyIO a). Using symmetry, this allows us to solve Coercible (MaybeT MyIO a) (MaybeT IO a).

Of course, this η-reduction must not prevent us from solving, for example, Coercible (MyIO Int) (IO Int). Therefore, we have the *type application rule* that allows us to use Coercible relations between types of higher kinds such as $* \rightarrow *$:

- ► If Coercible t1 t2, where t1, t2 :: k1 $\rightarrow$ k2, then Coercible (t1 x) (t2 x).

What about the very similar-looking rule "If Coercible a b then Coercible (t a) (t b)", where t is a type variable (of kind k1 $\rightarrow$ k2)? Such a lifting rule for type variables would be unsound. For example, the variable could be instantiated with a type constructor that has a nominal parameter, such as EncText, which would allow us to coerce (erroneously) between EncText HTML and EncText String.

Therefore, the parameters of type variables are always assumed to have nominal role, and no lifting rule is available. This inability to abstract over types while retaining information about their parameter's roles has some practical consequences; see Section 8.

3 Roles, abstraction, and coherence

The purpose of the HTML type from the introduction is to prevent the confusion of unescaped strings and HTML fragments. However, because these types have the same representation, confusing them does not lead to unsoundness in the type system. Instead, programs that make this mistake do not preserve the user-defined abstraction of the HTML type.

While the previous section describes how the Coercible formation rules ensure that Coercible types share runtime representations, this section discusses two other properties that guide the design of this mechanism: *type abstraction* (Section 3.1) and *class coherence* (Section 3.2).

3.1 Preserving abstraction

Haskell programmers define *abstract types* by hiding the constructors of newtypes and datatypes. In this case, the creation of values of a type like HTML is controlled by a code in a single module, so programmers can establish invariants about those values. Because coerce can also construct values of type HTML, the unwrapping coercion associated with this newtype is available if and only if the newtype constructor MkHTML is in scope.

However, what about the interaction between the *lifting rule* and type abstraction? It turns out that, even when module authors have carefully hidden the constructors of a type, sometimes they want to make the lifting rule available for that type, but at other times they would like to restrict it.

To illustrate the former case, we would like to permit coercions between IORef HTML to IORef String, even though IORef is an abstract type. Similarly, consider a library for non-empty lists:

```
module NonEmptyListLib( NE, singleton, ... ) where

data NE a = MkNE [a]
singleton :: a → NE a
...etc...
```

The type must be exported abstractly; otherwise, the non-empty property can be broken by its users. Nevertheless lifting a coercion through NE, i.e. coercing NE HTML to NE String, does not break this invariant.

To illustrate the case where one would want to restrict the lifting rule, consider the data type Map k v. This type 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 coerce from (Map Age v) to (Map Int v), because a valid tree with regard to the ordering of Age might be bogus when using the ordering of Int. Functions that manipulate Maps use an Ord k constraint and thus use the Ord instance for the type k; nothing in Haskell requires that instances Ord Int and Ord Age behave similarly.

To prevent coercing (Map Age v) to (Map Int v), we allow the programmer to specify a *role annotation*, thus:

```
type role Map nominal representational
```

As explained in Section 2.2, these roles produce the abstraction-preserving lifting rule

- If Coercible a b then Coercible (Map k a) (Map k b)

which allows the coercion from Map k HTML to Map k String.

Note that in the declaration of Map the parameters k and v are used in exactly the same way, so this distinction cannot be made by the compiler; it can only be specified by the programmer. However, the compiler ensures that programmer-specified role annotations cannot violate type safety: if the annotation specifies an unsafe role, the compiler will reject the program.

3.2 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 careful design 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 (MkHTML 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 (MkHTML "Hello")
"HTML:Hello"
λ> showH badShow (MkHTML "Hello")
"Hello"
```

In the last interaction we applied show to a value of type HTML, but the Show instance for String (coerced to (Show HTML)) was used. This example shows the problem that derives from the lack of coherence – we used coerce to construct a second instance of the Show class for the HTML type.

To avoid this confusion, the parameters of a type class are assigned a *nominal* role.[2] 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 actually produce a sound type system? 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. The version of FC that we develop in

[2] A role annotation can be used to override this default, but the user must specify GHC's `-XIncoherentInstances` extension to do so.

Metavariables:

x	term	a,b	type	c	coercion
C	axiom	D	data type	N	newtype
F	type family	K	data constructor		

e	$::= \lambda c{:}\phi.e \mid e\,\gamma \mid e \triangleright \gamma \mid \cdots$	terms
τ,σ	$::= a \mid \tau_1\,\tau_2 \mid \forall a{:}\kappa.\ \tau \mid H \mid F(\overline{\tau})$	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 data types
ϕ	$::= \tau \sim^{\kappa}_{\rho} \sigma$	propositions
γ,η	$::=$	coercions
	$\mid \langle\tau\rangle \mid \langle\tau,\sigma\rangle_{\mathsf{P}} \mid \mathbf{sym}\,\gamma \mid \gamma_1 \fatsemi \gamma_2$	equivalence
	$\mid H(\overline{\gamma}) \mid F(\overline{\gamma}) \mid \gamma_1\,\gamma_2 \mid \forall a{:}\kappa.\ \gamma$	congruence
	$\mid c \mid C(\overline{\tau})$	assumption
	$\mid \mathbf{nth}^{i}\,\gamma \mid \mathbf{left}\,\gamma \mid \mathbf{right}\,\gamma \mid \gamma@\tau$	decomposition
	$\mid \mathbf{sub}\,\gamma$	sub-roling
ρ	$::= \mathsf{N} \mid \mathsf{R} \mid \mathsf{P}$	roles
Γ	$::= \varnothing \mid \Gamma,a{:}\kappa \mid \Gamma,c{:}\phi \mid \Gamma,x{:}\tau$	typing contexts
Ω	$::= \varnothing \mid \Omega,a{:}\rho$	role contexts

Figure 4. An excerpt of the grammar of System FC

this paper derives from much prior work.[3] However, for clarity we give a self-contained description of the system and do not assume familiarity with previous versions.

Figure 4 gives the syntax of System FC. The starting point is a conventional typed, polymorphic lambda calculus inspired by System F. We therefore elide most of the syntax of terms e, giving the typing judgement for terms in Appendix A.2. Types τ are also conventional, except that we add (saturated) type-family applications $F(\overline{\tau})$, to reflect their addition to source Haskell (Chakravarty *et al.*, 2005a; Chakravarty *et al.*, 2005b). Types are classified by kinds κ as usual; the kinding judgement $\Gamma \vdash \tau : \kappa$ appears in Appendix A.2. This judgement is syntax directed: from the context Γ and type τ, we can determine the unique kind κ (if one exists). To avoid clutter we use only monomorphic kinds, but it is easy to add kind polymorphism along the lines of Yorgey *et al.* (2012), and our implementation does so.

[3] 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".

FC is an *explicitly-typed* language. By using System F's explicit type abstraction and application, an FC program can by typechecked by a simple, syntax-directed algorithm, despite the presence of impredicative polymorphism. Type inference is not required.

4.1 Roles and casts

FC's distinctive feature is a type-safe cast $(e \triangleright \gamma)$ (Figure 4), which uses a *coercion* γ to cast a term from one type to another. The explicit coercions and casts in System FC ensure that type checking remains simple and syntax-directed, despite the presence of GADTs and type functions.

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 5, 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 important feature of this paper; it is a development of earlier work, as Section 7 describes. There are precisely three roles (see Figure 4), written N, R, and P, with the following meaning:

Nominal equality, written $\sim_{\mathsf{N}}$, is the equality that the source Haskell 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 $\mathsf{Int} \sim_{\mathsf{N}} \mathsf{Int}$ but **not** $\mathsf{Int} \sim_{\mathsf{N}} \mathsf{Age}$. Type families introduce new nominal equalities. So, if we have **type instance** $\mathsf{F}\ \mathsf{Int} = \mathsf{Bool}$, then $\mathsf{F}\ \mathsf{Int} \sim_{\mathsf{N}} \mathsf{Bool}$.

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, $\mathsf{HTML} \sim_{\mathsf{R}} \mathsf{String}$. A Coercible constraint in Haskell corresponds to a proposition of representational equality in FC.

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 rule for type-safe cast:

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

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 sense: we can treat an expression of one type τ_1 as an expression of some other type τ_2 when those types share a representation.

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

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

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

$$\frac{\begin{array}{l} C : [\overline{a{:}\kappa}].\sigma_1 \sim_{\rho} \sigma_2 \\ \vdash \Gamma \qquad \overline{\Gamma \vdash \tau : \kappa} \end{array}}{\Gamma \vdash C(\overline{\tau}) : \sigma_1[\overline{\tau/a}] \sim_{\rho} \sigma_2[\overline{\tau/a}]}\ \textsc{Co\_Axiom} \qquad \frac{\vdash \Gamma \qquad c{:}\tau \sim_{\rho} \sigma \in \Gamma}{\Gamma \vdash c : \tau \sim_{\rho} \sigma}\ \textsc{Co\_Var}$$

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

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

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

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

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

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

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

Figure 5. $\Gamma \vdash \gamma : \phi$: Formation rules for coercions

4.2 Coercions

Coercions (Figure 4) and their typing rules (Figure 5) are the heart of System FC. The basic typing judgement for coercions is $\Gamma \vdash \gamma : \tau \sim_{\rho}^{\kappa} \sigma$. This judgement is also syntax directed: when this judgement holds we can determine the unique proposition $\tau \sim_{\rho}^{\kappa} \sigma$ that is justified by a particular coercion γ in a given context Γ. Furthermore, in this case τ and σ must be well formed and have the same kind κ. We often omit the kind annotation in our presentation when it is not important.

We can understand the typing rules in Figure 5, by thinking about the equalities that they define.

4.2.1 Nominal equality implies representational equality

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

4.2.2 Phantom equality relates all types

The coercion form $\langle \tau, \sigma \rangle_{\mathsf{P}}$ (shown in rule Co_Phantom) proves that any two types τ and σ are equal at role P.

4.2.3 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 derive representational reflexivity using **sub**. Phantom equality trivially includes reflexivity through rule Co_Phantom.

4.2.4 Axioms for equality

Each newtype declaration and type-family instance gives rise to an *axiom*; newtypes give rise to representational axioms, and type-family instances give rise to nominal axioms.[4] For example, the declarations

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

produce the axioms

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

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 Maybeσ (meaning that the two are considered to be the same type by the type checker). In C_2, the notation "$[a{:}\star]$." binds a 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.

4.2.5 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 4) include coercion abstraction $\lambda c{:}\phi.e$ and application $e\,\gamma$. These are the introduction and elimination

[4] For simplicity, we restrict ourselves to *open* type families. Closed type families (Eisenberg *et al.*, 2014) could also be accommodated.

```
newtype HTML = MkHTML String

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

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

Figure 6. Congruence and roles example code

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 A.2).

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

4.2.6 Equality is congruent

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 definitions in Figure 6. With these definitions in hand, what equalities should be derivable? (Recall the intuitive meanings of the different roles in Section 4.1.)

1. Should $\mathsf{Maybe\,HTML} \sim_{\mathsf{R}} \mathsf{Maybe\,String}$ 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 $\mathsf{Maybe\,HTML} \sim_{\mathsf{N}} \mathsf{Maybe\,String}$ hold?
   Certainly not. These two types are entirely distinct to Haskell programmers and its type checker.
3. Should $\mathsf{T\,HTML} \sim_{\mathsf{R}} \mathsf{T\,String}$ hold?
   Certainly not. We can see, by unfolding the definition for T, that the representations of the two types are different.
4. Should $a\,\mathsf{HTML} \sim_{\mathsf{R}} a\,\mathsf{String}$ hold, for a type variable a?
   It depends on the instantiation of a! If a becomes Maybe, then "yes"; if a becomes T, then "no". Since we may be abstracting over a, we do not know which of the two will happen, so we take the conservative stance and say that $a\,\mathsf{HTML} \sim_{\mathsf{R}} a\,\mathsf{String}$ does *not* hold.

This last point is critical. The alternative is to express a's argument roles in its kind, but that leads to a more complicated system; see related work in Section 7. A distinguishing feature of this paper is the simplification we obtain by attributing roles only to the arguments to type constants (H, in the grammar), and not to abstracted type variables. We lose a little expressiveness; see Sections 7.1 and 8.1 for discussion and examples.

To support both (1) and (4) requires two coercion forms and corresponding 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 the application $\gamma_1\,\gamma_2$.

What if we wish to prove a nominal equality such as $\mathsf{Maybe}\,(\mathsf{F}\,\mathsf{String}) \sim_{\mathsf{N}} \mathsf{Maybe}\,\mathsf{Int}$? We can't use the $H(\overline{\gamma})$ form, which proves only representational equality, but we can use the $\gamma_1\,\gamma_2$ form, with $\langle\mathsf{Maybe}\rangle$ for γ_1.

Congruence of type family application Rule Co_TyFam proves the equality of two type-family applications. It requires nominal coercions among all the arguments because type families can inspect their (type) arguments and branch on them. It would be unsound to derive an 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.

4.2.7 Equality can be decomposed

If we have a proof of $\mathsf{Maybe}\,\sigma \sim_\rho \mathsf{Maybe}\,\tau$, 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 necessarily injective with respect to representational equality. (Like all datatypes in Haskell, newtypes are injective with respect to nominal equality.) For example, consider these definitions:

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

Here, *roles*$(\mathsf{App}) = \mathsf{R},\mathsf{N}$. (The roles are inferred during compilation; see Section 4.5.) Yet, we can see the following chain of equalities:

$$\mathsf{App}\,\mathsf{Phant}\,\mathsf{Int} \sim_{\mathsf{R}} \mathsf{Phant}\,\mathsf{Int} \sim_{\mathsf{R}} \mathsf{Phant}\,\mathsf{Bool} \sim_{\mathsf{R}} \mathsf{App}\,\mathsf{Phant}\,\mathsf{Bool}$$

By transitivity, we can derive a coercion γ witnessing

$$\mathsf{App}\,\mathsf{Phant}\,\mathsf{Int} \sim_{\mathsf{R}} \mathsf{App}\,\mathsf{Phant}\,\mathsf{Bool}$$

If we could use **nth**^{2} on γ, we would get $\mathsf{Int} \sim_{\mathsf{N}} \mathsf{Bool}$: 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. Consider this 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, $\mathsf{EitherInt\ a} \sim_{\mathsf{R}} (\mathsf{Either\ a\ Int})$. Suppose we could apply **left** and **right** to coercions formed from this axiom. Using **left** would get us a proof of $\mathsf{EitherInt} \sim_{\mathsf{R}} (\mathsf{Either\ a})$, which could then be used to show, say, $(\mathsf{Either\ Char}) \sim_{\mathsf{R}} (\mathsf{Either\ Bool})$ and then (using **nth**) $\mathsf{Char} \sim_{\mathsf{N}} \mathsf{Bool}$. Using **right** would get us a proof of $\mathsf{a} \sim_{\mathsf{R}} \mathsf{Int}$, for *any* a. These are both clearly disastrous. So, we forbid using these coercion formers on representational coercions.[5]

The reader might wonder why **nth** is restricted to decompose only representational coercions, and never nominal ones. This is a simple design decision, and we could have chosen otherwise. Because nominal coercions can be decomposed via **left** and **right**, there is no need for (but also no harm in) using **nth** on nominal coercions.

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 gives the 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 wrong for the global environment to claim that Maybe's parameter is phantom because then we could prove that $\mathsf{Maybe\,Int} \sim_{\mathsf{R}} \mathsf{Maybe\,Bool}$ 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

[5] Although the forms **left** and **right** were originally part of FC, for simplicity they were omitted in previous papers (Weirich *et al.*, 2011) and in the implementation (GHC 7.2-7.6). However, Haskell users (e.g. Trac #7205) reported that some programs no longer type checked after this change, so these forms were re-introduced in GHC 7.8.

$$\boxed{\overline{\rho} \models H} \quad \text{“}\overline{\rho}\text{ are appropriate roles for }H\text{.”}$$

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

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

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

$$\boxed{\Omega \vdash \tau : \rho} \quad \text{“Assuming }\Omega\text{, }\tau\text{ can be used at role }\rho\text{.”}$$

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

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

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

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

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

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

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

$$\boxed{\rho_1 \leq \rho_2} \quad \text{“}\rho_1\text{ is a sub-role of }\rho_2\text{.”}$$

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

Figure 7. Rules asserting a correct assignment of roles to data types

for this judgement and two auxiliary judgements appear in Figure 7. Note that this judgement defines a *relation* between roles and data types. Our role inference algorithm (Section 4.5) determines the most permissible roles for this relation, but often other, less permissive roles, such as those specified by role annotations, are also included by this relation.

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

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 $a{:}\rho' \in \Omega$ and if $\sigma_1 \sim_{\rho'} \sigma_2$ then $\tau[\sigma_1/a] \sim_{\rho} \tau[\sigma_2/a]$. 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.

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 $\overline{a}$, existential type variables $\overline{b}$, 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 $(\rightarrow)$ 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 the congruence of the respective equality relations.

These definitions lead to a powerful theorem:

Theorem (Roles assignment narrowing)**.** *If $\overline{\rho} \models H$, where H is a data type or newtype, and $\overline{\rho}'$ is such that $\rho'_i \leq \rho_i$ (for $\rho_i \in \overline{\rho}$ and $\rho'_i \in \overline{\rho}'$), then $\overline{\rho}' \models H$.*

Proof. Straightforward induction on $\Omega \vdash \tau : \rho$. □

This theorem states that, given a sound role assignment for H, any more restrictive role assignment is also sound. This property of our system here is one of its distinguishing characteristics from our prior work on roles – see the end of Section 8.1.1 for discussion.

4.4 Progress and preservation

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. (For example, the **nth** decomposition coercion is necessary for the S_KPUSH rule of the operational semantics, shown in Appendix A.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 full proof of type safety appears in the appendix; it exhibits no new proof techniques. The structure of the proofs follows previous work, such as Weirich *et al.* (2011) or Yorgey *et al.* (2012).

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[6] 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.

4.5 Role inference

We have assumed throughout this discussion a global context where we can look up roles via $\mathit{roles}(H)$ and that these roles are appropriate, i.e. $\mathit{roles}(H) \models H$ for all H. We give here the algorithm that populates the environment $\mathit{roles}(H)$:

- Primitive type constructors like $(\rightarrow)$ and $(\sim^{\kappa}_{\rho})$ have predefined roles for their parameters (Figure 7).
- Type families (Section 2.5) have nominal roles for all parameters.
- The roles of **class**, **data** type, or **newtype** parameters are determined by a role inference algorithm, which we describe next.

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

1. Populate $\mathit{roles}(T)$ (for all T) with user-supplied annotations; omitted role annotations default to phantom for **data** and **newtype** and to nominal for **class**. Other than this default, classes are treated identically to datatypes, as they are implemented in FC via datatypes representing dictionaries (Sulzmann *et al.*, 2007; Hall *et al.*, 1996). (See Section 6.4 for discussion about this choice of default.)

[6] As in prior work (Eisenberg *et al.*, 2014), we ensure that the rewrite relation is confluent by restricting type families to have only linear patterns. If non-linear patterns were allowed in type families (that is, with a repeated variable on the left-hand side), combined with non-termination, our rewrite system would not be confluent. Losing confluence does not necessarily threaten consistency – it just threatens the particular proof technique that we use. However, a more powerful proof appears to be an open problem in the term rewriting community. Specifically, 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.

2. For every data type D, every constructor for that data type 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:

```
$\mathsf{walk}(T,a)$        := mark the $a$ parameter to $T$ as $\mathsf{R}$, when $a$ is unmarked.
$\mathsf{walk}(T,H\,\overline{\tau})$     := 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 $\mathsf{N}$;
                     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 $\mathsf{N}$.
$\mathsf{walk}(T,F(\overline{\tau}))$   := mark all variables free in the $\overline{\tau}$ as $\mathsf{N}$.
$\mathsf{walk}(T,\forall b{:}\kappa.\ \tau)$ := $\mathsf{walk}(T,\tau)$.
```

When marking variables, we ignore those that are not parameters to the data type T in question or have been previously been marked as N. The first case deals with existential and local ($\forall$-bound) type variables and the second with the case where a variable is used both in a nominal and in a representational context.

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 G.

5 Type inference with Coercible constraints

Section 2 describes a programmer-level view of when types are $\mathsf{Coercible}$; this section describes the portion of GHC's type inference algorithm that solves these constraints. This algorithm also produces the coercion evidence as described in Section 4, but we elide the details of evidence creation as this process is straightforward.

Type inference in GHC is accomplished via the OUTSIDEIN(X) algorithm, as described by Vytiniotis *et al.* (2011). This algorithm is a constraint-based type inference algorithm (Pottier & Rémy, 2005), that first generates a set of constraints during a pass over the Haskell source code and then solves these constraints separately. OUTSIDEIN(X) is parameterised by a constraint language and associated

$$\begin{array}{lcll}
L & & & \text{metavariable for classes}\\
\xi & ::= & a \mid \xi_1\,\xi_2 \mid H \mid \forall a{:}\kappa.\ \xi & \text{function-free types}\\
Q & ::= & \epsilon \mid Q_1 \wedge Q_2 \mid L\,\overline{\tau} \mid \tau_1 \sim_{\mathsf{N}} \tau_2 \mid \tau_1 \sim_{\mathsf{R}} \tau_2 & \text{constraints}\\
\mathbb{Q} & ::= & Q \mid \mathbb{Q}_1 \wedge \mathbb{Q}_2 & \text{top-level axiom schemes}\\
 & \mid & \forall \overline{a{:}\kappa}.\ Q \Rightarrow L\,\overline{\tau} & \quad\text{(constrained) class instance}\\
 & \mid & \forall \overline{a{:}\kappa}.\ F(\overline{\tau}) \sim_{\mathsf{N}} \sigma & \quad\text{type family instance}\\
 & \mid & \forall \overline{a{:}\kappa}.\ N\,\overline{a} \sim_{\mathsf{R}} \sigma & \quad\text{newtype axiom}\\
\ell & ::= & \mathsf{g} \mid \mathsf{w} & \text{constraint flavours}
\end{array}$$

Figure 8. Grammar for our constraint system

$$\begin{array}{ll}
\mathbb{Q} \wedge Q \Vdash Q & \text{reflexivity}\\
\mathbb{Q} \wedge Q_1 \Vdash Q_2 \text{ and } \mathbb{Q} \wedge Q_2 \Vdash Q_3 \text{ implies } \mathbb{Q} \wedge Q_1 \Vdash Q_3 & \text{transitivity}\\
\mathbb{Q} \Vdash Q_2 \text{ implies } \theta(\mathbb{Q}) \Vdash \theta(Q_2) & \text{substitutivity}\\
\mathbb{Q} \Vdash \tau \sim_{\mathsf{N}} \tau & \text{nominal eq. reflexivity}\\
\mathbb{Q} \Vdash \tau_1 \sim_{\mathsf{N}} \tau_2 \text{ implies } \mathbb{Q} \Vdash \tau_2 \sim_{\mathsf{N}} \tau_1 & \text{nominal eq. symmetry}\\
\mathbb{Q} \Vdash \tau_1 \sim_{\mathsf{N}} \tau_2 \text{ and } \mathbb{Q} \Vdash \tau_2 \sim_{\mathsf{N}} \tau_3 \text{ implies } \mathbb{Q} \Vdash \tau_1 \sim_{\mathsf{N}} \tau_3 & \text{nominal eq. transitivity}\\
\mathbb{Q} \Vdash Q_1 \text{ and } \mathbb{Q} \Vdash Q_2 \text{ implies } \mathbb{Q} \Vdash Q_1 \wedge Q_2 & \text{conjunctions}\\
\mathbb{Q} \Vdash \tau_1 \sim_{\mathsf{N}} \tau_2 \text{ implies } \mathbb{Q} \Vdash \tau[\tau_1/a] \sim_{\mathsf{N}} \tau[\tau_2/a] & \text{nominal eq. congruence}
\end{array}$$

Figure 9. Requirements of the entailment relation $\mathbb{Q} \Vdash Q$, adapted from Figure 3 of Vytiniotis *et al.* (2011).

constraint solver; the X in OutsideIn(X). Our work fits into this framework by introducing a new `Coercible t1 t2` constraint and extending the constraint solver to handle this constraint.

5.1 A constraint system with representational equality

The grammar for our instantiation of X appears in Figure 8. A constraint Q can be empty (trivially satisfied), a conjunction of constraints, a class constraint $L\,\overline{\tau}$, or an equality constraint. A nominal equality constraint $\tau_1 \sim_{\mathsf{N}} \tau_2$ is the standard type equality constraint already present in OutsideIn(X); a representational one $\tau_1 \sim_{\mathsf{R}} \tau_2$ is the encoding of `Coercible` $\tau_1\,\tau_2$. (Phantom equality constraints $\tau_1 \sim_{\mathsf{P}} \tau_2$ are unnecessary.)

The constraint system X defines an entailment relation $\mathbb{Q} \Vdash Q$, a judgement that holds whenever the assumptions $\mathbb{Q}$ imply the constraint Q. Note that the grammar for $\mathbb{Q}$ includes a conjunction of both regular constraints Q as well as top-level axioms. In our case, these axioms take one of three forms as shown in Figure 8: a class instance, a type family instance, or a newtype axiom. The OutsideIn(X) framework expects the entailment relation to uphold the properties listed in Figure 9.

In our case, the entailment relation essentially duplicates Figure 5, leaving out the form of the coercions themselves. Added onto those rules are rules for type classes, which do not concern us here. It can easily be shown that this entailment relation satisfies the properties of Figure 9. In particular, note that the substitutivity property of entailment is directly implied by a standard substitution lemma over coercions.

5.2 *An overview of* **OUTSIDEIN(X)**

We start with a brief overview of the OUTSIDEIN(X) algorithm, somewhat simplified from its original presentation.[7] Our goal is not to provide a complete explanation of OUTSIDEIN(X), but to provide enough context to explain the modifications required by the new Coercible constraint. Due to the complexities they add to the algorithm, polytypes (headed by ∀) are excluded from this presentation; their complexity is orthogonal to roles'.

OUTSIDEIN(X) uses a judgement $\Gamma \mapsto e : \tau \rightsquigarrow Q_{\mathrm{w}}$ to generate constraints in the language X. We can view $\Gamma \mapsto e : \tau \rightsquigarrow Q_{\mathrm{w}}$ as an algorithm whose inputs are Γ and e and whose outputs are τ, the type of the expression e and Q_{w}, the "wanted" constraint. By "wanted", we mean that the constraint Q_{w} must be satisfiable for e to have type τ. The constraint Q_{w} is then run through a constraint solver, in an attempt to reduce Q_{w} to the empty constraint ϵ via simplifications and substitutions.

Constraints can also be "given" constraints. These constraints arise from user type annotations. For example, if the user has declared foo :: Coercible a b ⇒ [a] → [b], then foo will be type-checked under an assumption that $a \sim_{\mathsf{R}} b$. This constraint will be considered a given.

5.2.1 *The solver pipeline*

The solver maintains a work list of simple constraints (that is, constraints without conjunctions), with given constraints prioritised over wanted ones. It proceeds by popping the first constraint off the work list (this constraint becomes the *work item*) and then processing it through the following pipeline:

1. Types in the work item are *flattened*, whereby a type τ, possibly with type functions, is converted into a type ξ devoid of type functions. Such types are easier to work with in subsequent steps. Flattening τ (essentially) creates a new type variable a for every type function application $F(\overline{\sigma})$ in τ, replacing the $F(\overline{\sigma})$ with a, and then adding the $F(\overline{\sigma}) \sim_{\mathsf{N}} a$ constraint to the work list. The details are, of course, more involved; see Vytiniotis *et al.* (2011).
2. The work item is then *canonicalised* (details are below), which reduces it to one of several simple forms. See Figure 10.

[7] Specifically, we omit implication constraints, touchable variables, and the flattening substitution.

$$\boxed{\vdash_{\mathsf{can}} Q} \quad \text{“Q is a canonical constraint.”}$$

$$\frac{}{\vdash_{\mathsf{can}} \epsilon}\ \text{CEMP} \qquad \frac{\vdash_{\mathsf{can}} Q_1 \qquad \vdash_{\mathsf{can}} Q_2}{\vdash_{\mathsf{can}} Q_1 \wedge Q_2}\ \text{CAND} \qquad \frac{}{\vdash_{\mathsf{can}} L\,\overline{\xi}}\ \text{CDICT}$$

$$\frac{}{\vdash_{\mathsf{can}} F(\overline{\xi}) \sim_{\mathsf{N}} a}\ \text{CFEQ} \qquad \frac{\begin{array}{l} a \notin \mathit{ftv}(\sigma) \\ \sigma \text{ is not headed by any type function } F \\ (\rho = \mathsf{R}) \Longrightarrow \sigma \text{ is not headed by any newtype } N \end{array}}{\vdash_{\mathsf{can}} a \sim_{\rho} \sigma}\ \text{CEQ}$$

Figure 10. Canonical constraints, $\vdash_{\mathsf{can}} Q$

3. The work item then undergoes binary interactions with *inert* constraints, where the inert constraints are those that have already gone through this pipeline. For example, if a given inert constraint is $a \sim_{\mathsf{N}} \mathsf{Int}$, then a work item of $\mathsf{Ord}\,a$ would be rewritten to $\mathsf{Ord}\,\mathsf{Int}$.
4. Lastly, the work item interacts with top-level axioms. This step includes type family reduction and class instance lookup.

While processing a work item, it is possible that we learn something new about a type variable, say, that a type variable b is now equal to Bool. When this happens, any inert constraint mentioning b is *kicked out* of the inert set and re-added to the work list. This step is necessary because the new knowledge about b may allow new interactions to occur.

5.2.2 *Canonicalisation*

The component of the solver that concerns us most is the *canonicalisation* algorithm. We write one step of this algorithm as $\mathit{canon}\,[\ell](Q_1) = Q_{2\bot}$, noting that canonicalisation can fail (that is, result in $\bot$) when the constraint is unsatisfiable. The parameter ℓ is a constraint *flavour*, which can be either given (“g”) or wanted (“w”).

Canonicalisation runs the *canon* algorithm until a fixpoint is reached; the result may or may not be canonical (according to the $\vdash_{\mathsf{can}} Q$ judgement in Figure 10). A constraint without a canonical form is not an error – perhaps later, the constraint solver will learn more and will be able to make more progress.

There are three basic canonical forms: a class constraint where all arguments are type-function-free (ξ is a metavariable for type-function-free types), an equality between a type function application and a type variable, and an equality between a type variable and a type that is not a type function application. The empty constraint and the conjunction of canonical constraints is also canonical.

The canonicalisation algorithm must be sound with respect to constraint entailment.

Property 1 (Canonicalisation soundness)**.**

Define $canon\,[\ell](Q_1) = Q_{2\perp}$ by top-to-bottom pattern matching as follows:

REFL	$canon\,[\ell](\tau \sim_{\mathsf{R}} \tau)$	$=$	ϵ
NEWL	$canon\,[\ell](N\,\overline{\tau} \sim_{\mathsf{R}} \sigma_2)$	$=$	$\sigma_1[\overline{\tau/a}] \sim_{\mathsf{R}} \sigma_2$
	newtype $N\,\overline{a} \rightsquigarrow^* \sigma_1$, where σ_1 is not headed by a newtype		
NEWR	$canon\,[\ell](\sigma_1 \sim_{\mathsf{R}} N\,\overline{\tau})$	$=$	$\sigma_1 \sim_{\mathsf{R}} \sigma_2[\overline{\tau/a}]$
	newtype $N\,\overline{a} \rightsquigarrow^* \sigma_2$, where σ_2 is not headed by a newtype		
DECOMPN	$canon\,[\ell](H\,\overline{\tau} \sim_{\mathsf{N}} H\,\overline{\sigma})$	$=$	$\overline{\tau \sim_{\mathsf{N}} \sigma}$
DECOMPR	$canon\,[\ell](H\,\overline{\tau} \sim_{\mathsf{R}} H\,\overline{\sigma})$	$=$	$\overline{\tau \sim_{\rho} \sigma}$
	$(H$ is not a newtype$) \wedge (\overline{\rho}$ is a prefix of $roles(H))$		
DECOMPFN	$canon\,[\ell](H_1\,\overline{\tau} \sim_{\mathsf{N}} H_2\,\overline{\sigma})$	$=$	$\perp$
DECOMPFR	$canon\,[\ell](H_1\,\overline{\tau} \sim_{\mathsf{R}} H_2\,\overline{\sigma})$	$=$	$\perp$
	$(H_1$ is not a newtype$) \wedge (H_2$ is not a newtype$)$		
DECOMPNEW	$canon\,[\mathrm{w}](N\,\overline{\tau} \sim_{\mathsf{R}} N\,\overline{\sigma})$	$=$	$\overline{\tau \sim_{\rho} \sigma}$
	(no givens might match) $\wedge$ $(\overline{\rho}$ is a prefix of $roles(N))$		
DECOMPD	$canon\,[\ell](H_1\,\overline{\tau} \sim_{\mathsf{R}} H_2\,\overline{\sigma})$	$=$	$H_1\,\overline{\tau} \sim_{\mathsf{R}} H_2\,\overline{\sigma}$
APPN	$canon\,[\ell](\tau_1\,\sigma_1 \sim_{\mathsf{N}} \tau_2\,\sigma_2)$	$=$	$\tau_1 \sim_{\mathsf{N}} \tau_2 \wedge \sigma_1 \sim_{\mathsf{N}} \sigma_2$
APPR	$canon\,[\ell](\tau_1\,\sigma_1 \sim_{\mathsf{R}} \tau_2\,\sigma_2)$	$=$	$\tau_1\,\sigma_1 \sim_{\mathsf{R}} \tau_2\,\sigma_2$
TVREFL	$canon\,[\ell](a \sim_{\rho} a)$	$=$	ϵ
OCCURL	$canon\,[\ell](a \sim_{\mathsf{N}} \sigma)$	$=$	$\perp$
	$a \in ftv(\sigma)$		
OCCURR	$canon\,[\ell](\sigma \sim_{\mathsf{N}} a)$	$=$	$\perp$
	$a \in ftv(\sigma)$		
TVL	$canon\,[\ell](a \sim_{\rho} \sigma)$	$=$	$a \sim_{\rho} \sigma$
TVR	$canon\,[\ell](\sigma \sim_{\rho} a)$	$=$	$a \sim_{\rho} \sigma$
FAIL	$canon\,[\ell](\tau_1 \sim_{\rho} \tau_2)$	$=$	$\perp$

Figure 11. The canonicalisation algorithm for equality constraints. These rules are explained in Section 5.3.

1. *When $canon\,[\mathrm{g}](Q_1) = Q_2$, it must be that $Q_1 \Vdash Q_2$. That is, we can derive evidence for Q_2 given evidence for Q_1.*
2. *When $canon\,[\mathrm{w}](Q_1) = Q_2$, it must be that $Q_2 \Vdash Q_1$. That is, by producing evidence for Q_2 we will be able to produce evidence for Q_1.*

5.3 Canonicalising equality constraints

Adding representational equalities to X requires changing the canonicalisation algorithm of OUTSIDEIN(X). Indeed, with the exception of a straightforward new binary interaction (Section 5.4), this is the *only* change necessary for the constraint solver.

Figure 11 presents the portion of the algorithm that works on equality constraints of the form $\tau_1 \sim_{\rho} \tau_2$. A result of $\perp$ (pronounced "failure") means that a definite type error can be reported. For example, attempting to canonicalise $\mathsf{Int} \sim_{\mathsf{R}} \mathsf{Bool}$ yields failure.

We describe the rules from the figure in order from top to bottom, except for REFL which we defer to Section 5.3.4.

5.3.1 Unwrapping newtypes

Rules NEWL and NEWR unwrap newtypes. They work only at the outermost level, unwrapping Age but not Maybe Age. Although not expressed in Figure 11, unwrapping newtypes happens only when the newtype's constructor is in scope (see Section 3.1).

Both rules also unwrap eagerly, continuing to unwrap outermost newtypes until a type without an outermost newtype is found. If no such type is found (because a newtype is recursive without an intervening non-newtype), then rule NEWL (or NEWR) does not apply. This is the meaning of the $\leadsto^*$ widget in the rules.

In the case of a recursive newtype, though, this unwrapping can diverge. For example, consider this unlikely construction:

newtype FixEither x = MkFE (Either x (FixEither x))

The role of FixEither's parameter will be inferred to be representational. Now suppose we are trying to canonicalise the wanted constraint

$$[\mathrm{w}]\ \mathsf{FixEither\ Age} \sim_{\mathsf{R}} \mathsf{FixEither\ Int}$$

Assuming the MkFE is in scope, unwrapping yields:

$$[\mathrm{w}]\ \mathsf{Either\ Age\ (FixEither\ Age)} \sim_{\mathsf{R}} \mathsf{Either\ Int\ (FixEither\ Int)}$$

Now decomposition yields $\mathsf{Age} \sim_{\mathsf{R}} \mathsf{Int}$ (which is easily solved), and the original constraint $\mathsf{FixEither\ Age} \sim_{\mathsf{R}} \mathsf{FixEither\ Int}$, so we are back where we started, having made no progress. In the actual implementation, a reduction counter notices the loop and reports an error.

5.3.2 Decomposition of applied type constants

The DECOMP rules implement decomposition of an applied type constant. Decomposition for nominal equality is straightforward: if the two constants match, decompose (DECOMPN); otherwise, fail (DECOMPFN). For representational equality, however, the rules are more subtle. Rule DECOMPR fires only for non-newtypes. It is easiest to understand this restriction by consider the "given" case separately from the "wanted" case.

We cannot decompose given newtype representational constraints. If the constraint in question is a given, then it would be unsound to decompose. Note rule CO_NTH from Figure 5, which forbids the type constant involved from being a newtype. See Section 4.2.7 for the details. Creating evidence for decomposing a given constraint of the form $H\,\overline{\tau} \sim_{\mathsf{R}} H\,\overline{\sigma}$ requires using **nth**, and so we are stuck.

Wanted newtype representational constraints are tricky. We must decompose wanted newtype representational constraints, even when unwrapping does not apply. For example it happens that the abstract type IO t is implemented by a newtype. Haskell programmers certainly want to coerce between (IO Int) and (IO Age). However, since IO is abstract, its data constructor is not visible to clients, and hence we cannot use newtype unwrapping (Section 5.3.1); so the only way forward is to decompose.

However we must tread carefully, because in certain situations it is just possible for decomposition to make a provable goal un-provable, which would compromise the completeness of type inference. Here is a contrived scenario illustrating the problem:

newtype ConstBool a = Mk Bool
type role ConstBool **representational**

Suppose the constructor Mk is not in scope. Now, consider the following constraints, where Greek letters denote unification variables:

$$\begin{array}{lll} [g] & \mathsf{ConstBool\ a} \sim_{\mathsf{R}} \mathsf{ConstBool\ b} & (1) \\ [w] & \mathsf{ConstBool}\,\alpha \sim_{\mathsf{R}} \mathsf{ConstBool\ b} & (2) \\ [w] & \alpha \sim_{\mathsf{R}} \mathsf{a} & (3) \end{array}$$

The wanted goal is certainly provable from the givens; just use (3) to substitute for α in (2), and then (2) is equal to (1). However, suppose the constraint solver happens to process (2) before (3). Because Mk is not in scope, ConstBool cannot be unwrapped. So we apply decomposition, yielding the un-satisfiable wanted goal $\alpha \sim_{\mathsf{R}}$ b, and hence (wrongly) report an error. This kind of incompleteness is particularly confusing to the programmer, because the goal we are trying to prove is practically equal to one of the givens. To avoid this confusion, we do one extra check before decomposing a wanted newtype representational equality, to make sure that no givens could possibly influence the wanted constraint. This is the informally-stated "no givens might match" side condition on the DECOMPNEW rule. To formalise the side condition we would need to pass to *canon* the set of (canonicalised) givens, which would clutter up Figure 11. Happily, there is no difficulty in the implementation.

There is another awkward consequence of decomposing wanted representational newtypes. Consider the FixEither example given in Section 5.3.1. As we saw there, canonicalising will loop if the data constructor MkFE is in scope. But it if it not, we will decompose to Age$\sim_{\mathsf{R}}$Int, which is easily soluble. This is a situation where importing the MkFE constructor makes a typeable program become ill-typed, rather unfortunately. However, this seems the best we can do.

Failure and stuck cases for decomposing representational equalities. In rule DECOMPFR, we fail (reporting an error) when canonicalising a representational equality between two different type constants, neither of which is a newtype.

On the other hand, to account for the ConstBool example discussed above, DECOMPD returns unchanged any remaining representational equality constraint between two applied type constants. If no earlier rule has fired, then we don't know enough about these types either to canonicalise fully or to be sure the program has a type error. These constraints will be examined again by the solver after it has learned more from other constraints.

5.3.3 Decomposition of applied type variables

Rule APPN decomposes an equality between two type applications, where the head of the type in the "function" position is just a type variable. (Note that the head of a nested type application must be either a type constant or a type variable; anything else would be ill-kinded.) This rule works only over nominal equality, as decomposing a representational equality of this form – say, $a\,\tau \sim_{\mathsf{R}} b\,\sigma$ – is unsound, for two reasons:

- We do not know the roles on a and b. Accordingly, should we reduce to $\tau \sim_{\mathsf{R}} \sigma$ or $\tau \sim_{\mathsf{N}} \sigma$? It is impossible to know, especially considering that we might learn, later on during solving, the concrete value for a or b.
- Perhaps more problematic, type variables may stand in for newtypes. If we learn, say, that $a \sim_{\mathsf{N}} N$ for some newtype N, then it is possible that τ and σ are unrelated, as $N\,\tau \sim_{\mathsf{R}} b\,\sigma$ might be solved via unwrapping N.

Decomposing a representational equality among such type applications is not possible, but neither is this an error. We thus simply fail to canonicalise such constraints, as shown in APPR, which returns the same constraint it is given.

5.3.4 The reflexivity check

Rule REFL checks for reflexivity, succeeding with an empty constraint if the equality is reflexive. This check is needed only for representational equality constraints, as it is redundant with later checks for nominal equality: any nominal equality constraint is decomposed into its atoms, which are then checked for reflexivity. For representational equality, however, this is not the case, both because of the possibility of recursive newtypes and of impossible-to-decompose type applications.

Here are examples of these cases. Suppose we have X:

```
newtype X = MkX (Int → X)
```

Further, suppose we have these (unrelated) constraints:

$$\begin{array}{ll} [\mathrm{w}] & \mathsf{X} \sim_{\mathsf{R}} \mathsf{X} \\ [\mathrm{w}] & \mathsf{f\ a} \sim_{\mathsf{R}} \mathsf{f\ a} \end{array}$$

Without the reflexivity check, canonicalising the first constraint would loop, in exactly the same way as the FixEither example of Section 5.3.1. Canonicalising the second constraint would simply be stuck without the reflexivity check, hitting rule APPR and making no progress. Programmers find it particularly frustrating if a compiler says that it is unable to prove that two syntactically-identical types are coercible, e.g. Coercible X X!

Note that rule REFL is tried first, before unwrapping newtypes, otherwise the X example above would loop through NEWL/NEWR.

5.3.5 Dealing with type variables

Rule TVREFL dispatches the case where we compare a type variable with itself, at either role. The *canon* algorithm then does an "occurs check" (rules OCCURL and OCCURR). The occurs check happens for nominal equality only because an occurs-check failure for representational equality is not necessarily an error. Suppose we have $a \sim_{\mathsf{R}} b\,a$, but then we later learn that $b \sim_{\mathsf{N}} \mathsf{Id}$, where **newtype** Id a = Id a. The $a \sim_{\mathsf{R}} b\,a$ equality now becomes easily solvable.

Because of the possibility of occurs-check failures, rules TVL and TVR do not necessarily produce canonical constraints over representational equalities. Canonical type variable equality constraints must pass the occurs check, even for representational equality constraints, because they are used for substitutions. Our $a \sim_{\mathsf{R}} b\,a$ constraint then remains non-canonical, but not otherwise harmful.

5.3.6 Correctness of canon

Theorem (Soundness of *canon*)**.** *The canon algorithm as presented in Figure 11 is sound, according to Property 1.*

Proof. For each rule in *canon*, it is possible to create a coercion witnessing the result from the input, and it is possible to create a coercion witnessing the input from the result, all using the coercion formation rules of Figure 5. These coercions are all straightforward to build. As the entailment relation $\mathcal{Q} \Vdash Q$ derives from the coercion formation rules, these coercions witness the entailments we desire. □

Note that we do not attempt to prove the algorithm complete – indeed, we know that with its treatment of recursive newtypes, type applications, and occurs-check failures, the algorithm is incomplete.

5.4 Substitution with representational equalities

Using the canonicalisation algorithm just described is nearly enough to have the OUTSIDEIN(X) solver work with representational equalities. The one remaining piece is to implement transitivity in the presence of assumptions. For example, we would like to be able to deduce $a_1 \sim_{\mathsf{R}} a_2 \wedge a_2 \sim_{\mathsf{R}} a_3 \Vdash a_1 \sim_{\mathsf{R}} a_3$. This is accomplished by allowing substitution by representational equalities in representational equality constraints. (Previously, only nominal equality constraints were used for substitution.) In this case, $a_1 \sim_{\mathsf{R}} a_2$ and $a_2 \sim_{\mathsf{R}} a_3$ are givens. These are already in canonical form. When solving the wanted $a_1 \sim_{\mathsf{R}} a_3$, we can use canonical type variable representational equality constraints to rewrite other representational equality constraints. We thus rewrite a_1 to a_2 and then a_2 to a_3 in $a_1 \sim_{\mathsf{R}} a_3$. We then get $a_3 \sim_{\mathsf{R}} a_3$ and are done.

This use of rewriting only works with *canonical* constraints. Transitivity is thus somewhat limited. For example, the following fails to type-check:

```
incomplete :: (Coercible (a b) (c d), Coercible (c d) (e f))
  ⇒ Proxy (c d) → a b → e f
```

```
incomplete _ = coerce
```

(The Proxy argument is just to make c and d unambiguous.) This definition *should* be accepted, but it is not, as *canon* cannot canonicalise the givens and then discover the transitivity. We conjecture that this source of incompleteness could be overcome with more engineering, but there seems to be little incentive to add the extra complexity to the solver in this obscure case.

6 Reflection and discussion

This section discusses some opportunities and choices that arose in the course of our work on Coercible.

6.1 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. Assume the programmer needs an instance of the type class 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 = MkHTML mempty
  mappend (MkHTML a) (MkHTML b) = MkHTML (mappend a b)
  mconcat xs = MkHTML (mconcat (map unHTML xs))
```

Note that this definition is not only verbose, but also non-trivial, as invocations of MkHTML and unHTML 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)
  mconcat = coerce (mconcat :: [String] → String)
```

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 = MkHTML 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.

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

Furthermore, GND was previously unsound (Weirich *et al.*, 2011). When combined with other extensions of GHC, such as type families (Chakravarty *et al.*, 2005a; Chakravarty *et al.*, 2005b) or GADTs (Cheney & Hinze, 2003), GND could be exploited to completely break the type system: Figure 12 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. Indeed, preventing abuse of GND was the entire subject of the previous work (Weirich *et al.*, 2011) the current paper is based on.

Similarly, it was possible to use GND to break invariants of abstract data types. The addition of coerce makes it yet easier to break such abstractions. As discussed in Section 3.1, these abuses can now be prevented via role annotations.

6.2 *Coercible and rewrite rules*

What if a client of module Html writes this?

```
....( map unHTML 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 unHTML) 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 unHTML) 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 (Peyton Jones *et al.*, 2001). 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 unHTML). The definition unHTML in module Html does not mention coerce, but both produce the same System FC code (a cast). So via cross-module inlining (more dissolution of abstraction boundaries) unHTML will be inlined, transforming the call to the equivalent of (map coerce), and that in turn fires the rewrite rule. Indeed even a nested call like map (map unHTML) will also be turned into 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.

6.3 Syntax for role annotations

Recall the Map example from Section 3.1, and its role annotation:

```
data Map k v = Leaf | Node k v (Map k v) (Map k v)
type role Map nominal representational
```

This is only one possible concrete syntax for role annotations, and we explored a number of others. In doing so we identified the following design criteria:

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, easing migration to the first version of GHC supporting roles (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 data type evolves.
7. Code is read much more often than it is written; favour readability over concision.

Our chosen syntax, with **type role** ..., satisfies criteria (1), (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, GHC supports C-style preprocessor directives, so 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. Breitner *et al.* (2014b) discusses alternatives to this syntax in Appendix B.1.

6.4 The role of role inference

Why did we add role inference to GHC, assigning the most permissive role to type constructors by default?

We did not have to design the language in this way. We could have required programmers to annotate the roles of every type, which GHC would check for consistency. However, in this case the burden on programmers seems drastic and migration to this system overwhelming, requiring all existing data type declarations to be annotated with roles.

Alternatively, we could specify that all unannotated roles default to nominal (thus removing the need for role inference). According to the specification of Figure 7, it is always sound to assign the nominal role to all parameters of a type constructor H. This choice would lead to greater abstraction safety by default. For example, the implementor of Map would not need to add a role annotation to guarantee abstraction.

However, we choose to use the most permissive roles by default for several reasons. First, for convenience: this choice increases the availability of coerce (as only those types with annotations would be Coercible otherwise), and it supports backward compatibility with the Generalized Newtype Deriving (GND) feature (see Section 6.1).

Furthermore, role inference also means that the majority of programmers do not need to learn about roles nor need to add role annotations. Users of coerce will need to consider roles, as will library implementors who use class-based invariants (see Section 3.1). Other users are unaffected by roles and will not be burdened by them.

Our choices in the design of the role system has generated vigorous debate.[8] This discussion is healthy for the Haskell community. The difficulty with abstraction is not new: with GND, it has always been possible to lift coercions through data types, potentially violating their class-based invariants. The features described in this paper make this subversion both more convenient (through the use of coerce) and, more importantly, now preventable (through the use of role annotations).

[8] To read some of this debate, see the thread beginning with this post: http://www.haskell.org/pipermail/libraries/2014-March/022321.html

6.5 Roles in Practice

We have described a mechanism to allow safe coercions among distinct types, and we have reimplemented GHC's previously unsafe Generalized Newtype Deriving extension in terms of these safe coercions. Naturally, this change causes some code that was previously accepted to be rejected. Given that Haskell has a large user base and a good deal of production code, how does this change affect the community?

Advance testing During the development of this feature, we tested it against several popular Haskell packages available through Hackage, an online Haskell open-source distribution site. These tests were all encouraging and did not find any instances of hard-to-repair code in the wild.

Compiling all of Hackage As of 30 September 2013, 3,234 packages on Hackage compiled with GHC 7.6.3, the last released version without roles. The development version of GHC at that time included roles. A total of only four packages failed to compile directly due to GND failure.[9] Of these, three of the failures were legitimate – the use of GND was indeed unsafe. For example, one case involved coercing a type variable passed into a type family; the author implicitly assumed that a newtype and its representation type were always considered equivalent with respect to the type family. Only one package – acme-schoenfinkel – failed to compile because of the gap in expressiveness between the roles in Weirich *et al.* (2011) and those here. No other Hackage package depends on this one, indicating it is not a key part of the Haskell open-source fabric. The example in Section 8.1.1 is along similar lines to the failure observed here.

These data were gathered almost two months after the implementation of roles was pushed into the development version of GHC, so active maintainers may have made changes to their packages before the study took place. Indeed, we are aware of a few packages that needed manual updates. In these cases, instances previously derived using GND had to be written by hand, but quite straightforwardly.

Rewrite rules Since GHC 7.10, the rewrite rule *"map/co"* from Section 6.2 has been added to the standard library, and indeed, it does fire: We analysed 1,077 packages[10]. In 64 of these packages, the *"map/co"* rule fired and eliminated a total of 272 calls to map (out of 13,991 calls that were not already dissolved by list fusion).

[9] These data come from Bryan O'Sullivan's work, described here: http://www.haskell.org/pipermail/ghc-devs/2013-September/002693.html That posting includes 3 additional GND failures; these were due to an implementation bug, since fixed.

[10] Stackage nightly 2015-05-21, excluding two packages with non-Haskell dependencies that were not fulfilled on our test system

7 Related work

Prior work 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 Weirich *et al.* (2011), as well as make connections to other systems.

7.1 Prior version of roles

The idea of *roles* was initially developed in Weirich *et al.* (2011) as a solution to the Generalized Newtype Deriving problem. That work introduces the equality relations $\sim_{\mathsf{R}}$ and $\sim_{\mathsf{N}}$ (called "type equality" and "code equality" resp. in Weirich *et al.* (2011)). However, the system presented in Weirich *et al.* (2011) was quite invasive: it required annotating every sub-tree of every kind with a role. Kinds in GHC are already quite complicated because of kind polymorphism, and a new form of role-annotated kinds would be more complex still.

In this paper, we present a substantially simplified version of the roles system of Weirich *et al.* (2011), requiring role information only on the parameters to data types. Our new design keeps roles and kinds modularly separate, so that roles can be handled almost entirely separately (both intellectually and in the implementation) from kinds. 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 (see Section 8.1).

Furthermore, the observation that it is sound to "assume the worst" and use parameterised types with less permissive roles opens the door to role annotations. In this work, programmers are allowed to deliberately specify less permissive roles, giving them the ability to preserve type abstractions.

Surprisingly, this flexibility means that our version of roles actually *increases* expressiveness compared to Weirich *et al.* (2011) in some places. In Weirich *et al.* (2011) 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.

Besides the simplification discussed above, this paper makes two other changes to the specification of roles presented in Weirich *et al.* (2011).

- The treatment of the phantom role is entirely novel; the rule CO_PHANTOM has no analogue in prior work.
- The coercion formation rules (Figure 5) are refactored so that the role on the coercion is an *output* of the (syntax-directed) judgement instead of an input. This is motivated by the implementation (which does not know the role at which coercions should be checked) and requires the addition of the CO_SUB rule.

There are, of course, other minor differences between this system and Weirich *et al.* (2011) in keeping with the evolution of System FC. The main significant change, unrelated to roles, is the re-introduction of **left** and **right** coercions; see Section 4.2.7.

One important non-difference relates to the linear-pattern requirement. Section 4.4 describes that our language is restricted to have only *linear* patterns in its type families. (GHC, on the other hand, allows non-linear patterns as well.) This restriction exists in the language in Weirich *et al.* (2011) as well. Section 4.2.2 of Weirich *et al.* (2011) defines so-called Good contexts as having certain properties. Condition 1 in this definition subtly implies that all type families have linear patterns – if a type family had a non-linear pattern, it would be impossible, in general, to establish this condition. The fact that the definition of Good implies linear patterns came as a surprise, further explored in Eisenberg *et al.* (2014). The language described in the present paper clarifies this restriction, but it is not a new restriction.

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 4.5 is a new contribution of this work.

7.2 *Prior version of* ***Coercible***

This paper describes Coercible as implemented in GHC 7.10, using a dedicated solver in the type checker to handle representational equality constraints (Coercible) as well as nominal equality constraints (∼). This approach differs from the initial design that was shipped with GHC 7.8 and discussed in an earlier version of this work (Breitner *et al.*, 2014a), where Coercible was presented as a type class instead of a special constraint.

In particular, our prior work explains the solving of Coercible constraints in terms of type class instances. The motivation was to make it possible for the programmer to predict and understand the behaviour of the compiler without special knowledge, assuming she is aware of type classes.

Unfortunately, that approach had a few drawbacks. Although it was sold as behaving "like a normal type class", that was never fully true, and the solver treated Coercible special in a few cases:

- It would refrain from building recursive evidence. Recursive evidence is common and useful with type classes, but for Coercible it would simply cause the program to loop when executed, so we gave a compile time error instead.
- It allowed constraints of the form Coercible (**forall** a. s) (**forall** a. t) which are forbidden for type classes, but required here to deal with newtypes such as **newtype** Sel = MkSel (**forall** a. [a] → a).
- While type class instances are always exported and unconditionally visible, the visibility of the newtype unwrapping instance depends on whether the constructor is in scope.

In the end we found it clearer to stop pretending Coercible is a type class and honestly call it a constraint of its own right, with its own rules and its own solver.

This also made the feature more powerful, as the instance-based approach is not able to decompose given Coercible constraints (Section 2.7).

7.3 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` $\leq$ τ `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 (2013) 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[11] that has been confirmed to date back several years. Garrigue (2013) 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.

8 Future directions

As of the date of writing (June 2015), roles seem not to have caused an undue burden to the community. The first release candidate for GHC 7.8 was released on 3 February 2014, followed by the full release on 9 April, and package authors had been updating their work to be compatible for some time. The authors of this paper are unaware of any major problems that Haskellers have had in updating existing code. However, two problems have been identified: the need for roles to work in higher-order scenarios, and the need for a better interaction between roles and Safe Haskell (Terei *et al.*, 2012). We also review some proposed expansions of the roles feature to more exotic Haskell constructs.

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

8.1 Roles for higher-order types

Some users wish to use roles in scenarios that are currently beyond the ability of roles to express. We focus on one such scenario, as it is representative of all examples we have seen, including the package that did not compile when testing all of Hackage (Section 6.5).

8.1.1 Adding join to Monad

Imagine adding the join method to the Monad class, as follows:

```
class Monad m where
  ...
  join :: forall a. m (m a) → m a
```

With this definition, GND would still work in many cases. For example, if we define

```
newtype M a = MkM (Maybe a)
  deriving Monad
```

GND will work without a problem. We would need to show Coercible (Maybe (Maybe a) → Maybe a) (M (M a) → M a), which is straightforward.

More complicated constructions run into trouble, though. Take this definition, written to restrict a monad's interface:

```
newtype Restr m a = MkRestr (m a)
  deriving Monad
```

To perform GND in this scenario, we must prove Coercible (m (m a) → m a) (Restr m (Restr m a) → Restr m a). In solving for this constraint, we eventually simplify to Coercible (m (m a)) (m (Restr m a). At this point, we are stuck, because we do not have any information about the role of m's parameter, so we must assume it is nominal. The GND feature is thus not available here. Similar problems arise when trying to use GND on monad transformers, a relatively common idiom.

How would this scenario play out under the system proposed in Weirich *et al.* (2011)? This particular problem wouldn't exist – m's kind could have the right roles – but a different problem would. A type's kind also stores its roles in Weirich *et al.* (2011). This means that Monad instances could be defined only for types that expect a representational parameter. Yet, it is sometimes convenient to define a Monad instance for a data type whose parameter is properly assigned a nominal role. The fact that the system described in this paper can accept Monad instances both for types with representational parameters and nominal parameters is a direct consequence of the *Role assignment narrowing* theorem (Section 4.3), which does not hold of the system in Weirich *et al.* (2011).

8.1.2 Implication constraints

Looking forward, there is a proposal to indeed add join to Monad, and so we want to be able to allow the use of GND on this enhanced Monad class. One promising approach to this problem is to allow *user-specified implication constraints*.

Continuing the example from above, imagine we could write the following:

```
deriving instance (Monad m, forall a b. Coercible a b ⇒ Coercible (m a) (m b))
  ⇒ Monad (Restr m)
```

When we are trying to simplify Coercible (m (m a)) (m (Restr m a)), we see that this constraint can be solved if Coercible (m a) (Restr m a), and so we simplify. This last constraint is easy to solve via the definition of Restr, and so we succeed.

The constraint **forall** a b. Coercible a b ⇒ Coercible (m a) (m b) is an *implication constraint* (Hinze & Peyton Jones, 2000), saying that Coercible (m a) (m b) holds whenever Coercible a b holds, for universally-quantified type variables a and b. These constraints do not currently exist in Haskell, but users have wanted them for some time.[12] With such constraints, it would seem that we can effectively assign roles to parameters of type variables, much like we already assign roles to parameters of type constants. For example, the implication constraint above gives the parameter to m a representational role. This role assignment is precisely what is needed to use GND with Monad and Restr.

The details of this have yet to be fully worked out, but we believe that the implementation could be straightforward, given that GHC already deals with internal implication constraints, derived from type-checking GADT pattern-matches.

8.2 Roles and Safe Haskell

Safe Haskell (Terei *et al.*, 2012) is a subset of Haskell known to have additional safety properties. Safe Haskell excludes constructs such as unsafeCoerce and unsafePerformIO, as these can be used to subvert the type system. It also excludes Template Haskell (Sheard & Peyton Jones, 2002), as that feature can look up type definitions and thus break abstraction. See the original paper for the details.

One of the consequences of the unsoundness of earlier versions of GND is that the feature was (quite rightly) excluded from the Safe Haskell subset. However, even with roles and GND written in terms of coerce, the feature *still* does not meet the Safe Haskell criteria. At issue is preserving datatype abstraction.

We describe in Section 3.1 that we allow coercions to happen even on data types 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 could use a more stringent check when satisfying a Coercible constraint when compiling in Safe mode, requiring all constructors of all data types to be coerced under to be visible. This means, essentially, traversing the entire tree of data type definitions, making sure all constructors of all data types, recursively,

[12] See https://ghc.haskell.org/trac/ghc/ticket/2256, which was created in 2008.

are available. This check is not implemented, however, because it would seem to be unreasonably non-performant. Furthermore, it would require that users import many constructors that remain unmentioned in their code, imported only to satisfy this requirement. We continue to look for a better solution to this problem; for some ideas, the reader is encouraged to consult https://ghc.haskell.org/trac/ghc/wiki/SafeRoles.

8.3 Conservativity of roles

8.3.1 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
```

It is safe to coerce a Bar Int to a Bar Bool. Unravelling definitions, we see that this is so. Yet, coercing Bar Int to Bar [Double] is clearly not safe. 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 instances correspond.

8.3.2 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. Consider the following type:

```
data T a b where
  MkT :: (a ~ b) ⇒ a → T a b
```

Role inference assigns both parameters to have nominal roles. Inspection of the type definition, however, shows us that the second parameter, b, is almost a phantom – it is used in only one place: the equality constraint with a. Also, note that the first parameter, a, is used representationally except in that same spot.

We can thus conclude that T x x has the same run-time representation as T y y whenever x has the same representation as y. Yet, the role mechanism is not expressive enough to prove this.

8.4 Extending roles to families

8.4.1 Roles on type and data families

In GHC today, all type and data family parameters have nominal roles, because 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 (a :: Bool) b c
type instance If True  b c = b
type instance If False b c = c
```

The above definition, though not accepted by our implementation, 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.

8.4.2 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, data types. 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 data type declaration, somewhat like this:

```
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, though it would seem to take only the will to do so. Instead, all type variables in a data family instance are assigned nominal roles. Why? Essentially because there is no

```
data Maybe a = Nothing | Just a
data Option a = None | Some a
data Few a = Zero | One a | Two a a

maybe2option :: Maybe a → Option a
maybe2option Nothing  = None
maybe2option (Just x) = Some x

maybe2few :: Maybe a → Few a
maybe2few Nothing = Zero
maybe2few (Just x) = One x
```

Figure 13. Data type conversions

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

8.5 What else is there to coerce?

The starting point of this work was the observation that there exist expressions, such as map MkHTML, which change the types, but not the representation of their arguments. We built a system to express and use this in a type-safe manner.

But Coercible and coerce currently cannot be used in all such situations. Consider the data types Maybe a and Option a in Figure 13, which have – up to the names of the constructors – identical definitions. For a given compiler, it may be the case that a value m::Maybe a has precisely the same representation as its counterpart (maybe2option m)::Option a. If so, we could replace maybe2option with a zero-cost coercion. We expect that it would be possible to extend our system to allow for Coercible (Maybe a) (Option a), in the situations where the compiler makes the two indistinguishable. Generic programming techniques (Rodriguez *et al.*, 2008) could, if tailored around this feature, gain performance boosts if the translation between the concrete to the generic representation no longer incurs a runtime cost.

One could go even further, however. The conversion function maybe2few in the same Figure may also (depending on the compiler) be operationally the identity. For example, if the first constructor is tagged 1, the second is tagged 2, and so on, then (Just x) and (One x) would have the same representation. However, the situation is now asymmetrical: we may be able to convert from Maybe a to Few a for free, but the reverse is certainly not true, because the value might use the constructor Two. Such uni-directional version of Coercible amounts to *explicit inclusive subtyping* and is more complicated than our current symmetric system:

For example, the lifting rule would have to take variance into account: For a type constructor T, does Coercible (T a) (T b) require Coercible a b, or Coercible b a, or both, or neither? Furthermore, we would have to adapt our internal language, FC, to work with explicit subtyping proofs (Crary, 2000; Rémy & Yakobowski, 2010; Cretin & Rémy, 2012).

9 Conclusion

Our focus has been on Haskell, for the sake of concreteness, but we believe that this work is important beyond the Haskell community. Any language that offers *both* generative type abstraction *and* type-level computation must deal with their interaction, and those interactions are extremely subtle. We have described one sound and tractable way to combine the two, including the source language changes, type inference, core calculus, and metatheory. In doing so we have given a concrete foundation for others to build upon.

Acknowledgements

Thanks to Antal Spector-Zabusky for contributing to this version of FC; and to Edward Kmett and Dimitrios Vytiniotis for discussion and feedback. This material is based upon work supported by the National Science Foundation under grant nos. CCF-1116620 and CCF-1319880. The first author was supported by the Deutsche Telekom Stiftung and the second author was supported by a Microsoft Research PhD Fellowship. This paper is typeset with the help of ott (Sewell *et al.*, 2010).

References

Breitner, Joachim, Eisenberg, Richard A., Peyton Jones, Simon, & Weirich, Stephanie. (2014a). Safe zero-cost coercions for haskell. *ICFP*.

Breitner, Joachim, Eisenberg, Richard A., Peyton Jones, Simon, & Weirich, Stephanie. (2014b). *Safe zero-cost coercions for Haskell (extended version)*. Tech. rept. MS-CIS-14-07. University of Pennsylvania.

Chakravarty, Manuel M. T., Keller, Gabriele, & Peyton Jones, Simon. (2005a). Associated type synonyms. *Pages 241–253 of: ICFP*. ACM.

Chakravarty, Manuel M. T., Keller, Gabriele, Peyton Jones, Simon, & Marlow, Simon. (2005b). Associated types with class. *Pages 1–13 of: POPL*. ACM.

Cheney, James, & Hinze, Ralf. (2003). *First-class phantom types*. Tech. rept. Cornell University.

Crary, Karl. (2000). Typed compilation of inclusive subtyping. *Pages 68–81 of: Icfp '00: Proceedings of the fifth acm sigplan international conference on functional programming*.

Cretin, Julien, & Rémy, Didier. (2012). On the power of coercion abstraction. *Pages 361–372 of: Proceedings of the 39th annual acm sigplan-sigact symposium on principles of programming languages*. POPL '12. New York, NY, USA: ACM.

Eisenberg, Richard A., Vytiniotis, Dimitrios, Peyton Jones, Simon, & Weirich, Stephanie. (2014). Closed type families with overlapping equations. *Pages 671–683 of: POPL*. ACM.

Garrigue, Jacques. 2013 (Sept.). *On variance, injectivity, and abstraction*. OCaml Meeting, Boston.

Hall, Cordelia V., Hammond, Kevin, Peyton Jones, Simon L., & Wadler, Philip L. (1996). Type classes in haskell. *Acm trans. program. lang. syst.*, **18**(2).

Hinze, Ralf, & Peyton Jones, Simon. (2000). Derivable type classes. *Haskell workshop*.

Marlow (editor), Simon. (2010). *Haskell 2010 language report*.

Milner, Robin, Tofte, Mads, Harper, Robert, & MacQueen, David. (1997). *The definition of Standard ML (revised)*.

Peyton Jones, Simon, Tolmach, Andrew, & Hoare, Tony. (2001). Playing by the rules: rewriting as a practical optimisation technique in GHC. *Pages 203–233 of: Haskell Workshop*.

Pottier, François, & Rémy, Didier. (2005). The essence of ML type inference. *Chap. 10, pages 389–489 of:* Pierce, Benjamin C. (ed), *Advanced topics in types and programming languages*. MIT Press.

Rémy, Didier, & Yakobowski, Boris. (2010). A church-style intermediate language for mlf. *Pages 24–39 of:* Blume, Matthias, Kobayashi, Naoki, & Vidal, Germán (eds), *Functional and logic programming*. Lecture Notes in Computer Science, vol. 6009. Springer Berlin Heidelberg.

Rodriguez, Alexey, Jeuring, Johan, Jansson, Patrik, Gerdes, Alex, Kiselyov, Oleg, & Oliveira, Bruno C. d. S. (2008). Comparing libraries for generic programming in haskell. *Pages 111–122 of: Proceedings of the first acm sigplan symposium on haskell*. Haskell '08. New York, NY, USA: ACM.

Scherer, Gabriel, & Rémy, Didier. (2013). GADTs meet subtyping. *Pages 554–573 of: ESOP*.

Sewell, Peter, Zappa Nardelli, Francesco, Owens, Scott, Peskine, Gilles, Ridge, Thomas, Sarkar, Susmit, & Strniša, Rok. (2010). Ott: Effective tool support for the working semanticist. *Journal of functional programming*, **20**(1).

Sheard, Tim, & Peyton Jones, Simon. (2002). Template meta-programming for Haskell. *Pages 1–16 of: Proc. 2002 acm sigplan workshop on haskell*. Haskell '02. ACM.

Sulzmann, Martin, Chakravarty, Manuel M. T., Peyton Jones, Simon, & Donnelly, Kevin. (2007). System F with type equality coercions. *Types in languages design and implementation*. TLDI '07. ACM.

Terei, David, Marlow, Simon, Peyton Jones, Simon, & Mazières, David. (2012). Safe haskell. *Haskell '12*. ACM.

Vytiniotis, Dimitrios, Peyton Jones, Simon, Schrijvers, Tom, & Sulzmann, Martin. (2011). OutsideIn(X) modular type inference with local assumptions. *Journal of functional programming*, **21**(4-5).

Weirich, Stephanie, Vytiniotis, Dimitrios, Peyton Jones, Simon, & Zdancewic, Steve. (2011). Generative type abstraction and type-level computation. *Pages 227–240 of: POPL*. ACM.

Yorgey, Brent A., Weirich, Stephanie, Cretin, Julien, Peyton Jones, Simon, Vytiniotis, Dimitrios, & Magalhães, José Pedro. (2012). Giving Haskell a promotion. *Pages 53–66 of: TLDI*. ACM.

A 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.

A.1 The remainder of the grammar

Φ	$::=$	$[\overline{a{:}\kappa}].\tau \sim_\rho \sigma$	axiom types
e	$::=$		expressions
	$\mid$	v	value
	$\mid$	x	variable
	$\mid$	$e_1\, e_2$	application
	$\mid$	$e\,\tau$	type application
	$\mid$	$e\,\gamma$	coercion application
	$\mid$	$\mathbf{case}_\tau\ e\ \mathbf{of}\ \overline{alt}$	pattern match
	$\mid$	$e \triangleright \gamma$	cast
	$\mid$	$\mathbf{contra}\,\gamma\,\tau$	absurdity
v	$::=$		expression values
	$\mid$	$\lambda x{:}\tau.e$	value abstraction
	$\mid$	$\Lambda a{:}\kappa.e$	type abstraction
	$\mid$	$\lambda c{:}\phi.e$	coercion abstraction
	$\mid$	$K\,\overline{\tau}\,\overline{\gamma}\,\overline{e}$	applied data constructor
alt	$::=$	$K\,\overline{a}\,\overline{c}\,\overline{x} \to e$	alternative in pattern match
ψ	$::=$		value types
	$\mid$	D	data type (*not* **newtype**s!)
	$\mid$	$(\to)$	arrow
	$\mid$	$(\Rightarrow)$	prop. arrow
	$\mid$	$(\sim_\rho^\kappa)$	equality
	$\mid$	$\forall a{:}\kappa.\ \tau$	polymorphism
	$\mid$	$\psi\,\tau$	application

A.2 Typing judgements

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\frac{\begin{array}{l}\Gamma \vdash e : \forall a{:}\kappa.\, \sigma \\ \Gamma \vdash \tau : \kappa\end{array}}{\Gamma \vdash e\, \tau : \sigma[\tau/a]} \ \textsc{Tm\_TApp}$$

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

$$\dfrac{\begin{array}{l}\Gamma \vdash e : (\sigma_1 \sim_\rho \sigma_2) \Rightarrow \tau \\ \Gamma \vdash \gamma : \sigma_1 \sim_\rho \sigma_2\end{array}}{\Gamma \vdash e\,\gamma : \tau} \quad \textsc{Tm\_CApp}$$

$$\dfrac{\vdash \Gamma \qquad K : \tau}{\Gamma \vdash K : \tau} \quad \textsc{Tm\_DataCon}$$

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

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

$$\dfrac{\begin{array}{l}\varnothing \vdash \gamma : H_1 \sim_{\mathsf{N}} H_2 \qquad H_1 \neq H_2 \\ \Gamma \vdash \tau : \star\end{array}}{\Gamma \vdash \mathbf{contra}\,\gamma\,\tau : \tau} \quad \textsc{Tm\_Contra}$$

A.3 Small-step operational semantics

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

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

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

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

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

$$\dfrac{}{(v \rhd \gamma_1) \rhd \gamma_2 \longrightarrow v \rhd (\gamma_1 \fatsemi \gamma_2)} \quad \textsc{S\_Trans}$$

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

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

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

$$\frac{e \longrightarrow e'}{\mathbf{case}_\tau\ e\ \mathbf{of}\ \overline{alt} \longrightarrow \mathbf{case}_\tau\ e'\ \mathbf{of}\ \overline{alt}}\quad \textsc{S\_Case\_Cong}$$

$$\frac{e \longrightarrow e'}{e \rhd \gamma \longrightarrow e' \rhd \gamma}\quad \textsc{S\_Cast\_Cong}$$

$$\frac{\begin{array}{l}\eta_1 = \mathbf{sym}\,(\mathbf{nth}^1\ \eta_0) \qquad\qquad \eta_2 = \mathbf{nth}^2\ \eta_0\\ \varnothing \vdash v : \sigma_1 \to \sigma_2\end{array}}{(v \rhd \eta_0)\,e' \longrightarrow v\,(e' \rhd \eta_1) \rhd \eta_2}\quad \textsc{S\_Push}$$

$$\frac{\begin{array}{l}\varnothing \vdash v : \forall a{:}\kappa.\ \sigma'\\ \varnothing \vdash \tau : \kappa\end{array}}{(v \rhd \gamma)\,\tau \longrightarrow v\,\tau \rhd \gamma@\tau}\quad \textsc{S\_TPush}$$

$$\frac{\begin{array}{l}\eta_{11} = \mathbf{nth}^1\ (\mathbf{nth}^1\ \eta_0) \qquad\qquad \eta_{12} = \mathbf{nth}^2\ (\mathbf{nth}^1\ \eta_0)\\ \eta_2 = \mathbf{nth}^2\ \eta \qquad\qquad \gamma'' = \eta_{11} \fatsemi \gamma' \fatsemi \mathbf{sym}\,\eta_{12}\\ \varnothing \vdash v : \sigma_1 \sim^\kappa_\rho \sigma_2 \Rightarrow \sigma_3 \qquad\qquad \varnothing \vdash \gamma' : \sigma_4 \sim^\kappa_\rho \sigma_5\end{array}}{(v \rhd \eta_0)\,\gamma' \longrightarrow v\,\gamma'' \rhd \eta_2}\quad \textsc{S\_CPush}$$

$$\frac{\begin{array}{l}\varnothing \vdash \eta : D\,\overline{\tau} \sim_{\mathsf{R}} D\,\overline{\tau}'\\ K : \forall\overline{a{:}\kappa}.\ \forall\overline{b{:}\kappa'}.\ \overline{(\sigma' \sim_\rho \sigma'')} \Rightarrow \overline{\tau}'' \to D\,\overline{a}\\ \overline{\varnothing \vdash \gamma : (\sigma' \sim_\rho \sigma'')[\overline{\tau/a}][\overline{\sigma/b}]}\\ \overline{\gamma' = \mathbf{sym}\,(\sigma'[\overline{\mathbf{nth}\,\eta/a}]_\rho) \fatsemi \gamma \fatsemi \sigma''[\overline{\mathbf{nth}\,\eta/a}]_\rho}\\ \overline{e' = e \rhd \tau''[\overline{\mathbf{nth}\,\eta/a}]_{\mathsf{R}}}\end{array}}{\mathbf{case}_{\tau_0}\ (K\,\overline{\tau}\,\overline{\sigma}\,\overline{\gamma}\,\overline{e}) \rhd \eta\ \mathbf{of}\ \overline{alt} \longrightarrow \mathbf{case}_{\tau_0}\ K\,\overline{\tau}'\,\overline{\sigma}\,\overline{\gamma}'\,\overline{e}'\ \mathbf{of}\ \overline{alt}}\quad \textsc{S\_KPush}$$

B 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 $C : [\overline{a{:}\kappa}].\tau \sim_\rho \sigma$:
   (a) $\overline{a{:}\kappa \vdash \tau : \kappa_0}$
   (b) $\overline{a{:}\kappa \vdash \sigma : \kappa_0}$
2. The global context may contain $T : \kappa$.
3. The global context may contain $K : \tau$:
   (a) $\tau = \forall\overline{a{:}\kappa}.\ \forall\overline{b{:}\kappa'}.\ \overline{\phi} \Rightarrow \overline{\sigma} \to D\,\overline{a}$
   (b) $\varnothing \vdash \tau : \star$
4. The global context may contain $F : [\overline{a{:}\kappa}].\kappa_0$.
5. For all H, $roles(H) \models H$.

C Properties of roles

Lemma 2 (Permutation of role checking). *If $\Omega \vdash \tau : \rho$ and Ω' is a permutation of Ω, then $\Omega' \vdash \tau : \rho$.*

Lemma 3 (Weakening of role checking). *If $\Omega \vdash \tau : \rho$, then $\Omega, a{:}\rho' \vdash \tau : \rho$.*

Lemma 4 (Strengthening of role checking). *If $\Omega, a{:}\rho' \vdash \tau : \rho$ and a does not appear free in τ, then $\Omega \vdash \tau : \rho$.*

Lemma 5 (Nominal roles are infectious). *Let $\overline{a}$ be the free variables in σ. We have $\Omega \vdash \sigma : \mathsf{N}$ if and only if every $a_i \in \overline{a}$ is at role N in Ω.*

Lemma 6 (Sub-roling). *If $\Omega \vdash \tau : \rho$ and $\rho \leq \rho'$, then $\Omega \vdash \tau : \rho'$.*

D Structural properties

D.1 Weakening

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

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

Lemma 7 (Type kinding weakening). *If $\Gamma, \Gamma' \vdash \tau : \kappa$ and $\vdash \Gamma, bnd, \Gamma'$, then $\Gamma, bnd, \Gamma' \vdash \tau : \kappa$.*

Lemma 8 (Coercion typing weakening). *If $\Gamma, \Gamma' \vdash \gamma : \phi$ and $\vdash \Gamma, bnd, \Gamma'$, then $\Gamma, bnd, \Gamma' \vdash \gamma : \phi$.*

Lemma 9 (Term typing weakening). *If $\Gamma, \Gamma' \vdash e : \tau$ and $\vdash \Gamma, bnd, \Gamma'$, then $\Gamma, bnd, \Gamma' \vdash e : \tau$.*

D.2 Substitution

Lemma 10 (Type variable substitution). *Suppose $\Gamma \vdash \sigma : \kappa_1$. Then:*

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

Lemma 11 (Type variable substitution in coercions). *If $\Gamma, a{:}\kappa, \Gamma' \vdash \gamma : \phi$ and $\Gamma \vdash \sigma : \kappa$, then $\Gamma, \Gamma'[\sigma/a] \vdash \gamma[\sigma/a] : \phi[\sigma/a]$.*

Lemma 12 (Type variable substitution in terms). *If $\Gamma, a{:}\kappa, \Gamma' \vdash e : \tau$ and $\Gamma \vdash \sigma : \kappa$, then $\Gamma, \Gamma'[\sigma/a] \vdash e[\sigma/a] : \tau[\sigma/a]$.*

Lemma 13 (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 14 (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 15 (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 16 (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 17 (Term strengthening in coercions). *If* $\Gamma, x{:}\tau, \Gamma' \vdash \gamma : \phi$*, then* $\Gamma, \Gamma' \vdash \gamma : \phi$.

Lemma 18 (Term substitution). *If* $\Gamma, x{:}\sigma, \Gamma' \vdash e : \tau$ *and* $\Gamma \vdash e' : \sigma$*, then* $\Gamma, \Gamma' \vdash e[e'/x] : \tau$.

D.3 Context regularity

Lemma 19 (Type context regularity for types). *If* $\Gamma \vdash \tau : \kappa$*, then* $\vdash \Gamma$.

Lemma 20 (Coercion context regularity). *If* $\Gamma \vdash \gamma : \phi$*, then* $\vdash \Gamma$.

Lemma 21 (Term context regularity). *If* $\Gamma \vdash e : \tau$*, then* $\vdash \Gamma$.

D.4 Classifier regularity

Lemma 22 (Coercion typing regularity). *If* $\Gamma \vdash \gamma : \tau \sim_\rho \sigma$*, then* $\Gamma \vdash \tau \sim_\rho \sigma : \star$.

Lemma 23 (Coercion homogeneity). *If* $\Gamma \vdash \gamma : \tau \sim_\rho \sigma$*, then* $\Gamma \vdash \tau : \kappa$ *and* $\Gamma \vdash \sigma : \kappa$.

Proof. Direct from Lemma 22. □

Lemma 24 (Term typing regularity). *If* $\Gamma \vdash e : \tau$*, then* $\Gamma \vdash \tau : \star$.

D.5 Determinacy

Lemma 25 (Uniqueness of type kinding). *If* $\Gamma \vdash \tau : \kappa_1$ *and* $\Gamma \vdash \tau : \kappa_2$ *then* $\kappa_1 = \kappa_2$.

Lemma 26 (Uniqueness of coercion typing). *If* $\Gamma \vdash c : \phi_1$ *and* $\Gamma \vdash c : \phi$ *then* $\phi_1 = \phi_2$.

Lemma 27 (Uniqueness of term typing). *If* $\Gamma \vdash e : \tau_1$ *and* $\Gamma \vdash e : \tau_2$ *then* $\tau_1 = \tau_2$.

Lemma 28 (Determinacy of evaluation). *If* $e \longrightarrow e_1$ *and* $e \longrightarrow e_2$ *then* $e_1 = e_2$.

E Preservation

E.1 Lifting

Lifting is defined by the following algorithm, with patterns to be tried in order from top to bottom. Note that the context Γ is an implicit argument of this function.

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

Lemma 29 (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* $\overline{b'}$ *is the type variables in* Γ, Γ'
   $\Omega = \overline{b'}{:}\mathsf{N}, \overline{b} : \mathit{roles}(H)$;
6. $\Gamma, \overline{b{:}\kappa}, \Gamma' \vdash \sigma_0 : \kappa'$; *and*
7. *Γ' contains only type variable bindings.*

then:

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

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 $\sigma_0[\overline{\mathbf{nth}\, \gamma/b}]_{\mathsf{P}} = \langle \sigma_0[\overline{\tau/b}], \sigma_0[\overline{\sigma/b}]\rangle_{\mathsf{P}}$, and we are done, applying Lemma 10.

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

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

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

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

Case ($a = b_i$):

Case ($\rho_0 = \rho_i$): Here, we have $\sigma_0[\overline{\mathbf{nth}\, \gamma/b}]_{\rho_0} = \mathbf{nth}^i\, \gamma$, $\sigma_0[\overline{\tau/b}] = \tau_i$, and $\sigma_0[\overline{\sigma/b}] = \sigma_i$. 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 $\Omega \vdash a : \mathsf{N}$. By inversion then, we know $a{:}\mathsf{N} \in \Omega$. Yet, we know that ρ_i is the ith role in $\mathit{roles}(H)$, and by the definition of Ω, $a{:}\rho_i \in \Omega$. This contradicts $\rho_i \neq \mathsf{N}$, and we are done.

Case ($a \notin \overline{b}$):

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

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

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

□

E.2 Preservation

Theorem 30 (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 $alt_i = K\overline{a}\,\overline{c}\,\overline{x} \to e'$. We must show $\Gamma \vdash e'[\overline{\sigma/a}][\overline{\gamma/c}][\overline{e/x}] : \tau_0$. By inversion on Tm_Case, we see

$$\Gamma \vdash K\overline{\tau}\,\overline{\sigma}\,\overline{\gamma}\,\overline{e} : D\,\overline{\tau}$$

$$K : \forall\overline{a':\kappa}.\ \forall\overline{b':\kappa'}.\ \overline{\phi} \Rightarrow \overline{\tau}' \to D\,\overline{a'}$$

$$\Gamma,\overline{a{:}\kappa'},\overline{c{:}\phi[\overline{\tau/a'}][\overline{a/b'}]},\overline{x{:}\tau'[\overline{\tau/a'}][\overline{a/b'}]} \vdash e' : \tau_0$$

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

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

This simplifies to

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

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

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

and

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

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

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

using the following facts

$$\begin{aligned} \tau_0[\overline{\sigma/a}] &= \tau_0 \\ \phi[\overline{\tau/a'}][\overline{a/b'}][\overline{\sigma/a}] &= \phi[\overline{\tau/a'}][\overline{\sigma/b'}] \\ \tau'[\overline{\tau/a'}][\overline{a/b'}][\overline{\sigma/a}] &= \tau'[\overline{\tau/a'}][\overline{\sigma/b'}] \end{aligned}$$

So, we have

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

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

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

and

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

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

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

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

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

as desired.

Case S_Trans: We know that $\Gamma \vdash (v \triangleright \gamma_1) \triangleright \gamma_2 : \tau$ and need to show that $\Gamma \vdash v \triangleright (\gamma_1 \fatsemi \gamma_2) : \tau$. Inversion gives us $\Gamma \vdash v : \sigma_1$, $\Gamma \vdash \gamma_1 : \sigma_1 \sim_{\mathsf{R}} \sigma_2$, and $\Gamma \vdash \gamma_2 : \sigma_2 \sim_{\mathsf{R}} \tau$. Straightforward use of typing rules shows that $\Gamma \vdash v \triangleright (\gamma_1 \fatsemi \gamma_2) : \tau$, as desired.

Congruence rules: By induction.

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

$$\frac{\eta_1 = \mathbf{sym}\,(\mathbf{nth}^1\,\eta_0) \qquad \eta_2 = \mathbf{nth}^2\,\eta_0 \\ \varnothing \vdash v : \sigma_1 \to \sigma_2}{(v \triangleright \eta_0)\,e' \longrightarrow v\,(e' \triangleright \eta_1) \triangleright \eta_2}\ \textsc{S\_Push}$$

We know that $\Gamma \vdash (v \triangleright \eta_0)\, e' : \sigma_4$ and we must show $\Gamma \vdash (v\,(e' \triangleright \eta_1)) \triangleright \eta_2 : \sigma_4$. Inversion tells us that $\Gamma \vdash \eta_0 : (\sigma_1 \to \sigma_2) \sim_{\mathsf{R}} (\sigma_3 \to \sigma_4)$ and $\Gamma \vdash e' : \sigma_3$. We can now see that $\Gamma \vdash \eta_1 : \sigma_3 \sim_{\mathsf{R}} \sigma_1$ and $\Gamma \vdash \eta_2 : \sigma_2 \sim_{\mathsf{R}} \sigma_4$. Thus, $\Gamma \vdash e' \triangleright \eta_1 : \sigma_1$ and $\Gamma \vdash v\,(e' \triangleright \eta_1) \triangleright \eta_2 : \sigma_4$ as desired.

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

$$\frac{\begin{array}{l}\varnothing \vdash v : \forall a{:}\kappa.\ \sigma' \\ \varnothing \vdash \tau : \kappa\end{array}}{(v \triangleright \gamma)\, \tau \longrightarrow v\, \tau \triangleright \gamma @ \tau}\ \text{S\_TPUSH}$$

We know that $\Gamma \vdash (v \triangleright \gamma)\, \tau : \tau'$ and we must show that $\Gamma \vdash v\, \tau \triangleright \gamma @ \tau : \tau'$. Inversion tells us that $\Gamma \vdash \gamma : (\forall a{:}\kappa.\ \sigma') \sim_{\mathsf{R}} (\forall a{:}\kappa.\ \sigma'')$ where $\tau' = \sigma''[\tau/a]$. We can see that $\Gamma \vdash \gamma @ \tau : \sigma'[\tau/a] \sim_{\mathsf{R}} \sigma''[\tau/a]$ and thus that $\Gamma \vdash v\, \tau \triangleright \gamma @ \tau : \tau'$ as desired.

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

$$\frac{\begin{array}{ll}\eta_{11} = \mathbf{nth}^1\ (\mathbf{nth}^1\ \eta_0) & \eta_{12} = \mathbf{nth}^2\ (\mathbf{nth}^1\ \eta_0) \\ \eta_2 = \mathbf{nth}^2\ \eta \qquad \gamma'' = \eta_{11} \mathbin{;} \gamma' \mathbin{;} \mathbf{sym}\, \eta_{12} & \\ \varnothing \vdash v : \sigma_1 \sim^{\kappa}_{\rho} \sigma_2 \Rightarrow \sigma_3 & \varnothing \vdash \gamma' : \sigma_4 \sim^{\kappa}_{\rho} \sigma_5\end{array}}{(v \triangleright \eta_0)\, \gamma' \longrightarrow v\, \gamma'' \triangleright \eta_2}\ \text{S\_CPUSH}$$

We know that $\Gamma \vdash (v \triangleright \eta_0)\, \gamma' : \sigma_6$ and we must show that $\Gamma \vdash v\, \gamma'' \triangleright \eta_2 : \sigma_6$. Inversion tells us that $\Gamma \vdash \eta_0 : (\sigma_1 \sim_\rho \sigma_2 \Rightarrow \sigma_3) \sim_{\mathsf{R}} (\sigma_4 \sim_\rho \sigma_5 \Rightarrow \sigma_6)$. We can now see the following:

$$\begin{array}{c}\Gamma \vdash \eta_{11} : \sigma_1 \sim_\rho \sigma_4 \\ \Gamma \vdash \eta_{12} : \sigma_2 \sim_\rho \sigma_5 \\ \Gamma \vdash \eta_2 : \sigma_3 \sim_\rho \sigma_6 \\ \Gamma \vdash \gamma'' : \sigma_1 \sim_\rho \sigma_2\end{array}$$

Thus $\Gamma \vdash v\, \gamma'' \triangleright \eta_2 : \sigma_6$ as desired.

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

$$\frac{\begin{array}{l}\varnothing \vdash \eta : D\, \overline{\tau} \sim_{\mathsf{R}} D\, \overline{\tau}' \\ K : \forall \overline{a{:}\kappa}.\ \forall \overline{b{:}\kappa'}.\ \overline{(\sigma' \sim_\rho \sigma'')} \Rightarrow \overline{\tau}'' \to D\, \overline{a} \\ \overline{\varnothing \vdash \gamma : (\sigma' \sim_\rho \sigma'')[\overline{\tau/a}][\overline{\sigma/b}]} \\ \overline{\gamma' = \mathbf{sym}\, (\sigma'[\overline{\mathbf{nth}\, \eta/a}]_\rho) \mathbin{;} \gamma \mathbin{;} \sigma''[\overline{\mathbf{nth}\, \eta/a}]_\rho} \\ \overline{e' = e \triangleright \tau''[\overline{\mathbf{nth}\, \eta/a}]_{\mathsf{R}}}\end{array}}{\mathbf{case}_{\tau_0}\ (K \overline{\tau}\, \overline{\sigma}\, \overline{\gamma}\, \overline{e}) \triangleright \eta\ \mathbf{of}\ \overline{alt} \longrightarrow \mathbf{case}_{\tau_0}\ K \overline{\tau}'\, \overline{\sigma}\, \overline{\gamma}'\, \overline{e}'\ \mathbf{of}\ \overline{alt}}\ \text{S\_KPUSH}$$

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 $\Gamma \vdash (K \overline{\tau}'\, \overline{\sigma}\, \overline{\gamma}'\, \overline{e}') : D\, \overline{\tau}'$. Note that τ_0 and the $\overline{alt}$ do not change, so we need not worry about them here.

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 23) (and more induction on $\overline{\tau}'$), we see that

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

Putting this together, we get

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

or

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

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

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

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

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

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

$$\overline{\Gamma \vdash \gamma : \phi[\overline{\tau/a}][\overline{\sigma/b}]}$$

$$\overline{\Gamma \vdash e : \tau''[\overline{\tau/a}][\overline{\sigma/b}]}$$

We need to conclude the following:

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

$$\overline{\Gamma \vdash e' : \tau''[\overline{\tau'/a}][\overline{\sigma/b}]}$$

We wish to use the lifting lemma (Lemma 29) to get types for $\overline{\sigma'[\overline{\mathbf{nth}\,\eta/a}]_\rho}$ and $\overline{\sigma''[\overline{\mathbf{nth}\,\eta/a}]_\rho}$. 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, $\Omega = \overline{b'{:}\mathsf{N}},\overline{a} : \mathit{roles}(D)$ where $\overline{b'}$ are the type variables bound in Γ, along with the existential variables $\overline{b}$. (That is, the Γ' in the statement of the lifting lemma is $\overline{b{:}\kappa'}$.) 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 6).
6. $\Gamma,\overline{a{:}\kappa},\overline{b{:}\kappa'} \vdash \sigma' : \kappa''$ and the same for σ'': This comes from the well-formedness of the global context, including the type of K.
7. $\overline{b{:}\kappa'}$ must contain only type variable bindings: It sure does.

Now, we can conclude

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

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

We then do type variable substitution to get

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

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

Now, by CO_TRANS, we can conclude

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

as desired.

To type the $\overline{e'}$, we need to apply the lifting lemma once again, this time to $\overline{\tau''[\overline{\mathbf{nth}\,\eta/a}]_{\mathsf{R}}}$. 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. $\overline{\Gamma, \overline{a{:}\kappa}, \overline{b{:}\kappa'} \vdash \tau'' : \kappa''}$: This comes from the well-formedness of the global context, including the type of K.

We then apply the lifting lemma to conclude that

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

We use type variable substitution to get

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

We can then conclude

$$\overline{\Gamma \vdash e' : \tau''[\overline{\tau'/a}][\overline{\sigma/b}]}$$

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.

□

F Progress

We prove progress by first establishing that the global context is *consistent* (defined below). We do this by placing further restrictions on the global context and proving that these imply consistency. However, these restrictions are needed only for consistency, and it is possible to relax or change these in future versions of FC, as long as the consistency property holds by some mechanism.

F.1 Restrictions on axioms

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

1. Newtype axioms: All of the following must hold.
   (a) $\tau = N\,\overline{a}$
   (b) $\rho = \mathsf{R}$
   (c) There must not be two axioms mentioning the same newtype N.
   (d) The length of $\mathit{roles}(N)$ must match the arity of the axiom C.
2. Type family axioms: All of the following must hold.
   (a) $\tau = F(\overline{\tau}')$
   (b) $\rho = \mathsf{N}$
   (c) The types $\overline{\tau}'$ must not mention type families.
   (d) Each $b \in \overline{a}$ must appear exactly once in the list $\overline{\tau}'$.
   (e) Consider two axioms $C_1 : [\overline{a{:}\kappa}].\tau_1 \sim_\rho \sigma_1$ and $C_2 : [\overline{b{:}\kappa'}].\tau_2 \sim_\rho \sigma_2$ (where variables are renamed so that $\overline{a} \cap \overline{b} = \varnothing$). Then, if there exists some θ with $\theta(\tau_1) = \theta(\tau_2)$, it must be that $\theta(\sigma_1) = \theta(\sigma_2)$.

F.2 Consistency

Definition 31 (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 $\tau_1 = \forall a{:}\kappa.\ \sigma$ then $\tau_2 = \forall a{:}\kappa.\ \sigma'$.*

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 32 (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 nondeterministic type reduction relation $\tau \rightsquigarrow_\rho \sigma$, show that the relation preserves value type heads (when ρ is not phantom), 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

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

$$\frac{\begin{array}{l}\tau_1 \rightsquigarrow_\rho \sigma_1\\ \tau_2 \rightsquigarrow_{\mathsf{N}} \sigma_2\end{array}}{\tau_1\,\tau_2 \rightsquigarrow_\rho \sigma_1\,\sigma_2}\ \textsc{Red\_App}$$

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

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

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

$$\frac{\begin{array}{l} C : [\overline{a{:}\kappa}].\tau_1 \sim_\rho \tau_2 \\ \rho \leq \rho' \end{array}}{\tau_1[\overline{\sigma/a}] \rightsquigarrow_{\rho'} \tau_2[\overline{\sigma/a}]}\quad \textsc{Red\_Axiom}$$

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

Lemma 33 (Simple rewrite substitution). *If $\tau_1 \rightsquigarrow_\rho \tau_2$, then $\tau_1[\sigma/a] \rightsquigarrow_\rho \tau_2[\sigma/a]$.*

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

Lemma 34 (Rewrite substitution). *Let $\overline{a}$ be the free variables in a type σ. If $\overline{a{:}\rho} \vdash \sigma : \mathsf{R}$:*

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

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

Case $\sigma = a$: There is thus only one free variable, a 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 $H\,\overline{\sigma}[\overline{\tau/a}] \rightsquigarrow_{\mathsf{R}} H\,\overline{\sigma}[\overline{\tau'/a}]$. We will use Red_TyConApp. Let $\overline{\rho}'$ be a prefix of *roles*(H) of the same length as $\overline{\sigma}$. We must show $\overline{\sigma[\overline{\tau/a}] \rightsquigarrow_{\rho'} \sigma[\overline{\tau'/a}]}$.
Fix i. We will show that $\sigma_i[\overline{\tau/a}] \rightsquigarrow_{\rho'_i} \sigma_i[\overline{\tau'/a}]$.

Case ($\rho'_i = \mathsf{N}$): In order to use the induction hypothesis, we must show that for every j such that a_j appears free in σ_i, $\rho_j = \mathsf{N}$. To use Lemma 5, 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 $\overline{a{:}\rho} \vdash H\,\overline{\sigma} : \mathsf{R}$ 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 4 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 6 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 $H\,\overline{\sigma}[\overline{\tau/a}] \rightsquigarrow_{\mathsf{N}} H\,\overline{\sigma}[\overline{\tau'/a}]$. 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 $(\sigma_1\,\sigma_2)[\overline{\tau/a}] \rightsquigarrow_{\mathsf{R}} (\sigma_1\,\sigma_2)[\overline{\tau'/a}]$. 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 $\sigma_1[\overline{\tau/a}] \rightsquigarrow_{\mathsf{R}} \sigma_1[\overline{\tau'/a}]$. To use the induction hypothesis for σ_2, we must first establish that, for every j such that a_j appears free in σ_2, $\tau_j \rightsquigarrow_{\mathsf{N}} \tau'_j$. Lemma 5 provides exactly this information, so we get $\sigma_2[\overline{\tau/a}] \rightsquigarrow_{\mathsf{N}} \sigma_2[\overline{\tau'/a}]$. We are done by RED_APP.

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

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

We are done by RED_APP.

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

Case $\sigma = \forall b{:}\kappa.\ \sigma'$: We assume that we have renamed variables so that $b \notin \overline{a}$. We see that inverting $\Omega \vdash \forall b{:}\kappa.\ \sigma' : \mathsf{R}$ gives us $\Omega, b{:}\mathsf{N} \vdash \sigma' : \mathsf{R}$, where $\overline{a}, b$ 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 5 to see that $\overline{\rho = \mathsf{N}}$. We then use the induction hypothesis repeatedly to get

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

We are now done by RED_TYFAM.

□

Lemma 35 (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 36 (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 35) 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 $i = (\text{length of } \overline{\tau}) + 1$. 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 35.

Case RED_APP: We get our first two desiderata from use of the induction hypothesis and our last from Lemma 35.

Case RED_TYCONAPP: Our first two desiderata come from the premises of RED_TYCONAPP, and the last one comes from Lemma 35.

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

□

Lemma 37 (Pattern)**.** *Let $\overline{a}$ be the free variables in a a type τ. We require that each variable a is mentioned exactly once in τ and that no type families appear in τ. Then, if, for some $\overline{\sigma}$, $\tau[\overline{\sigma/a}] \rightsquigarrow_{\mathsf{N}} \tau'$, then there exist $\overline{\sigma}'$ such that $\tau' = \tau[\overline{\sigma'/a}]$ and $\overline{\sigma \rightsquigarrow_{\mathsf{N}} \sigma'}$.*

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

Case $\tau = a$: There is just one free variable (a), 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 $\overline{b_1}$ that appear in τ_1 and $\overline{b_2}$ 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 $\tau_1[\overline{\sigma_1/b_1}]\,\tau_2[\overline{\sigma_2/b_2}] \rightsquigarrow_{\mathsf{N}} \tau'$. Thus must be by RED_APP (noting that all newtype axioms are at role R). Thus, $\tau' = \tau_1'\,\tau_2'$ and $\tau_1[\overline{\sigma_1/b_1}] \rightsquigarrow_{\mathsf{N}} \tau_1'$ and $\tau_2[\overline{\sigma_2/b_2}] \rightsquigarrow_{\mathsf{N}} \tau_2'$. We then use the induction hypothesis to get $\overline{\sigma}_1'$ and $\overline{\sigma}_2'$ such that $\tau_1' = \tau_1[\overline{\sigma_1'/b_1}]$ and $\tau_2' = \tau_2[\overline{\sigma_2'/b_2}]$. 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 $\tau = \forall b{:}\kappa.\ \tau_0$: We first note that, according to the definition of $\overline{a}$, $b \notin \overline{a}$. We wish to use the induction hypothesis, but we must be careful because τ_0 may mention b multiple times. So, we linearise τ_0 into τ_0', replacing every occurrence of b with fresh variables $\overline{b'}$. (Note that $\overline{b'}$ can be empty.) We know that $(\forall b{:}\kappa.\ \tau_0)[\overline{\sigma/a}] \rightsquigarrow_{\mathsf{N}} \tau'$. We note that $(\forall b{:}\kappa.\ \tau_0)[\overline{\sigma/a}] = \forall b{:}\kappa.\ (\tau_0[\overline{\sigma/a}]) = \forall b{:}\kappa.\ (\tau_0'[\overline{\sigma/a}][\overline{b/b'}])$. (We have abused notation somewhat in the second substitution. There is only one b; it is substituted for every variable in $\overline{b'}$.) Let $\overline{\sigma}''$ be $\overline{\sigma}$ appended with the right number of copies of b. Let $\overline{a'}$ be $\overline{a}$ appended with $\overline{b'}$. Then, we can say $\forall b{:}\kappa.\ (\tau_0'[\overline{\sigma''/a'}]) \rightsquigarrow_{\mathsf{N}} \tau'$. We invert to get that $\tau' = \forall b{:}\kappa.\ \tau''$ and $\tau_0'[\overline{\sigma''/a'}] \rightsquigarrow_{\mathsf{N}} \tau''$. We can now use the induction hypothesis to get $\overline{\sigma}'''$ such that $\tau' = \tau[\overline{\sigma'''/a'}]$ and $\overline{\sigma'' \rightsquigarrow_{\mathsf{N}} \sigma'''}$. But, we can see that, b steps only to itself. Thus, the last entries in $\overline{\sigma}'''$ must be the same list of bs that $\overline{\sigma}''$ has. We let σ' be the prefix of $\overline{\sigma}'''$ without the bs, and we are done.

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

□

Lemma 38 (Patterns)**.** *Let $\overline{a}$ be the free variables in a list of types $\overline{\tau}$. Assume each variable a is mentioned exactly once in $\overline{\tau}$ and that no type families appear in $\overline{\tau}$. If, for some $\overline{\sigma}$, $\overline{\tau[\overline{\sigma/a}] \rightsquigarrow_{\mathsf{N}} \tau'}$, then there exist $\overline{\sigma'}$ such that $\overline{\tau' = \tau[\overline{\sigma'/a}]}$ 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 39 (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 $\tau = a$: 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 36 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 $C : [\overline{a{:}\kappa}].H\,\overline{a} \sim_{\mathsf{R}} \sigma_0$, where $\sigma_2 = \sigma_0[\overline{\tau_0/a}]$.

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 restrictions on axioms say 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 $\sigma_3 = \sigma_0[\overline{\sigma'/a}]$ by RED_AXIOM. Now, we must show $\sigma_2 \rightsquigarrow_{\mathsf{R}} \sigma_3$. We wish to apply the rewrite-substitution lemma (Lemma 34). We must show that $\overline{\tau_0 \rightsquigarrow_\rho \sigma'}$, where $\overline{a{:}\rho} \vdash \sigma_0 : \mathsf{R}$. 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 $\mathit{roles}(H)$, by context well-formedness. So, we can use Lemma 36 to get $\overline{\tau_0 \rightsquigarrow_{\rho} \sigma'}$, as desired. We then apply Lemma 34 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 $\mathit{roles}(H)$ and $\overline{\tau_0 \rightsquigarrow_{\rho} \sigma'}$. We also see that $C : [\overline{a{:}\kappa}].H\,\overline{a} \sim_{\mathsf{R}} \sigma_0$ and that $\sigma_2 = \sigma_0[\overline{\tau_0/a}]$.

Let $\sigma_3 = \sigma_0[\overline{\sigma'/a}]$. We can see that $\sigma_1 \rightsquigarrow_{\mathsf{R}} \sigma_3$ by Red_Axiom. And, by Lemma 34 (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: $C : [\overline{a{:}\kappa}].H\,\overline{a} \sim_{\mathsf{R}} \sigma_0$. 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 $\sigma_1 = \sigma_0[\overline{\sigma/a}]$ and $\sigma_2 = \sigma_0[\overline{\sigma'/a}]$. 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 $\tau = \forall a{:}\kappa.\ \tau'$: 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 $\sigma_2 = \sigma_0[\overline{\sigma'/a}]$ where $C : [\overline{a{:}\kappa}].F(\overline{\tau}') \sim_{\mathsf{N}} \sigma_0$ and $\overline{\tau} = \overline{\tau}'[\overline{\sigma'/a}]$.

We wish to use Red_Axiom to reduce $F(\overline{\sigma})$. We apply Lemma 38 to get $\overline{\sigma}''$ such that $\overline{\sigma} = \overline{\tau}'[\overline{\sigma''/a}]$ and $\overline{\sigma' \rightsquigarrow_{\mathsf{N}} \sigma''}$. We then use Red_Axiom to get $\sigma_1 \rightsquigarrow_{\mathsf{N}} \sigma_3$, where $\sigma_3 = \sigma_0[\overline{\sigma''/a}]$. Now, we must show that $\sigma_2 \rightsquigarrow_{\mathsf{N}} \sigma_3$. This comes directly from Lemma 34, and we are done.

Case Red_Axiom/Red_Axiom:

We have $C_1 : [\overline{a{:}\kappa}].F(\overline{\tau}_1) \sim_{\mathsf{N}} \sigma_1'$ and $C_2 : [\overline{b{:}\kappa'}].F(\overline{\tau}_2) \sim_{\mathsf{N}} \sigma_2'$. We also know that $\tau = F(\overline{\tau}_1)[\overline{\sigma'/a}]$ and $\tau = F(\overline{\tau}_2)[\overline{\sigma''/b}]$. Thus, $F(\overline{\tau}_1)[\overline{\sigma'/a}] = F(\overline{\tau}_2)[\overline{\sigma''/b}]$. Thus, $[\overline{\sigma}',\overline{\sigma}''/\overline{a},\overline{b}]$ is a unifier for $F(\overline{\tau}_1)$ and $F(\overline{\tau}_2)$. Thus, by context well-formedness, we have $\sigma_1'[\overline{\sigma'/a}] = \sigma_2'[\overline{\sigma''/b}]$. But, $\sigma_1 = \sigma_1'[\overline{\sigma'/a}]$ and $\sigma_2 = \sigma_2'[\overline{\sigma''/b}]$, 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 40 (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 39. □

Lemma 41 (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 42 (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 41, 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 43 (Completeness of the rewrite relation)**.** *If Γ binds no coercion variables and $\Gamma \vdash \gamma : \tau_1 \sim_\rho \tau_2$, then $\tau_1 \Leftrightarrow_\rho \tau_2$.*

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

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:

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

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 40 (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 has no coercion variables.
Case Co_Axiom: We are done by Red_Axiom.
Case Co_Nth: We adopt the variable names in the rule:

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

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 36, 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:

$$\frac{\Gamma \vdash \gamma : \tau_1\,\tau_2 \sim_{\mathsf{N}} \sigma_1\,\sigma_2 \qquad \Gamma \vdash \tau_1 : \kappa \qquad \Gamma \vdash \sigma_1 : \kappa}{\Gamma \vdash \mathbf{left}\,\gamma : \tau_1 \sim_{\mathsf{N}} \sigma_1}\ \text{CO\_LEFT}$$

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:

$$\frac{\Gamma \vdash \gamma : \forall a{:}\kappa.\ \tau_1 \sim_\rho \forall a{:}\kappa.\ \sigma_1 \qquad \Gamma \vdash \tau : \kappa}{\Gamma \vdash \gamma @ \tau : \tau_1[\tau/a] \sim_\rho \sigma_1[\tau/a]}\ \text{CO\_INST}$$

We know by induction that $\forall a{:}\kappa.\ \tau_1 \Leftrightarrow_\rho \forall a{:}\kappa.\ \sigma_1$. We can easily see by inspection of the rewrite relation that the common reduct must have the form $\forall a{:}\kappa.\ \tau_0$ 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 $\tau_1[\tau/a] \rightsquigarrow^*_\rho \tau_0[\tau/a]$ and $\sigma_1[\tau/a] \rightsquigarrow^*_\rho \tau_0[\tau/a]$. These facts come from an induction over the lengths of the derivations and the use of the simple rewrite substitution lemma, Lemma 33.

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

$$\frac{\Gamma \vdash \gamma : \tau \sim_{\mathsf{N}} \sigma}{\Gamma \vdash \mathbf{sub}\,\gamma : \tau \sim_{\mathsf{R}} \sigma}\ \text{CO\_SUB}$$

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 35.

□

Lemma 44 (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 43), we see that $\tau_1 \Leftrightarrow_{\mathsf{R}} \tau_2$. But, the rewrite relation consistency lemma (Lemma 42) tells us that τ_1 and τ_2 are consistent. Thus, the context admits only consistent coercions and is itself consistent. □

F.3 Progress

Lemma 45 (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* $\varnothing \vdash v : \forall a{:}\kappa.\ \tau$*, then* v *is either* $\Lambda a{:}\kappa.e'$ *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* $K\,\overline{\tau}\,\overline{\gamma}\,\overline{e}$*.*

Lemma 46 (Value types)**.** *If* $\Gamma \vdash v : \tau$*, then* τ *is a value type.*

Proof. If v is an abstraction, then the result is trivial. So, we assume that $v = K\,\overline{\tau}\,\overline{\gamma}\,\overline{e}$. Induction on the lengths of the lists of arguments yields

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

We can see (again, by induction on the argument lists) that no matter what K is applied to, its type will always be a value type, headed by one of $\forall$, $\Rightarrow$, $\to$ or D, all of which form value types. □

Theorem 47 (Progress)**.** *If* $\varnothing \vdash e : \tau$*, then either* e *is a value or a coerced 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, a coerced 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.

If e_1 is a coerced value $v \rhd \gamma$, then by the value types lemma (Lemma 46) and the consistency lemma (Lemma 44), the type of v must be headed by $(\to)$. We are done by S_PUSH.
Case TM_TAPP: Similar to previous case.
Case TM_CAPP: Similar to previous cases.
Case TM_DATACON: e is a value.
Case TM_CASE: We adopt the variable names from the rule:

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

We know by induction that e is a value, a coerced value, or $e \longrightarrow e'$ for some e'. If e steps, then we are done by S_CASE_CONG.
We know that T must actually be a data type (not a newtype), because it has a constructor. Thus, e has a value type. Therefore, if it has the form $v \rhd \gamma$, the value

v has a type headed by T as well. Thus $v = K\,\overline{\tau}\,\overline{\gamma}\,\overline{e}$ and we apply S_KPUSH, noting that the premises are all satisfied by straightforward use of typing judgements. The final case is 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, a coerced value, or $e \longrightarrow e'$.
If e steps, we are done by S_CAST_CONG.
If e is a value, then $e \triangleright \gamma$ is a coerced value, and we are done.
If e is a coerced value, then we are done by S_TRANS.

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

$$\frac{\begin{array}{ll}\varnothing \vdash \gamma : H_1 \sim_{\mathsf{N}} H_2 & \quad H_1 \neq H_2 \\ \Gamma \vdash \tau : \star & \end{array}}{\Gamma \vdash \textbf{contra}\,\gamma\,\tau : \tau}\ \textsc{Tm\_Contra}$$

By completeness of the rewrite relation (Lemma 43), 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.

□

G Role inference

Lemma 48 (Walking)**.** *Let $\overline{a}$ be the parameters to some type constant T. For some type σ, let $\overline{b}$ be the free variables in σ that are not in $\overline{a}$. Let $\overline{\rho}$ be a list of roles of the same length as $\overline{a}$. Let $\Omega = \overline{a{:}\rho}, \overline{b{:}\mathsf{N}}$.*

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

Proof. By induction on the structure of σ:

Case $\sigma = a'$: By assumption, it must be that $a'{:}\mathsf{R} \in \Omega$ or $a'{:}\mathsf{N} \in \Omega$. In either case, we can derive $\Omega \vdash a' : \mathsf{R}$, 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 $\overline{\rho}' = \mathit{roles}(H')$. 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 5, 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 5.

Case $\sigma = H$: We are done by immediate application of RTY_TYCONAPP.

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

Case $\sigma = F(\overline{\tau})$: Repeated use of Lemma 5 tells us that $\overline{\Omega \vdash \tau : \mathsf{N}}$. We are done by RTY_TYFAM.

□

Theorem 49. *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 50 (Role inference is sound)**.** *After running the role inference algorithm, $roles(H) \models H$ will hold for all H.*

Proof. We handle the data type 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

$$K : \forall \overline{a{:}\kappa}.\ \forall \overline{b{:}\kappa'}.\ \overline{\phi} \Rightarrow \overline{\sigma} \rightarrow D\ \overline{a}$$

Let $\overline{\rho} = roles(D)$ and $\Omega = \overline{a{:}\rho}, \overline{b{:}\mathsf{N}}$. We have satisfied the premises of the walking lemma (Lemma 48), 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 51 (Stumbling)**.** *Let $\overline{a}$ be the parameters to some type constant T. For some type σ, let $\overline{b}$ be the free variables in σ that are not in $\overline{a}$. Let $\overline{\rho}$ be a list of roles of the same length as $\overline{a}$. Let $\Omega = \overline{a{:}\rho}, \overline{b{:}\mathsf{N}}$.*

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

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 5. 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 6.

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 5) and thus we cannot use RTy_App.

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

Case $\sigma = \forall a'{:}\kappa. \, \sigma_1$: We are done by induction, noting that in RTy_ForAll, a' gets assigned role N when checking σ_1. This matches our expectations that the type variables $\overline{b}$ 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 5. Thus, we would be unable to use RTy_TyFam.

□

Theorem 52 (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 data types, so we need only concern ourselves with Ts, not general Hs. In both the data type and newtype cases, showing $roles(T) \models T$ requires showing $\Omega \vdash \sigma : \mathsf{R}$, where $\Omega = \overline{a{:}\rho}, \overline{b{:}\mathsf{N}}$ and $\overline{a}$ are the parameters to T and $\overline{b}$ are the remaining free variables of σ. (In the newtype case, $\overline{b}$ 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 51 shows and so we are done. □