A Role for Dependent Types in Haskell

STEPHANIE WEIRICH, University of Pennsylvania, USA
PRITAM CHOUDHURY, University of Pennsylvania, USA
ANTOINE VOIZARD, University of Pennsylvania, USA
RICHARD A. EISENBERG, Bryn Mawr College, USA

Modern Haskell supports *zero-cost* coercions, a mechanism where types that share the same run-time representation may be freely converted between. To make sure such conversions are safe and desirable, this feature relies on a mechanism of *roles* to prohibit invalid coercions. In this work, we show how to incorporate roles into dependent types systems and prove, using the Coq proof assistant, that the resulting system is sound. We have designed this work as a foundation for the addition of dependent types to the Glasgow Haskell Compiler, but we also expect that it will be of use to designers of other dependently-typed languages who might want to adopt Haskell's safe coercions feature.

CCS Concepts: • **Software and its engineering** → *Functional languages*; *Polymorphism*; • **Theory of computation** → *Type theory*.

Additional Key Words and Phrases: Haskell, Dependent Types

ACM Reference Format:
Stephanie Weirich, Pritam Choudhury, Antoine Voizard, and Richard A. Eisenberg. 2019. A Role for Dependent Types in Haskell. *Proc. ACM Program. Lang.* 3, ICFP, Article 101 (August 2019), 29 pages. https://doi.org/10.1145/3341705

1 COMBINING SAFE COERCIONS WITH DEPENDENT TYPES

A *newtype* in Haskell[1] is a user-defined algebraic datatype with exactly one constructor; that constructor takes exactly one argument. Here is an example:

```
newtype HTML = MkHTML String
```

This declaration creates a generative abstraction; the `HTML` type is *new* in the sense that it is not equal to any existing type. We call the argument type (`String`) the *representation type*. Because a newtype is isomorphic to its representation type, the Haskell *compiler* uses the same in-memory format for values of these types. Thus, creating a value of a newtype (i.e., calling `MkHTML`) is free at runtime, as is unpacking it (i.e., using a pattern-match).

However, the Haskell *type checker* treats the newtype and its representation type as wholly distinct, meaning programmers cannot accidentally confuse `HTML` objects with `String` objects. We thus call newtypes a *zero-cost* abstraction: a convenient compile-time distinction with no cost

[1]In this paper, we use "Haskell" to refer to the language implemented by the Glasgow Haskell Compiler (GHC), version 8.6.

Authors' addresses: Stephanie Weirich, Computer and Information Science, University of Pennsylvania, 3330 Walnut St, Philadelphia, PA, 19104, USA, sweirich@cis.upenn.edu; Pritam Choudhury, Computer and Information Science, University of Pennsylvania, USA, pritam@seas.upenn.edu; Antoine Voizard, Computer and Information Science, University of Pennsylvania, USA, voizard@seas.upenn.edu; Richard A. Eisenberg, Computer Science, Bryn Mawr College, 101 N. Merion Ave, Bryn Mawr, PA, 19010, USA, rae@cs.brynmawr.edu.

2475-1421/2019/8-ART101
https://doi.org/10.1145/3341705

to runtime efficiency. A newtype exported from a module without its constructor is an abstract datatype; clients do not have access to its representation.

Inside the defining module, a newtype is a translucent abstraction: you can see through it with effort. The *safe coercions* [Breitner et al. 2016] extension to the Glasgow Haskell Compiler (GHC) reduces this effort through the availability of the **coerce** primitive. For example, as `HTML` and `String` are represented by the same bits in memory, so are the lists `[HTML]` and `[String]`. Therefore, we can define a no-cost operation that converts the former to the latter by coercing between representationally equal types.

```
unpackList :: [HTML] -> [String]
unpackList = coerce
```

However, **coerce** must be used with care. Not every structure is appropriate for conversion. For example, converting a `Map HTML Int` to a `Map String Int` would be disastrous if the ordering relation used for keys differs between `HTML` and `String`. Even worse, allowing **coerce** on types that use the **type family** feature [Chakravarty et al. 2005] leads to unsoundness. Haskell thus includes *role annotations* for type constructors that indicate whether it is appropriate to lift newtype-coercions through abstract structures, such as `Map`.

The key idea of *safe coercions* is that there are two different notions of type equality at play—the usual definition of type equality, called *nominal* equality, that distinguishes between the types `HTML` and `String`, and *representational* equality that identifies types suitable for coercion. Some type constructor arguments are not congruent with respect to representational equivalence, so role annotations prohibit the derivation of these undesired equalities.

1.1 Extending GHC with Dependent Types

Recent work has laid out a design for Haskell extended with *dependent types* [Eisenberg 2016; Gundry 2013; Weirich et al. 2013, 2017] and there is ongoing work dedicated to implementing this theory [Xie and Eisenberg 2018].[2] Dependent types are desirable for Haskell because they increase the ability to create abstractions. Indexing types by terms allows datatypes to maintain application-specific invariants, increasing program reliability and expressiveness [Oury and Swierstra 2008; Weirich 2014, 2017].

However, even though dependent type theories are fundamentally based on a rich definition of type equality, it is not clear how best to incorporate roles and safe coercions with these systems. In the context of GHC, this omission has interfered with the incorporation of dependent types. To make progress we must reconcile the practical efficiency of safe, zero-cost coercions with the power of dependent types. We need to know how a use of **coerce** interacts with type equality, and we must resolve how roles can be assigned in the presence of type dependency.

1.2 Contribution

The primary contribution of this work is System DR, a language that safely integrates dependent types and roles. The starting point for our design is System D, the core language for Dependent Haskell from Weirich et al. [2017]. Though this language has full-spectrum dependent types, it is not a standard dependent type theory: it admits logical inconsistency and the $\star : \star$ axiom, along with support for equality assumptions and type erasure. System DR extends System D with roles, based on the existing design [Breitner et al. 2016] as it must be realizable in GHC.

Integrating roles with Dependent Haskell's type system is not straightforward. Unpacking the point above, our paper describes the following aspects of our contribution:

[2]Also, see https://github.com/ghc-proposals/ghc-proposals/pull/102

- To model the two different notions of type equality, we index the type system's definition of equality by *roles*, using the declared roles of abstract constructors to control the sorts of equivalences that may be derived. In Section 3 we describe how roles and newtype axioms interact with a minimal dependent type system. In particular, type equality is based on computation, so we also update the operational semantics to be role-sensitive. Newtypes evaluate to their representations only at the representational role; at the nominal role, they are values. In contrast, type family axioms step to their definitions at all roles.
- In Section 4 we extend the basic system with support for the features of GHC. We start with a discussion of the interaction of roles with System D's features of *coercion abstraction* (Section 4.1) and *irrelevant arguments* (Section 4.2).
- Supporting GHC's type families requires an operation for intensional type-analysis as type families branch on the head constructors of types. Therefore, in Section 4.3 we add a *case* expression to model this behavior. Because our language is dependently-typed, this expression supports both compile-time type analysis (as in type families) and run-time type analysis (i.e. typecase).
- Our type equality includes *nth* projections, a way to decompose equalities between type constants. We describe the rules that support these projections and how they interact with roles in Section 4.4.
- Our mechanized proof in Coq, available online,[3] is presented in Section 5. Proving safety is important because the combination of **`coerce`** and type families, without roles, is known to violate type safety.[4] This work provides the first mechanically checked proof of the safety of this combination.
- Our work solves a longstanding issue in GHC, known as the *Constraint vs. Type problem*. In Section 6.1 we describe this problem and how defining `Constraint` as a `newkind` resolves this tension.
- Our work sheds new light on the semantics of safe-coercions. Prior work [Breitner et al. 2016], includes a *phantom* role, in addition to the nominal and representational roles. This role allows free conversion between the parameters of type constructors that do not use their arguments. In 6.2 we show that this role need not be made primitive, but instead can be encoded using irrelevant arguments.
- We also observe that although our work integrates roles and dependent types at the level of GHC's core intermediate language, we lack a direct specification of source Haskell augmented with the **`coerce`** primitive. The problem, which we describe in detail in Section 6.3, is that it is difficult to give an operational semantics of **`coerce`**: reducing it away would violate type preservation, but it quite literally has no runtime behavior. Instead, in 6.4 we argue that our core language can provide a (type-sound) specification through elaboration.

Although our work is tailored to our goal of adding dependent types to Haskell, an existing language with safe-coercions, we also view it as a blueprint for adding safe-coercions to dependently-typed languages. Many dependently-typed languages include features related to the ones discussed here. For example, some support semi-opaque definitions, such as Coq's `Opaque` and `Transparent` commands. Such definitions often guide type-class resolution [Brady 2013; Devriese and Piessens 2011; Sozeau and Oury 2008], so precise control over their unfolding is important. Cedille includes zero-cost coercions [Diehl et al. 2018] and Idris has recently added experimental support for

[3] https://github.com/sweirich/corespec
[4] This safety violation was originally reported as http://ghc.haskell.org/trac/ghc/ticket/1496.

typecase[5]. Because the design considerations of these languages differ from that of GHC, we compare our treatment of roles to modal dependent type theory in Section 8.

Our Coq proof is a significant extension of prior work [Weirich et al. 2017]. Our work is fully integrated—the same source is used to generate the LaTeXinference rules, Coq definitions, and variable binding infrastructure. We use the tools Ott [Sewell et al. 2010] and LNgen [Aydemir and Weirich 2010] to represent the binding structure using a *locally nameless representation* [Aydemir et al. 2008]. Our code includes over 21k nonblank, noncomment lines of Coq definitions and proofs. This total includes 1.8k lines generated directly from our Ott definitions, and 7k lines generated by LNgen.

In the next section, we review existing mechanisms for newtypes and safe coercions in GHC in more detail and lay out the considerations that govern their design. We present our new system starting in Section 3.

2 NEWTYPES AND SAFE COERCIONS IN HASKELL

2.1 Newtypes Provide Zero-Cost Abstractions

We first flesh out the HTML example of the introduction, by considering this Haskell module:

```haskell
module Html( HTML, text, unHTML, ... ) where
newtype HTML = MkHTML String

unHTML :: HTML -> String
unHTML (MkHTML s) = s

text :: String -> HTML
text s = MkHTML (escapeSpecialCharacters s)

instance IsString HTML where
  fromString = text
```

As above, HTML is a newtype; its representation type is String. This means that HTMLs and Strings are represented identically at runtime and that the MkHTML constructor compiles into the identity function. However, the type system keeps HTML and String separate: a function that expects an HTML will *not* accept something of type String.

Even in this small example, the Haskell programmer benefits from the newtype abstraction. The module exports the type HTML but not its data constructor MkHTML. Accordingly, outside the module, the only way to construct a value of this type is to use the text function, which enforces the invariant of the data structure. By exporting the type HTML without its data constructor, the module ensures that the type is abstract—clients cannot make arbitrary strings into HTML—and thereby prevents, for instance, cross-site scripting attacks.

Naturally, the author of this module will want to reuse functions that work with Strings to also work with values of type HTML—even if those functions actually work with, say, lists of Strings. To support this reuse, certain types, including functions (->) and lists [], allow us to *lift* coercions between String and HTML. For example, suppose we wish to break up chunks of HTML into their constituent lines. We define

```haskell
linesH :: HTML -> [HTML]
linesH = coerce lines
```

[5]https://gist.github.com/edwinb/25cd0449aab932bdf49456d426960fed

Using Haskell's standard `lines :: String -> [String]` function, we have now, with minimal effort, lifted its action to work over `HTML`. Critically, the use of **`coerce`** above is allowed only when the `MkHTML` constructor is in scope; in other words, `linesH` can be written in the `Html` module but not outside it. In this way, the author of `HTML` has a chance to ensure that any invariants of the `HTML` type are maintained if an `HTML` chunk is split into lines.

2.2 Newtypes Guide Type-Directed Programming

Newtypes allow more than just abstraction; they may also be used to guide type-directed programming. For example, the sorting function in the base library has the following type.

```
sort :: Ord a => [a] -> [a]
```

The `Ord` type class constraint means that sorting requires a comparison function. When this function is called, the standard comparison function for the element type will be used. In other words, the type of the list determines how it is sorted.

Suppose our application sometimes must work with sorted lists of `HTML` chunks. For efficiency reasons, we wish to partition our sorted lists into a region where all chunks start with a tag (that is, the `'<'` character) and a region where no chunk starts with a tag. To that end, we define a custom `Ord` instance that will sort all `HTML` chunks that begin with < before all those that do not.

```
instance Ord HTML where
  MkHTML left `compare` MkHTML right
    | tagged left == tagged right = left `compare` right
    | tagged left                 = LT
    | otherwise                   = GT
    where
      tagged ('<':_) = True
      tagged _       = False
```

Now, when we `sort` a list of chunks, we can be confident that the sorting algorithm will use our custom comparison operation. The validity of this approach vitally depends on the generative nature of newtypes: if the type-checker could confuse `HTML` with `String`, we could not be sure whether type inference would select our custom ordering or the default lexicographic ordering.

Newtypes can also be used to locally override the behavior of the sorting operation. For example, the newtype `Down a`, defined in the Haskell base library, is isomorphic to its representation `a`, but reverses the comparison in its instance for `Ord`. Therefore, to sort a list in reverse order, use **`coerce`** to change the element type from `a` to `Down a`, thus modifying the comparison operation used by `sort`.

```
sortDown :: forall a. Ord a => [a] -> [a]
sortDown x = coerce (sort (coerce x :: [Down a]))
```

More generally, GHC's recent `DerivingVia` extension [Blöndal et al. 2018], based on **`coerce`**, uses newtypes and their zero-cost coercions to extend this idea. This extension allows programmers to effectively write templates for instances; individual types need not write their own class instances but can select among the templates, each one embodied in a newtype.

2.3 The Problem with Unfettered coerce

We have shown that functions and lists support lifting coercions, but doing so is not safe for all types. Consider this (contrived) example:

```
type family Discern t where
  Discern String = Bool
  Discern HTML   = Char
data D a = MkD (Discern a)
```

The `Discern` type family [Chakravarty et al. 2005; Eisenberg et al. 2014] behaves like a function, where `Discern String` is `Bool` and `Discern HTML` is `Char`. Thus, a `D String` wraps a `Bool`, while a `D HTML` wraps a `Char`. Being able to use **`coerce`** to go between `D String` and `D HTML` would be disastrous: those two types have *different* runtime representations (in contrast to `[String]` and `[HTML]`). The goal of roles is to permit safe liftings (like for lists) and rule out unsafe ones (like for `D`).

Therefore, to control the use of **`coerce`**, all datatype and newtype parameters are assigned one of two roles: *nominal* and *representational*.[6] In a nominally-roled parameter, the *name* of the type provided is material to the definition, not just its representation. The one parameter of `D` is assigned a nominal role because the definition of `D` distinguishes between the names `String` and `HTML`. We cannot safely coerce between `D String` and `D HTML`, because these two types have different representations. In contrast, the type parameter of list has a representational role; coercing between `[String]` and `[HTML]` is indeed safe.

Roles are assigned either by user annotation or by *role inference* [Breitner et al. 2016, Section 4.6]. The safety of user-provided role annotations is ensured by the compiler; the user would be unable to assign a representational role to the parameter of `D`.

2.4 Representational Equality

The full type of **`coerce`** is **`Coercible`** `a b` **`=>`** `a -> b`. That is, we can safely coerce between two types if those types are **`Coercible`**. The pseudo-class **`Coercible`** (it has custom solving rules and is thus not a proper type class) is an equivalence relation; we call it *representational equality*. We can thus coerce between any two representationally equal types. Representational equality is coarser than Haskell's standard type equality (also called *nominal equality*): not only does it relate all pairs of types that are traditionally understood to be equal, it also relates newtypes to their representation types.

Crucially, representational equality relates datatypes whose parameters have the relationship indicated by the datatype's parameters' roles. Thus, because the list type's parameter has a representational role, `[ty1]` is representationally equal to `[ty2]` iff `ty1` is representationally equal to `ty2`. And because `D`'s parameter has a nominal role, `D ty1` is representationally equal to `D ty2` iff `ty1` is nominally equal to `ty2`.

In addition to the *lifting* rules sketched above, representational equality also relates a newtype to its representation type, but with this caveat: this relationship holds only when the newtype's constructor is in scope. This caveat is added to allow the `Html` module to enforce its abstraction barrier. If a newtype were *always* representationally equal to its representation, then any client of `Html` could use **`coerce`** in place of the unavailable constructor `MkHTML`, defeating the goal of abstraction.

2.5 Design Considerations

The system for safe-coercions laid out in Breitner et al. [2016] is subject to design constraints that arise from the context of integration with the Haskell language. In particular, safe coercions are

[6]The implementation in GHC supports a third role, *phantom*, which behaves somewhat differently from the other two. We ignore it for the bulk of this paper, returning to it in Section 6.2.

Grammar

term/type variables	x		
constants	F, T		
roles	R	$::=$	$\mathbf{Nom} \mid \mathbf{Rep}$
application flags	ν	$::=$	$R \mid +$
terms, types	a, b, A, B	$::=$	$\star \mid x \mid F \mid \lambda x.b \mid a\ b^{\nu} \mid \Pi x\!:\!A.B$
contexts	Γ	$::=$	$\varnothing \mid \Gamma, x\!:\!A$
signatures	Σ	$::=$	$\varnothing \mid \Sigma \cup \{T : A \mathbin{@} \overline{R}\} \mid \Sigma \cup \{F : A \mathbin{@} \overline{R}\ \text{where}\ p \sim_R a\}$
patterns	p	$::=$	$F \mid p\ x^R$

Fig. 1. Syntax of core language

considered an advanced feature and should have minimal interaction with the rest of the language. In other words, Haskell programmers should not need to think about roles if they never use `coerce`.

This separation between types and kinds was not present in the first design of a role system for Haskell [Weirich et al. 2011]. Due to its complexity, that first system was never integrated into GHC. Instead, by keeping roles separate from types, Breitner et al. [2016] simplified both the implementation of `coerce` (i.e., it was easier to extend the compiler) and the language specification, as programmers who do not use `coerce` need not understand roles.

Keeping types and roles separate imposes two major constraints on the design of System DR.

- First, the type checking judgment should not also check roles. In the system that we present in the next section, the type checking judgment $\Gamma \vDash a : A$ does not depend on the role-checking judgment $\Omega \vDash a : R$. The only interaction between these two systems is confined to checking the role annotations on top-level axioms. (In contrast, in the first version of the system, type and role checking occurred together in a single judgment.)
- Second, the syntax of types and kinds should not include roles. In the first version of the system, the kinds of type constructors included role information for parameters. However, this means that all users of higher-order polymorphism must understand (and choose) these roles. Instead, Breitner et al. [2016] does not modify the syntax of kinds, safely approximating role information with the nominal role when necessary.

In practice, the loss of expressiveness due to this simplification has not been significant and roles have proven to be a popular extension in GHC. However, we return to this discussion in Section 8, when we compare our design with the framework provided by modal dependent type theory.

3 A CALCULUS WITH DEPENDENT TYPES AND ROLES

We now introduce System DR, a dependently-typed calculus with role-indexed equality. To make our work more approachable, we present this calculus incrementally, starting with the core ideas. In this section, we start with a minimal calculus that contains only dependent functions, constants and axioms. In Section 4, we extend the discussion to full System DR, including case analysis, irrelevant arguments, coercion abstraction, and decomposition rules.

System DR is intended to model an extension of FC [Sulzmann et al. 2007], the explicitly-typed intermediate language of GHC. As an intermediate language, it does not need to specify Haskell's type inference algorithm or include features, like type classes, that exist only in Haskell's source language.

$\boxed{\Gamma \vDash a : A}$

$$\textsc{AE-Star}\ \frac{\vdash \Gamma}{\Gamma \vDash \star : \star} \qquad \textsc{AE-Var}\ \frac{\vdash \Gamma \qquad x : A \in \Gamma}{\Gamma \vDash x : A} \qquad \textsc{AE-Pi}\ \frac{\Gamma, x{:}A \vDash B : \star}{\Gamma \vDash \Pi x{:}A.B : \star} \qquad \textsc{AE-Abs}\ \frac{\Gamma, x{:}A \vDash a : B}{\Gamma \vDash \lambda x.a : \Pi x{:}A.B}$$

$$\textsc{AE-App}\ \frac{\Gamma \vDash b : \Pi x{:}A.B \qquad \Gamma \vDash a : A}{\Gamma \vDash b\ a^{+} : B\{a/x\}} \qquad \textsc{AE-TApp}\ \frac{\Gamma \vDash b : \Pi x{:}A.B \qquad \Gamma \vDash a : A \qquad \text{Roles}\ (b) = R, \overline{R}}{\Gamma \vDash b\ a^{R} : B\{a/x\}}$$

$$\textsc{AE-Conv}\ \frac{\Gamma \vDash a : A \qquad \Gamma \vDash A \equiv_{\mathbf{Rep}} B : \star}{\Gamma \vDash a : B} \qquad \textsc{AE-Const}\ \frac{\vdash \Gamma \qquad T : A\ @\ \overline{R} \in \Sigma_0}{\Gamma \vDash T : A} \qquad \textsc{AE-Fam}\ \frac{\vdash \Gamma \qquad F : A\ @\ \overline{R}\ \text{where}\ p \sim_{R_1} a \in \Sigma_0}{\Gamma \vDash F : A}$$

$\boxed{\Gamma \vDash a \equiv_R b : A}$

$$\textsc{AE-Sub}\ \frac{\Gamma \vDash a \equiv_{R_1} b : A \qquad R_1 \leq R_2}{\Gamma \vDash a \equiv_{R_2} b : A} \qquad \textsc{AE-Beta}\ \frac{\Gamma \vDash a_1 : B \qquad \vDash a_1 \to^{\beta}_{R} a_2}{\Gamma \vDash a_1 \equiv_R a_2 : B}$$

$$\textsc{AE-PiCong}\ \frac{\Gamma \vDash A_1 \equiv_R A_2 : \star \qquad \Gamma, x{:}A_1 \vDash B_1 \equiv_R B_2 : \star}{\Gamma \vDash (\Pi x{:}A_1.B_1) \equiv_R (\Pi x{:}A_2.B_2) : \star} \qquad \textsc{AE-AbsCong}\ \frac{\Gamma, x{:}A \vDash b_1 \equiv_R b_2 : B}{\Gamma \vDash (\lambda x.b_1) \equiv_R (\lambda x.b_2) : (\Pi x{:}A.B)}$$

$$\textsc{AE-AppCong}\ \frac{\Gamma \vDash a_1 \equiv_R b_1 : \Pi x{:}A.B \qquad \Gamma \vDash a_2 \equiv_{\mathbf{Nom}} b_2 : A}{\Gamma \vDash a_1\ {a_2}^{+} \equiv_R b_1\ {b_2}^{+} : B\{a_2/x\}}$$

$$\textsc{AE-TAppCong}\ \frac{\begin{array}{c}\Gamma \vDash a_1 \equiv_{R'} b_1 : \Pi x{:}A.B \qquad \Gamma \vDash a_2 \equiv_{R \wedge R'} b_2 : A \\ \text{Roles}\ (a_1) = R, \overline{R} \qquad \text{Roles}\ (b_1) = R, \overline{R} \qquad \Gamma \vDash b_1\ {b_2}^{R} : B\{a_2/x\}\end{array}}{\Gamma \vDash a_1\ {a_2}^{R} \equiv_{R'} b_1\ {b_2}^{R} : B\{a_2/x\}}$$

$$\textsc{AE-Refl}\ \frac{\Gamma \vDash a : A}{\Gamma \vDash a \equiv_R a : A} \qquad \textsc{AE-Sym}\ \frac{\Gamma \vDash b \equiv_R a : A}{\Gamma \vDash a \equiv_R b : A} \qquad \textsc{AE-Trans}\ \frac{\Gamma \vDash a \equiv_R a_1 : A \qquad \Gamma \vDash a_1 \equiv_R b : A}{\Gamma \vDash a \equiv_R b : A}$$

$\boxed{\text{Roles}\ (a) = \overline{R}}$

$$\textsc{RolePath-AbsConst}\ \frac{F : A\ @\ \overline{R} \in \Sigma_0}{\text{Roles}\ (F) = \overline{R}} \qquad \textsc{RolePath-Const}\ \frac{F : A\ @\ \overline{R}\ \text{where}\ p \sim_{R_1} a \in \Sigma_0}{\text{Roles}\ (F) = \overline{R}}$$

$$\textsc{RolePath-TApp}\ \frac{\text{Roles}\ (a) = R_1, \overline{R}}{\text{Roles}\ (a\ b^{R_1}) = \overline{R}} \qquad \textsc{RolePath-App}\ \frac{\text{Roles}\ (a) = \mathbf{Nom}, \overline{R}}{\text{Roles}\ (a\ b^{+}) = \overline{R}}$$

Fig. 2. Typing and definitional equality for core language

Furthermore, the goal of our design of System DR is to describe what terms type check and how they evaluate. Like System D from prior work [Weirich et al. 2017], this calculus need not have decidable type checking for this purpose. Instead, once we have determined the language that we want, we can then figure out how to annotate it in the implementation with enough information to make type checking simple and syntax directed. The connection between System D and System DC in prior work provides a roadmap for this (fairly mechanical) process. Furthermore, this process is also constrained by implementation details of GHC that are beyond the scope of this paper, so we do not include an annotated system here.

Therefore, the core calculus that we start with is a Curry-style dependently-typed lambda-calculus with a single sort $\star$. The syntax is shown in Figure 1. As in pure type systems [Barendregt 1991], we have a shared syntax for terms and types, but, as we don't require decidable type checking, there are no typing annotations on function binders. This syntax has been decorated with role information in two places—applications are marked by flags (ν) and declarations of data type and newtype constants in the signature (Σ) include role annotations. Application flags are not needed for source Haskell—they are easily added via elaboration and their presence here is a mere technical device to make role information easily accessible.

Roles R are drawn from a lattice, with bottom element **Nom** for Haskell's nominal role and top element **Rep** for the representational role. We use $R_1 \leq R_2$ to denote the ordering within the lattice. The operation $(R_1 \wedge R_2)$ calculates the greatest lower bound of the two roles. For concreteness, this paper fixes that lattice to the two element lattice, which is all that is needed for GHC. However, treating this lattice abstractly allows us to define the type system more uniformly.

The rules for typing and definitional equality for this fragment are shown in Figure 2.[7] These rules are implicitly parameterized over a global signature Σ_0 of type constant declarations.

Typing Relation. Most rules of the typing relation are standard for dependently-typed languages. Because Haskell includes nontermination, we do not need include a universe hierarchy, instead using the $\star : \star$ axiom [Cardelli 1986]. The novel rules are the application rules (rules AE-App and AE-TApp), the conversion rule (rule AE-Conv) and the rules for constants and axioms (rules AE-Const and AE-Fam), all discussed below.

Role-Indexed Type Equality. In System DR, the equality relation is indexed by a role that determines whether the equality is *nominal* or *representational*. The judgment $\Gamma \vDash a \equiv_R b : A$ defines when the terms a and b, of type A, are equal *at role R*. The rules for this judgment appear in the middle of Figure 2. This relation is defined as essentially the language's small-step operational semantics, closed over reflexivity, symmetry, transitivity, and congruence. (Note that because System DR allows nontermination, the equality judgment is not a decidable relation.)

The role sensitivity of the equality relation derives from the fact that System DR's small-step operational semantics, written $a \rightarrow^{\beta}_{R} b$, is also role-sensitive. Specifically, the role in the small-step relation determines whether top-level definitions can unfold to their right-hand sides or are kept abstract (see rule ABeta-Axiom, in Figure 3).

For example, we have that **HTML** $\rightarrow^{\beta}_{\mathbf{Rep}}$ **String**, but at role **Nom**, the expression **HTML** is treated as a value. This step is reflected into the equality relation via rule AE-Beta.

Conversion. Dependently-typed languages use definitional equality for conversion: allowing the types of terms to be implicitly replaced with equal types. In source Haskell, conversion is available for all types that are nominally equal. The **coerce** primitive is required to convert between types

[7]For the purposes of presentation, these rules presented in this section are simplified versions of the rules that we use in our proofs. The complete listing of rules is available in the extended version of this paper [Weirich et al. 2019].

that are representationally equal. This primitive ensures that newtype distinctions are maintained by default but are erasable when desired.

However, System DR is intended to define GHC's intermediate language, so we can assume that the source language type checker has already made sure that users do not confuse `HTML` and `String`. Instead, the optimizer is free to conflate these types, for great benefit.

Therefore System DR does not include a **coerce** primitive. Instead, the conversion rule, rule AE-Conv, allows conversion using the coarsest relation, *representational* equality. This choice simplifies the design because all uses of coercion are implicit; there are no special rules in the equality relation or operational semantics. The downside of this design is that System DR is not a definition of source Haskell, an issue that we return to in Section 6.4.

3.1 Role Annotations and Application Congruence

Haskell allows data type and newtype constants to be optionally ascribed with *role annotations* for their parameters. (Definitions without role ascriptions get their roles inferred [Breitner et al. 2016, Section 4.6].) These role annotations control what equalities can be derived about these constants. For example, `Maybe` has a representational parameter so `Maybe HTML` is representationally equal to `Maybe String`. However, the type `Set HTML` can be prevented from being coercible to `Set String` by annotating its parameter with the nominal role.

In System DR, a constant, like **Maybe**, is an opaque declaration in the signature Σ_0 of the form $T : A \mathbin{@} \overline{R}$. This declaration specifies the type A of the constant T, as well as a list of roles $\overline{R}$ for its parameters. (We assume that role inference has already happened, so all constants include role annotations.) For example, the signature might declare constants $\mathbf{Set} : \star \to \star \mathbin{@} \mathbf{Nom}$ and $\mathbf{Maybe} : \star \to \star \mathbin{@} \mathbf{Rep}$ with their usual types and roles.

The key idea from Breitner et al. [2016] is that the equality rule for applications headed by constants uses these declared role annotations to determine how to compare their arguments. We adapt this idea in this context using *application flags*, ν, marking arguments in applications. Consistent usage of flags is checked by the typing judgment using rule AE-TApp. If the head of the application is a constant, then this rule ensures that the flag must be the one calculated by the (partial) function Roles $(a) = \overline{R}$, shown at the bottom of Figure 2. For example, $\mathbf{Set}\ \mathbf{String}^{\mathbf{Nom}}$ and $\mathbf{Maybe}\ \mathbf{Int}^{\mathbf{Rep}}$ are valid terms. However, role annotations are optional and can be replaced by the application flag $+$, in which case rule AE-App is used to check the term.

Application Congruence. The equality rule AE-TAppCong defines when two role-annotated applications $a_1\ a_2{}^R$ and $b_1\ b_2{}^R$ are equal at some (other) role R'. This rule is most interesting when R' is **Rep**—it explains why `Maybe HTML` and `Maybe String` are representationally equal but `Set String` and `Set HTML` are not. Here, applications such as `Maybe String` use role annotations on their arguments to enable this rule. In the case of `Maybe`, the R above should also be **Rep** because `Maybe` is declared with a representational argument.

The first two premises of the rule specify that the corresponding components of the application must be equal. Importantly, the role used for equality between the arguments of the application (a_2 and b_2) is the minimum of the current role R' and the declared role for the argument R. For example, for `Set` the declared role is **Nom**, so the arguments must be nominally equal, as $\mathbf{Nom} \wedge \mathbf{Rep}$ is **Nom**, but for `Maybe` they may be representationally equal.

The use of the minimum role in this rule forces the nominal equality judgment (i.e. when R' is **Nom**), to compare all subterms using nominal equality while allowing representational equality when both the context *and* the argument are representational.

The last premise of the rule is a subtle aspect of the combination of roles and dependent types—it ensures that definitional equality is homogeneous. In the conclusion of the rule, we want to ensure that both terms have the same type, even when the type may be dependent. In this case, we know that $a_2 \equiv b_2$ at role $R \wedge R'$, but this does not necessarily imply that the types $B\{a_2/x\}$ and $B\{b_2/x\}$ are representationally equal. For example, given a new type T with dependent type $\Pi x{:}\star.F\ x^{\mathbf{Nom}} \to \star$, then we can show

$$\vdash T\ \mathbf{String}^{\mathbf{Rep}} : (F\ \mathbf{String}^{\mathbf{Nom}}) \to \star$$

and

$$\vdash T\ \mathbf{HTML}^{\mathbf{Rep}} : (F\ \mathbf{HTML}^{\mathbf{Nom}}) \to \star$$

In this case, the terms are equal at role **Rep** but the types are not.[8] Therefore, the rule ensures that both sides have the same type by fiat.

In comparison, the (non-roled) application congruence rule AE-AppCong always uses nominal equality for arguments, following Breitner et al. [2016]. Lacking any other source of role information about the parameters (such as roles annotating the function types), this rule defaults to requiring that they be equal using the finest equality (i.e. **Nom**).

Whether definitional equality uses rule AE-TAppCong or rule AE-AppCong depends on the *application flag* annotating the syntax of the term. The typing rules (rule AE-App and rule AE-TApp) ensure that the application flag is appropriate. Some arguments have a choice of application flag: they can either use the one specified by the roles in the signature, or they can use + (which defaults to **Nom** for congruence). Mostly however, application flags are a technical device for our proofs as they duplicate information that is already available in the abstract syntax tree.

3.2 Type Families and Newtypes via Axioms

This calculus uses *axioms* to model type families and newtypes in GHC. An axiom declaration appears in the top-level signature and has the following form.

$$F : A\ @\ \overline{R}\ \text{where}\ p \sim_R a$$

The axiom introduces a contructor F of type A with parameter roles $\overline{R}$. It also declares that the pattern p (which must be headed by F) can be equated at role R to the right-hand side term a.

Patterns come from the following subgrammar of terms and are composed of a sequence of applications.

$$p ::= F \mid p\ x^R$$

Each variable in the pattern is bound in the right hand side of the axiom. The variables in the pattern are annotated with their roles, which are repeated in the list $\overline{R}$ for convenience.

For example, compare the axiom for the type family definition F on the left side below with the one for the newtype declaration T on the right. In each case, the pattern is headed by the corresponding constant and binds the variable x at role **Nom**.

```
type family F a where
    F a = Maybe a
```

$$F : \star \to \star\ @\ \mathbf{Nom}\ \text{where}\quad F\ x^{\mathbf{Nom}} \sim_{\mathbf{Nom}} (\mathbf{Maybe}\ x^{\mathbf{Rep}})$$

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

$$T : \star \to \star\ @\ \mathbf{Nom}\ \text{where}\quad T\ x^{\mathbf{Nom}} \sim_{\mathbf{Rep}} (F\ x^{\mathbf{Nom}})$$

The important distinction is the role marking the $\sim$ in the axiom declarations: it is **Nom** for type families and **Rep** for newtypes. This role determines whether a definition should be treated opaquely

[8]In a system that annotates parameter roles on function types, we could check that the parameter is used consistently with the role annotation on its type. This would allow us to drop this premise from the application rule.

$$\boxed{\vDash a \to^{\beta}_{R} b}$$

$$\textsc{ABeta-AppAbs}\quad \frac{}{\vDash (\lambda x.a)\ b \to^{\beta}_{R} a\{b/x\}} \qquad \textsc{ABeta-Axiom}\quad \frac{F : A \mathbin{@} \overline{R} \text{ where } p \sim_{R_1} b \in \Sigma_0 \qquad \text{MatchSubst } (a, p, b) = b' \qquad R_1 \leq R}{\vDash a \to^{\beta}_{R} b'}$$

$$\boxed{\text{MatchSubst } (a, p, b_1) = b_2}$$

$$\textsc{MatchSubst-Const}\quad \frac{}{\text{MatchSubst } (F, F, b) = b} \qquad \textsc{MatchSubst-AppRelR}\quad \frac{\text{MatchSubst } (a_1, p_1, b_1) = b_2}{\text{MatchSubst } ((a_1\ a^R), (p_1\ x^R), b_1) = (b_2\{a/x\})}$$

$$\boxed{\vdash \Sigma}$$

$$\textsc{Sig-Empty}\quad \frac{}{\vDash \varnothing} \qquad \textsc{Sig-ConsConst}\quad \frac{\vDash \Sigma \qquad \varnothing \vDash A : \star \qquad F \notin \operatorname{dom}\Sigma}{\vDash \Sigma \cup \{F : A \mathbin{@} \overline{R}\}}$$

$$\textsc{SSig-ConsAx}\quad \frac{\vdash \Sigma \qquad F \notin \operatorname{dom}\Sigma \qquad \text{PatCtx } (p, F{:}A) = \Gamma; B; \Omega \qquad \varnothing \vDash A : \star \qquad \Gamma \vDash a : B \qquad \Omega \vDash a : R}{\vdash \Sigma \cup \{F : A \mathbin{@} \operatorname{rng}\Omega \text{ where } p \sim_R a\}}$$

$$\boxed{\text{PatCtx } (p, F{:}A) = \Gamma; B; \Omega}$$

$$\textsc{SPatCtx-Const}\quad \frac{}{\text{PatCtx } (F, F{:}A) = \varnothing; A; \varnothing} \qquad \textsc{SPatCtx-PiRel}\quad \frac{\text{PatCtx } (p, F{:}A) = \Gamma; (\Pi x{:}A'.B); \Omega}{\text{PatCtx } (p\ x^R, F{:}A) = (\Gamma, x{:}A'); B; (\Omega, x{:}R)}$$

$$\boxed{\Omega \vDash a : R}$$

$$\textsc{Srole-a-Star}\quad \frac{\mathit{uniq}(\Omega)}{\Omega \vDash \star : R} \qquad \textsc{Srole-a-Var}\quad \frac{\mathit{uniq}(\Omega) \qquad x : R \in \Omega \qquad R \leq R_1}{\Omega \vDash x : R_1} \qquad \textsc{Srole-a-Const}\quad \frac{\begin{array}{c}\mathit{uniq}(\Omega)\\ F : A \mathbin{@} \overline{R} \in \Sigma_0\end{array}}{\Omega \vDash F : R_1}$$

$$\textsc{Srole-a-Fam}\quad \frac{\begin{array}{c}\mathit{uniq}(\Omega)\\ F : A \mathbin{@} \overline{R} \text{ where } p \sim_R a \in \Sigma_0\end{array}}{\Omega \vDash F : R_1} \qquad \textsc{Srole-a-Abs}\quad \frac{\Omega, x{:}\mathbf{Nom} \vDash a : R}{\Omega \vDash (\lambda x.a) : R} \qquad \textsc{Srole-a-App}\quad \frac{\Omega \vDash a : R \qquad \Omega \vDash b : \mathbf{Nom}}{\Omega \vDash (a\ b^{+}) : R}$$

$$\textsc{Srole-a-TApp}\quad \frac{\Omega \vDash a : R \qquad \Omega \vDash b : R_1 \wedge R}{\Omega \vDash (a\ b^{R_1}) : R} \qquad \textsc{Srole-a-Pi}\quad \frac{\Omega \vDash A : R \qquad \Omega, x{:}\mathbf{Nom} \vDash B : R}{\Omega \vDash (\Pi x{:}A.B) : R}$$

Fig. 3. Axioms and Role-checking

or transparently by definitional equality. For example, at the nominal role, we have $F\ \mathbf{Int}^{\mathbf{Nom}}$ equal to $\mathbf{Maybe}\ \mathbf{Int}^{\mathbf{Rep}}$ and $T\ \mathbf{Int}^{\mathbf{Nom}}$ distinct from $F\ \mathbf{Int}^{\mathbf{Nom}}$. The representational role equates all of these types.

These equalities are derived via the operational semantics. The types $F\ \mathbf{Int}^{\mathbf{Nom}}$ and $\mathbf{Maybe}\ \mathbf{Int}^{\mathbf{Rep}}$ are equal because the former reduces to the latter. In other words, the operational semantics matches a term of the form $F\ \mathbf{Int}^{\mathbf{Nom}}$ against a pattern in an axiom declaration $F\ x^{\mathbf{Nom}}$, producing a substitution $\{\mathbf{Int}/x\}$ that is applied to the right-hand side of the axiom $(\mathbf{Maybe}\ x^{\mathbf{Rep}})\{\mathbf{Int}/x\}$.

More generally, this behavior is specified using rule ABeta-Axiom This rule uses an auxiliary relation $\mathsf{MatchSubst}\ (a, p, b) = b'$ to determine whether the scrutinee a matches the pattern p, and if so, substitutes for the pattern variables in the right-hand side. This rule is also only applicable as long as the role of the axiom R_1 is less-than or equal to the role R that is used for evaluation. For example, if R is **Nom** and R_1 is **Rep**, then this rule does not apply (and a is an opaque value). Alternatively, such as when R_1 is **Nom** as in a type family, then this rule will replace the application headed by a constant with its definition.

Axiom constructors must be *saturated* in order to reduce to their right hand side. In other words, a constructor F must be applied to at least as many arguments as specified by the pattern in the axiom. Non-saturated constructor applications are treated as values by the semantics, no matter their role.

Axioms extend the notion of *definitions* from Weirich et al. [2017], which were always transparent and consequently had no need of role annotations. As before, signatures are unordered and definitions may be recursive—each right hand side may refer to any name in the entire signature. As a result, axioms may be used to define a fixed point operator or other functions and types that use general recursion.

3.3 Role-Checking

The role checking judgment $\Omega \vDash a : R$ is used by rule SSig-ConsAx, which checks the well-formedness of axioms. This rule uses the auxiliary function $\mathsf{PatCtx}\ (p, F\!:\!A) = \Gamma; B; \Omega$ to determine the context Γ and type B to use to type check the right-hand side of the axiom. This function also determines Ω, the context to use when role-checking the right-hand side. The notation $\mathsf{rng}\Omega$ converts the role-checking context into a list of roles that is used when checking application flags.

Unlike opaque constants (which are inert) role annotations for the parameters to an axiom must be checked by the system. In other words, if a newtype axiom declares that it has a representational parameter, then there are restrictions on how that parameter may be used. We check role annotations using the role-checking judgment $\Omega \vDash a : R$, shown at the bottom of Figure 3.

The role-checking context Ω assigns roles to variables. When we check the well-formedness of axioms, the role-checking context is derived from the annotations in pattern p. Note, the roles declared in the pattern need not be the *most permissive* roles for a. Even if the term would check at role **Rep**, the pattern may specify role **Nom** instead.

The rules of the role checking judgment appear at the bottom of Figure 3. The rule Srole-a-Var specifies that the role of a variable must be greater-than or equal to its role in the context. In rule Srole-a-TApp, the role marking an annotated, relevant argument determines how it will be checked. If the role annotation is not present, then arguments must be checked at role **Nom**, as in rule Srole-a-App. Analogously, when role-checking an abstraction, the bound variable enters the context at role **Nom**, as this is the most conservative choice.

4 FULL SYSTEM DR

The previous section presented the complete details of a minimal calculus to provide a solid basis for understanding about the interaction between roles and dependent types in System DR. In

relevance	ρ	::=	$+ \mid -$
application flags (terms)	ν	::=	$R \mid \rho$
application flags (any)	υ	::=	$\nu \mid \bullet$
equality constraints	ϕ	::=	$a \sim_R b : A$
terms, types	a, b, A, B	::=	$\star \mid x \mid F \mid \lambda^\rho x.b \mid a\ b^\nu \mid \Pi^\rho x{:}A \to B$
		$\mid$	$\Box \mid \Lambda c.b \mid a\ \bullet \mid \forall c{:}\phi.B$
		$\mid$	$\mathsf{case}\ a\ \mathsf{of}\ F\,\overline{\upsilon} \to b_1 \| \_ \to b_2$
contexts	Γ	::=	$\varnothing \mid \Gamma, x{:}A \mid \Gamma, c{:}\phi$
patterns	p	::=	$F \mid p\ x^R \mid p\ x^+ \mid p\ \Box^- \mid p\ \bullet$

Fig. 4. Syntax of full language

Typing	$\Gamma \vDash a : A$
Definitional equality (terms)	$\Gamma; \Delta \vDash a \equiv_R b : A$
Proposition well-formedness	$\Gamma \vDash \phi$ ok
Definitional equality (propositions)	$\Gamma; \Delta \vDash \phi_1 \equiv \phi_2$
Context well-formedness	$\vDash \Gamma$
Signature well-formedness	$\vDash \Sigma$
Primitive (β-)reduction	$\vDash a \to^{\beta}_{R} b$
One-step reduction	$\vDash a \leadsto_R b$

Fig. 5. Major judgment forms for System DR.

this section, we zoom out and complete the story at a higher level, providing an overview of the remaining features of the language. The syntax of the full language appears in Figure 4 and the major judgment forms are summarized in Figure 5. For reference, the full specification of System DR is available in the extended version of this paper [Weirich et al. 2019].

4.1 Coercion Abstraction

An essential feature of internal languages capable of compiling Haskell is coercion abstraction, which is used to generalize over equality propositions [Sulzmann et al. 2007]. Coercion abstraction is the basis for the implementation of *generalized algebraic datatypes* [Peyton Jones et al. 2006; Xi et al. 2003] in GHC. For example, a datatype definition, such as

```
data T :: Type -> Type where MkT :: T Int
```

can be encoded by supplying MkT with a constraint about its parameter.

```
data T :: Type -> Type where MkT :: forall a. (a ~ Int) => T a
```

Pattern matching an argument of type T b brings the equality constraint b ~ Int into scope.

```
f :: T a -> a
f MkT = 42   -- we have an assumption (a ~ Int) in the context
```

In System DR, definitional equality is indexed by a role R, so we also allow equality propositions, written $a \sim_R b : A$, to include roles. When R is **Nom**, this proposition corresponds to Haskell's equality constraint, such as $a \sim$ **Int** above. When the role is **Rep**, it corresponds to the **`Coercible`** (§2.4) constraint.

Coercion abstraction brings equality constraints into the context and coercion application discharges those assumptions when the equality can be satisfied. As in extensional type theory [Martin-Löf 1971], equality propositions can be used implicitly by definitional equality. If an equality assumption between two types is available in the context, then those two types are defined to be equal.

$$\textsc{E-Assn}\quad \frac{\vDash \Gamma \qquad c : (a \sim_R b : A) \in \Gamma \qquad c \in \Delta}{\Gamma; \Delta \vDash a \equiv_R b : A}$$

For technical reasons, discussed in Weirich et al. [2017], the full judgment for definitional equality in System DR is written $\Gamma; \Delta \vDash a \equiv_R b : A$. The additional component is a set Δ that tracks which coercions are actually available, used in the rule above. This set need not concern us here; feel free simply to assume that Δ equals the domain of Γ, written $\widetilde{\Gamma}$.

The extension of the System D rules with roled equality constraints is straightforward, though care must be taken to ensure that the roles are used consistently. Note that when role-checking, all variables that appear in a nominal equality constraint must have role **Nom**. This corresponds to the requirement in GHC that the constrained parameters to GADTs have nominal arguments.

4.2 Irrelevant Arguments

A dependently-typed intermediate language for GHC must include support for *irrelevant* arguments as well as relevant arguments [Miquel 2001] in order to implement the type-erasure aspect of parametric polymorphism. In Haskell, polymorphic functions cannot dispatch on types, so these may be erased prior to runtime. In (Curry-style) System DR, irrelevant arguments are therefore elided from the abstract syntax. We extend the calculus by adding a new application flag "–" to indicate that an argument is irrelevant. Furthermore, we add a flag ρ to function types to indicate whether the argument to the function is relevant or irrelevant.

The typing rules for the introduction of an irrelevant abstraction requires that the bound variable not actually appear in the body of the term. When an irrelevant function is used in an application, the argument must be the trivial term, $\square$. Note that the argument is only elided from the term however—it is still substituted in the result type.[9]

$$\textsc{AE-IAbs}\quad \frac{\Gamma, x{:}A \vDash a : B \qquad x \notin \mathsf{fv}\, a}{\Gamma \vDash \lambda^{-} x.a : (\Pi^{-} x{:}A \to B)} \qquad\qquad \textsc{E-IApp}\quad \frac{\Gamma \vDash b : \Pi^{-} x{:}A \to B \qquad \Gamma \vDash a : A}{\Gamma \vDash b\, \square^{-} : B\{a/x\}}$$

Role annotations may *only* apply to relevant arguments, even though constants and newtypes may have both relevant and irrelevant parameters. Irrelevant arguments have their own congruence rule for applications. Because irrelevant arguments never appear in the syntax of terms, an equality between two irrelevant applications only need compare the function components—the arguments are always trivially equal.

$$\textsc{E-IAppCong}\quad \frac{\Gamma; \Delta \vDash a_1 \equiv_R b_1 : (\Pi^{-} x{:}A \to B) \qquad \Gamma \vDash a : A}{\Gamma; \Delta \vDash a_1\, \square^{-} \equiv_R b_1\, \square^{-} : (B\{a/x\})}$$

[9]In System DC, the annotated version of the language with syntax-directed type checking, the argument does appear in term but is eliminated via an erasure operation, following Barras and Bernardo [2008].

$$\textsc{Beta-PatternTrue}\quad \frac{a \leftrightarrow_{\mathbf{Nom}} F\,\overline{v} \qquad \mathrm{ApplyArgs}(a, b_1) = b_1'}{\vDash (\mathrm{case}\ a\ \mathrm{of}\ F\,\overline{v} \to b_1 \| \_ \to b_2) \to_R^{\beta} b_1'\ \bullet}$$

$$\textsc{Beta-PatternFalse}\quad \frac{\mathrm{Value}_{\mathbf{Nom}}\ a \qquad \neg(a \leftrightarrow_{\mathbf{Nom}} F\,\overline{v})}{\vDash (\mathrm{case}\ a\ \mathrm{of}\ F\,\overline{v} \to b_1 \| \_ \to b_2) \to_R^{\beta} b_2}$$

Fig. 6. Case analysis

Overall, there is little interaction between irrelevant arguments and roles. However, there is one important benefit of having both capabilities in the same system. We can use irrelevant quantification to model the *phantom* role from prior work; details are in Section 6.2.

4.3 Case Expressions

The soundness issue described in Section 2.3 arises through the use of the `Discern` type family, which returns different results based on whether its argument is `String` or `HTML`. To ensure that System DR is not susceptible to a similar issue, we include a pattern matching term of the following form.[10]

$$\mathrm{case}\ a\ \mathrm{of}\ F\,\overline{v} \to b_1 \| \_ \to b_2$$

Operationally, the pattern matching term reduces the scrutinee a to a value and then compares it against the pattern specified by $F\,\overline{v}$. If there is a match, the expression steps to b_1. In all other cases, the expression steps to b_2. Pattern matching is not nested—only the head constructor can be observed. In Haskell, type families do both axiom unfolding and discrimination. We separate these features in System DR for orthogonality and eventual unification of pattern matching with Haskell's existing **case** expression. (The semantics of this expression is not exactly the same as that of Haskell's **case**; more details are in Section 7.)

In this syntax, the scrutinee must match the pattern of arguments specified by $F\,\overline{v}$, where $\overline{v}$ is a list of application flags. Note that in the full language, these application flags can include roles, +, -, or indicate a coercion argument.[11] We specify the behavior of case with the rules shown in Figure 6. In the first rule, the $a \leftrightarrow_R F\,\overline{v}$ judgment holds when the scrutinee matches the pattern; i.e. when the scrutinee is an application headed by F with arguments specified by $\overline{v}$. The constructor F must be a constant at role **Nom**; it cannot be a type family axiom. If this judgment holds, the second premise passes those arguments to the branch b_1. In the conclusion of the rule, b_1' is further applied to an elided coercion $\bullet$; this coercion witnesses the equality between the head of a and the pattern, implementing dependent pattern matching.

The second rule, rule Beta-PatternFalse triggers when the scrutinee is a value, yet the comparison $a \leftrightarrow_R F\,\overline{v}$ does not hold. It steps directly to b_2.

Dependent case analysis mean that when the scrutinee matches the head constructor, not only does the expression step to the first branch, but the branch is type checked under the assumption of an equality proposition between the scrutinee and the pattern.

Simple examples of this behavior are possible in source Haskell today, using the `TypeInType` extension.

```
type family F k (a :: k) :: Bool where
   F Bool  x = x      -- here we have (k ~ Bool)
   F _     x = False
```

[10]Unlike source Haskell, patterns in System DR axiom definitions may only include variables and thus may not dispatch on their arguments.
[11]This nameless form of pattern-matching helps with our formalization. The typing rule for case requires the branch b_1 to start with a sequence of abstractions that matches the form specified by the list of flags $\overline{v}$.

Because System DR is dependently-typed, and full-spectrum, the pattern matching term described in this section can also be used for run-time type analysis, as well as dispatch during type checking. We view this as a key benefit of our complete design: we retain the ability to erase (most) types during compilation by abstracting them via irrelevant quantification, but can support run-time dispatch on types when desired.

Case Analysis is Nominal. One part of our design that we found surprising is the fact that case analysis must use the nominal role to evaluate the scrutinee, as we see in the following rule:[12]

$$\textsc{E-Pattern}\quad \frac{\vDash a \rightsquigarrow_{\mathbf{Nom}} a'}{\vDash (\text{case } a \text{ of } F\,\overline{v} \to b_1 \| \_ \to b_2) \rightsquigarrow_R (\text{case } a' \text{ of } F\,\overline{v} \to b_1 \| \_ \to b_2)}$$

Indeed, our original draft of the system also allowed a form of "representational" case analysis, which first evaluated the scrutinee to a representational value before pattern matching. This case analysis could "see through" newtype definitions and would match on the underlying definition.

For example, with representational case analysis, the term

```
case-rep HTML of
   String -> True
   _      -> False
```

would evaluate to `True`.

Unfortunately, we found that representational case analysis is unsound in our system. Consider the following term, which uses a representational analysis to first match the outer structure of its argument, and then uses an inner, nominal analysis for the parameter. System DR always assigns nominal roles to variables bound in a case-match, so this axiom would role-check with a representational argument.

```
F x = case-rep x of
       [y] -> case-nom y of
                HTML -> True
                _    -> False
        _  -> False
```

With this definition, we would be able to show `F [HTML]` representationally equal to `F [String]` because `F`'s parameter is representational. However, these two expressions evaluate to different results. Disaster!

Extending the system to include a safe version of representational case analysis requires a way to rule out the nominal case analysis of `y` above. This means that the type system must record `y`'s role as representational (as it is the argument to the list constructor) and furthermore use the role-checking judgment to ensure that `y` does not appear in a nominal context (such as in the scrutinee of a nominal case analysis). But doing so would violate our design constraint of keeping role checking completely separate from type checking (cf. Section 2.5), thus we have not pursued this extension.

This example also demonstrates that the soundness of *nominal* case analysis depends on the use of the $R \wedge R'$ operation when checking the roles in applications. Because of this operator, all

[12]This rule belongs to a different judgment than the rules in Figure 6. We separate our primitive β-reduction rules from the congruence rules for stepping in our semantics. Only the β-reduction rules are used in our equality relation, relying on the equality relation's congruence rules to correspond to the stepping relation's congruence rules.

subterms of the scrutinee are checked at the nominal role, so we would not be able to show that `F [HTML]` is nominally equal to `F [String]`.

4.4 Constructor Injectivity

System DR is a syntactic type theory. As a result, it supports equality rules for injectivity. If two types are equal, then corresponding subterms of those types should also be equal. In prior work [Weirich et al. 2017], the injectivity of function types was witnessed by rules that allowed an equality between two function types to be decomposed.

This work augments those rules with the correct role components. For example, in rule E-PiSnd, shown below, when we pull out an equality between the co-domain types of a function type, we must provide an equality between the arguments at role **Nom**. This is because we have no knowledge about how the parameter x is used inside the types B_1 and B_2 and so we must be conservative.[13]

E-PiSnd
$$\frac{\Gamma;\Delta \vDash \Pi^{\rho} x\!:\!A_1 \to B_1 \equiv_R \Pi^{\rho} x\!:\!A_2 \to B_2 : \star \qquad \Gamma;\Delta \vDash a_1 \equiv_{\mathbf{Nom}} a_2 : A_1}{\Gamma;\Delta \vDash B_1\{a_1/x\} \equiv_R B_2\{a_2/x\} : \star}$$

This work also extends the reasoning about injectivity to *abstract types*. An equality of the form $F\ {a_1}^{R_1} \ldots a_n^{Rn} = F\ {b_1}^{R_1} \ldots b_n^{Rn}$ can be decomposed to equalities between the corresponding arguments at the roles specified for F. For example, the rule E-Right, shown below shows that we can extract an equality between b and b' when we have an equality between $a\ b^R$ and $a'\ b'^R$.

E-Right
$$\frac{\begin{array}{c}\mathsf{CasePath}_{R_2}\ (a\ b^{R_1}) = F \\ \mathsf{CasePath}_{R_2}\ (a'\ b'^{R_1}) = F \qquad \Gamma \vDash a : \Pi^{+} x\!:\!A \to B \qquad \Gamma \vDash b : A \qquad \Gamma \vDash a' : \Pi^{+} x\!:\!A \to B \\ \Gamma \vDash b' : A \qquad \Gamma;\Delta \vDash a\ b^{R_1} \equiv_{R_2} a'\ b'^{R_1} : B\{b/x\} \qquad \Gamma;\widetilde{\Gamma} \vDash B\{b/x\} \equiv_{\mathbf{Rep}} B\{b'/x\} : \star\end{array}}{\Gamma;\Delta \vDash b \equiv_{R_1 \wedge R_2} b' : A}$$

The first two premises of this rule require that the equation is between two applications, headed by the same constructor F, which cannot be matched to an axiom at role R_2. The next four premises describe the types of the components of the applications. These premises ensure that the equality relation is *homogeneous*, i.e. that only terms of equal types are related.

The key part of this rule is that the equality role of the conclusion is determined by both the original role of the equality R_2 and the annotated role of the application R_1. This is the dual of rule AE-TAppCong, the congruence rule for applications. In that rule, we can use the fact that b is representationally equivalent to b' to show that $a\ b^{\mathbf{Rep}}$ is representationally equal to $a'\ b'^{\mathbf{Rep}}$. Here, we can invert that reasoning.

5 PROPERTIES OF SYSTEM DR

The main result of our Coq development is the proof of type soundness for full System DR. Given the size of the language, the delicate interactions between its features, and number of iterations we have gone through in its development, we could not have done it without mechanical assistance.

This type soundness proof follows from the usual preservation and progress lemmas. Both of these lemmas are useful properties for an intermediate language. The preservation property holds even for open terms. Therefore, it implies that simple, reduction-based optimizations, such as inlining, do not produce ill-typed terms. Our proof of the progress lemma relies on showing that a particular reduction relation is confluent, which itself provides a simplification process for terms and (semi-decidable) algorithm for showing them equivalent.

[13]If the function type were annotated with a role, we would not be limited to **Nom** in this rule.

$\boxed{\mathsf{Value}_R\ a}$ *(Values)*

$$\textsc{Value-Star}\ \frac{}{\mathsf{Value}_R\ \star}\qquad \textsc{Value-Pi}\ \frac{}{\mathsf{Value}_R\ \Pi^{\rho}x\!:\!A \to B}\qquad \textsc{Value-CPi}\ \frac{}{\mathsf{Value}_R\ \forall c\!:\!\phi.B}\qquad \textsc{Value-UAbsRel}\ \frac{}{\mathsf{Value}_R\ \lambda^{+}x.a}\qquad \textsc{Value-UAbsIrrel}\ \frac{\mathsf{Value}_R\ a}{\mathsf{Value}_R\ \lambda^{-}x.a}$$

$$\textsc{Value-UCAbs}\ \frac{}{\mathsf{Value}_R\ \Lambda c.a}\qquad \textsc{Value-Path}\ \frac{\mathsf{CasePath}_R\ a = F}{\mathsf{Value}_R\ a}$$

Fig. 7. Values

From the original design of FC [Sulzmann et al. 2007], we inherit a separation between the proofs of the preservation and progress lemmas. In this system it is possible to prove preservation without relying on the consistency of the system. This means that preservation holds in any context, including ones with contradictory assumptions (such as `Int ~ Bool`). As a result, GHC can apply, e.g., inlining regardless of context.

In this section, we provide an overview of the main results of our Coq development. However, we omit many details. Excluding automatically generated proofs, our scripts include over 700 lemmas and 250 auxiliary definitions.

5.1 Values and Reduction

We define the values of this language using the role-indexed relation $\mathsf{Value}_R\ a$, shown in Figure 7. Whether a term is a value depends on the role: a newtype `HTML` is a value at role **Nom** but reduces at role **Rep**. This relation depends on the auxiliary judgment $\mathsf{CasePath}_R\ a = F$ (not shown, available in the extended version of this paper [Weirich et al. 2019]) which holds when a is a path headed by the constant F that cannot reduce at role R. (This may be because F is a constant, or if the role on F's definition is greater than than R, or if F has not been applied to enough arguments.)

Note that the value relation is contravariant with respect to roles. If a term is a value at some role, it is a value at all smaller roles.

Lemma 5.1 (SubRole-Value[14]). *If $R_2 \leq R_1$ and $\mathsf{Value}_{R_1}\ a$ then $\mathsf{Value}_{R_2}\ a$.*

Alternatively, if a term steps at some evaluation role, and we make some of the definitions more transparent, then it will continue to step, but it could step to a different term. This discrepancy is due to the fact that β-reduction only applies when functions are values. Irrelevant functions are values only when their bodies are values—so changing to a larger role could allow an abstraction to step further.

Lemma 5.2 (SubRole-Step[15]). *If $R_1 \leq R_2$ and $\vDash a \leadsto_{R_1} a'$ then $\exists a'', \vDash a \leadsto_{R_2} a''$.*

That said, the operational semantics is deterministic at a fixed role.

Lemma 5.3 (Deterministic[16]). *If $\vDash a \leadsto_R a'$ and $\vDash a \leadsto_R a''$ then $a' = a''$.*

[14] ext_red_one.v:nsub_Value
[15] ext_red_one.v:sub_red_one
[16] ext_red_one.v:reduction_in_one_deterministic

5.2 Role-Checking

The role-checking judgment $\Omega \vDash a : R$ satisfies a number of important properties. For example, we can always role-check at a larger role.

Lemma 5.4 (SubRole-ing[17]). *If* $\Omega \vDash a : R_1$ *and* $R_1 \leq R_2$ *then* $\Omega \vDash a : R_2$.

Furthermore, the following property says that users may always downgrade the roles of the parameters to their abstract types.

Lemma 5.5 (Role assignment narrowing[18]). *If* $\Omega_1, x : R_1, \Omega_2 \vDash a : R$ *and* $R_2 \leq R_1$ *then* $\Omega_1, x{:}R_2, \Omega_2 \vDash a : R$.

Finally, well-typed terms are always well-roled at **Nom**, when all free variables have role **Nom**.

Lemma 5.6 (Typing/Roleing[19]). *If* $\Gamma \vDash a : A$ *then* $\Gamma_{\mathsf{Nom}} \vDash a : \mathbf{Nom}$, *where* Γ_{Nom} *is the role-checking context that assigns* **Nom** *to every term variable in the domain of* Γ.

5.3 Structural Properties

Lemma 5.7 (Typing Regularity[20]). *If* $\Gamma \vDash a : A$ *then* $\Gamma \vDash A : \star$.

Definition 5.8 (Context equality). Define $\vDash \Gamma_1 \equiv \Gamma_2$ with the following inductive relation:

$$\textsc{CE-Empty}\quad \frac{}{\vDash \varnothing \equiv \varnothing}$$

$$\textsc{CE-ConsTm}\quad \frac{\Gamma_1; \Delta \vDash A_1 \equiv_{\mathbf{Rep}} A_2 : \star \qquad \Gamma_2; \Delta \vDash A_1 \equiv_{\mathbf{Rep}} A_2 : \star \qquad \vDash \Gamma_1 \equiv \Gamma_2}{\vDash \Gamma_1, x{:}A_1 \equiv \Gamma_2, x{:}A_2}$$

$$\textsc{CE-ConsCo}\quad \frac{\Gamma_1; \Delta \vDash \phi_1 \equiv \phi_2 \qquad \Gamma_2; \Delta \vDash \phi_1 \equiv \phi_2 \qquad \vDash \Gamma_1 \equiv \Gamma_2}{\vDash \Gamma_1, c{:}\phi_1 \equiv \Gamma_2, c{:}\phi_2}$$

Lemma 5.9 (Context Conversion[21]). *If* $\Gamma_1 \vDash a : A$ *and* $\vDash \Gamma_1 \equiv \Gamma_2$ *then* $\Gamma_2 \vDash a : A$.

Lemma 5.10 (DefEq Regularity[22]). *If* $\Gamma; \Delta \vDash a \equiv_R b : A$ *then* $\Gamma \vDash a : A$ *and* $\Gamma \vDash b : A$.

5.4 Preservation

We prove the preservation lemma simultaneously with the property that one-step reduction is contained within definitional equality. (This property is not trivial because definitional equality only includes the primitive reductions directly, and relies on congruence rules for the rest.) The reason that we need to show these results simultaneously is due to our typing rule for dependent pattern matching.

Lemma 5.11 (Preservation[23]). *If* $\vDash a \rightsquigarrow_R a'$ *and* $\Gamma \vDash a : A$ *then* $\Gamma \vDash a' : A$ *and* $\Gamma; \Delta \vDash a \equiv_R a' : A$.

5.5 Progress

We prove progress by extending the proof in prior work [Weirich et al. 2017] with new rules for axiom reduction and case analysis. This proof, based on a technique of Tait and Martin-Löf, proceeds first by developing a confluent, role-indexed, parallel reduction relation $\Rightarrow_R$ for terms and then showing that equal terms must be joinable under parallel reduction [Barendregt 1984]. Furthermore, this relation also tracks the roles of free variables using a role-checking context Ω.

[17] ett_roleing.v:roleing_sub
[18] ett_roleing.v:roleing_ctx_weakening
[19] ett_roleing.v:Typing_roleing
[20] ext_invert.v:Typing_regularity
[21] ext_invert.v:context_DefEq_typing
[22] ext_invert.v:DefEq_regularity
[23] ext_red.v:reduction_preservation

We need this role checking context because of the following substitution lemma, necessary to show the confluence lemma below.

Lemma 5.12 (Parallel reduction substitution[24]). *If* $\Omega' \vDash a \Rightarrow_{R_1} a'$ *and* $\Omega', x : R_1 \vDash b : R_2$ *then* $\Omega' \vDash b\{a/x\} \Rightarrow_{R_2} b\{a'/x\}$

We know that some term a reduces and we want to show that we can reconstruct that reduction after that term has been substituted into some other term b. However, the variable x could appear anywhere in b, perhaps as the argument to a function. As a result, the role that we use to reduce a may not be the same role as the one that we use for b.

The parallel reduction relation must be consistent with the role-checking relation. Although our definition of parallel reduction is not typed (it is independent of the type system) it maintains a strong connection to the role-checking judgment.

Lemma 5.13 (Parallel Reduction Role-Checking[25]). *If* $\Omega \vDash a \Rightarrow_R a'$ *then* $\Omega \vDash a : R$ *and* $\Omega \vDash a' : R$.

This property explains why rule SSig-ConsAx, which checks axiom declarations, uses the role on the declaration to role check the right-hand side of the axiom. In other words, type families must role check at **Nom**, whereas newtypes must role check at **Rep**. We could imagine trying to weaken this requirement and role-check all axioms at the most permissive role **Rep**. However, then the above property would not hold. We need to know that when an axiom unfolds, the term remains well-formed at that role.

Lemma 5.14 (Confluence[26]). *If* $\Omega \vDash a \Rightarrow_R a_1$ *and* $\Omega \vDash a \Rightarrow_R a_2$ *then there exists some* b *such that* $\Omega \vDash a_1 \Rightarrow_R b$ *and* $\Omega \vDash a_2 \Rightarrow_R b$.

The confluence proof allows us to show the usual canonical forms lemmas, which are the key to showing the progress lemma.

Lemma 5.15 (Progress[27]). *If* $\varnothing \vDash a : A$ *then either* $\mathsf{Value}_R\ a$ *or there exists some* a' *such that* $\vDash a \leadsto_R a'$.

6 PRACTICALITIES

6.1 Constraint vs. Type

Haskell differentiates the kind `Constraint` from the kind `Type`; the former classifies constraints that appear to the left of an `=>` in Haskell (thus, we have `Eq a :: Constraint`) while the latter classifies ordinary types, like `Int`. This separation between `Constraint` and `Type` is necessary for at least two reasons: we want to disallow users from confusing these two, rejecting types such as `Int => Int` and `Bool -> Eq Char`; and types in kind `Constraint` have special rules in Haskell (allowing definition only via classes and instances) to keep them coherent.

However, we do not want to distinguish `Constraint` from `Type` in the core language. Inhabitants of types of both kinds can be passed to and from functions freely, and we also want to allow (homogeneous) equalities between elements of `Constraint` and elements of `Type`. These equalities come up when the user defines a class with exactly one member, such as

```
class HasDefault a where
  deflt :: a
```

[24] ett_par.v:subst1
[25] ett_par.v:Par_roleing_tm_fst,ett_par.v:Par_roleing_tm_snd
[26] ext_consist.v:confluence
[27] ext_consist.v:progress

Given that the evidence for a `HasDefault a` instance consists only of the `deflt :: a` member, GHC compiles this class declaration into a newtype definition, producing an axiom equating `HasDefault a` with `a`, at the representational role. Some packages[28] rely on this encoding, and it would be disruptive to the Haskell ecosystem to alter this arrangement.

We are thus left with a conundrum: how can we keep `Constraint` distinct from `Type` in Haskell but identify them in the internal language? This situation clearly has parallels with the need for newtypes: a newtype is distinct from its representation in Haskell but is convertible with its representation in the internal language. We find that we can connect `Constraint` with `Type` by following the same approach, but in kinds instead of in types.

This would mean defining `Constraint` along with an axiom stating that `Constraint` is representationally equal to `Type`. That solves the problem: the Haskell type-checker already knows to keep representationally equal types distinct, and all of the internal language functionality over `Type`s would now work over `Constraint`s, too. Because the internal language—System DR—allows conversion using representational equality, an axiom relating, say, `HasDefault a :: Constraint` to `a :: Type` would be homogeneous, as required. The implementors of GHC are eager for System DR in part because it solves this thorny problem.[29]

6.2 The Phantom Role

Prior work [Breitner et al. 2016] includes a third role, the phantom role. Consider the following newtype definition, which does not make use of its argument.

```
newtype F a = MkF Int
```

All values of type `F a` are representationally equal, for any `a`. By giving this newtype the phantom role for its parameter, Haskell programmers can show that `F Int` is representationally equal to `F Bool` even when the `MkF` constructor is not available.

It is attractive to think about the phantom role when thinking of roles as indexing a set of equivalence relations, but that doesn't work out for System DR. In that interpretation, the phantom role is the coarsest relation that identifies all terms of the same type, so it should be placed at the top of the role lattice (above **Rep**). However, with this addition, we do not get the desired semantics for the phantom role.

First, we would need a special definition for evaluating at the phantom role. The difference between nominal and representational evaluation is determined solely by whether axioms are transparent or opaque. However, we cannot use this mechanism to define what it means to evaluate at the phantom role. We would need something else entirely.

Second, arguments with phantom roles require special treatment in the congruence rule. Logically, phantom would be above representational in the role hierarchy, as the corresponding equivalence is coarser. However, rule AE-TAppCong uses $R_1 \wedge R$ to equate arguments. But the minimum of **Rep** and phantom is **Rep**, not phantom, so we would need a different congruence rule for this case.

However, the most compelling reason why we do not include phantom as a role is because it is already *derivable* using irrelevant arguments. In the example above, we can implement the desired behavior via two levels of newtype definition. First, we define a constant, say F', with an irrelevant argument; this is the representation of the newtype above.

$$F' : \star \,@\ \text{ where } F'\ \square^{-} \sim_{\mathbf{Rep}} \mathbf{Int}$$

[28]Notably, the reflection package by Edward Kmett.

[29]See https://ghc.haskell.org/trac/ghc/ticket/11715#comment:64 and https://github.com/ghc-proposals/ghc-proposals/pull/32#issuecomment-315082072.

(Note that F' lacks a role annotation. Only relevant arguments are annotated with roles.) We make this newtype abstract by not exporting this axiom.

Then, we define the phantom type by wrapping the irrelevant argument with a relevant one, which is ignored.

$$F : \star \mathbin{@} \mathbf{Rep} \text{ where } F\ x^{\mathbf{Rep}} \sim_{\mathbf{Rep}} F'\ \Box^{-}$$

When we use F in a nominal role, we will not be able to show that $F\ \mathbf{Int}^{\mathbf{Rep}}$ is equal to $F\ \mathbf{Bool}^{\mathbf{Rep}}$, as is the case in Haskell. However, at the representational role, we can unfold the definition of F in both sides to $F'\ \Box^{-}$, equating the two types. Furthermore, the actual definition of the type can stay hidden, just as in the example above.

6.3 An explicit coerce Term

One simplifying idea we use in System DR, taken from Breitner et al. [2016], is the separation between the role-checking and type-checking judgments. This design, overall, leads to a simpler system because it limits the interactions between the type system and roles. Furthermore, it is also compatible with the current implementation of role checking in GHC.

However, one might hope for a more expressive system by combining the role-checking and type-checking judgments together, as was done in the system of Weirich et al. [2011]. In fact, this was our first approach to this work, primarily because we wanted to explore a design that factored conversion into implicit and explicit parts.

$$\text{AR-Conv}\quad \frac{\Gamma \vDash_R a : A \qquad \Gamma \vDash A \equiv_R B : \star}{\Gamma \vDash_R a : B} \qquad\qquad \text{AR-Cast}\quad \frac{\Gamma \vDash_R a : A \qquad \Gamma \vDash A \equiv_{\mathbf{Rep}} B : \star}{\Gamma \vDash_R \mathbf{coerce}\ a : B}$$

In the conversion rule on the left, the role on the *typing* judgment (indexing the typing judgment by a role is new here) determines the equality that can be used. If this role is **Nom**, then only nominal equality is permitted and coercing between representationally equal types requires an explicit use of coercion, via the rule on the right. Alternatively, if the role is **Rep** then all type conversions are allowed (and using the **coerce** primitive is unnecessary).

This system is attractive because it resembles the design of source Haskell. In contrast, in the current System DR, if an expression has type `HTML`, then it also has type `String`—precisely the situation newtypes were meant to avoid. We return to the question of what **`coerce`** means for source Haskell in the next subsection.

However, after struggling with various designs of the system for some time, we ultimately abandoned this approach. In particular, we were unhappy with a number of aspects of the design.

- How should **coerce** a reduce at role **Nom**? It cannot reduce to a: that would violate type preservation. The solution to this problem is "push" rules, as in System FC [Sulzmann et al. 2007]. These complicate the semantics by moving uses of **coerce** when they block normal reduction. For example, if we have `(coerce (\x -> x)) 5`, we cannot use our normal rule for β-reduction, as the **coerce** intervenes between the λ-expression and its argument. Instead, a push rule is required to reduce the term to `(\x -> coerce x) (coerce 5)`, allowing β-reduction to proceed. However, these push rules are complex and the complexity increases with each feature added to the language; see Weirich et al. [2013, Section 5] for a telling example of how bad they can be.
- Push rules prevent **coerce** from creating stuck terms, but they are not the only evaluation rules for **coerce** that we could want. In particular, we would like the operational semantics to eliminate *degenerate* coercions, which step **coerce** a to a in the case when the coercion does

not actually change the type of a. However, this sort of reduction rule would be type-directed: it would apply only when the two types involved are definitionally equal. Such an operational behavior is at odds with our Curry-style approach and would complicate our treatment of irrelevance.

- Because we are in a dependent setting, we must also consider the impact of **coerce** on the equality relation. For example, what is the relationship between **coerce** a and a? Are they nominally equal? Are they representationally equal? In our explorations of the possibilities, none worked out well. (See also Eisenberg [2015, Section 5].)

We also hoped that working with the combined role-/type-checking system would lead to greater expressiveness in other parts of the language. In particular, the current treatment of roles in GHC was believed to be incompatible with putting the function `join :: Monad m => m (m a) -> m a` in the `Monad` type-class.[30] However, the combined role-/type-checking does not help with this problem. Fortunately, the new `QuantifiedConstraints` extension [Bottu et al. 2017], available in GHC 8.6, provides a new solution,[31] resolving the problem in a much less invasive way.

6.4 What is Source Haskell?

As described above, System DR fails to give a direct semantics for the **coerce** primitive in Haskell. (This is not an issue specific to System DR; no prior work does this [Breitner et al. 2016; Weirich et al. 2011].) However, all is not lost. We propose instead that it is better to understand the **coerce** term in the Haskell source language through an *elaboration semantics*.

More concretely, we can imagine a specification for source Haskell where source terms can automatically convert types with nominal equalities and **coerce** is needed for representational equalities.

$$\textsc{S-Conv}\ \frac{\Gamma \vDash_{\mathsf{src}} a : A \qquad \Gamma \vDash A \equiv_{\mathbf{Nom}} B : \star}{\Gamma \vDash_{\mathsf{src}} a : B} \qquad\qquad \textsc{S-Coerce}\ \frac{\Gamma \vDash_{\mathsf{src}} a : A \qquad \Gamma \vDash A \equiv_{\mathbf{Rep}} B : \star}{\Gamma \vDash_{\mathsf{src}} \mathbf{coerce}\ a : B}$$

However, this language would not have its own small-step semantics (which would require "push rules" as described above). Instead, we would understand its semantics directly through translation to System DR. In other words, we would specify a relation $\Gamma \vDash_{\mathsf{src}} a \rightsquigarrow a' : A$ that translates source Haskell terms a to System DR terms a'. The key step of this translation is that it compiles away all uses of the **coerce** term.

$$\textsc{ST-Coerce}\ \frac{\Gamma \vDash_{\mathsf{src}} a \rightsquigarrow a' : A \qquad \Gamma \vDash A \equiv_{\mathbf{Rep}} B : \star}{\Gamma \vDash_{\mathsf{src}} \mathbf{coerce}\ a \rightsquigarrow a' : B}$$

Because we know that System DR is type-sound, we would automatically get a type soundness property for well-typed source terms.

The drawback of this approach is that without an operational semantics, the source language type system would fall back to System DR for conversion, as in the rules above. Any terms that appear in types would be checked according to System DR instead of with the source language rules. However, this discrepancy would affect only the most advanced Haskell programmers: those that use `coerce` and dependent types together. In the absence of the use of either feature, the systems would coincide.

[30] See https://ghc.haskell.org/trac/ghc/ticket/9123, which was originally titled "Need for higher kinded roles".
[31] https://ryanglscott.github.io/2018/03/04/how-quantifiedconstraints-can-let-us-put-join-back-in-monad/

7 FUTURE WORK

Our work in this paper lays out a consistent point in the design space of interactions between roles and dependent types. However, we plan to continue refining our definitions and extending our system to enhance its expressiveness. Our efforts will be along the following lines:

Annotated Language. We adopted a Curry-style language design in this paper, where the syntax of terms includes only computationally relevant terms. This style of language makes a lot of sense for reasoning about representational equivalence [Diehl et al. 2018]. It also simplifies our design process as we need not worry about propagating annotations at the same time as developing the semantics.

However, this language, taken at face value, cannot be implemented: type-checking is undecidable and non-deterministic. In order to extend GHC's core language along the lines of System DR, we will need to design a system of annotations that will make type checking simple and syntax-directed, much like System DC [Weirich et al. 2017]. Given prior work (including all the work on System FC [Breitner et al. 2016; Sulzmann et al. 2007]) as exemplars, this task should be straightforward; the most challenging parts are choosing a system of annotations that is easy for GHC to manipulate, and showing that the annotated language satisfies properties that are useful for the implementation, such as substitution and preservation.

Other Roles. Our focus in this work has been on the roles **Rep** and **Nom**. However, the rules are generic enough to support arbitrary roles in between these two extremes. Furthermore, perhaps **Nom** is not the right role at the bottom of the lattice—it still allows type families to unfold, for example. Indeed, GHC internally implements an even finer equality—termed `pickyEqType`—that does not even unfold type synonyms. Today, this equality is used only to control error message printing, but perhaps giving users more control over unfolding would open new doors.

Closed Types. The rules of System DR do not check for the exhaustiveness of case analysis. Instead, every pattern match must have a "wildcard" case to provide an option when the scrutinee does not match (see rule Beta-PatternFalse). This design is appropriate when all types are open and extensible through the use of declarations in the signature. It ensures that if a term type checks it will also type check in any extended signature (a necessary property for separate compilation). However, we would like to investigate an extension of the system with *closed types*, those that do not allow the definition of new constants or axioms in the signature. The types could then be the basis for a form of case analysis that does not require the wildcard case and can be shown to be exhaustive.

Extended Pattern Matching. Although System DR allows the definition of *parameterized* datatypes through the use of constants in the signature, the case analysis term is not expressive enough to allow pattern matching for these datatypes. For example, we cannot use this term with the type **Maybe Int**. We omit this capability only to constrain the scope of this work—our rules for pattern matching are complex enough already and are targeted to scrutinees of type $\star$. However, we would like to extend this system with this ability in future work and do not foresee any difficulties.

8 RELATED WORK

Roles. We have reviewed the connection between this work and its two main antecedents [Breitner et al. 2016; Weirich et al. 2017] throughout. Furthermore, Sections 2.5 and 6.3 compare with the approach taken in Weirich et al. [2011].

Eisenberg [2015] discusses a way to integrate roles with a system having some features of dependent types, though that work does not present a dependently typed system. Instead, the type system under study is an extension of the non-dependent System FC used in other prior work on Haskell (and introduced originally by Sulzmann et al. [2007]), but with merged types and kinds [Weirich et al. 2013]. The main challenge that work faces is with the fact that the system described is *heterogeneous*, in that equalities can relate types of different kinds. By contrast, our work here studies only *homogeneous* equality, removing much of the complication Eisenberg discusses. In addition, our work here studies an implicit calculus, allowing us to simplify the presentation even further over the explicit calculus (with decidable type-checking) of Eisenberg.

A recent client for roles and `coerce` is Winant and Devriese [2018]. This work employs roles in a critical way to build a mechanism for explicit instantiation and manipulation of type-class dictionaries without imperiling class coherence.

Modal Dependent Type Theory. Like this work, Pfenning [2001] presents a dependent type theory that simultaneously supports two different forms of equality in conjunction with irrelevance. However, Pfenning's equality relations differ from ours (his system includes α-equivalence and $\alpha\beta\eta$-equivalence), and the type system is a conservative extension of LF [Harper et al. 1993]. Furthermore, Pfenning uses a modal typing discipline to internalize these different concepts.

More recently, modal type theories have refined our understanding of parametric, erasable and irrelevant arguments in dependent type theories [Abel and Scherer 2012; Abel et al. 2017; Mishra-Linger and Sheard 2008; Nuyts et al. 2017].[32] In particular, Nuyts and Devriese [2018] presents a unified framework that includes these and other modalities. It is possible that roles could also be understood as a new modality in this sense, and, once this is accomplished, enjoy a unified treatment with irrelevant quantification.

A starting point for the first task would be a language that includes roles (as a modality) annotated on Π-types. This design could draw on the type system of Weirich et al. [2011], extended with dependent types but without the explicit **coerce** term described in Section 6.3. This framework would then avoid some of the approximations made by System DR caused by the lack of role information on Π-types (noted in the descriptions of rules AE-AppCong, Srole-a-App, and E-PiSnd) and the separation of type- and role- checking (noted in the description of the case expression).

The uniform system of Nuyts and Devriese [2018] could form the basis of the second task. More specifically, this system includes structures that can resolve the differences in the treatment of co-domain of Π types between irrelevant and roled arguments. Although irrelevant arguments cannot appear in lambda terms, they can freely appear in the result type of the lambda term. In contrast, roled arguments must appear according to their indicated role everywhere. Furthermore, Nuyts and Devriese [2018] includes a subsumption relationship between modalities that could generalize sub-roleing.

However, even with the addition of role-annotated Π-types, there would be differences in the treatment of roles in System DR and in the general treatment of modalities in Nuyts and Devriese [2018] that would need to be resolved. First, the general system would need to be extended by a role-indexed operational semantics; roles control the unfolding of axioms and must be considered in any reduction relation. Second, the application congruence rule (rule AE-TAppCong) uses the $R \wedge R'$ operator when comparing the arguments; instead, a modal system would only use the role of argument R and would be independent of R', the role of the context. System DR uses this operator to make nominal equality line up with Haskell's type equality. It is not clear how the general framework can accommodate this behavior.

[32]Note that the distinctions between parametric, erasable and irrelevant quantification are not present in System DR. Our work does not cover parametricity, so cannot determine whether functions take related inputs to related outputs. Furthermore, because our equality judgment is not type directed, our system does not need to distinguish between irrelevant [Abel and Scherer 2012] and shape-irrelevant [Abel et al. 2017] quantification.

Overall, it does not make sense in GHC to combine the treatment of irrelevant quantification and roles. Irrelevant quantification is the basis for the implementation of parametric polymorphism in System DR. Parametric polymorphism is used everywhere, even by novice Haskellers, so it cannot be restricted—so the system must mark relevance on Π types. On the other hand, roles are used to implement the safe-coercions, an advanced feature of GHC, and are subject to the design considerations described in Section 2.5.

Conversions. Recent work by Diehl et al. [2018] studies the possibility of zero-cost conversions in a Curry-style, dependently typed language. However, that work allows these conversions only when two types differ in their *irrelevant* parameters only. In effect, their work supports nominal equality and phantom equality, but not representational equality. Their work does not consider roles. The main application of their work is code re-use, a challenging and painful problem in the domain of dependent types, but one we do not explore here.

The work of Bernardy and Moulin [2013] is similar, and considers color-directed erasure (where arguments of types can be colored, making them optionally irrelevant). In turn, the use of colors allows the generation of coercions for the corresponding arguments. This work provides a mechanism for distinguishing between different sorts of phantom equalities, thanks to the ability to combine several colors and perform selective erasure of arguments.

Our work is tangentially connected to the recent study of *cubical type theory* [Cohen et al. 2018]. In cubical type theory, equality or isomorphism of types is described as a path from one to the other. These equalities can be lifted through type definitions, similar to how roles permit the lifting of equalities through datatypes. However, there is a key difference: roles are about describing *zero-cost* conversions, whereas cubical type theory describes computationally relevant conversions.

Like this work, 2-level type theory [Altenkirch et al. 2016; Capriotti 2016; Voevodsky 2013] is a variant of Martin-Löf type theory with two different equality types. In this case, the "outer" equality contains a strict equality type former, with unique identity proofs, while the "inner" one is some version of Homotopy Type Theory (HoTT). As here, this work reconciles two conflicting definitions of equality in the same system. However, the specific notions of equality are differ.

Typecase. Our case expression implements a form of `typecase` or intensional analysis of types [Harper and Morrisett 1995]. In this setting, type discrimination is available both in types (as in Haskell today) and at run-time. We do not attempt to summarize the immense literature related to run-time type analysis here. Crary and Weirich [1999] consider a similar problem as we do here: the sound preservation of typecase through translation.

9 CONCLUSION

In this work, we provide a solid foundation for two popular extensions of Haskell, dependent types and safe coercions. We have worked out how these features can combine soundly in the same system and have used a proof assistant to verify that we have not missed any subtle interactions. Translating roles into a new framework (dependent type theory, with irrelevance and coercion abstraction) has given us new insight into their power and limitations. Our focus has been Haskell, but we believe that this work is important for more compilers than GHC.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1319880, Grant No. 1521539, and Grant No. 1704041. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Andreas Abel and Gabriel Scherer. 2012. On Irrelevance and Algorithmic Equality in Predicative Type Theory. *Logical Methods in Computer Science* 8, 1 (2012). https://doi.org/10.2168/LMCS-8(1:29)2012

Andreas Abel, Andrea Vezzosi, and Théo Winterhalter. 2017. Normalization by evaluation for sized dependent types. *PACMPL* 1, ICFP (2017), 33:1–33:30. https://doi.org/10.1145/3110277

Thorsten Altenkirch, Paolo Capriotti, and Nicolai Kraus. 2016. Extending Homotopy Type Theory with Strict Equality. In *25th EACSL Annual Conference on Computer Science Logic, CSL 2016, August 29 - September 1, 2016, Marseille, France (LIPIcs)*, Jean-Marc Talbot and Laurent Regnier (Eds.), Vol. 62. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 21:1–21:17. https://doi.org/10.4230/LIPIcs.CSL.2016.21

Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. 2008. Engineering Formal Metatheory. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 3–15.

Brian Aydemir and Stephanie Weirich. 2010. *LNgen: Tool Support for Locally Nameless Representations.* Technical Report MS-CIS-10-24. Computer and Information Science, University of Pennsylvania.

Henk Barendregt. 1991. Introduction to generalized type systems. *J. Funct. Program.* 1, 2 (1991), 125–154.

H. P. Barendregt. 1984. *The Lambda Calculus: Its Syntax and Semantics.* Elsevier.

Bruno Barras and Bruno Bernardo. 2008. The Implicit Calculus of Constructions as a Programming Language with Dependent Types. In *Foundations of Software Science and Computational Structures*, Roberto Amadio (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 365–379.

Jean-Philippe Bernardy and Guilhem Moulin. 2013. Type-theory in color. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, Greg Morrisett and Tarmo Uustalu (Eds.). ACM, 61–72. https://doi.org/10.1145/2500365.2500577

Baldur Blöndal, Andres Löh, and Ryan Scott. 2018. Deriving via: or, how to turn hand-written instances into an anti-pattern. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2018, St. Louis, MO, USA, September 27-17, 2018*, Nicolas Wu (Ed.). ACM, 55–67. https://doi.org/10.1145/3242744.3242746

Gert-Jan Bottu, Georgios Karachalias, Tom Schrijvers, Bruno C. d. S. Oliveira, and Philip Wadler. 2017. Quantified Class Constraints. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell (Haskell 2017)*. ACM, New York, NY, USA, 148–161. https://doi.org/10.1145/3122955.3122967

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

Joachim Breitner, Richard A. Eisenberg, Simon Peyton Jones, and Stephanie Weirich. 2016. Safe zero-cost coercions for Haskell. *Journal of Functional Programming* 26 (2016). https://doi.org/10.1017/S0956796816000150

Paolo Capriotti. 2016. *Models of type theory with strict equality.* Ph.D. Dissertation. School of Computer Science, University of Nottingham, Nottingham, UK. http://arxiv.org/abs/1702.04912

Luca Cardelli. 1986. *A Polymorphic Lambda Calculus with Type:Type.* Technical Report 10. Digital Equipment Corporation, SRC.

Manuel M. T. Chakravarty, Gabriele Keller, and Simon L. Peyton Jones. 2005. Associated type synonyms. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005*, Olivier Danvy and Benjamin C. Pierce (Eds.). ACM, 241–253. https://doi.org/10.1145/1086365.1086397

Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. 2018. Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom. In *21st International Conference on Types for Proofs and Programs (TYPES 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*, Tarmo Uustalu (Ed.), Vol. 69. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 5:1–5:34. https://doi.org/10.4230/LIPIcs.TYPES.2015.5

Karl Crary and Stephanie Weirich. 1999. Flexible Type Analysis. In *Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP)*. Paris, France, 233–248.

Dominique Devriese and Frank Piessens. 2011. On the bright side of type classes: instance arguments in Agda. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy (Eds.). ACM, 143–155. https://doi.org/10.1145/2034773.2034796

Larry Diehl, Denis Firsov, and Aaron Stump. 2018. Generic zero-cost reuse for dependent types. *PACMPL* 2, ICFP (2018), 104:1–104:30. https://doi.org/10.1145/3236799

Richard A. Eisenberg. 2015. *An Overabundance of Equality: Implementing Kind Equalities into Haskell.* Technical Report MS-CIS-15-10. University of Pennsylvania. http://www.cis.upenn.edu/~eir/papers/2015/equalities/equalities.pdf

Richard A. Eisenberg. 2016. *Dependent Types in Haskell: Theory and Practice.* Ph.D. Dissertation. University of Pennsylvania.

Richard A. Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. 2014. Closed Type Families with Overlapping Equations. In *Principles of Programming Languages (POPL '14)*. ACM.

Adam Gundry. 2013. *Type Inference, Haskell and Dependent Types.* Ph.D. Dissertation. University of Strathclyde.

Robert Harper, Furio Honsell, and Gordon Plotkin. 1993. A Framework for Defining Logics. *J. ACM* 40, 1 (Jan. 1993), 143–184. https://doi.org/10.1145/138027.138060

Robert Harper and J. Gregory Morrisett. 1995. Compiling Polymorphism Using Intensional Type Analysis. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, Ron K. Cytron and Peter Lee (Eds.). ACM Press, 130–141. https://doi.org/10.1145/199448.199475

Per Martin-Löf. 1971. A Theory of Types. (1971). Unpublished manuscript.

Alexandre Miquel. 2001. *The Implicit Calculus of Constructions Extending Pure Type Systems with an Intersection Type Binder and Subtyping.* Springer Berlin Heidelberg, Berlin, Heidelberg, 344–359. https://doi.org/10.1007/3-540-45413-6_27

Nathan Mishra-Linger and Tim Sheard. 2008. Erasure and Polymorphism in Pure Type Systems. In *Foundations of Software Science and Computational Structures, 11th International Conference, FOSSACS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29 - April 6, 2008. Proceedings (Lecture Notes in Computer Science)*, Roberto M. Amadio (Ed.), Vol. 4962. Springer, 350–364. https://doi.org/10.1007/978-3-540-78499-9_25

Andreas Nuyts and Dominique Devriese. 2018. Degrees of Relatedness: A Unified Framework for Parametricity, Irrelevance, Ad Hoc Polymorphism, Intersections, Unions and Algebra in Dependent Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, Anuj Dawar and Erich Grädel (Eds.). ACM, 779–788. https://doi.org/10.1145/3209108.3209119

Andreas Nuyts, Andrea Vezzosi, and Dominique Devriese. 2017. Parametric quantifiers for dependent type theory. *PACMPL* 1, ICFP (2017), 32:1–32:29. https://doi.org/10.1145/3110276

Nicolas Oury and Wouter Swierstra. 2008. The power of Pi. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, James Hook and Peter Thiemann (Eds.). ACM, 39–50. https://doi.org/10.1145/1411204.1411213

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

Frank Pfenning. 2001. Intensionality, Extensionality, and Proof Irrelevance in Modal Type Theory. In *Proceedings of the 16th Annual Symposium on Logic in Computer Science (LICS'01)*, J. Halpern (Ed.). IEEE Computer Society Press, Boston, Massachusetts, 221–230.

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

Matthieu Sozeau and Nicolas Oury. 2008. First-Class Type Classes. In *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings (Lecture Notes in Computer Science)*, Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar (Eds.), Vol. 5170. Springer, 278–293. https://doi.org/10.1007/978-3-540-71067-7_23

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

Vladimir Voevodsky. 2013. A simple type system with two identity types. *Unpublished note* (2013), 8 pages. https://www.math.ias.edu/vladimir/sites/math.ias.edu.vladimir/files/HTS.pdf

Stephanie Weirich. 2014. Depending on Types. Invited keynote given at ICFP 2014.

Stephanie Weirich. 2017. The Influence of Dependent Types. Invited keynote given at POPL 2017.

Stephanie Weirich, Pritam Choudhury, Antoine Voizard, and Richard A. Eisenberg. 2019. A Role for Dependent Types in Haskell (Extended Version). https://arxiv.org/abs/1905.13706

Stephanie Weirich, Justin Hsu, and Richard A. Eisenberg. 2013. System FC with explicit kind equality. In *Proceedings of The 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. Boston, MA, 275–286.

Stephanie Weirich, Antoine Voizard, Pedro Henrique Avezedo de Amorim, and Richard A. Eisenberg. 2017. A Specification for Dependent Types in Haskell. *Proc. ACM Program. Lang.* 1, ICFP, Article 31 (Aug. 2017), 29 pages. https://doi.org/10.1145/3110275

Stephanie Weirich, Dimitrios Vytiniotis, Simon Peyton Jones, and Steve Zdancewic. 2011. Generative Type Abstraction and Type-level Computation. In *POPL 11: 38th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, January 26–28, 2011. Austin, TX, USA.* 227–240.

Thomas Winant and Dominique Devriese. 2018. Coherent explicit dictionary application for Haskell. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2018, St. Louis, MO, USA, September 27-17, 2018*, Nicolas Wu (Ed.). ACM, 81–93. https://doi.org/10.1145/3242744.3242752

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

Ningning Xie and Richard A. Eisenberg. 2018. Coercion Quantification. In *Haskell Implementors' Workshop*.