Draft submitted to ICFP '15

An overabundance of equality: Implementing kind equalities into Haskell

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

Abstract

Haskell, as embodied by version 7.8.4 of the Glasgow Haskell Compiler (GHC), supports reasoning about equality among types, via generalized algebraic datatypes (GADTs) and type families. However, these features are not available among the kinds that classify the types. Motivated by a concrete example of how kind equalities can help programmers today, this paper presents the challenges and solutions encountered in integrating kind equalities into GHC, an industrial-strength compiler. The challenges addressed here all surround the many notions of type equality that exist in GHC today, and in particular around GHC's role mechanism. These many different relations on types complicate the theory considerably and should serve as a caution to designers of other languages considering adding new equality-like relations among types.

An update of GHC supporting reasoning about kind equalities is a key part of this work.

1. Introduction

Today's Haskell[1] has an intriguing limitation.

The following declaration, straight from the standard library module *Data.Type.Equality*, defines an equality witness:

```
data (a :: k) :~: (b :: k) where
  Refl :: ∀ (a :: k). a :~: a
```

Pattern matching on this generalized algebraic datatype (GADT) [11, 20] allows a programmer to assert the equality between two types:

```
castWith :: ∀ (a :: k) (b :: k). (a :~: b) → a → b
castWith Refl x = x
```

In the definition of *castWith*, we pattern-match on *Refl*. This exposes the fact that a and b must, in fact, be the same. Then, GHC happily uses x of type a in a context expecting something of type b. All is good.

However, the following, very similar definition, is rejected:

```
data (a :: k1) :≈: (b :: k2) where
  HRefl :: a :≈: a
```

The only difference is in the kinds of the type arguments. Homogeneous equality (:~:) takes two parameters, a and b, both of some kind k. Heterogeneous equality (:≈:), on the other hand, takes its parameters of *different* kinds, k_1 and k_2. Thus, pattern-matching on *HRefl* should yield both that the kinds k_1 and k_2 are equal and that the types a and b are equal. Such a definition is quite useful for enabling Haskell to operate in a distributed, cloud-based setting; see Section 2.1 for the details, as well as other motivation.

The restriction above exists because GHC reasons about only type equality, never kind equality. When a programmer uses *Refl* at type $a :\sim: b$ in an expression, GHC must create a proof that a indeed equals b. This proof is then passed as an implicit argument to the *Refl* constructor. Conversely, when matching on *Refl*, GHC unpacks this implicit argument and is able to use it when type-checking the body of the pattern match. However, GHC currently has no notion of kind equalities, so there is no equivalent proof that k_1 equals k_2 to pack and unpack. Relatedly, today's Haskell does not support kind families – functions that take and return kinds – nor promoting GADTs to the kind level [21].

The solution to all of these problems is simple to state: merge the concepts of *type* and *kind*. If types and kinds are the same, then we surely have kind equalities. We can go one step further: adding the $\star :: \star$ axiom and avoiding an infinite hierarchy of sorts; see Section 9.1 for discussion about this design choice.

The solution introduced above is described in detail in previous work [19], which develops an enhanced internal language and proves it type safe. The current work goes beyond in that it deals in pragmatics. GHC/Haskell is a large, complex beast: it supports unlifted types (Section 4), roles (Sections 5-6), open and closed type families (Section 7), among many other features. (GHC also requires type inference, which is beyond the scope of this work, but see Gundry [7].)

In this paper, I describe how the internal language from that previous work fits into GHC, resolving the thorny issues that arose in the process. A common thread among these pain points is that they all deal with different notions of type equality. As GHC has evolved, it has required several distinct notions of equality among types, and it is here that the challenge of implementing kind equality comes to a head.

I see this work as a cautionary tale about the complexities that can arise when a language supports multiple notions of equality. Though the technical content of this paper concerns Haskell, this note of caution applies to any language with a strong typing discipline.

I offer the following contributions:

- Sections 4 through 7 present the primary challenges in incorporating kind equalities into GHC, along with their solutions.

[1] Throughout this paper, I use "Haskell" to describe the language implemented by the Glasgow Haskell Compiler (GHC) version 7.8.4.

[Copyright notice will appear here once 'preprint' option is removed.]

- In order to overcome those challenges, it has been necessary to augment GHC's internal language, System FC. Section 3 introduces an updated version of the language, and the relevant lemmas of the proof of type safety appears in the submitted auxiliary material.
- I have made available an implementation of kind equalities in GHC.[2] The implementation is capable of compiling all of the examples in Weirich et al. [19], GHC itself, and the standard libraries, and it fares admirably on the GHC testsuite. See also Section 8 for implementation notes.

2. Motivation

2.1 Cloud Haskell

Cloud Haskell [5] is an ongoing project, aiming to support writing a Haskell program that can operate on several machines in parallel, communicating over a network. Naturally, we would like to do so in a type-safe manner. To do so, we need a way of communicating types over a wire – a runtime type representation. Here is our desired representation:

```
data TyCon (a :: k)
    -- abstract; Int is represented by a TyCon Int
data TypeRep (a :: k) where
  TyCon :: TyCon a → TypeRep a
  TyApp :: TypeRep a → TypeRep b → TypeRep (a b)
```

For every new type declared, the compiler would supply an appropriate value of the *TyCon* datatype. The type representation library would supply also the following function, which computes equality over *TyCon*s, returning the heterogeneous equality from the introduction:

```
eqTyCon :: ∀ (a :: k1) (b :: k2).
           TyCon a → TyCon b → Maybe (a :≈: b)
```

It is critical that this function returns (:≈:), not (:∼:). This is because *TyCon*s exist at many different kinds. For example, *Int* is at kind $\star$, and *Maybe* is at kind $\star \to \star$. Thus, when comparing two *TyCon* representations for equality, we want to learn whether the types *and the kinds* are equal. If we used (:∼:) here, then the *eqTyCon* could be used only when we know, from some other source, that the kinds are equal.

We can now easily write an equality test over these type representations:

```
eqT :: ∀ (a :: k1) (b :: k2).
       TypeRep a → TypeRep b → Maybe (a :≈: b)
eqT (TyCon t1)     (TyCon t2)     = eqTyCon t1 t2
eqT (TyApp a1 b1) (TyApp a2 b2)
  | Just Refl ← eqT a1 a2
  , Just Refl ← eqT b1 b2          = Just Refl
eqT _              _               = Nothing
```

Now that we have a type representation with computable equality, we can package that representation with a chunk of data, and so form a dynamically typed package:

```
data Dyn where
  Dyn :: ∀ (a :: ⋆). TypeRep a → a → Dyn
```

The *a* type variable there is an *existential* type variable. We can think of this type as being part of the data payload of the *Dyn* constructor; it is chosen at construction time and unpacked at pattern-match time. Because of the *TypeRep a* argument, we can learn more about *a* after unpacking. (Without the *TypeRep a* or some other type-level information about *a*, the unpacking code must treat *a* as an unknown type and must be parametric in the choice of type for *a*.)

Using *Dyn*, we can pack up arbitrary data along with its type, and push that data across a network. The receiving program can then make use of the data, without needing to subvert Haskell's type system. The type representation library must be trusted to recreate the *TypeRep* on the far end of the wire, but the equality tests above and other functions below can live outside the trusted code base.

Suppose we were to send an object with a function type, say *Bool* → *Int* over the network. For the time being, let's ignore the complexities of actually serializing a function – there is a solution to that problem[3], but here we are concerned only with the types. We would want to apply the received function to some argument. What we want is this:

```
dynApply :: Dyn → Dyn → Maybe Dyn
```

The function *dynApply* applies its first argument to the second, as long as the types line up. The definition of this function is fairly straightforward:

```
dynApply :: Dyn → Dyn → Maybe Dyn
dynApply (Dyn (TyApp
                 (TyApp (TyCon tarrow) targ)
                 tres)
              fun)
         (Dyn targ' arg)
    | Just Refl ← eqTyCon tarrow (tyCon :: TyCon (→))
    , Just Refl ← eqT targ targ'
    = Just (Dyn tres (fun arg))
dynApply _ _ = Nothing
```

We first match against the expected type structure – the first *Dyn* argument must be a function type. We then confirm that the *TyCon tarrow* is indeed the representation for (→) (the construct *tyCon* :: *TyCon* (→) retrieves the compiler-generated representation for (→)) and that the actual argument type matches the expected argument type. If everything is good so far, we succeed, applying the function in *fun arg*.

Heterogeneous equality is necessary throughout this example. It first is necessary in the definition of *eqT*. In the *TyApp* case, we compare a_1 to a_2. If we had only homogeneous equality, it would be necessary that the types represented by a_1 and a_2 be of the same kind. Yet, we can't know this here! Even if the types represented by *TyApp* a_1 b_1 and *TyApp* a_2 b_2 have the same kind, it is possible that a_1 and a_2 would not. (For example, maybe the type represented by a_1 has kind $\star \to \star$ and the type represented by a_2 has kind *Bool* $\to \star$.) With only homogeneous equality, we cannot even write an equality function over this form of type representation. The problem repeats itself in the definition of *dynApply*, when calling *eqTyCon tarrow TArrow*. The call to *eqT* in *dynApply*, on the other hand, *could* be homogeneous, as we would know at that point that the types represented by *targ* and *targ'* are both of kind $\star$.

In today's Haskell, the lack of heterogeneous equality means that *dynApply* must rely critically on *unsafeCoerce*. With heterogeneous equality, we can see that *dynApply* can remain safely outside the trusted code base.

2.2 Toward dependent types

A further motivation for adding kind equalities is that this seems a necessary step on our way to having full dependent types. For a concrete example of what dependent types in Haskell might look like, see Gundry [7], which explores the concept thoroughly – in-

[2] `https://github.com/goldfirere/ghc`, on the `nokinds` branch. The top-level `README.md` file has more information.

[3] `https://ghc.haskell.org/trac/ghc/wiki/StaticPointers`

a, b	Type variables	c	Coercion variables
D	**data** types	N	**newtype**s
K	Data constructors	F	Type families
$\mathcal{F}$	Type family axioms	$\mathcal{N}$	Newtype axioms

Figure 1. Metavariables for System FC

cluding type inference. That work is, in turn, closely based on Weirich et al. [19], from which the current paper has sprung forth. With kind equalities, I conjecture that an arbitrary dependently typed program can be converted to Haskell, in a type-preserving manner, using the singletons construction. Monnier and Haguenauer [10] write a proof that such a translation is possible, and it seems that the version of Haskell proposed in this paper is expressive enough to apply their work.

There is much motivation for dependent types in the folklore. In particular, an interested reader is encouraged to consider Idris [1], a language very similar to Haskell but with dependent types. Despite Idris's relative youth – dating back only to 2008 – it appears to be gaining a strong following and has generated much buzz, in part owing to its use of dependent types. Despite Idris's success in this area, it would also be nice to have dependent types in Haskell, as GHC is much more mature, and is considered to be industrial strength. Putting dependent types into Haskell would allow more programmers access to dependent types' power.

Kind equalities are necessary because dependent types require having term-level constructs available in types. Currently, certain constructor expressions in Haskell are indeed expressible in both terms and types. For example, *Just True* is both a term and a type in Haskell, thanks to GHC's promotion mechanism [21]. However, only a subset of constructors are promotable – constructors of GADTs, for example, are not. The inability to promote a GADT constructor is directly due to the lack of kind equalities. Just as a term-level GADT constructor wraps up a type equality, a type-level GADT constructor would have to wrap up a kind equality. Accordingly, we need kind equalities to be able to promote GADTs, and take this next step toward dependent types.

For further motivation around kind equalities, the reader is encouraged to see Section 2 of Weirich et al. [19]. Now that we are motivated, let's dive into some details.

3. System FC

System FC [13] is GHC's internal, typed language. It tracks the kind equalities that concern us in this paper. Previous work [2, 19] has a more detailed treatment of System FC than what appears here, but readers are not expected to know more than this introduction to understand the remainder of this paper.

System FC[4] extends System F, adding coercions, type families, and algebraic datatypes. See Figures 1-3 for a primer on the language; I have highlighted the parts that are new to this paper. The treatment of algebraic datatypes in System FC is standard. Coercions are a new form, written with the metavariable γ, that represent a proof that two types are equal. Coercions are typed by a judgment of the form

$$\Sigma; \Gamma \vdash \gamma : \tau_1 \sim_\rho \tau_2$$

where Σ is a signature containing definitions of type constants and Γ is a typing context containing assumptions. The ρ subscript to the equality operator $\sim$ is a role, as explained in Section 5. I use the

[4] This system has grown over the years and through a number of publications. Some of these extensions have given new names to the system, such as $\mathrm{F}_\mathrm{C}^\uparrow$ and FC_2. I refer to the evolving system under one name: FC.

H	$::=$	$\mathit{TYPE} \mid (\to) \mid {}'K \mid T$	Type constants
T	$::=$	$D \mid N$	Algebraic types
e, u	$::=$	$\dots$	Expressions
τ, σ, κ	$::=$	$a \mid H \mid \tau\,\psi \mid \forall\,\delta.\tau \mid F[\overline{\tau}] \mid \tau \rhd \gamma$	Types
ξ	$::=$	$a \mid H \mid \xi_1\,\xi_2 \mid \xi\,c \mid \xi \rhd c$	Type patterns
δ	$::=$	$a{:}\kappa \mid c{:}\phi$	Binders
ψ	$::=$	$\tau \mid \gamma$	Arguments
γ, η	$::=$	$\langle\tau\rangle \mid \mathbf{sym}\,\gamma \mid \gamma_1 \fatsemi \gamma_2 \mid \gamma_1\ \gamma_{2(\eta)}$	Coercions
	$\mid$	$\gamma_1\ (\gamma_2, \gamma_3)_\chi \mid \forall_\eta(a_1, a_2, c).\gamma \mid \forall_\chi(c_1, c_2).\gamma$	
	$\mid$	$\mathbf{kind}\,\gamma \mid \mathbf{kapp}\,\gamma \mid \mathbf{kapp}_1\,\gamma \mid \mathbf{kapp}_2\,\gamma$	
	$\mid$	$c \mid \mathcal{F}[i][\overline{\psi}] \mid \mathcal{N}[\overline{\tau}] \mid \mathbf{sub}\,\gamma \mid \gamma \rhd \eta \mid \dots$	
ϕ	$::=$	$\tau_{1(\kappa_1)} \sim_\rho \tau_{2(\kappa_2)}$	Propositions
χ	$::=$	$\langle\eta_1, \eta_2\rangle_\rho$	Higher-order cos.
ρ	$::=$	$\mathsf{N} \mid \mathsf{R}$	Roles
Φ	$::=$	$[\Delta].F[\overline{\xi}] \mapsto \sigma$	T.F. equations
Ψ	$::=$	$\overline{\Phi}$	T.F. axiom types
Σ	$::=$	$\varnothing \mid \Sigma, \mathit{sbnd}$	Signatures
sbnd	$::=$	$T : \forall\overline{a{:}_\rho\kappa}.\,\star \mid K : \tau \mid F : [\Delta].\kappa$	Sig. bindings
	$\mid$	$\mathcal{F} : \Psi \mid \mathcal{N} : [\overline{a{:}\kappa}].N\ \overline{a} \mapsto_\kappa \sigma$	
Γ, Δ	$::=$	$\varnothing \mid \Gamma, \delta$	Contexts
t, p	$::=$	$\dots$	Erased types / args.

The type $\tau_1 \to \tau_2$ is syntactic sugar for $(\to)\,\tau_1\,\tau_2$.
I write $\tau_1 \sim_\rho \tau_2$ for $\tau_{1(\kappa_1)} \sim_\rho \tau_{2(\kappa_2)}$ when kinds are unimportant.

Figure 2. Grammar for System FC. *TYPE* is discussed in Section 4.3; the new coercion forms are discussed in Section 6 (in particular, Section 6.3.2); erased types and type patterns are discussed in Section 7.2; and the new form χ is discussed in Section 3.1.

$\Sigma \vdash H \Leftarrow \overline{\rho}$	"Roles $\overline{\rho}$ are appropriate for H."
$\Sigma; \Gamma \vdash \mathbf{ctx}$	Context validity
$\Sigma; \Gamma \vdash \tau : \kappa$	Type kinding
$\Sigma; \Gamma \vdash_{\mathsf{tel}} \overline{\psi} \Leftarrow \Delta$	Telescope validity
$\Sigma; \Gamma \vdash e : \tau$	Expression typing
$\Sigma \vdash \mathsf{no\_conflict}(\Psi, \Phi, \overline{t}, i)$	"Equation Φ in axiom Ψ at erased types $\overline{\tau}$ is not prevented from reducing be equations i or below."
$\Sigma \vdash e \longrightarrow e'$	Small-step reduction

Figure 3. Typing judgment schemas for System FC

$\boxed{\Sigma; \Gamma \vdash \chi : \phi_1 \sim_\rho \phi_2}$ Higher-order coercion typing

$$\frac{\begin{array}{c}\Sigma; \Gamma \vdash \eta_1 : \sigma_1 \sim_\rho \sigma_3 \\ \Sigma; \Gamma \vdash \eta_2 : \sigma_2 \sim_\rho \sigma_4 \\ \phi_1 = \sigma_1 \sim_{\rho_2} \sigma_2 \\ \phi_2 = \sigma_3 \sim_{\rho_2} \sigma_4\end{array}}{\Sigma; \Gamma \vdash \langle\eta_1, \eta_2\rangle_{\rho_2} : \phi_1 \sim_\rho \phi_2}\ \textsc{HCo\_Pair}$$

$\boxed{\Sigma; \Gamma \vdash \phi\ \mathsf{ok}}$ Proposition validity

$$\frac{\begin{array}{c}\Sigma; \Gamma \vdash \tau_1 : \kappa_1 \\ \Sigma; \Gamma \vdash \tau_2 : \kappa_2\end{array}}{\Sigma; \Gamma \vdash \tau_{1(\kappa_1)} \sim_\rho \tau_{2(\kappa_2)}\ \mathsf{ok}}\ \textsc{Prop\_Equality}$$

Figure 4. Selected FC typing judgments, in full

$$\frac{\Sigma;\Gamma \vdash \tau : \kappa}{\Sigma;\Gamma \vdash \langle\tau\rangle : \tau \sim_{\mathsf{N}} \tau}\ \textsc{Co\_Refl} \qquad \frac{\Sigma;\Gamma \vdash \gamma : \tau_2 \sim_\rho \tau_1}{\Sigma;\Gamma \vdash \mathbf{sym}\,\gamma : \tau_1 \sim_\rho \tau_2}\ \textsc{Co\_Sym} \qquad \frac{\begin{array}{c}\Sigma;\Gamma \vdash \gamma_1 : \tau_1 \sim_\rho \tau_2 \\ \Sigma;\Gamma \vdash \gamma_2 : \tau_2 \sim_\rho \tau_3\end{array}}{\Sigma;\Gamma \vdash \gamma_1 \mathbin{⨟} \gamma_2 : \tau_1 \sim_\rho \tau_3}\ \textsc{Co\_Trans}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash \gamma_1 : \tau_1 \sim_\rho \tau_2 \\ \Sigma;\Gamma \vdash \gamma_2 : \sigma_1 \sim_{\mathsf{N}} \sigma_2 \\ \Sigma;\Gamma \vdash \eta : \kappa_1 \sim_\rho \kappa_2 \\ \Sigma;\Gamma \vdash \tau_1\,\sigma_1 : \kappa_0 \qquad \Sigma;\Gamma \vdash \tau_2\,\sigma_2 : \kappa_0'\end{array}}{\Sigma;\Gamma \vdash \gamma_1\ \gamma_{2(\eta)} : \tau_1\,\sigma_1 \sim_\rho \kappa_2\,\sigma_2}\ \textsc{Co\_AppTy} \qquad \frac{\begin{array}{l}\Sigma;\Gamma \vdash \eta : \kappa_1 \sim_\rho \kappa_2 \\ \Sigma;\Gamma, a_1{:}\kappa_1, a_2{:}\kappa_2, c{:}a_1 \sim_{\mathsf{N}} a_2 \vdash \gamma : \tau_1 \sim_\rho \tau_2 \\ \Sigma;\Gamma \vdash \forall\, a_1{:}\kappa_1.\tau_1 : \sigma_1 \qquad \Sigma;\Gamma \vdash \forall\, a_2{:}\kappa_2.\tau_2 : \sigma_2\end{array}}{\Sigma;\Gamma \vdash \forall_\eta(a_1, a_2, c).\gamma : \forall\, a_1{:}\kappa_1.\tau_1 \sim_\rho \forall\, a_2{:}\kappa_2.\tau_2}\ \textsc{Co\_ForAllTy}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash \gamma_0 : \tau_1 \sim_\rho \tau_2 \\ \Sigma;\Gamma \vdash \gamma_1 : \phi_1 \qquad \Sigma;\Gamma \vdash \tau_1\,\gamma_1 : \kappa_1' \\ \Sigma;\Gamma \vdash \gamma_2 : \phi_2 \qquad \Sigma;\Gamma \vdash \tau_2\,\gamma_2 : \kappa_2' \\ \Sigma;\Gamma \vdash \chi : \phi_1 \sim_\rho \phi_2\end{array}}{\Sigma;\Gamma \vdash \gamma_0\ (\gamma_1, \gamma_2)_\chi : \tau_1\,\gamma_1 \sim_\rho \tau_2\,\gamma_2}\ \textsc{Co\_AppCo} \qquad \frac{\begin{array}{l}\Sigma;\Gamma \vdash \chi : \phi_1 \sim_\rho \phi_2 \\ \Sigma;\Gamma, c_1{:}\phi_1, c_2{:}\phi_2 \vdash \gamma : \tau_1 \sim_\rho \tau_2 \\ \Sigma;\Gamma \vdash \forall\, c_1{:}\phi_1.\tau_1 : \sigma_1 \qquad \Sigma;\Gamma \vdash \forall\, c_2{:}\phi_2.\tau_2 : \sigma_2\end{array}}{\Sigma;\Gamma \vdash \forall_\chi(c_1, c_2).\gamma : \forall\, c_1{:}\phi_1.\tau_1 \sim_\rho \forall\, c_2{:}\phi_2.\tau_2}\ \textsc{Co\_ForAllCo}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash \gamma : \tau_1 \sim_\rho \tau_2 \\ \Sigma;\Gamma \vdash \tau_1 \rhd \eta : \kappa_1'\end{array}}{\Sigma;\Gamma \vdash \gamma \rhd \eta : (\tau_1 \rhd \eta) \sim_\rho \tau_2}\ \textsc{Co\_Coherence} \qquad \frac{\begin{array}{l}\Sigma;\Gamma \vdash \gamma : \tau_1\,\sigma_1 \sim_{\mathsf{N}} \tau_2\,\sigma_2 \\ \Sigma;\Gamma \vdash \sigma_1 : \kappa_1 \\ \Sigma;\Gamma \vdash \sigma_2 : \kappa_2\end{array}}{\Sigma;\Gamma \vdash \mathbf{kapp}\,\gamma : \kappa_1 \sim_{\mathsf{N}} \kappa_2}\ \textsc{Co\_KAppTy}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash \gamma : \tau_1\,\gamma_1 \sim_{\mathsf{N}} \tau_2\,\gamma_2 \\ \Sigma;\Gamma \vdash \gamma_1 : \sigma_1 \sim_\rho \sigma_2 \\ \Sigma;\Gamma \vdash \gamma_2 : \sigma_3 \sim_\rho \sigma_4\end{array}}{\Sigma;\Gamma \vdash \mathbf{kapp}_1\,\gamma : \sigma_1 \sim_{\mathsf{N}} \sigma_3}\ \textsc{Co\_KAppCo1} \qquad \frac{\begin{array}{l}\Sigma;\Gamma \vdash \gamma : \tau_1\,\gamma_1 \sim_{\mathsf{N}} \tau_2\,\gamma_2 \\ \Sigma;\Gamma \vdash \gamma_1 : \sigma_1 \sim_\rho \sigma_2 \\ \Sigma;\Gamma \vdash \gamma_2 : \sigma_3 \sim_\rho \sigma_4\end{array}}{\Sigma;\Gamma \vdash \mathbf{kapp}_2\,\gamma : \sigma_2 \sim_{\mathsf{N}} \sigma_4}\ \textsc{Co\_KAppCo2}$$

$$\frac{\begin{array}{c}c{:}\phi \in \Gamma \\ \Sigma;\Gamma \vdash \mathbf{ctx}\end{array}}{\Sigma;\Gamma \vdash c : \phi}\ \textsc{Co\_Var} \qquad \frac{\Sigma;\Gamma \vdash \gamma : \tau_{1(\kappa_1)} \sim_\rho \kappa_{2(\kappa_2)}}{\Sigma;\Gamma \vdash \mathbf{kind}\,\gamma : \kappa_1 \sim_{\mathsf{R}} \kappa_2}\ \textsc{Co\_Kind} \qquad \frac{\Sigma;\Gamma \vdash \gamma : \tau_1 \sim_{\mathsf{N}} \tau_2}{\Sigma;\Gamma \vdash \mathbf{sub}\,\gamma : \tau_1 \sim_{\mathsf{R}} \tau_2}\ \textsc{Co\_Sub}$$

$$\frac{\begin{array}{l}\mathcal{F} : \Psi \in \Sigma \qquad \Psi[i] = [\Delta].F[\overline{\xi}] \mapsto \sigma \\ F : [\Delta_2].\kappa' \in \Sigma \qquad \Sigma;\Gamma \vdash_{\mathsf{tel}} \overline{\psi} \Leftarrow \Delta \\ \mathsf{erase}_{\Sigma;\Gamma}(\overline{\xi}[\overline{\psi}/\Delta]) \rightsquigarrow \overline{t} \qquad \Sigma;\Gamma \vdash \kappa'[\overline{\xi}[\overline{\psi}/\Delta]/\Delta_2] : \star \\ \Sigma \vdash \mathsf{no\_conflict}(\Psi, \Psi[i], \overline{t}, i)\end{array}}{\Sigma;\Gamma \vdash \mathcal{F}[i][\overline{\psi}] : F[\overline{\xi}][\overline{\psi}/\Delta] \sim_{\mathsf{N}} \sigma[\overline{\psi}/\Delta]}\ \textsc{Co\_TFAxiom} \qquad \frac{\begin{array}{l}\mathcal{N} : [\overline{a}{:}\overline{\kappa}].N\ \overline{a} \mapsto_{\kappa_0} \sigma \in \Sigma \\ \Sigma;\Gamma \vdash_{\mathsf{tel}} \overline{\tau} \Leftarrow \overline{a}{:}\overline{\kappa} \\ \Sigma;\Gamma \vdash \kappa_0[\overline{\tau}/\overline{a}] : \star\end{array}}{\Sigma;\Gamma \vdash \mathcal{N}[\overline{\tau}] : N\,\overline{\tau} \sim_{\mathsf{R}} \sigma[\overline{\tau}/\overline{a}]}\ \textsc{Co\_NTAxiom}$$

Figure 5. Selected rules of coercion formation: $\Sigma;\Gamma \vdash \gamma : \phi$

metavariable ϕ to refer to equality propositions, such as $\tau_1 \sim_\rho \tau_2$ above.

Coercions can be formed in a multitude of ways. Figure 5 contains selected rules from the typing judgment, though we will focus only on the highlighted parts. Figure 4 contains several auxiliary judgments referred to in the coercion typing judgment.

Coercions are used in casts, such as the expression form $e \rhd \gamma$, with the following typing rule:

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash e : \tau_1 \\ \Sigma;\Gamma \vdash \gamma : \tau_1 \sim \tau_2\end{array}}{\Sigma;\Gamma \vdash e \rhd \gamma : \tau_2}\ \textsc{Tm\_Cast (ver. i)}$$

We see here that we can take an expression e of type τ_1 and treat it as an expression of type τ_2 by inserting an explicit cast using the cast operator $\rhd$.

Coercions arise from two sources: axioms in the environment and local assumptions. A type family instance compiles to an axiom $\mathcal{F}$. For example, the declaration

type instance *F Int* = *Bool*

yields an axiom *axF* : *F*[*Int*] $\mapsto$ *Bool*, which can be made into a coercion proving *F*[*Int*] $\sim$ *Bool*. We see here that type families in System FC are presented with a first-order syntax *F*[...]; this is because type families must always appear saturated, both in Haskell and in FC.

Newtypes also compile to axioms $\mathcal{N}$. The declaration

newtype *Age* = *MkAge Int*

compiles to *axAge* : *Age* $\mapsto$ *Int*. The *MkAge* constructor does not appear in FC.

The use of GADTs lead to coercion assumptions. Consider the type *G*:

data *G a* **where**
  MkGBool :: *G Bool*

The *MkGBool* :: *G Bool* constructor is elaborated to have type *MkGBool* :: $\forall$ *a*. *a* $\sim$ *Bool* $\Rightarrow$ *G a*. This transformation is critical when analyzing **case** expressions, where we need the result types of all constructors to be uniform in the datatype parameters. (In other words, the result type of a constructor must be the datatype name followed by a correctly sized list of unrepeated type variables, such as *G a*.) The type $\forall$ *a*. *a* $\sim$ *Bool* $\Rightarrow$ *G a* is understood to mean that *MkGBool* takes one parameter when elaborated in System FC – a coercion between type *a* and *Bool*. Then, when unpacking the constructor in a pattern match, this coercion is available within the body of the pattern match. For example, consider this function:

$$\begin{array}{l} \mathit{match} :: \forall\ a.\ G\ a \to a \\ \mathit{match}\ \mathit{MkGBool} = \mathit{True} \end{array}$$

The function *match* elaborates as follows:

$$\begin{array}{l} \mathit{match} :: \forall\ a.\ G\ a \to a \\ \mathit{match} = \Lambda(a :: \star) \to \lambda(x :: G\ a) \to \textbf{case}\ x\ \textbf{of} \\ \quad \mathit{MkGBool}\ (c :: a \sim \mathit{Bool}) \to \mathit{True} \rhd \mathbf{sym}\ c \end{array}$$

The cast is necessary because *match* must return a result of type a; the $\mathbf{sym}$ operator we see there reverses the order of the coercion c, proving that $\mathit{Bool} \sim a$, as desired.

3.1 Adding kind equalities

In Weirich et al. [19], my co-authors and I extended this ability of casts to include casting on types, using the following rule:

$$\frac{\Sigma;\Gamma \vdash \tau : \kappa_1 \qquad \Sigma;\Gamma \vdash \gamma : \kappa_1 \sim \kappa_2}{\Sigma;\Gamma \vdash \tau \rhd \gamma : \kappa_2}\ \text{TY\_CAST (VER. I)}$$

This rule behaves identically to the term-level rule, but one level up. It is this ability to use an equality to change the kind of a type that forms the essential difference between the System FC with kind equalities and other versions of the language.

There are several knock-on effects of adding kind equalities, summarized here:

- Promoting a GADT constructor leads to a type constant, written $'K$, that takes a coercion argument. Accordingly, System FC allows type application to either types $\tau\ \sigma$ or coercions $\tau\ \gamma$. This ability is more succinctly expressed in terms of ψ, which stands for either a type or a coercion. Thus, the application form looks like $\tau\ \psi$.
- In two places in the grammar (seen in rules CO_APPCO and CO_FORALLCO) it is necessary to prove an equality between two propositions. I call this a *higher-order coercion*, and write it χ. A higher-order coercion, composed of two ordinary coercions, is a proof that the corresponding parts of two propositions are equal. See Figure 4 for the typing rule.

During compilation, GHC takes a source Haskell program and performs type inference on it, producing an annotated version of the source program. At this stage, any programmer errors should be caught and reported. In an error-free program, GHC then elaborates the annotated source program to a System FC program. The elaborated program is then optionally type checked. A type error in the System FC program would indicate a bug in the GHC implementation, as all user errors should have already been detected and reported. This redundancy is why type checking System FC is optional. However, having a typed internal language has proved to be a great asset to the GHC implementation, ferreting out a plenitude of bugs during development. System FC is also amenable to formal reasoning, allowing GHC/Haskell to rest on a solid type-theoretical framework.

3.2 Metatheory

The metatheory of System FC is well explored in the literature [2, 4, 13, 19]. In the submitted auxiliary material, I update several of the lemmas to work with this new system. The two key lemmas needed to prove type safety are *lifting* (for preservation) and *consistency* (for progress).

The lifting lemma says that types are preserved when a cast stands in the way of a pattern match. It is significantly complicated by kind equalities, as explored in Weirich et al. [19]. Here, it is updated with roles.

The consistency lemma says, essentially, that there is no coercion (in an empty context) proving $\mathit{Int} \sim \mathit{Bool}$. This is a key part of proving progress.

Theorem (Preservation)**.** *Assume for all* $H \in \mathsf{dom}(\Sigma)$, $\Sigma \vdash H \Leftarrow \overline{\rho}$. *If* $\Sigma;\Gamma \vdash e : \tau$ *and* $\Sigma \vdash e \longrightarrow e'$, *then* $\Sigma;\Gamma \vdash e' : \tau$.

In System FC, a value is a lambda-abstraction or an expression headed by a constructor.

Theorem (Progress)**.** *Assume for all* $H \in \mathsf{dom}(\Sigma)$, $\Sigma \vdash H \Leftarrow \overline{\rho}$. *IF* $\Sigma;\Gamma \vdash e : \tau$, *then either* $\Sigma \vdash e \longrightarrow e'$ *for some* e', e *is a value, or* e *is a casted value.*

The full definition of this version of System FC, with the statement and proofs of the key lemmas, appears in the submitted auxiliary material.

4. Unlifted types and eliminating sub-kinding

4.1 Lifted types vs. unlifted types

Haskell, a non-strict language, differentiates between *lifted* types and *unlifted* types. A lifted type is one that contains *undefined* – the canonical $\bot$ term in Haskell – and an unlifted type does not. As a consequence, evaluating expressions of unlifted types is always strict.

Inhabitants of lifted types may be *thunks* – unevaluated portions of a program. It is thus absolutely necessary for inhabitants of lifted types to be stored via pointers. The runtime system can distinguish between thunks and evaluated data before following the pointer. Inhabitants of unlifted types, however, can be represented without any indirection. Because we know an unlifted type is always strict, we can manipulate it directly. Haskell library-writers take advantage of this fact by using unlifted types in performance-critical code.

The difference in representation between lifted and unlifted types implies another distinguishing characteristic: it is not possible to abstract over unlifted types. For example, the value $3\#$ is of type $\mathit{Int}\#$, yet writing $\mathit{id}\ 3\#$ (where $\mathit{id} :: \forall\ a.\ a \to a$ is the polymorphic identity) is erroneous. Upon reflection, we see that this code must be wrong:[5] what machine code could be generated for *id*? All lifted types are represented by pointers, so it is straightforward to have an implementation for *id* that works on any lifted type. Yet, unlifted types can have any size and layout in memory; the machine code for one function cannot deal with this variety.

Haskell uses its type system to prevent constructions like $\mathit{id}\ 3\#$ by putting unlifted types into their own kind. The kind $\star$ classifies lifted types (such as *Int*, *Bool*, and *Maybe Double*). The kind $\#$ classifies unlifted types (such as $\mathit{Int}\#$, $\mathit{Word32}\#$, and $\mathit{Array}\ \#\ \mathit{Float}\#$). Accordingly, the full type of *id* should be written $\forall\ (a :: \star).\ a \to a$. When I say $\mathit{id}\ 3\#$, I get a kind error:

$$\begin{array}{l} \mathit{Kind\ incompatibility\ when\ matching\ types} : \\ \quad a \qquad :: \star \\ \quad \mathit{Int}\ \# \ :: \# \end{array}$$

4.2 Sub-kinding

Despite the separation between $\star$ and $\#$, the following code type-checks:

$$\begin{array}{l} \mathit{quux} :: \mathit{Bool} \to \mathit{Int}\ \# \\ \mathit{quux}\ \_ = \mathit{undefined} \end{array}$$

This correct code begs the questions: What is the kind of $(\to)$? And what is the type of *undefined*? Both $(\to)$ and *undefined* are used above in conjunction with unlifted types, seemingly successfully.

[5] I am discounting the possibility of specializing *id* to concrete unlifted types here, which GHC does not currently do.

The answer to these questions is that today's GHC has a sub-kinding feature. When used fully applied, the $(\rightarrow)$ type has the kind $\mathit{OpenKind} \rightarrow \mathit{OpenKind} \rightarrow \star$, where $\#$ and $\star$ are sub-kinds of $\mathit{OpenKind}$. Similarly, the type of *undefined* is properly $\forall\ (a :: \mathit{OpenKind}).\ a$. The type $\mathit{Bool} \rightarrow \mathit{Int}\#$ is thus well-kinded, according to the sub-kinding relationship; and using *undefined* at type $\mathit{Int}\#$ is also well-kinded. *OpenKind* also appears during type inference when checking a lambda-term. As we're checking $\lambda x \rightarrow ...$, we don't yet know whether x's type will be unlifted or lifted; GHC uses a unification variable of kind *OpenKind* to pull this off.

This use of sub-kinding has worked moderately well, but it does present a few oddities:

- A student of type systems can learn about the $\star$ and $\#$ kinds, and then be quite perplexed that *undefined* works at type $\mathit{Int}\#$. This question comes up with some regularity on the Haskell mailing lists.[6]
- The function $\mathit{error} :: \forall\ (a :: \mathit{OpenKind}).\ \mathit{String} \rightarrow a$ works similarly to *undefined*, but prints a user-specified error message when evaluated. It is sometimes convenient to define a function such as $\mathit{flooError}\ s = \mathit{error}$ (`"Module Floo: "` $+\!\!+\, s$). Yet, user definitions cannot tap into the sub-kinding magic, and so *flooError* would be restricted to types of kind $\star$, making it less useful than *error*.
- From an implementation standpoint, the presence of sub-kinds complicates various parts of type inference, and has always seemed to be a less-than-ideal solution.

As we examine adding kind equalities, sub-kinding becomes even more of a thorn. For example, if we have an proof that $k \sim \star$ (for some k), then what is k's relationship to *OpenKind*? What is the proof for that relationship? Perhaps there is a story to be told here, but it would be complex.

4.3 Eliminating sub-kinds via *levity polymorphism*

Instead, merging types with kinds facilitates getting rid of sub-kinds altogether. The idea behind the new approach is to replace sub-kinding with polymorphism, over a type variable of the new kind *Levity*:

$$\textbf{data}\ \mathit{Levity} = \mathit{Lifted} \mid \mathit{Unlifted}$$

We then say that $\star$ is just a synonym for $\mathit{TYPE}\ '\mathit{Lifted}$ and $\#$ is a synonym for $\mathit{TYPE}\ '\mathit{Unlifted}$, where *TYPE* is a new primitive constant in the type system. All the uses of sub-kinding now are easily expressed using polymorphism:

$$\begin{array}{ll}
\mathit{undefined} & :: \forall\ (v :: \mathit{Levity})\ (a :: \mathit{TYPE}\ v).\ a \\
\mathit{error} & :: \forall\ (v :: \mathit{Levity})\ (a :: \mathit{TYPE}\ v).\ \mathit{String} \rightarrow a \\
(\rightarrow) & :: \forall\ (v_1 :: \mathit{Levity})\ (v_2 :: \mathit{Levity}). \\
 & \quad \mathit{TYPE}\ v_1 \rightarrow \mathit{TYPE}\ v_2 \rightarrow \star
\end{array}$$

As we can see, *TYPE* has type $\mathit{Levity} \rightarrow \mathit{TYPE}\ '\mathit{Lifted}$, recalling that $\mathit{TYPE}\ '\mathit{Lifted}$ – that is, $\star$ – is the classification of all kinds. This strange typing relationship (where a constant is mentioned in its own type) can be accommodated by a custom typing rule, much like that for $(\rightarrow)$. It presents no trouble in the formalization or proof.

4.4 Levity polymorphism in type inference

One of the signs that levity polymorphism is a good approach is how well it works with the existing type inference scheme. When GHC must reason about an as-yet-unknown type, it invents *unification variables*, written α or β. When GHC learns, say, that α should be *Int*, it just does an in-place mutable update of $\alpha := \mathit{Int}$.

Suppose GHC is inferring the type of x in the expression $\lambda x \rightarrow \mathit{swizzle}\ x$, where $\mathit{swizzle} :: \mathit{Int} \rightarrow \mathit{Bool}$. When type-checking the pattern x, GHC will invent a levity unification variable $\alpha :: \mathit{Levity}$ and a type unification variable $\beta :: \mathit{TYPE}\ \alpha$, where $x :: \beta$. In the course of normal type inference, GHC will discover that β must be the type that *swizzle* expects, *Int*. Before setting $\beta := \mathit{Int}$, however, GHC does a kind check. (This kind check exists in today's Haskell, too.) It ensures that the type of β, $\mathit{TYPE}\ \alpha$, matches the type of *Int*, which is $\mathit{TYPE}\ '\mathit{Lifted}$. In so doing, GHC discovers that it should set $\alpha := '\mathit{Lifted}$. Naturally, GHC does *another* kind check, but that straightforwardly confirms that the type of α is the same as the type of $'\mathit{Lifted}$: they are both *Levity*. Thus, GHC sets $\alpha := '\mathit{Lifted}$ and then $\beta := \mathit{Int}$. What is encouraging about all of this is that *nothing* in this part of the type inference algorithm had to change to support levity polymorphism.

The one area in GHC that did take some care is in the handling of defaulting. Suppose we are checking $\lambda y \rightarrow y$. GHC invents unification variables $\alpha :: \mathit{Levity}$ and $\beta :: \mathit{TYPE}\ \alpha$. It then discovers nothing at all to constrain these. A naive possibility is that GHC would assign the type $(\lambda y \rightarrow y) :: \forall\ (a :: \mathit{Levity})\ (b :: \mathit{TYPE}\ a).\ b \rightarrow b$, but this would be disastrous! Abstracting over a levity variable in this way means that we can now abstract over types in kind $\#$, which we must avoid. If we somehow did produce that type, the machine code generator would be hamstrung in trying to deal with it.

Instead, levity polymorphism requires updating GHC's existing kind-defaulting mechanism, where unconstrained kind default to $\star$. This defaulting mechanism exists in today's GHC, where we do not want to infer $(\lambda y \rightarrow y) :: \forall\ (b :: \mathit{OpenKind}).\ b \rightarrow b$. Changing this defaulting mechanism to set unconstrained levity unification variables to $'\mathit{Lifted}$ was actually a simplification over the old mechanism.

4.5 Discussion

Using levity polymorphism to eliminate sub-kinding is a clear win for GHC, in both theory and practice. While the *TYPE* constant is perhaps more complex than $\star$ or $\#$, avoiding sub-kinding simplifies the type system by removing this extra relation among types. Implementing the feature also removed several checks to make sure the sub-kinding relation was being respected. This seems like a solid way to deal with a type system that admits multiple kinds of inhabited types, like $\star$ and $\#$.

However, levity polymorphism would not, on its own, be implementable in today's GHC without some pain. The problem is that levity variables appear in kinds. Today's GHC maintains an invariant that all variables that appear in kinds have type $\Box$, the sort of kinds. Indeed, decisions based on a variable's kind happen quite often, as today's GHC treats kind variables somewhat differently than type variables. (For example, type variables are printed in error messages; kind variables are not.)This is not an easy limitation to work around. While I conjecture that the engineering challenges could be overcome with effort, integrating levity polymorphism fits more naturally in a system with a fully-merged grammar of types and kinds.

5. Roles on kind equalities

A critical step in the integration of kind equalities into GHC is figuring out a way to combine roles [2, 18] with kind equalities. It turns out that this combination leads to subtle interactions and forces some hard design choices.

[6] For example: `https://www.haskell.org/pipermail/ghc-devs/2015-February/008222.html`

5.1 An introduction to roles

Haskell supports the **newtype** construct, which allows the programmer to declare that one type shares a representation with another:

newtype *Age* = *MkAge Int*

This declaration says that the runtime representation of *Age* is identical to that of *Int*, although these types remain distinct to the type checker. This construct is a boon to programmers, because they can use **newtype**s to enforce abstraction in their code without paying a runtime penalty.

The usefulness of **newtype**s has led Haskellers to demand the ability to lift the free conversions through types. That is, we would want to be able to convert *Maybe* [*Age*] to *Maybe* [*Int*] with no runtime penalty. We will soon discover, however, that this is not safe with *all* types. For example, consider these declarations:

```
type family F a where
  F Int  = Bool
  F Age = Double
data X a = MkX (F a)
```

We clearly cannot convert an *X Int* to an *X Age* – the former stores a *Bool* and the latter a *Double*! How can we tell the difference between safe coercions, based on types like [ ] and *Maybe*, and unsafe ones, based on types like *X*?

To see the difference between *Maybe* and *X*, we must consider the two different notions of equality in play. We want safe coercions allowed between types that have the same representation at runtime. Yet, a type family can distinguish among such types, as we see above. *Age* and *Int* are equal in one sense, and distinct in another. We must carefully track these two equality relations to get this right.

To allow these free coercions safely, Weirich et al. [18] proposed distinguishing the two equality relations at work here. Breitner et al. [2] expanded upon that work and simplified it so that it was suitable for implementation.

5.2 Nominal and representational equality

The two equality relations at work are $(\sim_{\mathsf{N}})$, called nominal equality; and $(\sim_{\mathsf{R}})$, called representational equality. Nominal equality is what Haskell programmers think of as type equality. Nominal equality includes the reduction of type functions, such that $F\ Int \sim_{\mathsf{N}} Bool$. Representational equality, on the other hand, is strictly coarser, relating also newtypes and their representations. Thus, $Age \sim_{\mathsf{R}} Int$ while $Age \not\sim_{\mathsf{N}} Int$. Breitner et al. [2] introduced the new function[7] $coerce :: a \sim_{\mathsf{R}} b \Rightarrow a \to b$, which allows programmers to freely convert one type to a representationally equal type.

In System FC, discerning between the two equality relations is done using an extra annotation on the $\sim$ operator that forms equality propositions. Thus, the typing judgment for coercions is more accurately stated as

$$\Sigma; \Gamma \vdash \gamma : \tau_1 \sim_\rho \tau_2$$

where ρ is a metavariable for roles; ρ may concretely become N or R.[8] Accordingly, every coercion proves either a nominal equality proposition or a representational one. The coercion former $\mathbf{sub}$ converts a nominal equality proof to a representational one. That is to say:

$$\frac{\Sigma; \Gamma \vdash \gamma : \tau_1 \sim_{\mathsf{N}} \tau_2}{\Sigma; \Gamma \vdash \mathbf{sub}\,\gamma : \tau_1 \sim_{\mathsf{R}} \tau_2} \quad \textsc{Co\_Sub}$$

[7] The operator $(\sim_{\mathsf{R}})$ is actually spelled *Coercible* in GHC programs.

[8] Here, and throughout most of this paper, I ignore the possibility of *phantom* coercions, as introduced in Breitner et al. [2].

We must now return and refine the typing rule for casting expressions, first encountered in Section 3. In order to support casting from *Age* to *Int*, we see that the cast operator must work with representational equality. We thus get this typing rule:

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

The only difference from the previous version is the addition of the R subscript to the equality proposition proved by γ.

With representational equality in hand, there is now a straightforward way to encode newtypes: every newtype declaration gives rise to an axiom for representational equality. For example, the declaration of *Age* in the introduction leads to the axiom *axAge* proving $Age \sim_{\mathsf{R}} Int$.

The addition of roles, unfortunately, complicates GHC's type system considerably, both internally, and in ways visible to programmers. This is a regrettable consequence of allowing both a way to convert among representationally equal types while having type-system features that can discern among representationally equal types.

5.3 Heterogeneous equality

System FC with kind equalities also directly supports *heterogeneous* equality. To see how this arises, consider the *Proxy* type:

data *Proxy* $(a :: k)$ = *P*

The type *Proxy* has kind $\forall\ k.\ k \to \star$. Now, consider that we have some assumption $c :: (k_0 \sim \star)$; that is, c is a proof that some kind k_0 is equal to $\star$, ignoring the nominal/representational distinction for the moment. Reflexive coercions can be built out of any type; the coercion $\langle Proxy \rangle$ proves $Proxy \sim Proxy$. We can then build the coercion $\langle Proxy \rangle\ c$, proving $Proxy\ k_0 \sim Proxy\ \star$. But, we can also see that $Proxy\ k_0 :: k_0 \to \star$ and $Proxy\ \star :: \star \to \star$. Thus, $\langle Proxy \rangle\ c$ is a heterogeneous coercion.

To support heterogeneous equality, we must refine the form of propositions once again to include the kinds of the types being related. An equality proposition, in full, is then written $\tau_{1(\kappa_1)} \sim \tau_{2(\kappa_2)}$, where τ_1 has kind κ_1 and τ_2 has kind κ_2. I will still often use the infix form $\tau_1 \sim \tau_2$ when the kinds are irrelevant.

Alongside heterogeneous equality, we would also like to say that casts are irrelevant in types, a property called *coherence*. We want to be able to prove that, for any τ and γ: $\tau \rhd \gamma \sim \tau$. This proof is embodied by a coherence coercion form, such as this rule from Weirich et al. [19]:

$$\frac{\Sigma; \Gamma \vdash \gamma : \tau_1 \sim \tau_2 \qquad \Sigma; \Gamma \vdash \tau_1 \rhd \gamma' : \kappa_1}{\Sigma; \Gamma \vdash \gamma \rhd \gamma' : (\tau_1 \rhd \gamma') \sim \tau_2} \quad \textsc{Co\_CoherenceNoRole}$$

Although this rule is asymmetric, it can be used with the rule for symmetry ($\mathbf{sym}$) to get coercions on either side of the $(\sim)$.

5.4 The role of kind coercions

Let's now add roles to kind coercions. The first problem to tackle is how to update the Ty_Cast rule, introduced in Section 3. Should that equality proposition be nominal or representational? I have chosen that it should be representational, yielding the following rule:

$$\frac{\Sigma; \Gamma \vdash \tau : \kappa_1 \qquad \Sigma; \Gamma \vdash \gamma : \kappa_1 \sim_{\mathsf{R}} \kappa_2}{\Sigma; \Gamma \vdash \tau \rhd \gamma : \kappa_2} \quad \textsc{Ty\_Cast}$$

This decision is not without consequences. Arguments in favor of each of the choices are below.

Casting by nominal coercions is much simpler. The only reason we have representational equality at all is to glean runtime benefits of having free conversions between newtypes and their representation types. Absent performance concerns, there would be little benefit of the representational equality relation. Yet, when reasoning about types, a performance argument falls flat – there is no need to make "free" conversions among types, as these conversions are never run at all. It would thus seem unnecessary to have representational equalities among kinds. Under this idea, a **newtype** declaration would be treated just as a **data** declaration would be when promoting data constructors to types. Even though the **newtype** constructor used at the term level is absent from a System FC program, that same constructor would be preserved when used at the type level.

Furthermore, casting types by nominal coercions avoids the thorny issues we explore below, caused by the choice of using representational equality in kind casts. The nominal equality choice would then lead to a significantly simpler system.

Casting by representational coercions is much more expressive. The chief argument in favor of casting by representational equalities is uniformity, echoing the decision to cast by representational equalities in terms. The work of integrating kind equalities into GHC is a key step along the way to dependent types, with a Π-quantifier, in Haskell. (See Section 2.2 for some more motivation.) As discussed in Gundry [7], having proper dependent types requires identifying a subset of the expression language that can also appear in types.

I believe all datatypes and newtypes should be in this subset. Because newtype constructors elaborate to representational coercions, in order to freely promote expressions using newtypes, representational coercions must be allowed to cast the kinds of types.[9]

Using representational coercions in kind casts also opens the door to future expansions of the representational equality concept. One way to view representational equality is that it is a type equality that is never inferred – it must always be annotated. The programmer does this by writing a newtype constructor, or by calling *coerce*. One can imagine more ways to let types equal one another, requiring that the programmer specifies when to make the conversion. For example, GHC's *Constraint* kind classifies (lifted) constraint types. As such, it is appropriately seen as indistinguishable from $\star$ in today's System FC. However, *Constraint* and $\star$ are *different* kinds to a Haskell programmer. This situation – when two kinds are distinct in a Haskell program but should be seen as the same under the hood – is exactly the situation with newtypes. Perhaps a future application of the ability to use representational coercions in kind casts is to let *Constraint* and $\star$ be representationally equal. This ability could be used to allow Haskell programs to convert regular values into dictionaries and back, a trick that currently requires *unsafeCoerce*.[10]

In the end, I decided to use representational coercions in kind casts, chiefly in order to support future dependent types work.

6. Extracting kind coercions from type coercions

Suppose we have $\gamma : \tau_{1(\kappa_1)} \sim_\rho \tau_{2(\kappa_2)}$. By definition, this means that γ shows that τ_1 and τ_2 are equal at role ρ. But, what does it say about κ_1 and κ_2? The system described in Weirich et al. [19] has the following rule:

$$\frac{\Sigma; \Gamma \vdash \gamma : \tau_{1(\kappa_1)} \sim \tau_{2(\kappa_2)}}{\Sigma; \Gamma \vdash \mathbf{kind}\,\gamma : \kappa_1 \sim \kappa_2} \quad \text{Co\_KindNoRoles}$$

We must now decide how to decorate this rule with roles. My decision, based on experience working with and implementing this system, is to have the conclusion have a representational role, regardless of the premise's role:

$$\frac{\Sigma; \Gamma \vdash \gamma : \tau_{1(\kappa_1)} \sim_\rho \kappa_{2(\kappa_2)}}{\Sigma; \Gamma \vdash \mathbf{kind}\,\gamma : \kappa_1 \sim_{\mathsf{R}} \kappa_2} \quad \text{Co\_Kind}$$

There appears to be, here, a free choice among several options. Though I have worked out the details only for the system with the rule immediately above, I conjecture that a similar system would support having the conclusion's role match the premise's role, and yet a different system would support removing this rule entirely. I consider these different options below.

6.1 Alternative: a nominal type equality implies a nominal kind equality

Here, we consider an alternate to Co_Kind above:

$$\frac{\Sigma; \Gamma \vdash \gamma : \tau_{1(\kappa_1)} \sim_\rho \tau_{2(\kappa_2)}}{\Sigma; \Gamma \vdash \mathbf{kind}\,\gamma : \kappa_1 \sim_\rho \kappa_2} \quad \text{Co\_KindAlt}$$

At first blush, this seems like the right option. If we know that two types are nominally equal – that is, the two types are considered wholly interchangeable in source Haskell – we would expect their kinds to have the same property. However, this choice, along with the choice to use representational coercions in casts, forces several other decisions, which eventually lead to a seemingly-insoluble type inference problem.

The first consequence of our decision is how it affects the coherence rule, which states that equality can ignore coercions in types. Adding role decorations to the rule as stated above gives this:

$$\frac{\begin{array}{l}\Sigma; \Gamma \vdash \gamma : \tau_1 \sim_\rho \tau_2 \\ \Sigma; \Gamma \vdash \tau_1 \rhd \gamma' : \kappa_1\end{array}}{\Sigma; \Gamma \vdash \gamma \rhd \gamma' : (\tau_1 \rhd \gamma') \sim_\rho \tau_2} \quad \text{Co\_Coherence (ver. i)}$$

I have chosen to label both the premise and conclusion with the same role metavariable ρ because both nominal equality and representational equality should be coherent.

However, this coherence rule leads to disaster. Suppose we have $\gamma_0 : \mathit{Int} \sim_{\mathsf{R}} \mathit{Age}$, built from *axAge*, introduced above (Section 5.2). Now, consider the coercion $\mathbf{kind}\,(\langle 3\rangle \rhd \gamma_0)$. Recall that $\langle\cdots\rangle$ is the notation to create a reflexive coercion at a given type. Accordingly, we can see that $\langle 3\rangle \rhd \gamma_0 : 3 \rhd \gamma_0 \sim_{\mathsf{N}} 3$ and that $\mathbf{kind}\,(\langle 3\rangle \rhd \gamma_0) : \mathit{Age} \sim_{\mathsf{N}} \mathit{Int}$. Thus, we have proved that *Age* is *nominally* equal to *Int*, a statement we wish to be false! Something is dreadfully wrong.

The problem is that, with the rules as given here, nominal equality is *not* coherent. The type $3 \rhd \gamma_0$ should not be nominally equal to 3 because their kinds are not nominally equal. Lack of coherence means that the placement of coercions in types is now relevant. But perhaps we can proceed regardless, if coherence is not a strict requirement.

6.1.1 A symmetric coherence rule

We can help ourselves here by defining a symmetric coherence rule, thus:

$$\frac{\begin{array}{l}\Sigma; \Gamma \vdash \gamma : \tau_{1(\kappa_1)} \sim_\rho \tau_{2(\kappa_2)} \\ \Sigma; \Gamma \vdash \eta_1 : \kappa_1 \sim_{\mathsf{R}} \kappa_1' \\ \Sigma; \Gamma \vdash \eta_2 : \kappa_2 \sim_{\mathsf{R}} \kappa_2' \\ \Sigma; \Gamma \vdash \eta : \kappa_1' \sim_\rho \kappa_2'\end{array}}{\Sigma; \Gamma \vdash \gamma \rhd_\eta (\eta_1, \eta_2) : (\tau_1 \rhd \eta_1) \sim_\rho (\tau_2 \rhd \eta_2)} \quad \text{Co\_SymCoh}$$

[9] Promoted **newtype**s can be used in today's Haskell by considering a **newtype** constructor akin to a **data** constructor. Such a treatment would fail, however, if GHC had to automatically promote certain expressions, which is necessary if it is to support Π-types. Today's promotion is done entirely by hand in the source Haskell program – GHC never converts a term-level expression into a type.

[10] For example, see Edward Kmett's reflection package, which makes use of this trick.

To use this rule to form a nominal coercion, we require a nominal coercion between the result kinds; the coercion is called η in the rule. (When this rule is used with $\rho = \mathsf{R}$, this η is wholly redundant.) This η is necessary to ensure that the $\mathbf{kind}\,\gamma$ coercion form does not form a nominal coercion between two types that should not be nominally equal. However, note that this rule does not make the type system completely coherent – it gives us a way to work with coercions in types, but not the complete freedom of an asymmetric rule.

Though I have not worked out the full details of the proof, I conjecture that CO_SYMCOH, along with CO_KINDALT, can form part of a logically sound type system.

6.1.2 A problem with type inference

Now that we have, presumably, built a sound System FC around CO_KINDALT and CO_SYMCOH, we need to implement type inference to elaborate source Haskell programs into System FC. And, this is where this hypothetical system breaks down. In short, having an incoherent system means that the success or failure of type inference hinges on the fundamentally brittle decision of where to put casts in types.

Type inference in GHC is a constraint-based algorithm, with constraints generated during a pass through a program's abstract syntax tree and then solved [16]. The constraint solver is what concerns us here, where a key step is canonicalization of constraints. The goal of canonicalization is to reduce constraints to one of several basic forms, which are then easy to either solve or to interact with one another. For example, one canonical form has the shape $a \sim_\rho \tau$, where we are guaranteed that the left-hand side is a bare variable and that both types have the same kind. This form can then be used to perform substitutions in other constraints.

Suppose we wish to solve the constraint $\gamma_1 : (\sigma \triangleright \gamma_0) \sim_{\mathsf{N}} \tau$, where $\sigma : \kappa$ and $\gamma_0 : \kappa \sim_{\mathsf{R}} \kappa'$. We first canonicalize. It would be convenient to strip off the cast on σ and canonicalize to $\gamma_2 : \sigma \sim_{\mathsf{N}} \tau$. However, for this decomposition to be sound, we need to have a way to take γ_2 and use it to build γ_1, proving $(\sigma \triangleright \gamma_0) \sim_{\mathsf{N}} \tau$. We should naturally use the coherence rule. To do so, though, we need a $\gamma_3 : \kappa' \sim_{\mathsf{N}} \star$ to use as the η coercion in the rule for CO_SYMCOH. We could decompose our original constraint $\gamma_1 : (\sigma \triangleright \gamma) \sim_{\mathsf{N}} \tau$ to $\gamma_2 : \sigma \sim_{\mathsf{N}} \tau$ *and* $\gamma_3 : \kappa' \sim_{\mathsf{N}} \star$. This would be sound, in that we can define γ_1 in terms of γ_2 and γ_3. But, critically, it would not lead to a complete algorithm, because the converse is not true: it is not possible to derive γ_2 and γ_3 from γ_1. The choice to decompose γ_1 to γ_2 and γ_3 involves something of a “guess”, in that γ_2 and γ_3 may impose more requirements on a solution than does γ_1 alone.

This incompleteness problem also manifests when dealing with *given* constraints. Because GHC reasons with respect to assumptions, often arising from the use of a GADT (see Section 3), it needs to canonicalize and work with both given assumptions and wanted constraints. The approach above fails with a given constraint of the form $(\sigma \triangleright \gamma_0) \sim_{\mathsf{N}} \tau$. There appears to be no way to decompose such an equality to a canonical form, restricting our ability to reason about such constraints.

There are other forms that are impossible to canonicalize, so this fact, alone, does not eliminate the line of inquiry we are headed down. However, consider the possibility that σ above is really a unification variable, α, and that $\kappa' = \star$. There is a straightforward solution here: $\alpha := \tau \triangleright \mathbf{sym}\,\gamma_0$. Yet without the ability to canonicalize, we would not find this easy solution. Even worse, it is quite possible that a tiny variation in the source Haskell program would produce the constraint $\alpha \sim_{\mathsf{N}} (\star \triangleright \mathbf{sym}\,\gamma_0)$, which is in fact canonical and would lead to setting α. It would be very unfortunate to have the typability of programs rest on so unsteady a foundation.

6.1.3 Conclusion: CO_KINDALT is untenable

From the sensible assumptions embodied in the definition of CO_KINDALT comes a fragile type inference algorithm. While it is possible that a different approach to inference could overcome this weakness, I have been unable to find one. The problem appears to stem from the fact that, with CO_KINDALT, nominal equality truly is incoherent – the placement of coercions in types matters. This fact means that we cannot “look through” casts during inference, despite the fact that it is often possible to move casts around (say, from one side of constraint to the other) without affecting a type's meaning. The importance of a fragile piece of information – exactly where the casts fall during constraint generation – leads to inevitable fragility in the overall algorithm.

6.2 Alternative: a nominal type equality implies nothing about the types' kinds

Though the details have not been worked out, my co-authors and I conjecture in Weirich et al. [19] that the $\mathbf{kind}\,\gamma$ coercion form is optional – it can be left out without ill effect. I see no reason that this fact would be changed in the system with roles. So, we must consider this possibility.

In the GHC implementation, the ability to extract a kind coercion from a type coercion is useful. This comes into play most often when it is necessary to form a *homogeneous* coercion. A key example is the implementation of type normalization. Given a type, possibly containing type family applications, we would like to produce a type without any reducible type family applications and the coercion that witnesses the reduction. Concretely, if we have **type instance** $F\ \mathit{Int} = \mathit{Bool}$, given $F\ \mathit{Int}$, we want to produce Bool and $\gamma : F\ \mathit{Int} \sim_{\mathsf{N}} \mathit{Bool}$. The coercion produced must be homogeneous, as only homogeneous equality is substitutive – that is, we must be able to replace the normalized type in for the original. Thus, during normalization, if $\tau_1 : \kappa_1$ normalizes to $\tau_2 : \kappa_2$ with $\gamma : \tau_1 \sim_{\mathsf{N}} \tau_2$, we can use $\tau_2 \triangleright \mathbf{sym}\,(\mathbf{kind}\,\gamma) : \kappa_1$ to substitute for τ_1.

Despite the convenience of using $\mathbf{kind}$ here, I believe it is strictly unnecessary. It seems quite possible to track changes of a type's kind during normalization (and other, similar algorithms in GHC) and then to homogenize using a separately-formed kind coercion. However, I have also been unable to find a simplification made possible by the omission of $\mathbf{kind}$, so it seems better to have $\mathbf{kind}$ than not.

6.3 Final decision: a nominal type equality implies a representational kind equality

I have thus adopted the following rules for kind extraction and coherence:

$$\frac{\Sigma;\Gamma \vdash \gamma : \tau_{1(\kappa_1)} \sim_\rho \kappa_{2(\kappa_2)}}{\Sigma;\Gamma \vdash \mathbf{kind}\,\gamma : \kappa_1 \sim_{\mathsf{R}} \kappa_2} \quad \text{CO\_KIND}$$

$$\frac{\begin{array}{c}\Sigma;\Gamma \vdash \gamma : \tau_1 \sim_\rho \tau_2 \\ \Sigma;\Gamma \vdash \tau_1 \triangleright \eta : \kappa_1'\end{array}}{\Sigma;\Gamma \vdash \gamma \triangleright \eta : (\tau_1 \triangleright \eta) \sim_\rho \tau_2} \quad \text{CO\_COHERENCE}$$

Regardless of the role of γ, $\mathbf{kind}\,\gamma$ is always a representational equality. This choice is not without its drawbacks, which I explore below. However, these drawbacks have, in practice, been more easily overcome than the drawbacks of other alternatives.

6.3.1 Nominal equality is more inclusive than it “should” be.

Suppose $\gamma : \mathit{Int} \sim_{\mathsf{R}} \mathit{Age}$, built from Age's newtype axiom. Then, the coercion $\langle 3 \rangle \triangleright \gamma$ proves $(3 \triangleright \gamma) \sim_{\mathsf{N}} 3$, a nominal equality. The fact that this proposition is provable is puzzling, because it involves a newtype axiom. Recall that nominal equality is source Haskell equality – it is intended that type inference can, without assistance, discover any nominal equality that can exist between two types. Yet, the whole point of newtypes is that type inference should not equate a newtype and its underlying representation, unless directed to do so by the programmer. Thus, our $\langle 3\rangle \triangleright \gamma$ coercion is suspect.

However, we can note a key fact here: the *kinds* of $3 \triangleright \gamma$ and 3 are different; the coercion $\langle 3\rangle \triangleright \gamma$ is heterogeneous. Thus, for a Haskell programmer to notice the fact that $3 \triangleright \gamma$ and 3 are nominally equal, there must be two nominally equal types, one of which expects a parameter of kind *Age* and the other of which expects a parameter of kind *Int*. This does not happen, and so we are saved. Understanding how this is so is best by way of a few examples.

Consider a basic *Proxy* type, with an explicit kind parameter:[11]

data *Proxy* $(k :: \star)$ $(a :: k) = P$

Both parameters k and a will be inferred to have representational roles. This means that we can prove *Proxy* k_1 a_1 is representationally equal to *Proxy* k_2 a_2 whenever $k_1 \sim_{\mathsf{R}} k_2$ and $a_1 \sim_{\mathsf{R}} a_2$.

Are* Proxy Age (MkAge 3) *and* Proxy Int 3 *representationally equal? Yes, as they should be. We can straightforwardly prove that $Age \sim_{\mathsf{R}} Int$ and use coherence to show that $3 \triangleright \gamma$ (which is the desugared form of *MkAge* 3) is representationally equal to 3.

Are* Proxy Age (MkAge 3) *and* Proxy Int 3 *nominally equal? No, because *Age* and *Int* are not nominally equal. The coherence rule allows us to prove that $3 \triangleright \gamma$ and 3 are nominally equal, but the overall types are still not nominally equal.

There is still a small lingering problem here. Even in today's Haskell, users can write *role annotations* which alter the default roles GHC infers for types [2]. For example, a programmer might write the following annotation for *Proxy*:

type *role Proxy representational nominal*

This annotation means that, to prove that *Proxy* k_1 a_1 and *Proxy* k_2 a_2 are representationally equal, k_1 can be representationally equal to k_2, but a_1 and a_2 are required to be nominally equal. With this annotation in place, we can *still* prove that *Proxy Age* (*MkAge* 3) is representationally equal to *Proxy Int* 3. This fact is slightly dissatisfying, because it means that GHC will "look through" the *MkAge* constructor. This fact may surprise some programmers, but the issue can come up only when a user has used a role annotation, and the problem does not threaten type safety. Furthermore, *Age* and *Int* are considered representationally equal only when the *MkAge* constructor is in scope, so the problem described here does not result in a loss of abstraction.

6.3.2 Type applications are hard to decompose.

Consider now trying to satisfy a wanted constraint $\beta_1 \; \beta_2 \sim_{\mathsf{N}}$ *Proxy* $\star$ *Int*, where $\beta_1 :: \alpha \to \star$ and $\beta_2 :: \alpha$, and Greek letters denote unification variables. The decomposition seems straightforward; we should reduce to $\beta_1 \sim_{\mathsf{N}}$ *Proxy* $\star$ and $\beta_2 \sim_{\mathsf{N}}$ *Int*. These equalities are almost canonical, except for the fact that they are heterogeneous. So, we create new wanted constraints from the kinds of the existing ones: $(\alpha \to \star) \sim_{\mathsf{R}} (\star \to \star)$ and $\alpha \sim_{\mathsf{R}} \star$. However, we see here that the kind equalities are representational, not nominal; this is a direct consequence of our choice of roles on $\mathbf{kind}\,\gamma$. After the constraint solver is done, α will still be left ambiguous, as a representational equality does not fix the value of a unification variable. Indeed, this ambiguity is correct here, as it is conceivable that the choice of α could have runtime significance through the selection of different class instances. (Consider, say, a **newtype** *Star* = *MkStar* $\star$, establishing a representational equality between *Star* and $\star$. Then, *Star* and $\star$ could have different instances for a certain class.)

We might imagine tweaking the canonicalization algorithm to produce nominal kind equalities here, but this would amount to "guessing" once again. Furthermore, when trying to canonicalize a *given* type application, we would be unable to produce a nominal kind equality in this scenario.

This issue is not merely a theoretical concern, either. This scenario came up when compiling the *Control.Arrow* module of GHC's standard library. Without the fix described shortly, that module failed to compile due to an ambiguous kind variable.

My solution to this problem is to add a new way of extracting a kind coercion from a type coercion, described by this rule:

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash \gamma : \tau_1\,\sigma_1 \sim_{\mathsf{N}} \tau_2\,\sigma_2 \\ \Sigma;\Gamma \vdash \sigma_1 : \kappa_1 \\ \Sigma;\Gamma \vdash \sigma_2 : \kappa_2\end{array}}{\Sigma;\Gamma \vdash \mathbf{kapp}\,\gamma : \kappa_1 \sim_{\mathsf{N}} \kappa_2} \quad \text{Co\_KAppTy}$$

Using $\mathbf{kapp}$, we can extract a nominal coercion relating the kinds of the arguments of a type application. Of course, simply adding this one rule would violate type safety – we must also add an extra requirement to the coercion form that relates type applications:

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash \gamma_1 : \tau_1 \sim_\rho \tau_2 \\ \Sigma;\Gamma \vdash \gamma_2 : \sigma_{1(\kappa_1)} \sim_{\mathsf{N}} \sigma_{2(\kappa_2)} \\ \Sigma;\Gamma \vdash \eta : \kappa_1 \sim_\rho \kappa_2 \\ \Sigma;\Gamma \vdash \tau_1\,\sigma_1 : \kappa_0 \qquad \Sigma;\Gamma \vdash \tau_2\,\sigma_2 : \kappa_0'\end{array}}{\Sigma;\Gamma \vdash \gamma_1\;\gamma_{2(\eta)} : \tau_1\,\sigma_1 \sim_\rho \tau_2\,\sigma_2} \quad \text{Co\_AppTy}$$

We see that to prove a nominal equality between $\tau_1\,\sigma_1$ and $\tau_2\,\sigma_2$, we must not only show that τ_1 and σ_1 are nominally equal to τ_2 and σ_2, respectively, but also that κ_1 equals κ_2. With this extra requirement on the application coercion form, rule Co_KAppTy becomes admissible, and therefore sound. See the submitted auxiliary material for the full details.

Having addressed the two shortcomings of choosing a representational role for $\mathbf{kind}\,\gamma$, I believe that this choice is the best and is the one implemented.

7. Maintaining sound type families

7.1 Overlap checking today

An open type family [3] in Haskell defines a function on types that can be extended. For example, we can write the following:

type family *F* $(a :: \star) :: \star$
type instance *F Int* = *Char* -- instance (1)

Then, perhaps in another module, we can add a new equation, thus:

type instance *F Bool* = *Double* -- instance (2)

With both **type instance** declarations in scope, GHC will reduce the type *F Int* to *Char* and *F Bool* to *Double*.

When adding equations to an open type family, it is necessary to make sure that the new equation does not overlap with the old one. For example, we cannot add the following instance to those above:

type instance *F a* = *String* -- instance (3)

This instance overlaps with both previous instances. If it were allowed, then type families would behave differently in different places, allowing a user to subvert the type system.

We thus need some way of eliminating the possibility of overlap. Because type family instances elaborate to axioms in System FC, our check must somehow ensure that the left-hand sides of the equations will never be nominally equal, even after arbitrary (well-typed) substitutions. In the case of the last type instance above, the

[11] This syntax – with explicit kinds – is actually legal in my implementation, as a natural consequence of combining types with kinds.

substitution $a \mapsto Bool$ would make the left-hand side overlap with the instance (2) above.

Our task is greatly simplified by the fact that type family equations cannot have type family applications on their left-hand side, much like how term-level patterns may not call functions. In today's Haskell, the overlap check is a simple, syntactic unification algorithm. If the left-hand sides fail to unify, then they must not overlap.

A similar check is needed in the reduction of closed type families. Consider the following:

```
type family G a where
  G Int = Bool
  G a   = Char
```

The equations in a closed type family are meant to be understood top-to-bottom. The definition for G means that $G\ Int$ reduces to $Bool$, and $G\ a$ reduces to $Char$, as long as a is *apart* from Int. By apart here, I mean that the argument to G must never become Int. For example, we cannot reduce $G\ (F\ a)$, because we don't (yet) know whether $F\ a$ is Int or not. The details of how apartness interacts with closed type family reduction are written out in full in Eisenberg et al. [4].

7.2 Type families with kind equalities

There are two aspects of the current system that cause trouble here: coercions can appear in types, and coercions can be abstracted over. The first problem is easily dispatched with. Simply erase coercions before checking for overlap and apartness. You can see this design choice in the coercion formation rules, Figure 5, which erases types before doing the closed-type-family apartness check, in rule CO_TFAXIOM.

The second problem – coercion abstraction – can be seen in an example:

```
type family G (a :: ⋆) :: ⋆
type instance G (∀ (c :: Int ∼ Bool). Bool) = Char
type instance G (∀ (c :: Bool ∼ Int). Int)  = Double
```

Those two left-hand sides do not unify, yet they can be proved nominally equal. We must reject these instance declarations.

Happily, the solution here is already existent in GHC: forbid quantified types in patterns. The part that comes with kind equalities is that we must *also* forbid this construction in System FC – hence the restricted grammar for type patterns ξ that we see in Figure 2.[12]

In today's Haskell (and in my implementation), quantified types are somewhat limited in source Haskell – Haskell remains predicative. Although inference with impredicativity has been tried in GHC [15], most of that functionality has been removed, as it had too many corner cases and induced too much confusing behavior.

System FC, on the other hand, has always been fully impredicative. It has been possible, then, to allow axioms to work with quantified types. With kind equalities, though, we must forbid this construction. Eliminating quantified types on the left-hand side of type family axioms is enough to establish the key *consistency* lemma, the most intricate part of the proof of the progress theorem. The statement and proof appear in the submitted auxiliary material.

8. A simpler equality to implement

Implementing a system such as the one described here is – for lack of a better word – fiddly. The problem is that the implementation works with yet a different equality relation than the ones discussed here: syntactic equality. GHC's function $eqType$ checks if two types are equal only up to α-equivalence.

What's challenging about syntactic equality is that it distinguishes between, say, Int and $Int \triangleright \langle\star\rangle$, and between τ, $\tau \triangleright \gamma \triangleright \mathbf{sym}\,\gamma$, and $\tau \triangleright (\gamma \mathbin{⨟} \mathbf{sym}\,\gamma)$. A key property of System FC – proof irrelevance – "fails" with syntactic equality, because the actual structure of proofs matters there.

These difficulties can be overcome, of course, with very careful management of casts, and many uses of coherence coercions. Yet, during implementation, the rigidity of syntactic equality was a constant source of bugs and engineering challenge.

The struggle to deal with syntactic equality led to a key insight: use a more relaxed equality internally. We thus want to replace syntactic equality with a new definitional equality. A first candidate for this new definitional equality is *erased equality* – when checking two types for equality, remove all casts and coercions first and then check. However, simple erased equality does not work, as it might relate types of different kinds; we need the definitional equality to be susbtitutive, so a heterogeneous equality just won't do.

Instead, I have chosen definitional equality to be erased equality, plus an additional kind check. Of course, the kinds are to be checked with the same choice of definitional equality. We avoid infinite regress by the fact that all kinds have type $\star$. (That is, it can be proved that if $\Sigma; \Gamma \vdash \tau : \kappa$, then $\Sigma; \Gamma \vdash \kappa : \star$.) This decision works very well in practice, and this redesign allowed me to remove over 1,000 lines of code with no apparent change in functionality.

Testsuite performance As of the time of writing, the implementation passes 3,561 of the 3,894 tests in the GHC testsuite. The vast majority of the remaining failures are due to changes in error messages; future work will include improving these. Notably, no program in the testsuite is accepted by my implementation in error; that is, every incorrect program is duly rejected at compile time.

9. Discussion

9.1 The $\star :: \star$ axiom and partial correctness

The language I present in this paper sports the $\star{::}\star$ axiom. This is in sharp contrast to more traditional dependently typed programming languages, such as Coq, Agda, and Idris, all of which have an infinite hierarchy of universes. In this infinite hierarchy, a standard type Int would have type $Type$, which in turn has type $Type_1$, which has $Type_2$, and so on. It has been established that the $\star :: \star$ axiom causes a system to be inconsistent as a logic [6] and can allow authors to write non-terminating type-level programs.

However, these flaws do not concern us in Haskell. Adding dependent-type features to Haskell is *not* an attempt to make GHC a proof assistant. All types are already inhabited, by $undefined$ at the term level, and by the following open **type family** $Any\ (k :: \star) :: k$ at the type level. If having $\star :: \star$ allows us another way to inhabit a type, it does not change the properties of the language. Due to its inconsistency as a logic, the best a Haskell programmer can hope for is *partial* correctness: if a Haskell program is ascribed to have a certain type, then it is known to have that type only if the program evaluates to a value in finite time. This guarantee has proved to be sufficient to Haskellers, who continue to use advanced type-level features despite the lack of total correctness.

There is intriguing related work, however, toward a Haskell termination checker [14] and pattern-match totality checker [9]. With these tools in place, it may be possible to provide even stronger compile-time guarantees to programmers. An additional exciting application is this related work is that it would allow GHC to optimize away certain parts of a program that exist only to prove type equality. Currently, all equality proofs must be executed at runtime; otherwise, we can't be sure that the equality is sound. If we could prove totality, though, running the proofs becomes unnecessary.

[12] Type patterns are also restricted in that all coercions must be bare coercion variables c. This is to allow the coercion to have any concrete form when instantiating.

More work needs to be done to make this a reality, but this is all an exciting direction to look to in the future.

9.2 Other related work

The previous paper describing the system implemented here [19] contains a thorough review of related work. The reader is encouraged to look there to see how System FC relates to the literature.

One new piece of related work is a recent proof by Kahrs and Smith [8]. That work contains a proof that non-overlapping rewrite systems have unique normal forms. If this proof can be adapted to the proof of type safety for System FC, it would close the gap in our proof, as detailed in Eisenberg et al. [4] and Breitner et al. [2].

9.3 Future work

There is much work left to do. Here are some starting points:

- The implementation discussed here necessarily does type inference to produce a well-typed System FC program from a source Haskell text. Although GHC's type inference algorithm did not require extensive modifications to make this work, the process of analyzing types is somewhat more involved than it was previously. Future work on type inference includes an in-depth analysis and explanation of the new algorithm.
- GHC supports *deferred type errors* [17], which allow a user to successfully compile a type-incorrect program. If the program then executes type-incorrect code, the running program halts with an error. With the kind equalities implemented here, it is possible to extend this idea to deferred kind errors. The details have yet to be worked out, however.
- With kind equalities in GHC, we are much closer to being able to implement a proper dependent quantifier into Haskell, along the lines of Gundry [7]. Working out the details and implementing is important future work along this line of research.

9.4 Conclusion

The fundamental tension that causes the incorporation of kind equalities into GHC to be challenging is that there are many notions of equality (and equality-like relations, such as sub-kinding) in the compiler. In particular, there seems to be tension between performance and abstraction; it is the interaction between these two desiderata that gave birth to **newtype**s and, later, roles, in Haskell.

The solutions proposed in this work vary in elegance. In my opinion, levity polymorphism seems like a nice solution to a slightly thorny problem. On the other hand, the lack of a clean fit between roles and kind equalities is dissatisfying, as is the incompleteness of the power of type family matching. Yet, I believe the solutions put forth here are adequately expressive, and provably implementable. With kind equalities now implemented, users can start to adopt yet richer types and continue to push the limits of practical, strongly typed programming.

Acknowledgments

Thanks to Stephanie Weirich for ongoing collaboration, encouragement, and mentorship. Thanks, also, to Simon Peyton Jones for collaboration on GHC and for the idea about levity polymorphism.

This paper is typeset with the help of lhs2TeX[13] and ott [12].

This work is supported by a Microsoft Research Graduate Student Fellowship and by the National Science Foundation under Grant No. 1116620.

[13] `http://www.andres-loeh.de/lhs2tex/`

References

[1] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Prog.*, 23, 2013.

[2] Joachim Breitner, Richard A. Eisenberg, Simon Peyton Jones, and Stephanie Weirich. Safe zero-cost coercions for Haskell. In *International Conference on Functional Programming*, ICFP '14. ACM, 2014.

[3] Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyon Jones. Associated type synonyms. In *International Conference on Functional Programming*, ICFP '05. ACM, 2005.

[4] Richard A. Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. Closed type families with overlapping equations. In *Principles of Programming Languages*, POPL '14. ACM, 2014.

[5] Jeff Epstein, Andrew P. Black, and Simon Peyton Jones. Towards haskell in the cloud. In *Haskell Symposium*. ACM, 2011.

[6] J.-Y. Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieu*. PhD thesis, Université Paris 7, 1972.

[7] Adam Gundry. *Type Inference, Haskell and Dependent Types*. PhD thesis, University of Strathclyde, 2013.

[8] Stefan Kahrs and Connor Smith. Non-ω-overlapping TRSs have unique normal forms. Draft, submitted for publication, 2015.

[9] Georgios Karachalias, Tom Schrijvers, Dimitrios Vytiniotis, and Simon Peyton Jones. Pattern match warnings meet GADTs, at last. Draft, submitted to ICFP 2015.

[10] Stefan Monnier and David Haguenauer. Singleton types here, singleton types there, singleton types everywhere. In *Programming languages meets program verification*, PLPV '10. ACM, 2010.

[11] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *International Conference on Functional Programming*, ICFP '06. ACM, 2006.

[12] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective tool support for the working semanticist. *J. Funct. Prog.*, 20, 2010.

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

[14] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton Jones. Refinement types for haskell. In *International Conference on Functional Programming*, ICFP '14. ACM, 2014.

[15] Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. Boxy types: Inference for higher-rank types and impredicativity. In *International Conference on Functional Programming*, ICFP '06. ACM, 2006.

[16] Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. OutsideIn(X): Modular type inference with local assumptions. *J. Funct. Prog.*, 21, 2011.

[17] Dimitrios Vytiniotis, Simon Peyton Jones, and José Pedro Magalhães. Equality proofs and deferred type errors: A compiler pearl. In *International Conference on Functional Programming*, ICFP '12. ACM, 2012.

[18] Stephanie Weirich, Dimitrios Vytiniotis, Simon Peyton Jones, and Steve Zdancewic. Generative type abstraction and type-level computation. In *POPL*. ACM, 2011.

[19] Stephanie Weirich, Justin Hsu, and Richard A. Eisenberg. System FC with explicit kind equality. In *International Conference on Functional Programming*, ICFP '13. ACM, 2013.

[20] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Principles of Programming Languages*, POPL '03. ACM, 2003.

[21] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving Haskell a promotion. In *Types in Language Design and Implementation*, TLDI '12. ACM, 2012.