—preprint—

Safe Coercions

Joachim Breitner
Karlsruhe Institute of Technology
breitner@kit.edu

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

Simon Peyton Jones
Microsoft Research
simonpj@microsoft.com

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

Abstract

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

1. Introduction

Modular languages support *generative type abstraction*, the ability for programmers to define application-specific types, and rely on the type system to distinguish between these new types and their underlying representation. Type abstraction is a powerful tool for programmers, enabling both flexibility (implementors can change representations) and security (implementors can maintain invariants about representations). Typed languages provide these mechanisms with zero run-time cost – there should be no performance penalty for creating abstractions – using mechanisms such as ML's module system [7] and Haskell's **newtype** declaration [6].

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

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

Although String values use the same patterns of bits in memory as HTML values, the two types are distinct. That is, a String will not be accepted by a function expecting an HTML. The constructor Mk converts a String to an HTML (see function text), while using Mk in a pattern converts in the other direction (see function unMk). Furthermore, by exporting the type HTML, but not its data constructor, module Html ensures that the type HTML is *abstract* – clients cannot make arbitrary strings into HTML – and thereby prevent cross-site scripting attacks.

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

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

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

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

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

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

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

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

- Our new approach also resolves a notorious and longstanding bug in GHC (#1496), which concerns the interaction of newtype coercions with type families (Section 7). While earlier work [12] was motivated by the same bug, it was too complicated to implement. Our new approach finds a sweet spot, offering a much simpler system in exchange for a minor loss of expressiveness (Section 8).
- We have implemented role inference and safe coercions in GHC (Section 6), and we show how the usual machinery of rewrite rules can be used to bring these improvements to existing, unmodified code. (Section 6.4)

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

2. The design and interface of Coercible

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

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

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

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

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

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

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

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

GHC generates the following instances of Coercible:

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

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

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

   the instance

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

Figure 1. Coercible instances

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

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

2.1 Coercing newtypes

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

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

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

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

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

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

```
instance Coercible a a
```

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

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

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

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

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

2.2 Coercing under type constructors

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

Roles are a new concept, not previously part of Haskell, and are a key contribution of this paper (although see related work in Section 8). There are three possible roles, *representational*, *phantom* or *nominal*, which are discussed in the following subsections.

2.2.1 Coercing representational type parameters

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

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

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

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

or even

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

Why a single instance is not enough

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

```
instance Coercible Int Age
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.2.2 Coercing phantom type parameters

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

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

both have b at a phantom role.

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

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

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

2.2.3 Coercing nominal type parameters

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

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

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

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

All the parameters of a type or data *family* have nominal role, since they are potentially inspected by the type-family instances. Type classes also use nominal role for their type parameters; see Section 3.4.

2.2.4 Coercing multiple type parameters

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

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

yields the instance

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

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

3. Abstraction and coherence

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

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

3.1 Abstraction and unwrapping newtypes

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

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

3.2 Lifting over abstractions

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

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

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

3.3 Abstraction through role annotations

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

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

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

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

```
type role Map nominal representational
```

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

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

that preserves the abstraction of Map.

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

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

3.4 Preserving class coherence

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

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

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

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

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

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

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

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

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

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

4. Ensuring type safety: System FC with roles

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

Core is an implementation of a calculus called System FC, itself an extension of the classical Girard/Reynolds System F. A full exposition of FC[2] is beyond the scope of this work, but it is well documented elsewhere (e.g. Yorgey et al. [13]).

Metavariables:

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

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

Figure 2. An excerpt of the grammar of System FC

Figure 2 gives the syntax of System FC. The starting point is an entirely conventional lambda calculus in the style of System F. We therefore elide most of the syntax of terms e, giving the typing judgement for terms in the extended version of this paper.[3] Types τ are also conventional, except that we add (saturated) type-family applications $F(\overline{\tau})$, to reflect their addition to source Haskell [1, 2]. Types are classified by kinds κ in the usual way; the kinding judgement $\Gamma \vdash \tau : \kappa$ on types is conventional and appears in the extended version of this paper. To avoid clutter we use only monomorphic kinds, but it is easy to add kind polymorphism along the lines of Yorgey et al. [13], and our implementation does so.

4.1 Roles and casts

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

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

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

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

[2] Several versions of System FC are described in published work. Some of these variants have had decorations to the FC name, such as FC$_2$ or F$_C^{\uparrow}$. We do not make these distinctions in the present work, referring instead to all of these systems – in fact, one evolving system – as "FC".

[3] http://www.cis.upenn.edu/~eir/papers/2014/coercible-ext.pdf

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

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

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

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

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

4.2 Coercions

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

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

4.2.1 Nominal implies representational

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

4.2.2 Equality is an equivalence relation

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

4.2.3 Equality is hypothetical

One of the key attributes of System FC is that it is possible to *assume* an equality. Specifically, each newtype declaration, and each type-family instance, gives rise to an FC *axiom*. Newtypes give rise to representational axioms, and type-family instances give rise to nominal axioms.[4] For example, the declarations

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

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

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

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

$$\frac{\begin{array}{l}\Gamma \vdash \gamma_1 : \tau_1 \sim_{\rho} \tau_2 \\ \Gamma \vdash \gamma_2 : \tau_2 \sim_{\rho} \tau_3\end{array}}{\Gamma \vdash \gamma_1 \fatsemi \gamma_2 : \tau_1 \sim_{\rho} \tau_3} \quad \text{Co\_Trans}$$

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

$$\frac{\begin{array}{l}\overline{\Gamma \vdash \gamma : \tau \sim_{\mathsf{N}} \sigma} \\ \Gamma \vdash F(\overline{\tau}) : \kappa \qquad\qquad \Gamma \vdash F(\overline{\sigma}) : \kappa\end{array}}{\Gamma \vdash F(\overline{\gamma}) : F(\overline{\tau}) \sim_{\mathsf{N}} F(\overline{\sigma})} \quad \text{Co\_TyFam}$$

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

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

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

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

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

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

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

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

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

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

Figure 3. Formation rules for coercions

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

produce the axioms

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

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 $\mathsf{Maybe}\,\sigma$ (meaning that the two are considered to be the same type by the type checker). In C_2, the notation "$[\alpha{:}\star]$." binds α in the types being equated. Uses of these axioms are governed by the rule CO_AXIOM. Axioms must always appear fully applied, and we assume that they live in a global context, separate from the local context Γ.

4.2.4 Equality can be abstracted

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

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

4.2.5 Equality is congruent

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

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

```
newtype HTML = Mk String

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

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

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

1. Should $\mathsf{Maybe\,HTML} \sim_{\mathsf{R}} \mathsf{Maybe\,String}$ 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 should be different.
4. Should $\alpha\,\mathsf{HTML} \sim_{\mathsf{R}} \alpha\,\mathsf{String}$ hold, for some type variable α? It depends on the instantiation of α! If α becomes Maybe, then "yes"; if α becomes T, then "no". Since we may be abstracting over α, we do not know which of the two will happen, so we take the conservative stance and say that $\alpha\,\mathsf{HTML} \sim_{\mathsf{R}} \alpha\,\mathsf{String}$ does *not* hold.

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

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

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

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

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

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

4.2.6 Equality can be decomposed

If we have a proof of $\mathsf{Maybe}\,\sigma \sim_\rho \mathsf{Maybe}\,\tau$, 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 injective! 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 5.) Yet, we can see the following chain of equalities:

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

By transitivity, we can derive a coercion γ witnessing

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

If we could use $\mathbf{nth}^2$ 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. We need a new newtype to see why this must be so:

```
newtype EitherInt a = MkEI (Either a Int)
```

This definition yields an axiom showing that, for all a, $\mathsf{EitherInt\ a} \sim_{\mathsf{R}} (\mathsf{Either\ a\ Int})$. 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]

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

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

4.3 Role attribution for type constants

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

All of this is standard except for roles. It is essential that the roles of D's parameters, $\mathit{roles}\,(D)$, are consistent with D's definition. For example, it would be utterly wrong for the global environment to claim that $\mathit{roles}\,(\mathsf{Maybe}) = P$, 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 $\mathit{roles}\,(H) \models H$ for all H. The rules for this judgement and two auxiliary judgements appear in Figure 4.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Finally $(\to)$ and $(\Rightarrow)$ have representational roles: functions care about representational equality but never branch on the nominal identity of a type. (For example, functions al-

[5] We note in passing that the forms **left** and **right** are present merely to increase expressivity. They are not needed anywhere in the metatheory to prove type soundness. Though originally part of FC, they were omitted in previous versions [12] and even in the implementation. Haskell users then found that some desirable program were no longer type-checking. Thus, these forms were re-introduced.

ways treat HTML and String identically.) We also see that the roles of the arguments to an equality proposition match the role of the proposition. This fact comes from congruence of the respective equality relations.

4.4 Metatheory

The preceding discussion gave several non-obvious examples where admitting *too many* coercions would lead to unsoundness. However, we must have *enough* coercions to allow us to make progress when evaluating a program. (We do not have space to elaborate, but a key example is the use of **nth** in rule S_KPUSH, presented in the extended version of this paper.) Happily, we can be confident that we have enough coercions, but not too many, because we prove the usual progress and preservations theorems for System FC. The structure of the proofs follows broadly that in previous work, such as Weirich et al. [12] or Yorgey et al. [13].

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 the rewrite system to be confluent and to preserve 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.

However, we are left with a major problem: the rewrite relation is *not*, confluent! The non-linear patterns allowed in type families (that is, with a repeated variable on the left-hand side), combined with non-termination, break the confluence property. Our previous work [4] gives more details. However, losing confluence does not necessarily threaten consistency – it just threatens the particular proof technique we use. However, a more powerful proof appears to be an open problem in the term rewriting community,[6] a problem we dodge by restricting type families to have only linear patterns, thus leading to confluence.

The full proof of type safety appears in the extended version of this paper; it exhibits no new proof techniques.

5. Role inference

In our type safety proof, we assume that $roles\,(H) \models H$ for all H, but where did these roles come from? We must do *role inference* during compilation to build the environment from which $roles\,(H)$ pulls its roles.

The role inference algorithm is quite straightforward. At a high level, it simply starts with the role information of the built-in constants $(\rightarrow)$, $(\Rightarrow)$, and $(\sim_\rho)$, and propagates the roles until it finds a fixpoint.

The exact algorithm appears below. In the description of the algorithm, we assume a mutable environment; $roles\,(H)$ pulls a list of roles from this environment. Only after the algorithm is complete will $roles\,(H) \models H$ hold.

1. Begin by assuming that all parameters to all datatypes have a phantom role. That is, let $roles\,(T)$ be an appropriately-sized list of P roles, for any T.
2. For every datatype D, every constructor for that datatype K, and every coercion type and term-level argument type σ to that constructor: run walk (D, σ).
3. For every newtype N with representation type σ, run walk (N, σ).
4. If the role of any parameter to any type constant changed in the previous steps, go to step 2.

The procedure walk (T, σ) is defined as follows, matching from top to bottom:

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

When marking, we must follow these two rules:

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

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

Theorem. *The role inference algorithm always terminates.*

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

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

Proofs of these theorems appear in the extended version of this paper.

6. Implementing Coercible

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

6.1 Coercible and coerce

When the compiler transforms Haskell to Core, type classes become regular types and typeclass constraints turn into function arguments [11]. Type classes without constraints become simple product types with one field per method. The built-in type class Coercible is a bit different: It wraps the primitive witness of representational equality $\sim_\mathsf{R}$ in a datatype:

```
data Coercible a b = MkCoercible (a ~R b)
```

The definition of coerce, which is possible to give only in Core, pattern-matches on MkCoercible to get hold of the equality witness, and then uses Core's primitive cast operation:

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

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

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

Why can we not simply expose the type $\sim_R$ to users, instead of introducing the type Coercible? Our primary motivator is to use Haskell's existing type-class mechanism (with its automatic instance lookup) to express representational equalities – we wouldn't want programmers to have to build the equalities themselves! This mechanism, in turn, requires *boxed* types, and $\sim_R$ is not boxed.

In GHC, a boxed type is one whose run-time representation is a pointer to the actual value of interest. Using such a representation allows for a straightforward implementation of polymorphism, for example, because we know the all boxed types take the same amount of space on the function stack. The type $\sim_R$, however, is unboxed. It is represented at run-time by a 0-bit field – that is, by nothing at all. After all, types are erased during compilation, so there is no need to keep around an equality proof in a running program. GHC requires all constraint types to be boxed. The requirement enables language features like constraint kinds, which allow for constraint polymorphism [13]. The relationship between $\sim_R$ and Coercible is exactly analogous to the wrapping of boxed nominal equalities necessary to implement deferred type errors [10].

With Coercible being just a regular data type, at first glance it seems that you can break type safety, by writing (in Core)

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

but because coerce evaluates the Coercible argument (see the definition of coerce above), looksUnsound will simply not terminate. Again, this follows the behavior of deferred type errors [10].

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

6.2 Instance generation and solving

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

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

When a newtype uses higher-rank types, as in

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

its newtype unwrapping instances take a form that is usually illegal, even with all GHC extensions enabled: The type used in the instance head is a forall-quantified type.

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

When do two forall-quantified types **forall** a. T1 a and **forall** a. T2 a have the same run-time representation? When they have the same representation, no matter what type is being used for a. Therefore, when GHC sees a constraint of the form Coercible (**forall** a. T1 a) (**forall** b. T2 b), it tries to solve the constraint Coercible (T1 a) (T2 a) where a is a fresh type variable. Because a is fresh, nothing can be assumed about it by the type checker and the result will be valid for any type used in place of a.

6.3 Preventing circular reasoning and diverging instances

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

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

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

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

But with Coercible, such circular reasoning would be problematic; we could then seemingly write unsafeCoerce:

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

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

In practice, however, the unsafeCoerce written above would not work: it would simply loop. This is exactly because coerce requires evaluating its equality witness (that is, instance of Coercible). In unsafeCoerce above, the equality witness is a looping term, so coerce itself would loop. We have not actually done an end-run around type safety.

However, the compiler still treats Coercible specially in this regard: while solving constraints, it keeps track of the *depth* of the constraint – that is, how many steps led to requiring this particular constraint. When solving a Coercible instance at depth n, only instances at depth greater than n are considered. GHC already uses a check to make sure that instance selection does not loop forever, using a depth cutoff. These checks together ensure that there are no cycles in the solver's reasoning about Coercible. With this in place, GHC eliminates the implementation of unsafeCoerce above at compile time instead of introducing a hard-to-find looping term in a user's program.

6.4 Coercible and rewrite rules

What if a client of module Html writes this?

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

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

The correctness of transforming (map unMk) to coerce depends on a theorem about map, which a compiler can hardly be expected to identify and prove all by itself. Fortunately GHC already comes with a mechanism that allows a library author to specify *rewrite rules* for their code [8]. 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/coerce"
        map coerce = coerce  #-}
```

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

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

7. Generalized Newtype Deriving done right

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

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

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

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

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

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

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

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

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

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

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

Now the process is very mechanical: Apply coerce to the method, instantiated at the base type by a type signature. And because it is so mechanical, the compiler can do it for her; all she has to do is to declare which instances of the base type should be transferred to the new type by listing them in the **deriving** clause:

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

Now, GHC can write the instance declaration shown above.

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 the corresponding manual instance and coerce.

Furthermore, GND was previously unsound [12]! When combined with other extensions of GHC, such as type families [1, 2] or GADTs [3], GND could be exploited to completely break the type system: Figure 5 shows how this notorious bug can allow any type to be coerced to any other. The clause "**deriving** (UnsafeCast b)" is the bogus use of GND, and now will generate the instance

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

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

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

8. Related work

Prior work [12] discusses the relationship between roles in FC and languages with generativity and abstraction, type-indexed constructs, as well as 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 that prior work, as well as make connections to other systems.

8.1 Prior version of roles

The idea of *roles* was initially developed by Weirich et al. [12] as a solution to the Generalized Newtype Deriving problem.

(We will refer to that work as WVPZ.) That work introduces the equality relations $\sim_{\mathsf{R}}$ and $\sim_{\mathsf{N}}$ (called "type equality" and "code equality" resp. in WVPZ). However, the system presented in WVPZ was drastic: it required annotating every sub-tree of every kind with a role. The pervasiveness of the change is one of the reasons that the system was never implemented.

In this paper, we present a substantially simplified version of the roles system, requiring role information only on the parameters to datatypes. Though the simplification decreases expressiveness in places, it also increases flexibility.

The key to this simplification is to "assume the worst" about higher-kinded parameters. Because this system only tracks roles for constants, compared to WVPZ it gives up the ability to abstract over type constructors with non-nominal argument roles. In practice, we have found few places where this limitation makes a difference.

One instance that would benefit from the increased precision of WVPZ comes from Edward Kmett's linear library which defines the type

newtype Point f a = P (f a)

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

The observation that we can treat all higher-kinded parameters as themselves taking nominal arguments does increase the flexibility of the system compared to WVPZ. In that work, 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 would be no problem as Maybe and T have the same kind.

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

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

8.2 OCaml and variance annotations

The interactions between sub-typing, type abstraction, and various type system extensions such as GADTs and parameter constraints also appear in the OCaml language. In that context, *variance annotations* act like roles; they ensure that subtype coercions between compatible types are safe. For example, the type α `list` of immutable lists is covariant in the parameter α: if $\sigma \leq \tau$ then σ `list` $\leq \tau$ `list`. Variances form a lattice, with *invariant*, the most restrictive, at the bottom; *covariant* and *contravariant* incomparable; and *bivariant* at the top, allowing sub-typing in both directions. It is tempting to identify invariant with nominal and bivariant with phantom, but the exact connection is unclear. Scherer and Rémy [9] 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[7] that has been confirmed to date back several years. Garrigue [5] discusses these issues. His proposed solution is to "assume that nothing is known about abstract types when they are used in parameter constraints and GADT return types" – akin to assigning nominal roles. However, this solution is too conservative, and in practice the OCaml 4.01 compiler relies on no fewer than *six* flags to describe the variance of type parameters. However, lacking anything equivalent to Core and its tractable metatheory, the OCaml developers cannot demonstrate the soundness of their solution in the way that we have done here.

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

Acknowledgements Thanks to Antal Spector-Zabusky for his contributions to this version of System FC and Edward Kmett for discussions on the design of Coercible and of roles.

References

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

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

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

[4] R. A. Eisenberg, D. Vytiniotis, S. Peyton Jones, and S. Weirich. Closed type families with overlapping equations. POPL '14, pages 671–683, New York, NY, USA, 2014. ACM.

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

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

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

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

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

[10] D. Vytiniotis, S. Peyton Jones, and J. P. Magalhães. Equality proofs and deferred type errors: A compiler pearl. ICFP '12, pages 341–352, New York, NY, USA, 2012. ACM.

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

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

[13] B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving Haskell a promotion. TLDI '12, pages 53–66, New York, NY, USA, 2012. ACM.

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