System FC with Explicit Kind Equality

Stephanie Weirich Justin Hsu Richard A. Eisenberg

Department of Computer and Information Science, University of Pennsylvania
Philadelphia, PA, USA
{sweirich,justhsu,eir}@cis.upenn.edu

Abstract

System FC, the core language of the Glasgow Haskell Compiler, is an explicitly-typed variant of System F with first-class type equality proofs called *coercions*. This extensible proof system forms the foundation for type system extensions such as type families (type-level functions) and Generalized Algebraic Datatypes (GADTs). Such features, in conjunction with kind polymorphism and datatype promotion, support expressive compile-time reasoning.

However, the core language lacks explicit *kind* equality proofs. As a result, type-level computation does not have access to kind-level functions or promoted GADTs, the type-level analogues to expression-level features that have been so useful. In this paper, we eliminate such discrepancies by introducing kind equalities to System FC. Our approach is based on dependent type systems with heterogeneous equality and the "Type-in-Type" axiom, yet it preserves the metatheoretic properties of FC. In particular, type checking is simple, decidable and syntax directed. We prove the preservation and progress theorems for the extended language.

Categories and Subject Descriptors F.3.3 [*Studies of Program Constructs*]: Type structure

General Terms Design, Languages

Keywords Haskell, Dependent types, Equality

1. Introduction

Is Haskell a dependently typed programming language? Many would say no, as Haskell fundamentally does not allow expressions to appear in types (a defining characteristic of dependently-typed languages). However, the type system of the Glasgow Haskell Compiler (GHC), Haskell's primary implementation, supports two essential features of dependently typed languages: *flow-sensitive typing* through Generalized Algebraic Datatypes (GADTs) (Peyton Jones et al. 2006; Schrijvers et al. 2009), and rich *type-level computation* through type classes (Jones 2000), type families (Chakravarty et al. 2005), datatype promotion and kind polymorphism (Yorgey et al. 2012). These two features allow clever Haskellers to encode programs that are typically reputed to need dependent types.

However, these encodings cannot accommodate all dependently-typed programs. GADTs and type families are supported in FC, GHC's typed intermediate language, through the use of first-class type equalities (Sulzmann et al. 2007). However, FC lacks first-class *kind equalities* limiting its expressiveness. As a result, *GADTs cannot be promoted*, because the type equalities in their definition cannot be lifted to kind equalities. Furthermore, *GADTs cannot be indexed by kinds*, which would require reasoning about kind equality. Finally, although type families permit types to be defined computationally, the lack of kind equalities means there are *no kind families* in GHC. Although these features seem esoteric, they are often necessary for encoding dependently-typed programs in GHC (Eisenberg and Weirich 2012). We give concrete examples that require these features in Section 2.

Our goal in this paper is to eliminate such nonuniformities with a single blow, by unifying types and kinds. In essence, we augment FC's type language with dependent kinds—kinds that can depend on types. This process is not without challenges—this dependency has complex interactions with type equality. However, our ultimate goal is to better support dependently typed programming in GHC, and resolving these issues is an critical step.

Specifically, we make the following technical contributions:

- We describe an explicitly-typed intermediate language with explicit equality proofs for both types *and kinds* (Sections 3 and 4). The language is no toy: it is an extension of the System FC intermediate language used by GHC (Sulzmann et al. 2007; Weirich et al. 2011; Yorgey et al. 2012; Vytiniotis et al. 2012).
- We extend the *type preservation* proof of FC to cover the new features (Section 5). The treatment of datatypes requires an important property: *congruence* for the equational theory. In other words, we can derive a proof of equality for any form of type or kind, given equality proofs of subcomponents. The computational content of this theorem, called *lifting*, generalizes the standard substitution operation. This operation is required in the operational semantics for datatypes.
- We prove the *progress* theorem in the presence of kind coercions and dependent coercion abstraction. The progress theorem holds under consistent sets of equality axioms. Our modifications require new conditions on axioms to ensure consistency, and proving consistency requires significant changes to the proof from prior work. We discuss these changes and their consequences in Section 6.

We have implemented our extensions to FC in a development branch[1] of GHC to demonstrate that our modifications are compatible with the existing system, and do not invalidate existing Haskell

[1] Available online, from
`http://www.cis.upenn.edu/~eir/packages/nokinds/`.

ICFP '13, September 25–27, 2013, Boston, MA, USA.

ACM 978-1-4503-2326-0/13/09…$15.00.
http://dx.doi.org/10.1145/2500365.2500599

programs. This implementation involves extensions to the core language syntax, type checker and stepper (used in optimizations).

Although our designs are inspired by the rich theory of dependent type systems, applying these ideas in the context of Haskell means that our language differs in many ways from existing work. We detail these comparisons in Section 7.

The scope of this paper only includes the design and implementation of kind equalities in the FC intermediate language; we have not yet modified GHC's source language, so promoted GADTs, kind-indexed GADTs and kind families are not (yet) available to programmers. Although the required syntactic extensions are minor, extending GHC's constraint solver requires careful integration with existing features. Furthermore, the encodings that work around Haskell's restriction that terms cannot appear in types often impose heavy syntactic overheads—improved source-level support for dependently-typed programming should also address this issue. We describe this important future work in Section 8.

For reasons of space, several technical details and the proofs are deferred to the extended version of the paper, available at `http://www.cis.upenn.edu/~sweirich/nokinds-extended.pdf`.

2. Why kind equalities?

Kind equalities enable new, useful features. In this section we use an extended example to demonstrate how *kind-indexed GADTs*, *promoted GADTs*, and *kind families* might be used in practice. Below, code snippets that require kind equalities in their compilation to FC are highlighted in gray—all other code snippets compile.[2]

The running example below defines "shallowly" and "deeply" indexed representations of types, and shows how they may be used for Generic Programming. The former use Haskell's types as indices (Crary et al. 1998; Yang 1998), whereas the latter use an algebraic datatype (also known as a *universe*) (Altenkirch and McBride 2002; Norell 2002). (Magalhães (2012) gives more details describing how extensions to Haskell, including the ones described in this paper, benefit generic programming.)

Shallow indexing Consider a GADT for type representations:

```
data TyRep :: * → * where
 TyInt  :: TyRep Int
 TyBool :: TyRep Bool
```

GADTs differ from ordinary algebraic datatypes in that they allow each data constructor to constrain the type parameters to the datatype. For example, the `TyInt` constructor requires that the single parameter to `TyRep` be `Int`.

We can use type representations for type-indexed programming—a simple example is computing a default element for each type.

```
zero :: ∀ a. TyRep a → a
zero TyInt  = 0           -- ‘a’ must be Int
zero TyBool = False       -- ‘a’ must be Bool
```

This code pattern matches the type representation to determine what value to return. Because of the nonuniform type index, pattern matching recovers the identity of the type variable `a`. In the first case, because the data constructor is `TyInt`, this parameter must be `Int`, so `0` can be returned. In the second case the parameter `a` must be equal to `Bool`, so returning `False` is well-typed.

However, the GADT above can only be used to represent types of kind $\star$. To represent type constructors with kind $\star \to \star$, such as `Maybe` or `[]`, we could create a separate datatype, perhaps called `TyRep1`. However, this approach is ugly and inflexible—what about tuples? Do we need a `TyRep2`, `TyRep3`, and more?

[2] with GHC 7.6 and the language extensions `PolyKinds, DataKinds, GADTs, ExplicitForAll` and `TypeFamilies`.

We might hope that kind polymorphism (Yorgey et al. 2012), which allows datatypes to be parameterized by kind variables as well as type variables, could be the solution. For example, the following kind polymorphic type takes two phantom arguments, a kind variable κ and a type variable a of kind κ.

```
data Proxy (a :: k) = P
```

However, kind polymorphism is not enough to unify the representations for `TyRep`—the type representation (shown below) should constrain its *kind* parameter.

```
data TyRep :: ∀ k. k → * where
  TyInt   :: TyRep Int
  TyBool  :: TyRep Bool
  TyMaybe :: TyRep Maybe
  TyApp   :: TyRep a → TyRep b → TyRep (a b)
```

This `TyRep` type takes two parameters, a kind `k` and a type of that kind (not named in the kind annotation). The data constructors constrain `k` to a concrete kind. For the example to be well-formed, `TyInt` must constrain the *kind* parameter to $\star$. Similarly, `TyMaybe` requires the kind parameter to be $\star \to \star$. We call this example a *kind-indexed GADT* because the datatype is indexed by both kind and type information.

Pattern matching with this datatype refines kinds as well as types—determining whether a type is of the form `TyApp` makes new kind and type equalities available. For example, consider the `zero` function extended with a default value of the `Maybe` type.

```
zero :: ∀ (a :: *). TyRep a → a
zero TyInt              = 0
zero TyBool             = False
zero (TyApp TyMaybe _) = Nothing
```

In the last case, the `TyApp` pattern introduces the kind variable `k`, the type variables `b :: k → *` and `c :: k`, and the type equality `a ∼ b c`. The `TyMaybe` pattern adds the kind equality `k ∼ *` and type equality `b ∼ Maybe`. Combining the equalities, we can show that `Maybe c`, the type of `Nothing`, is well-kinded and equal to `a`.[3]

Deep indexing Kind equalities enable additional features besides kind-indexed GADTs. The previous example used Haskell types directly to index type representations. With datatype promotion, we can instead define a datatype (a *universe*) for type information.

```
data Ty = TInt | TBool
```

We can use this datatype to index the representation type.

```
data TyRep :: Ty → * where
 TyInt  :: TyRep TInt
 TyBool :: TyRep TBool
```

Note that the kind of the parameter to this datatype is `Ty` instead of `*`—datatype promotion allows the type `Ty` to be used as a kind and allows its constructors, `TyInt` and `TyBool`, to appear in types.

To use these type representations, we describe their connection with Haskell types via a *type family* (a function at the type level).

```
type family I (t :: Ty) :: *
type instance I TInt  = Int
type instance I TBool = Bool
```

`I` is a function that maps the (promoted) data constructor `TInt` to the Haskell type `Int`, and similarly `TBool` to `Bool`.

[3] Note that although this definition of `zero` is exhaustive, it is unlikely that an extended version of GHC will be able to determine that fact automatically.

We can use these type representations to define type-indexed operations, like before.

```
zero :: ∀ (a :: Ty). TyRep a → I a
zero TyInt  = 0
zero TyBool = False
```

Pattern matching `TyInt` refines `a` to `TInt`, which then uses the type family definition to show that the result type is equal to `Int`.

Dependently typed languages do not require an argument like `TyRep` to implement operations such as `zero`—they can match directly on the type of kind `Ty`. This is not allowed in Haskell, which maintains a separation between types and expressions. The `TyRep` argument is an example of a *singleton* type, a standard way of encoding dependently typed operations in Haskell.

Note that this representation is no better than the shallow version in one respect—`I` must produce a type of kind $\star$. What if we wanted to encode `TMaybe` with `Ty`?

To get around this issue, we use a GADT to represent different kinds of types. We first need a universe of kinds.

```
data Kind = Star | Arr Kind Kind
```

`Kind` is a normal datatype that, when promoted, can be used to index the `Ty` datatype, making it a (standard) GADT.

```
data Ty :: Kind → * where
  TInt   :: Ty Star
  TBool  :: Ty Star
  TMaybe :: Ty (Arr Star Star)
  TApp   :: Ty (Arr k1 k2) → Ty k1 → Ty k2
```

This indexing means that `Ty` can only represent well-kinded types. For example `TMaybe` has type `Ty (Arr Star Star)` and `TApp TMaybe TBool` has type `Ty Star`, while the value `TApp TInt` would be rejected. Although this GADT can be expressed in GHC, the corresponding `TyRep` type requires two new extensions: *promoted GADTs* and *kind families*.

With the current design of FC, only a subset of Haskell 98 datatypes can be promoted. In particular, GADTs cannot be used to index other GADTs. The extensions proposed in this work allow the GADT `Ty` above to be used as an index to `TyRep` or to be interpreted by the type family `I`, as shown below.

```
data TyRep (k :: Kind) (t :: Ty k) where
  TyInt   :: TyRep Star TInt
  TyBool  :: TyRep Star TBool
  TyMaybe :: TyRep (Arr Star Star) TMaybe
  TyApp   :: TyRep (Arr k1 k2) a → TyRep k1 b
          → TyRep k2 (TApp a b)
```

We now need to adapt the type family `I` to work with the new promoted GADT `Ty`. To do so, we must classify its return kind, and for that, we need a *kind family*—a function that produces a kind by pattern matching a type or kind argument. For example, we can interpret values of the `Kind` datatype as Haskell kinds like so:

```
kind family IK (k :: Kind)
kind instance IK Star = *
kind instance IK (Arr k1 k2) = IK k1 → IK k2
```

This interpretation of kinds is necessary to define the interpretation of types—without it, this definition does not "kind-check":

```
type family I (t :: Ty k) :: IK k
type instance I TInt       = Int
type instance I TBool      = Bool
type instance I TMaybe     = Maybe
type instance I (TApp a b) = (I a) (I b)
```

However, once `I` has been defined, `Ty` and `TyRep` can be used in type-indexed operations as before.

```
zero :: ∀ (a :: Ty Star). TyRep Star a → I a
zero TyInt              = 0
zero TyBool             = False
zero (TyApp TyMaybe _) = Nothing
```

The examples above demonstrate all three features that kind equalities enable: kind-indexed GADTs, kind families, and promoted GADTs. While these examples are all derived from generic programming, we have also been able to use these features to express dependently typed programs from McBride (2012) and Oury and Swierstra (2008). We omit these examples for lack of space.

We note that the Haskell syntax used in the gray boxes above is hypothetical, as we have not extended the surface language. However, an important first step is to enhance the core language, System FC, so that it is expressive enough to support these features. We now turn to this task.

3. System FC

System FC is the typed intermediate language of GHC. GHC's advanced features, such as GADTs and type families, are compiled into FC as type equalities. This section reviews the current status of System FC, describes that compilation, and puts our work in context. FC has evolved over time, from its initial definition (Sulzmann et al. 2007), to extensions FC_2 (Weirich et al. 2011), and $F_C^{\uparrow}$ (Yorgey et al. 2012). In this paper, we use the name FC for the language and all of its variants. Our technical discussion contrasts our new extensions with the most recent prior version, $F_C^{\uparrow}$.

Along with the usual kinds (κ), types (τ) and expressions (e), FC contains coercions (γ) that are proofs of type equality. The judgement

$$\Gamma \vdash_{\mathsf{co}} \gamma \ : \ \tau_1 \sim \tau_2$$

checks that the coercion γ proves types τ_1 and τ_2 equal. These proofs are used to change the types of expressions. For example, if γ is a proof of $\tau_1 \sim \tau_2$, and the expression e has type τ_1, then the expression $e \triangleright \gamma$ (pronounced "e casted by γ") has type τ_2.

Making type conversion explicit ensures that the FC typing relation $\Gamma \vdash_{\mathsf{tm}} e \ : \ \tau$ is syntax-directed and decidable. This is not the case in the source language; there type checking requires nonlocal reasoning, such as unification and type class resolution. Furthermore, in the presence of certain flags (such as `UndecidableInstances`), it may not terminate.

Straightforward type checking is an important sanity check on the internals of GHC—transformations and optimizations must preserve typability. Therefore, all information necessary for type checking is present in FC expressions. This information includes explicit type abstractions and applications (System FC is an extension of System F_ω (Girard 1972)) as well as explicit proofs of type equality.

For example, type family definitions are compiled to *axioms* about type equality that can be used in FC coercion proofs. A type family declaration and instance in source Haskell

```
type family F a :: *
type instance F Bool = Int
```

generates the following FC axiom declaration:

$$\mathsf{axF} : \mathsf{F\,Bool} \sim \mathsf{Int}$$

When given a source language function of type

```
g :: ∀ a. a → F a → Char
```

the expression `g True 3` translates to the FC expression

$$g\ \mathsf{Bool}\ \mathsf{True}\ (3 \triangleright \mathbf{sym}\ \mathsf{axF})$$

that instantiates g at type Bool and coerces 3 to have type $\mathsf{F}\ \mathsf{Bool}$. The coercion $\mathbf{sym}\ \mathsf{axF}$ is a proof that $\mathsf{Int} \sim \mathsf{F}\ \mathsf{Bool}$.

GADTs are compiled into FC so that pattern matching on their data constructors introduces *type equality assumptions* into the context. For example, consider the following simple GADT.

```
data T :: * → * where
   TInt :: T Int
```

This declaration could have also been written as a normal datatype where the type parameter is constrained to be equal to `Int`.

```
data T a = (a ∼ Int) ⇒ TInt
```

In fact, all GADTs can be rewritten in this form using equality constraints. Pattern matching makes this constraint available to the type checker. For example, the type checker concludes below that 3 has type `a` because the type `Int` is known to be equal to `a`.

```
f :: T a → a
f TInt = 3
```

In the translation to FC, the TInt data constructor takes this equality constraint as an explicit argument.

$$\mathsf{TInt} : \forall\, a{:}\, \star\, .\, (a \sim \mathsf{Int}) \Rightarrow T\ a$$

When pattern matching on values of type $T\ a$, this proof is available for use in a cast.

$$\begin{aligned} f = \Lambda a{:}\,\star\,.\lambda x{:}\ T\ a.\ &\mathbf{case}\ x\ \mathbf{of} \\ &\mathsf{TInt}\ (c{:}\ a \sim \mathsf{Int}) \to (3 \triangleright \mathbf{sym}\ c) \end{aligned}$$

Coercion assumptions and axioms can be composed to form larger proofs. FC includes a number of forms in the coercion language that witness the reflexivity, symmetry and transitivity of type equality. Furthermore, equality is a congruent relation over types. For example, if we have proofs of $\tau_1 \sim \tau_2$ and $\tau_1' \sim \tau_2'$, then we can form a proof of the equality $\tau_1\, \tau_1' \sim \tau_2\, \tau_2'$. Finally, composite coercion proofs can be decomposed. For example, data constructors T are injective, so given a proof of $T\, \tau_1 \sim T\, \tau_2$, a proof of $\tau_1 \sim \tau_2$ can be produced.

Explicit coercion proofs are like explicit type arguments: they are erasable from expressions and do not effect the operational behavior of an expression. (We make this precise in Section 5.3.) To ensure that coercions do not suspend computation, FC includes "push rules". For example, when a coerced value is applied to an argument, the coercion must be "pushed" to the argument and result of the application so that β-reduction can occur.

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma\ :\ \sigma_1 \to \sigma_2 \sim \tau_1 \to \tau_2}{(v \triangleright \gamma)\ e \longrightarrow (v\ (e \triangleright \mathbf{sym}\ (\mathbf{nth}^1\ \gamma))) \triangleright \mathbf{nth}^2\ \gamma}\ \textsc{S\_Push}$$

In this rule, if the expression $(v \triangleright \gamma)\ e$ is well typed, then γ must be a proof of the equality $\sigma_1 \to \sigma_2 \sim \tau_1 \to \tau_2$. The coercions $\mathbf{sym}\ (\mathbf{nth}^1\ \gamma)$ and $\mathbf{nth}^2\ \gamma$ decompose this proof into coercions for the argument $(\tau_1 \sim \sigma_1)$ and result $(\sigma_2 \sim \tau_2)$ of the application.

4. System FC with kind equalities

The main idea of this paper is to augment FC with proofs of equality between kinds and to use these proofs to explicitly coerce the kinds of types. We do so via new type form: if type τ has kind κ_1, and γ is a proof that kind κ_1 equals kind κ_2, then $\tau \triangleright \gamma$ is type τ casted to kind κ_2. There are several challenges to this extension, which we address with the following technical solutions.

H	$::=$		Type constants
	$\mid$	$(\to)$	Arrow
	$\mid$	$\star$	Type/Kind
	$\mid$	T	Type constructor
	$\mid$	K	Promoted data constructor
w	$::=$		Type-level names
	$\mid$	a	Type variables
	$\mid$	F	Type functions
	$\mid$	H	Type constants
$\sigma,\ \tau,\ \kappa$	$::=$		Types and Kinds
	$\mid$	w	Names
	$\mid$	$\forall\, a{:}\, \kappa.\, \tau$	Polymorphic types
	$\mid$	$\forall\, c{:}\, \phi.\, \tau$	Coercion abstr. type
	$\mid$	$\tau_1\, \tau_2$	Type/kind application
	$\mid$	$\tau \triangleright \gamma$	Casting
	$\mid$	$\tau\, \gamma$	Coercion application
ϕ	$::= \sigma \sim \tau$		Propositions (coercion kinds)
$\gamma,\ \eta$	$::=$		Coercions
	$\mid$	c	Variables
	$\mid$	$C\, \overline{\rho}$	Axiom application
	$\mid$	$\langle \tau \rangle$	Reflexivity
	$\mid$	$\mathbf{sym}\, \gamma$	Symmetry
	$\mid$	$\gamma_1 \mathbin{⨾} \gamma_2$	Transitivity
	$\mid$	$\forall_\eta (a_1, a_2, c).\gamma$	Type/kind abstr. cong.
	$\mid$	$\forall_{(\eta_1, \eta_2)} (c_1, c_2).\gamma$	Coercion abstr. cong.
	$\mid$	$\gamma_1\, \gamma_2$	Type/kind app. cong.
	$\mid$	$\gamma(\gamma_2, \gamma_2')$	Coercion app. cong.
	$\mid$	$\gamma \triangleright \gamma'$	Coherence
	$\mid$	$\gamma @ \gamma'$	Type/kind instantiation
	$\mid$	$\gamma @ (\gamma_1, \gamma_2)$	Coercion instantiation
	$\mid$	$\mathbf{nth}^i\, \gamma$	nth argument projection
	$\mid$	$\mathbf{kind}\, \gamma$	Kind equality extraction
ρ	$::= \tau \mid \gamma$		Type or coercion
$e,\ u$	$::=$		Expressions
	$\mid$	x	Variables
	$\mid$	$\lambda x{:}\, \tau.\, e$	Abstraction
	$\mid$	$e_1\, e_2$	Application
	$\mid$	$\Lambda a{:}\, \kappa.\, e$	Type/kind abstraction
	$\mid$	$e\, \tau$	Type/kind application
	$\mid$	$\lambda c{:}\, \phi.\, e$	Coercion abstraction
	$\mid$	$e\, \gamma$	Coercion application
	$\mid$	$e \triangleright \gamma$	Casting
	$\mid$	K	Data constructors
	$\mid$	$\mathbf{case}\ e\ \mathbf{of}\ \overline{p \to u}$	Case analysis
	$\mid$	$\mathbf{contra}\, \gamma\, \tau$	Absurdity
p	$::= K\, \Delta\, \overline{x{:}\tau}$		Patterns
Δ	$::=$		Telescopes
	$\mid$	$\varnothing$	Empty
	$\mid$	$\Delta,\ a{:}\, \kappa$	Type variable binding
	$\mid$	$\Delta,\ c{:}\, \phi$	Coercion variable binding

Figure 1. Basic Grammar

- *Unifying kinds and types.* A language with kind polymorphism, kind equalities, kind coercions, type polymorphism, type equalities and type coercions quickly becomes redundant (and somewhat overwhelming).

  Therefore, we follow pure type systems (Barendregt 1992) and unify the syntax of types and kinds, allowing us to reuse type coercions as kind coercions.[4] Although there is no syntactic distinction between types and kinds, we informally use the word *type* (metavariables τ and σ) for those members that classify runtime expressions, and *kind* (metavariable κ) for those members that classify expressions of the type language.

  As in pure type systems, types and kinds share semantics—there is a common judgement for the validity of both. Furthermore, our rules include the $\star{:}\star$ axiom which means that there is no real distinction between types and kinds. This choice simplifies many aspects of the language design.

  Languages such as Coq and Agda avoid the $\star{:}\star$ axiom because it introduces inconsistency, but that is not an issue here. The FC type language is already inconsistent in the sense that all kinds are inhabited. The type safety property of FC depends on the consistency of its *coercion* language, not its *type* language. See Section 6 and Section 7 for more discussion of this issue.[5]

- *Making type equality "heterogeneous".* As kinds classify types, kind equality has nontrivial interactions with type equality.

  Because kind coercions are explicit, there are equivalent types that do not have syntactically identical kinds. Therefore, like McBride's "John Major" equality (2002), our definition of type equality $\tau_1 \sim \tau_2$ is heterogeneous—the types τ_1 and τ_2 could have kinds κ_1 and κ_2 that have no syntactic relation to each other. A proof γ of $\tau_1 \sim \tau_2$ implies not only that τ_1 and τ_2 are equal, but also that their kinds are equal. The new coercion form $\mathbf{kind}\,\gamma$ extracts the proof of $\kappa_1 \sim \kappa_2$ from γ.

  Another difficulty comes from the need to equate polymorphic types that have coercible but not syntactically equal kinds for the bound variable. We discuss the modification to this coercion form in Section 4.3.1.

- *Coercion irrelevance.* Coercions should be irrelevant to both the operational semantics *and* type equivalence. The fact that a coercion is used to change the type of an expression, or the kind of a type, should not influence the evaluation of the expression or the equalities available for the type. For the former, we maintain irrelevance by updating FC's "push rules" to the new semantics (see Section 5 for details). For the latter, we carefully construct our coercion forms to ignore coercions inside types (Section 4.3.2).

- *Dependent coercion abstraction.* As in prior versions of FC, coercions are first class—they can be passed as arguments to functions and stored in data structures (as the arguments to data constructors of GADTs). However, this system differs from earlier versions in that the type form for these objects, written $\forall\, c{:}\, \phi.\, \tau$, *names* the abstracted proof with the variable c and allows the type τ to refer to this coercion.

  This extension is necessary for some kind-indexed GADTs. For example, consider the following datatype, which is polymorphic over a kind and type parameter.

  ```
  data T :: ∀ k. k → * where
     K :: ∀ (b :: *). b → T b
  ```

  The single data constructor K constrains the kind to be $\star$ but does not otherwise constrain the type.

  After translation, the data constructor should be given the following FC type, where the abstracted kind coercion c is used to cast the kind of the parameter k.

  $$\mathsf{K}{:}\,\forall\, k{:}\star,\ b{:}\,k.\,\forall\, c{:}\,(k \sim \star).\,(b \triangleright c) \to \mathsf{T}\, k\, b$$

4.1 Type system overview

The next few subsections go into more detail about these technical points. We start with a quick tour of the type system.

The new syntax for FC appears in Figure 1; forms that are new or modified in this paper are highlighted—these modifications are primarily in the type and coercion languages. Also, note that $\star$ is a new type constant and κ is a metavariable for types. The only difference in the grammar for expressions is that type abstractions and kind abstractions have been merged. In general, the type system and operational semantics for the expression language is the same here as in prior versions of FC.

A context Γ is a list of assumptions for term variables (x), type variables/datatypes/data constructors (w), coercion variables (c), and coercion axioms (C).

$$\Gamma ::= \varnothing \mid \Gamma,\ x{:}\,\tau \mid \Gamma,\ w{:}\,\kappa \mid \Gamma,\ c{:}\,\phi \mid \Gamma,\ C{:}\,\forall\,\Delta.\,\phi$$

The type system includes the following judgements:

$\vdash_{\mathsf{wf}} \Gamma$	Context validity	(Figure 5)
$\Gamma \vdash_{\mathsf{ty}} \tau : \kappa$	Type/kind validity	(Figure 2)
$\Gamma \vdash_{\mathsf{pr}} \phi\ \mathsf{ok}$	Proposition validity	(Figure 3)
$\Gamma \vdash_{\mathsf{tm}} e : \tau$	Expression typing	(extended version)
$\Gamma \vdash_{\mathsf{co}} \gamma : \phi$	Coercion validity	(Figure 4)
$\Gamma \vdash_{\mathsf{tel}} \overline{\rho} \Leftarrow \Delta$	Telescope arg. validity	(extended version)

Each of the judgements is syntax directed: given the information before the colon (if present), a simple algorithm determines if the judgement holds, and recovers the information after the colon.

4.2 Type and kind formation

We next describe our extensions and modifications to the rules classifying FC types into kinds, which appear in Figure 2. Some of these rules are unchanged or only slightly modified from prior versions of FC.

For example, rule K_VAR looks up the kind of a type-level name from the typing context. Unlike previous systems, this rule now covers the kinding of promoted constructors, since w ranges over them. Recall that datatype promotion allows data constructors, such as `TInt`, to appear in types and be the arguments of type functions. Previously, the types of data constructors had to be explicitly promoted to kinds (Yorgey et al. 2012). Now, *any* data constructor may freely be used as a type. When the constructor is used as a type, its kind is the same as the type of the constructor when used as a term.

Rule K_ARROW gives the expected kind for the arrow type constructor. We use the usual syntactic sugar for arrow types, writing $\tau_1 \to \tau_2$ for $(\to)\,\tau_1\,\tau_2$. Note that the kind of the arrow type constructor is itself an arrow type. However, that circularity does not cause difficulty. After that, the rule K_ALLT describes when polymorphic types are well formed.

The next two rules describe when type application is well-formed. Application is overloaded in these rules, but the system is still syntax-directed—the type of the first component determines which rule applies. We do not combine function types $\sigma_1 \to \sigma_2$

[4] GHC already uses a shared datatype for types and kinds, so this merge brings the formalism closer to the actual implementation.

[5] If a consistent type language were desired for FC for other reasons, we believe that the ideas presented in this paper are adaptable to the stratification of $\star$ into *universe levels* (Luo 1994), as is done in Coq and Agda.

$\boxed{\Gamma \vdash_{\mathsf{ty}} \tau : \kappa}$

$$\frac{\vdash_{\mathsf{wf}} \Gamma \quad w{:}\kappa \in \Gamma}{\Gamma \vdash_{\mathsf{ty}} w : \kappa}\ \textsc{K\_Var}$$

$$\frac{\vdash_{\mathsf{wf}} \Gamma}{\Gamma \vdash_{\mathsf{ty}} (\to) : \star \to \star \to \star}\ \textsc{K\_Arrow}$$

$$\frac{\Gamma,\, a{:}\kappa \vdash_{\mathsf{ty}} \tau : \star \quad \Gamma \vdash_{\mathsf{ty}} \kappa : \star}{\Gamma \vdash_{\mathsf{ty}} \forall\, a{:}\kappa.\, \tau : \star}\ \textsc{K\_AllT}$$

$$\frac{\Gamma \vdash_{\mathsf{ty}} \tau_1 : \kappa_1 \to \kappa_2 \quad \Gamma \vdash_{\mathsf{ty}} \tau_2 : \kappa_1}{\Gamma \vdash_{\mathsf{ty}} \tau_1\, \tau_2 : \kappa_2}\ \textsc{K\_App}$$

$$\frac{\Gamma \vdash_{\mathsf{ty}} \tau_1 : \forall\, a{:}\kappa_1.\, \kappa_2 \quad \Gamma \vdash_{\mathsf{ty}} \tau_2 : \kappa_1}{\Gamma \vdash_{\mathsf{ty}} \tau_1\, \tau_2 : \kappa_2[\tau_2/a]}\ \textsc{K\_Inst}$$

$$\frac{\vdash_{\mathsf{wf}} \Gamma}{\Gamma \vdash_{\mathsf{ty}} \star : \star}\ \textsc{K\_StarInStar}$$

$$\frac{\Gamma \vdash_{\mathsf{ty}} \tau_1 : \forall\, c{:}\phi.\, \kappa \quad \Gamma \vdash_{\mathsf{co}} \gamma_1 : \phi}{\Gamma \vdash_{\mathsf{ty}} \tau_1\, \gamma_1 : \kappa[\gamma_1/c]}\ \textsc{K\_CApp}$$

$$\frac{\Gamma,\, c{:}\phi \vdash_{\mathsf{ty}} \tau : \star \quad \Gamma \vdash_{\mathsf{pr}} \phi\ \mathsf{ok}}{\Gamma \vdash_{\mathsf{ty}} \forall\, c{:}\phi.\, \tau : \star}\ \textsc{K\_AllC}$$

$$\frac{\Gamma \vdash_{\mathsf{ty}} \tau : \kappa_1 \quad \Gamma \vdash_{\mathsf{co}} \eta : \kappa_1 \sim \kappa_2 \quad \Gamma \vdash_{\mathsf{ty}} \kappa_2 : \star}{\Gamma \vdash_{\mathsf{ty}} \tau \rhd \eta : \kappa_2}\ \textsc{K\_Cast}$$

Figure 2. Kind and type formation rules

$\boxed{\Gamma \vdash_{\mathsf{pr}} \phi\ \mathsf{ok}}$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{ty}} \sigma_1 : \kappa_1 \\ \Gamma \vdash_{\mathsf{ty}} \sigma_2 : \kappa_2\end{array}}{\Gamma \vdash_{\mathsf{pr}} \sigma_1 \sim \sigma_2\ \mathsf{ok}}\ \textsc{Prop\_Equality}$$

Figure 3. Proposition formation rule

and polymorphic types $\forall\, a{:}\kappa.\, \sigma$ into a single form because of type erasure: term arguments are necessary at runtime, whereas type arguments may be erased. Although this distinction is meaningless at the kind level, it is benign. Identifying these forms at the kind level while retaining the distinction at the term level would needlessly complicate the language.

The rules K_StarInStar, K_Cast and K_CApp and K_AllC check the new type forms. The first says that $\star$ has kind $\star$.

To preserve the syntax-directed nature of FC, we must make the use of kind equality proofs explicit. We do so via the new form $\tau \rhd \gamma$ of kind casts: when given a type τ of kind κ_1 and a proof γ that kind κ_1 equals kind κ_2, the cast produces a type of kind κ_2. Because equality is heterogeneous, the K_Cast rule requires a third premise to ensure that the new kind has the correct classification, so that inhabited types have kind $\star$.

To promote GADTs we must be able to promote data constructors that take coercions as arguments, requiring the new application form $\tau\, \gamma$. For example, the data constructor TInt (from Section 3) requires a type argument τ and a proof that $\tau \sim \mathsf{Int}$. Note that there is no type-level abstraction over coercion—the form $\tau\, \gamma$ can only appear when the head of τ is a promoted datatype constructor.

4.3 Coercions

Coercions are proof terms witnessing the equality between types (and kinds), and are classified by propositions ϕ. The rules under which the proofs can be derived appear in Figure 4, with the validity

$\boxed{\Gamma \vdash_{\mathsf{co}} \gamma : \phi}$

$$\frac{\Gamma \vdash_{\mathsf{ty}} \tau : \kappa}{\Gamma \vdash_{\mathsf{co}} \langle \tau \rangle : \tau \sim \tau}\ \textsc{CT\_Refl}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \sim \tau_2}{\Gamma \vdash_{\mathsf{co}} \mathbf{sym}\, \gamma : \tau_2 \sim \tau_1}\ \textsc{CT\_Sym}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma_1 : \tau_1 \sim \tau_2 \quad \Gamma \vdash_{\mathsf{co}} \gamma_2 : \tau_2 \sim \tau_3}{\Gamma \vdash_{\mathsf{co}} \gamma_1 \mathbin{;} \gamma_2 : \tau_1 \sim \tau_3}\ \textsc{CT\_Trans}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{co}} \gamma_1 : \tau_1' \sim \tau_2' \quad \Gamma \vdash_{\mathsf{co}} \gamma_2 : \tau_1 \sim \tau_2 \\ \Gamma \vdash_{\mathsf{ty}} \tau_1'\, \tau_1 : \kappa_1 \quad \Gamma \vdash_{\mathsf{ty}} \tau_2'\, \tau_2 : \kappa_2\end{array}}{\Gamma \vdash_{\mathsf{co}} \gamma_1\, \gamma_2 : \tau_1'\, \tau_1 \sim \tau_2'\, \tau_2}\ \textsc{CT\_App}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{co}} \gamma_1 : \tau_1 \sim \tau_1' \\ \Gamma \vdash_{\mathsf{ty}} \tau_1\, \gamma_2 : \kappa \quad \Gamma \vdash_{\mathsf{ty}} \tau_1'\, \gamma_2' : \kappa'\end{array}}{\Gamma \vdash_{\mathsf{co}} \gamma_1(\gamma_2, \gamma_2') : \tau_1\, \gamma_2 \sim \tau_1'\, \gamma_2'}\ \textsc{CT\_CApp}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{co}} \eta : \kappa_1 \sim \kappa_2 \\ \Gamma,\, a_1{:}\kappa_1,\, a_2{:}\kappa_2,\, c{:}\, a_1 \sim a_2 \vdash_{\mathsf{co}} \gamma : \tau_1 \sim \tau_2 \\ \Gamma \vdash_{\mathsf{ty}} \forall\, a_1{:}\kappa_1.\, \tau_1 : \star \quad \Gamma \vdash_{\mathsf{ty}} \forall\, a_2{:}\kappa_2.\, \tau_2 : \star\end{array}}{\Gamma \vdash_{\mathsf{co}} \forall_{\eta}(a_1, a_2, c).\gamma : (\forall\, a_1{:}\kappa_1.\, \tau_1) \sim (\forall\, a_2{:}\kappa_2.\, \tau_2)}\ \textsc{CT\_AllT}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{co}} \eta_1 : \sigma_1 \sim \sigma_1' \quad \phi_1 = \sigma_1 \sim \sigma_2 \\ \Gamma \vdash_{\mathsf{co}} \eta_2 : \sigma_2 \sim \sigma_2' \quad \phi_2 = \sigma_1' \sim \sigma_2' \\ c_1 \# |\gamma| \quad c_2 \# |\gamma| \\ \Gamma,\, c_1{:}\phi_1,\, c_2{:}\phi_2 \vdash_{\mathsf{co}} \gamma : \tau_1 \sim \tau_2 \\ \Gamma \vdash_{\mathsf{ty}} \forall\, c_1{:}\phi_1.\, \tau_1 : \star \quad \Gamma \vdash_{\mathsf{ty}} \forall\, c_2{:}\phi_2.\, \tau_2 : \star\end{array}}{\Gamma \vdash_{\mathsf{co}} \forall_{(\eta_1, \eta_2)}(c_1, c_2).\gamma : (\forall\, c_1{:}\phi_1.\, \tau_1) \sim (\forall\, c_2{:}\phi_2.\, \tau_2)}\ \textsc{CT\_AllC}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \sim \tau_2 \quad \Gamma \vdash_{\mathsf{ty}} \tau_1 \rhd \gamma' : \kappa}{\Gamma \vdash_{\mathsf{co}} \gamma \rhd \gamma' : \tau_1 \rhd \gamma' \sim \tau_2}\ \textsc{CT\_Coh}$$

$$\frac{c{:}\phi \in \Gamma \quad \vdash_{\mathsf{wf}} \Gamma}{\Gamma \vdash_{\mathsf{co}} c : \phi}\ \textsc{CT\_Var}$$

$$\frac{C{:}\, \forall\, \Delta.\, (\tau_1 \sim \tau_2) \in \Gamma \quad \Gamma \vdash_{\mathsf{tel}} \overline{\rho} \Leftarrow \Delta}{\Gamma \vdash_{\mathsf{co}} C\, \overline{\rho} : \tau_1[\overline{\rho}/\Delta] \sim \tau_2[\overline{\rho}/\Delta]}\ \textsc{CT\_Axiom}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{co}} \gamma : H\, \overline{\rho} \sim H\, \overline{\rho}' \\ \rho_i = \tau \quad \rho_i' = \tau'\end{array}}{\Gamma \vdash_{\mathsf{co}} \mathbf{nth}^i\, \gamma : \tau \sim \tau'}\ \textsc{CT\_Nth}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma_1 : (\forall\, a_1{:}\kappa_1.\, \tau_1) \sim (\forall\, a_2{:}\kappa_2.\, \tau_2)}{\Gamma \vdash_{\mathsf{co}} \mathbf{nth}^1\, \gamma_1 : \kappa_1 \sim \kappa_2}\ \textsc{CT\_Nth1TA}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{co}} \gamma_1 : (\forall\, a_1{:}\kappa_1.\, \tau_1) \sim (\forall\, a_2{:}\kappa_2.\, \tau_2) \\ \Gamma \vdash_{\mathsf{co}} \gamma_2 : \sigma_1 \sim \sigma_2 \\ \Gamma \vdash_{\mathsf{ty}} \sigma_1 : \kappa_1 \quad \Gamma \vdash_{\mathsf{ty}} \sigma_2 : \kappa_2\end{array}}{\Gamma \vdash_{\mathsf{co}} \gamma_1 @ \gamma_2 : \tau_1[\sigma_1/a_1] \sim \tau_2[\sigma_2/a_2]}\ \textsc{CT\_Inst}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : (\forall\, c{:}\kappa_1 \sim \kappa_2.\, \tau) \sim (\forall\, c'{:}\kappa_1' \sim \kappa_2'.\, \tau')}{\Gamma \vdash_{\mathsf{co}} \mathbf{nth}^1\, \gamma : \kappa_1 \sim \kappa_1'}\ \textsc{CT\_Nth1CA}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : (\forall\, c{:}\kappa_1 \sim \kappa_2.\, \tau) \sim (\forall\, c'{:}\kappa_1' \sim \kappa_2'.\, \tau')}{\Gamma \vdash_{\mathsf{co}} \mathbf{nth}^2\, \gamma : \kappa_2 \sim \kappa_2'}\ \textsc{CT\_Nth2CA}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{co}} \gamma : (\forall\, c_1{:}\phi_1.\, \tau_1) \sim (\forall\, c_2{:}\phi_2.\, \tau_2) \\ \Gamma \vdash_{\mathsf{co}} \gamma_1 : \phi_1 \quad \Gamma \vdash_{\mathsf{co}} \gamma_2 : \phi_2\end{array}}{\Gamma \vdash_{\mathsf{co}} \gamma @ (\gamma_1, \gamma_2) : \tau_1[\gamma_1/c_1] \sim \tau_2[\gamma_2/c_2]}\ \textsc{CT\_InstC}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \sim \tau_2 \quad \Gamma \vdash_{\mathsf{ty}} \tau_1 : \kappa_1 \quad \Gamma \vdash_{\mathsf{ty}} \tau_2 : \kappa_2}{\Gamma \vdash_{\mathsf{co}} \mathbf{kind}\, \gamma : \kappa_1 \sim \kappa_2}\ \textsc{CT\_Ext}$$

Figure 4. Coercion formation rules

rule for ϕ appearing in Figure 3. These rules establish properties of the type equality relation:

- Equality is an *equivalence relation*, as seen in rules CT_REFL, CT_SYM, and CT_TRANS.
- Equality is *congruent*—types with equal subcomponents are equal. Every type formation rule (except for the base cases like variables and constants) has an associated congruence rule. The exception is kind coercion $\tau \triangleright \gamma$, where the congruence rule is derivable (see Section 4.3.2). The congruence rules are mostly straightforward; we discuss the rules for quantified types (rules CT_ALLT and CT_ALLC) in Section 4.3.1.
- Equality can be *assumed*. Coercion variables and axioms add assumptions about equality to the context and appear in proofs (using rules CT_VAR and CT_AXIOM respectively). These axioms for type equality are allowed to be *axiom schemes*—they may be parameterized and must be instantiated when used.

  The general form of the type of an axiom, $C{:}\ \forall\,\Delta.\,\phi$ gathers multiple parameters in a *telescope*, a context denoted with Δ of type and coercion variables, each of which scope over the remainder of the telescope as well as the body of the axiom. We specify the list of instantiations for a telescope with $\overline{\rho}$, a mixed list of types and coercions. When type checking an axiom application, we must type check its list of arguments $\overline{\rho}$ against the given telescope. The judgement form $\Gamma \vdash_{\mathsf{tel}} \overline{\rho} \Leftarrow \Delta$ (presented in the extended version of this paper) checks each argument ρ in turn against the binding in the telescope, scoping variables appropriately.
- Equality can be *decomposed* using the next six rules. For example, because we know that datatypes are injective type functions, we can decompose a proof of the equivalence of two datatypes into equivalence proofs for any pair of corresponding type parameters (CT_NTH). Furthermore, the equivalence of two polymorphic types means that the kinds of the bound variables are equivalent (CT_NTH1TA), and that all instantiations of the bound variables are equivalent (CT_INST). The same is true for coercion abstraction types (rules CT_NTH1CA, CT_NTH2CA, and CT_INSTC).
- Equality is *heterogeneous*. If γ is a proof of the equality $\tau_1 \sim \tau_2$, then $\mathbf{kind}\,\gamma$ extracts a proof of equality between the kinds of τ_1 and τ_2.

4.3.1 Congruence rules for quantified types

In prior versions of FC, the coercion $\forall a{:}\,\kappa.\gamma$ proved the equality proposition $\forall\,a{:}\,\kappa.\,\tau_1 \sim \forall\,a{:}\,\kappa.\,\tau_2$, using the following rule:

$$\frac{\Gamma \vdash_{\mathsf{ty}} \kappa \ :\ \star \qquad \Gamma,\ a{:}\,\kappa \vdash_{\mathsf{co}} \gamma\ :\ \tau_1 \sim \tau_2}{\Gamma \vdash_{\mathsf{co}} \forall\,a{:}\,\kappa.\,\gamma\ :\ (\forall\,a{:}\,\kappa.\,\tau_1) \sim (\forall\,a{:}\,\kappa.\,\tau_2)}\ \text{CT\_ALLTX}$$

This rule sufficed because the only quantified types that could be shown equal had the same syntactic kinds κ for the bound variable. However, we now have a nontrivial equality between kinds. We need to be able to show a more general proposition, $\forall\,a{:}\,\kappa_1.\,\tau_1 \sim \forall\,a{:}\,\kappa_2.\,\tau_2$, even when κ_1 is not syntactically equal to κ_2.

Without this generality, the language does not satisfy the preservation theorem, which requires that the equality relation be substitutive—given a valid type σ where a appears free, and a proof $\Gamma \vdash_{\mathsf{co}} \gamma\ :\ \tau_1 \sim \tau_2$, we must be able to derive a proof between $\sigma[\tau_1/a]$ and $\sigma[\tau_2/a]$. For this property to hold, if a occurs in the kind of a quantified type (or coercion) variable $\forall\,b{:}\,a.\,\tau$, then we must be able to derive $\forall\,b{:}\,\tau_1.\,\tau \sim \forall\,b{:}\,\tau_2.\,\tau$.

Rule CT_ALLT shows when two polytypes are equal. The first premise requires a proof η that the kinds of the bound variables are equal. But, these two kinds might not be *syntactically* equal, so we must have two type variables, a_1 and a_2, one of each kind. The

$\boxed{\vdash_{\mathsf{wf}} \Gamma}$

$$\frac{\Gamma \vdash_{\mathsf{ty}} \forall\,\overline{a{:}\,\kappa}.\,\star\ :\ \star \qquad T \# \Gamma}{\vdash_{\mathsf{wf}} \Gamma,\ T{:}\,\forall\,\overline{a{:}\,\kappa}.\,\star}\ \text{GWF\_TYDATA}$$

$$\frac{\Gamma \vdash_{\mathsf{ty}} \forall\,\overline{a{:}\,\kappa}.\,\forall\,\Delta.\,(\overline{\sigma} \to T\ \overline{a})\ :\ \star \qquad K \# \Gamma}{\vdash_{\mathsf{wf}} \Gamma,\ K{:}\,\forall\,\overline{a{:}\,\kappa}.\,\forall\,\Delta.\,(\overline{\sigma} \to T\ \overline{a})}\ \text{GWF\_CON}$$

$$\frac{\Gamma, \Delta \vdash_{\mathsf{pr}} \phi\ \mathsf{ok} \qquad C \# \Gamma}{\vdash_{\mathsf{wf}} \Gamma,\ C{:}\ \forall\,\Delta.\,\phi}\ \text{GWF\_AX}$$

Figure 5. Context formation rules (excerpted)

second premise of the rule adds both bindings $a_1{:}\,\kappa_1$ and $a_2{:}\,\kappa_2$ to the context as well as an assertion c that a_1 and a_2 are equal. The polytypes themselves can only refer to their own variables, as verified by the last two premises of the rule.

The other type form that includes binding is the coercion abstractions, $\forall\,c{:}\,\phi.\,\tau$. The rule CT_ALLC constructs a proof that two such types of this form are equal. We can only construct such proofs when the abstracted propositions relate correspondingly equal types, as witnessed by proofs η_1 and η_2. The proof term introduces two coercion variables into the context, similar to the two type variables above. Due to proof irrelevance, there is no need for a proof of equality between coercions themselves. Note that the kind of c_1 is *not* that of η_1: the kind of c_1 is built from types in both η_1 and η_2.

The rule CT_ALLC also restricts how the variables c_1 and c_2 can be used in γ. The premises $c_1 \# |\gamma|$ and $c_2 \# |\gamma|$ prevent these variables from appearing in the relevant parts of γ. (The freshness operator $\#$ requires its two arguments to have disjoint sets of free variables.) This restriction stems from our proof technique for the consistency of this proof system; we define the erasure operation $|\cdot|$ and discuss this issue in more detail in Section 6.

4.3.2 Coercion irrelevance and coherence

Although the type system includes a judgement for type equality, and types may include explicit coercion proofs, the system does not include a judgement that states when two coercions proofs are equal. The reason is that this relation is trivial—all coercions should be considered equivalent. As a result, coercion proofs are irrelevant to type equality.

This "proof irrelevance" is guaranteed by several of the coercion rules. Consider the congruence rule for coercion application, CT_CAPP: there are no restrictions on γ_2 and γ_2' other than well-formedness. Another example is rule CT_INSTC—again, no relation is required between the coercions γ_1 and γ_2.

Not only is the identity of coercion proofs irrelevant, but it is always possible to equate a type with a casted version of itself. The coherence rule, CT_COH, essentially says that the use of kind coercions can be ignored when proving type equalities. Although this rule seems limited, it is sufficient to derive the elimination and congruence rules for coerced types, as seen below.

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma\ :\ \tau_1 \sim \tau_2 \qquad \Gamma \vdash_{\mathsf{ty}} \tau_1 \triangleright \eta_1\ :\ \kappa_1 \qquad \Gamma \vdash_{\mathsf{ty}} \tau_2 \triangleright \eta_2\ :\ \kappa_2}{\Gamma \vdash_{\mathsf{co}} (\mathbf{sym}\,((\mathbf{sym}\,\gamma) \triangleright \eta_2)) \triangleright \eta_1\ :\ \tau_1 \triangleright \eta_1 \sim \tau_2 \triangleright \eta_2}$$

(Again, note that there is no relation required between η_1 and η_2.) We use the syntactic sugar $\gamma \triangleright \eta_1 \sim \eta_2$ to abbreviate the coercion $(\mathbf{sym}\,((\mathbf{sym}\,\gamma) \triangleright \eta_2)) \triangleright \eta_1$.

4.4 Datatypes

Because we focus on the treatment of equality in the type language, we omit most of the discussion of the expression language and its operational semantics. However, since we have collapsed types and kinds, we must revise the treatment of datatypes, whose constructors can contain types and kinds as arguments. Previously, the arguments to datatype constructors were ordered with all kind arguments occurring before all type arguments (Yorgey et al. 2012). In this language, we cannot divide up the arguments in this way. Therefore, we again use the technique of telescopes to describe the more complex dependency between arguments.

The validity rules for contexts (see Figure 5) restrict datatype constants T to have kind $\forall\, \overline{a{:}\,\kappa}.\, \star$. We call the variables $\overline{a}$ the *parameters* of the datatype. For example, the kind of the datatype List is $\forall\, a{:}\,\star.\,\star$ and the kind of the datatype TyRep (the first version from Section 2) is $\forall\, k{:}\,\star,\ t{:}\,k.\,\star$. Furthermore, datatypes can only be parameterized by types and kinds, not coercions.

Likewise, the same validity rules force data constructors K to have types/kinds of the form

$$\forall\, \overline{a{:}\,\kappa}.\, \forall\, \Delta.\, (\overline{\sigma} \to\ T\ \overline{a}).$$

Each data constructor K must produce an element of T applied to all of its parameters $\overline{a{:}\,\kappa}$. Above, the form $\forall\, \Delta.\, \tau$ is syntactic sugar for a list of nested quantified types. The scope of the bound variables includes both the remainder of the telescope Δ and the form within the quantification (in this case, $\overline{\sigma} \to\ T\ \overline{a}$).

The telescope Δ describes the *existential* arguments to the data constructor. These arguments may be either coercions or types, and because of the dependency, must be allowed to freely intermix. For example, the data constructor TyInt from Section 2 (a data constructor belonging to TyRep $:\ \forall\, k{:}\,\star,\ t{:}\,k.\,\star$) includes two coercions in its telescope, one asserting that the kind parameter k is $\star$, the second asserting that the type parameter t is Int:

$$\mathsf{TyInt} : \forall\, k{:}\,\star,\ t{:}\,k.\, \forall\, c_1{:}\,k \sim \star,\ c_2{:}\,t \sim \mathsf{Int}.\, \mathsf{TyRep}\, k\, t$$

Likewise, the data constructor TyApp existentially binds k', a, b, and c—one kind and two type variables followed by a coercion.

$$\begin{array}{c}\mathsf{TyApp} : \forall k{:}\,\star, t{:}\,k. \forall k'{:}\,\star, a{:}\,k' \to k, b{:}\,k', c{:}\,t \sim a\ b. \\ \mathsf{TyRep}\,(k' \to k)\ a \to \mathsf{TyRep}\, k'\ b \to \mathsf{TyRep}\, k\, t\end{array}$$

A datatype value is of the form $K\ \overline{\tau}\ \overline{\rho}\ \overline{e}$, where $\overline{\tau}$ denotes the parameters (which cannot include coercions), $\overline{\rho}$ instantiate the existential arguments, and $\overline{e}$ is the list of usual expression arguments to the data constructor.

5. The "push" rules and the preservation theorem

Now that we have defined our extensions, we turn to the metatheory: preservation and progress. While the operational semantics is largely unchanged from prior work, we detail here a few key differences. The most intricate part of the operational semantics of FC are the "push" rules, which ensure that coercions do not interfere with the small step semantics. Coercions are "pushed" into the subcomponents of values whenever a coerced value appears in an elimination context. System FC has four push rules, one for each such context: term application, type application, coercion application, and pattern matching on a datatype. The first three are straightforward and are detailed in previous work (Yorgey et al. 2012). In this section, we focus on pattern matching and the S_KPUSH rule.

5.1 Pushing coercions through constructors

When pattern matching on a coerced datatype value of the form $K\ \overline{\tau}\ \overline{\rho}\ \overline{e} \rhd \gamma$, the coercion must be distributed over all of the arguments of the data constructor, producing a new scrutinee $K\ \overline{\tau'}\ \overline{\rho}'\ \overline{e'}$ as shown in Figure 6. In the rest of this section, we explain the rule by describing the formation of the *lifting context* Ψ and its use in the definition of $\overline{\tau'}$, $\overline{\rho}'$ and $\overline{e'}$.

The S_KPUSH rule uses a *lifting* operation $\Psi(\cdot)$ on expressions which coerces the type of its argument ($\overline{e}$ in Figure 6). For example,

$$\frac{\begin{array}{l} K{:}\, \forall\, \overline{a{:}\,\kappa}.\, \forall\, \Delta.\, \overline{\sigma} \to (T\ \overline{a})\ \in\ \Gamma \\ \Psi = \mathsf{extend}(\mathsf{context}(\gamma); \overline{\rho}; \Delta) \\ \overline{\tau'} = \Psi_2(\overline{a}) \\ \overline{\rho}' = \Psi_2(dom\,\Delta) \\ \text{for each } e_i \in \overline{e}, \\ \qquad e_i' = e_i \rhd \Psi(\sigma_i) \end{array}}{\begin{array}{l}\mathbf{case}\ ((K\ \overline{\tau}\ \overline{\rho}\ \overline{e}) \rhd \gamma)\ \mathbf{of}\ \overline{p \to u} \longrightarrow \\ \quad \mathbf{case}\ (K\ \overline{\tau'}\ \overline{\rho}'\ \overline{e'})\ \mathbf{of}\ \overline{p \to u}\end{array}} \quad \text{S\_KPUSH}$$

Figure 6. The S_KPUSH rule

suppose we have a data constructor K of type $\forall\, a{:}\,\star\,.\, F\ a \to T\ a$ for some type function F and some type constructor T. Consider what happens when a case expression scrutinee $(K\ \mathsf{Int}\ e) \rhd \gamma$, where γ is a coercion of type $T\ \mathsf{Int} \sim\ T\ \tau'$. The push rule should convert this expression to $K\ \tau'\ (e \rhd \gamma')$ for some new coercion γ' showing $F\ \mathsf{Int}\ \sim\ F\ \tau'$. To produce γ', we need to lift the type $F\ a$ to a coercion along the coercion $\mathbf{nth}^1\ \gamma$, which shows $\mathsf{Int} \sim \tau'$.

In previous work, lifting was written $\sigma[a \mapsto \gamma]$, defined by analogy with substitution. Because of the similar syntax of types and coercion proofs, we could think of lifting as replacing a type variable with a coercion to produce a new coercion. That intuition holds true here, but we require more machinery to make this precise.

Lifting contexts We define lifting with respect to a *lifting context* Ψ, which maps type variables to triples (τ_1, τ_2, γ) and coercion variables to pairs (η_1, η_2). The forms τ_1 and η_1 refer to the original, uncoerced parameters to the data constructor (Int in our example). The forms τ_2 and η_2 refer to the new, coerced parameters to the data constructor (like τ' in our example). Finally, the coercion γ witnesses the equality of τ_1 and τ_2. No witness is needed for the equality between η_1 and η_2—equality on proofs is trivial.

The lifting operation is defined by structural recursion on its type argument. This operation is complicated by type forms that bind fresh variables: $\forall\, a{:}\,\kappa.\, \tau$ and $\forall\, c{:}\,\phi.\, \tau$. Lifting over these types introduces new mappings in the lifting context, marked with $\overset{\bullet}{\mapsto}$.

$$\begin{array}{rclcl} \Psi ::= \emptyset & \mid & \Psi, a{:}\,\kappa \mapsto (\tau_1, \tau_2, \gamma) & \mid & \Psi, c{:}\,\phi \mapsto (\gamma_1, \gamma_2) \\ & \mid & \Psi, a{:}\,\kappa \overset{\bullet}{\mapsto} (a_1, a_2, c) & \mid & \Psi, c{:}\,\phi \overset{\bullet}{\mapsto} (c_1, c_2) \end{array}$$

(We use the notation $\overset{?}{\mapsto}$ to refer to a mapping created either with $\mapsto$ or with $\overset{\bullet}{\mapsto}$.) A lifting context Ψ induces two multisubstitutions $\Psi_1(\cdot)$ and $\Psi_2(\cdot)$, as follows:

Definition 5.1 (Lifting context substitution). *$\Psi_1(\cdot)$ and $\Psi_2(\cdot)$ are multisubstitutions, applicable to types, coercions, telescopes, typing contexts, and even other lifting contexts.*

1. *For each $a{:}\,\kappa \overset{?}{\mapsto} (\tau_1, \tau_2, \gamma)$ in Ψ, $\Psi_1(\cdot)$ maps a to τ_1 and $\Psi_2(\cdot)$ maps a to τ_2.*
2. *For each $c{:}\,\phi \overset{?}{\mapsto} (\gamma_1, \gamma_2)$ in Ψ, $\Psi_1(\cdot)$ maps c to γ_1 and $\Psi_2(\cdot)$ maps c to γ_2.*

The two substitution operations satisfy straightforward substitution lemmas, defined and proved in the extended version of this paper. The usual substitution lemmas, which substitute a single type or coercion, are a direct corollary of these lemmas.

We can now define *lifting*:[6]

Definition 5.2 (Lifting). *We define the lifting of types to coercions, written $\Psi(\tau)$, by induction on the type structure. The following equations, to be tried in order, define the operation. (Note that the last line uses the syntactic sugar introduced in Section 4.3.2.)*

[6] This definition is not just for the proof—it is implemented in GHC as part of the optimizer to reduce case expressions.

$$
\begin{array}{ll}
\Psi(a) & = \gamma \text{ when } a{:}\,\kappa \overset{?}{\mapsto} (\tau_1, \tau_2, \gamma) \in \Psi \\
\Psi(\tau) & = \langle \tau \rangle \text{ when } \tau \,\#\, \mathit{dom}(\Psi) \\
\Psi(\tau_1\, \tau_2) & = \Psi(\tau_1)\, \Psi(\tau_2) \\
\Psi(\tau\, \gamma) & = \Psi(\tau)(\Psi_1(\gamma), \Psi_2(\gamma)) \\
\Psi(\forall\, a{:}\,\kappa.\, \tau) & = \forall_{\Psi(\kappa)}(a_1, a_2, c).\Psi'(\tau) \\
& \quad \text{where } \Psi' = \Psi, a{:}\,\kappa \overset{\bullet}{\mapsto} (a_1, a_2, c) \\
& \quad \text{and } a_1, a_2, c \text{ are fresh} \\
\Psi(\forall\, c{:}\,\sigma_1 \sim \sigma_2.\, \tau) & = \forall_{(\Psi(\sigma_1), \Psi(\sigma_2))}(c_1, c_2).\Psi'(\tau) \\
& \quad \text{where } \Psi' = \Psi, c{:}\,\sigma_1 \sim \sigma_2 \overset{\bullet}{\mapsto} (c_1, c_2) \\
& \quad \text{and } c_1, c_2 \text{ are fresh} \\
\Psi(\tau \triangleright \gamma) & = \Psi(\tau) \triangleright \Psi_1(\gamma) \sim \Psi_2(\gamma)
\end{array}
$$

The lifting lemma establishes the correctness of the lifting operation and shows that equality is congruent.

Lemma 5.3 (Lifting Lemma)**.** *If Ψ is a valid lifting context for context Γ and the telescope Δ, and $\Gamma, \Delta \vDash_{\mathsf{ty}} \tau : \kappa$, then*

$$\Gamma \vDash_{\mathsf{co}} \Psi(\tau) \;:\; \Psi_1(\tau) \sim \Psi_2(\tau)$$

Lifting context creation In the S_KPUSH rule, the actual context Ψ used for lifting is built in two stages. First, $\mathsf{context}(\gamma)$ defines a lifting context with coercions for the parameters to the datatype.

Definition 5.4 (Lifting context generation)**.** *If $\Gamma \vDash_{\mathsf{co}} \gamma : T\,\overline{\sigma} \sim T\,\overline{\sigma'}$, and $T{:}\,\forall\,\overline{a{:}\,\kappa}.\star \in \Gamma$, where the lists $\overline{\sigma}$, $\overline{\sigma'}$, and $\overline{a{:}\,\kappa}$ are all of length n, then define $\mathsf{context}(\gamma)$ as*

$$\mathsf{context}(\gamma) = \overline{a_i{:}\,\kappa_i \mapsto (\sigma_i, \sigma'_i, \mathbf{nth}^i\,\gamma)}^{\,i \in 1..n}$$

Intuitively, $(\mathsf{context}(\gamma))_1(\tau)$ replaces all parameters a in τ with the corresponding type on the left of $\sim$ in the type of γ. Similarly, $(\mathsf{context}(\gamma))_2(\tau)$ replaces a with the corresponding type on the right of $\sim$.

Next, this initial lifting context is extended with coercions using the operation $\mathsf{extend}(\cdot)$, which adds mappings for the variables in Δ, the existential parameters to the data constructor K. Due to the dependency, we define the operation recursively. The intuition still holds: $(\mathsf{extend}(\Psi; \overline{\rho}; \Delta))_1(\tau)$ replaces free variables in τ with their corresponding "from" types, while $(\mathsf{extend}(\Psi; \overline{\rho}; \Delta))_2(\tau)$ replaces a variables with their corresponding "to" types.

Definition 5.5 (Lifting context extension)**.** *Define the operation of lifting context extension, written $\mathsf{extend}(\Psi; \overline{\rho}; \Delta)$, as:*

$$
\begin{array}{lcl}
\mathsf{extend}(\Psi; \varnothing; \varnothing) & = & \Psi \\
\mathsf{extend}(\Psi; \overline{\rho}, \tau; \Delta,\ a{:}\,\kappa) & = & \\
\quad \Psi', a{:}\,\kappa \mapsto (\tau, \tau \triangleright \Psi'(\kappa), \mathbf{sym}\,(\langle\tau\rangle \triangleright \Psi'(\kappa))) & & \\
\quad \text{where } \Psi' = \mathsf{extend}(\Psi; \overline{\rho}; \Delta) & & \\
\mathsf{extend}(\Psi; \overline{\rho}, \gamma; \Delta,\ c{:}\,\sigma_1 \sim \sigma_2) & = & \\
\quad \Psi', c{:}\,\sigma_1 \sim \sigma_2 \mapsto (\gamma, \mathbf{sym}\,(\Psi'(\sigma_1)) \,{\scriptstyle{}^\circ_9}\, \gamma \,{\scriptstyle{}^\circ_9}\, \Psi'(\sigma_2)) & & \\
\quad \text{where } \Psi' = \mathsf{extend}(\Psi; \overline{\rho}; \Delta) & &
\end{array}
$$

5.2 Type preservation

Now that we have explained the most novel part of the operational semantics, we can state the preservation theorem.

Theorem 5.6 (Preservation)**.** *If $\Gamma \vDash_{\mathsf{tm}} e : \tau$ and $e \longrightarrow e'$ then $\Gamma \vDash_{\mathsf{tm}} e' : \tau$.*

The proof of this theorem is by induction on the typing derivation, with a case analysis on the small-step. Most of the rules are straightforward, following directly by induction or by substitution. The "push" rules require reasoning about coercion propagation. We include the details of the rules that differ from previous work (Weirich et al. 2010) in the extended version of this paper.

5.3 Correctness of push rules: The type erasure theorem

We care not only that the push rules preserve types, but that they do "the right thing." Do these rules reduce to no-ops if we erase types and coercions?

To state this formally, we define an erasure operation $|\cdot|$ over expressions. This operation erases types, coercions, and equality propositions to trivial forms $\bullet_{\mathsf{ty}}$, $\bullet_{\mathsf{co}}$ and $\bullet_{\mathsf{prop}}$ and removes all casts. The full definition of this operation appears in the extended version of this paper, and we present only the interesting cases here:

$$|e\,\tau| = |e|\,\bullet_{\mathsf{ty}} \qquad |e\,\gamma| = |e|\,\bullet_{\mathsf{co}} \qquad |e \triangleright \gamma| = |e|$$

With this operation, we can state that erasing types, coercions and casts does not change how expressions evaluate e.

Theorem 5.7 (Type erasure)**.** *If $e \longrightarrow e'$, then either $|e| = |e'|$ or $|e| \longrightarrow |e'|$.*

6. Consistency and the progress theorem

The proof for the progress theorem follows the same course as in previous work (Weirich et al. 2010). The progress theorem holds only for *closed*, *consistent* contexts. A context is *closed* if it does not contain any expression variable bindings—as usual, open expressions could be stuck. We use the metavariable Σ to denote closed contexts.

Theorem 6.1 (Progress)**.** *Assume Σ is a closed, consistent context. If $\Sigma \vDash_{\mathsf{tm}} e_1 : \tau$ and e_1 is not a value v or a coerced value $v \triangleright \gamma$, then there exists an e_2 such that $e_1 \longrightarrow e_2$.*

The definition of consistent contexts is stated using the notions of uncoerced *values* and their types, *value types*. Formally, we define values v and value types ξ, with the following grammars:

$$
\begin{array}{lcl}
v & ::= & \lambda x{:}\,\sigma.\, e \mid \Lambda a{:}\,\kappa.\, e \mid \lambda c{:}\,\phi.\, e \mid K\,\overline{\tau}\,\overline{\rho}\,\overline{e} \\
\xi & ::= & \sigma_1 \to \sigma_2 \mid \forall\, a{:}\,\kappa.\,\sigma \mid \forall\, c{:}\,\phi.\,\sigma \mid T\,\overline{\sigma}
\end{array}
$$

Definition 6.2 (Consistency)**.** *A context Γ is consistent if ξ_1 and ξ_2 have the same head form whenever $\Gamma \vDash_{\mathsf{co}} \gamma : \xi_1 \sim \xi_2$.*

Although the extensions in this paper have little effect on the structure of this proof compared to prior work, there is still work to do: we need an new notion of acceptable contexts to allow kind equalities, and we must prove that these contexts are consistent.

Our consistency argument proceeds in four steps:

1. Because coercion proofs are irrelevant to type equivalence, we start with an *implicitly coerced* version of the language, where all coercion proofs have been erased. Derivations in the explicit language can be matched up with derivations in the implicit language (Definition 6.3) so showing consistency in the latter implies consistency in the former.
2. We define a *rewrite relation* that reduces types in the implicit system by firing axioms in the context (Figure 7).
3. We specify a sufficient condition, which we write $\mathbf{Good}\,\Gamma$ (Definition 6.5), for a context to be consistent. This condition allows the axioms produced by type and kind family definitions.
4. We show that good contexts are consistent by arguing that the joinability of the rewrite relation is complete with respect to the implicit coercion proof system. Since the rewrite relation and erasure preserve the head form of value types, this gives consistency for both the implicit and explicit systems.

Since we don't want consistency to depend on particular proofs of kind equality, we prove our results with an implicit version of the type language. This implicit language elides coercion proofs and casts from the type language, and has judgements (denoted with a turnstile $\models$) analogous to the explicit language but for a few key

$\boxed{\Gamma \models \tau \rightsquigarrow \tau'}$

$$\frac{}{\Gamma \models \tau \rightsquigarrow \tau} \quad \text{TS\_REFL}$$

$$\frac{\Gamma, \Gamma' \models \kappa \rightsquigarrow \kappa' \quad \Gamma,\ c{:}\, a_1 \sim a_2, \Gamma' \models \sigma \rightsquigarrow \sigma'}{\Gamma, \Gamma' \models \forall\, a_1{:}\, \kappa.\, \sigma \rightsquigarrow \forall\, a_2{:}\, \kappa'.\, \sigma'} \quad \text{TS\_ALLT}$$

$$\frac{\Gamma \models \tau_1 \rightsquigarrow \tau_1' \quad \Gamma \models \tau_2 \rightsquigarrow \tau_2' \quad \Gamma \models \sigma \rightsquigarrow \sigma'}{\Gamma \models \forall\, c{:}\, \tau_1 \sim \tau_2.\, \sigma \rightsquigarrow \forall\, c{:}\, \tau_1' \sim \tau_2'.\, \sigma'} \quad \text{TS\_ALLC}$$

$$\frac{\begin{array}{c} C{:}\, \forall\, \Delta.\, (F\, \overline{\tau} \sim \tau') \in \Gamma \\ \overline{\sigma_1 = \tau[\overline{\rho}/\Delta]} \quad \sigma_1' = \tau'[\overline{\rho}/\Delta] \end{array}}{\Gamma \models F\, \overline{\sigma_1} \rightsquigarrow \sigma_1'} \quad \text{TS\_RED}$$

$$\frac{c{:}\, a \sim \tau \in \Gamma}{\Gamma \models a \rightsquigarrow \tau} \quad \text{TS\_VARRED}$$

$$\frac{\Gamma \models \tau \rightsquigarrow \tau' \quad \Gamma \models \sigma \rightsquigarrow \sigma'}{\Gamma \models \tau\, \sigma \rightsquigarrow \tau'\, \sigma'} \quad \text{TS\_APP}$$

$$\frac{\Gamma \models \tau \rightsquigarrow \tau'}{\Gamma \models \tau \bullet_{\mathsf{co}} \rightsquigarrow \tau' \bullet_{\mathsf{co}}} \quad \text{TS\_CAPP}$$

Figure 7. Rewrite relation

differences where coercions are dropped from types. To connect the explicit and implicit systems, we define an erasure operation:

Definition 6.3 (Coercion Erasure)**.** *Given an explicitly typed term τ or coercion γ, we define its* erasure*, denoted $|\tau|$ or $|\gamma|$, by induction on its structure. The interesting cases follow:*

$$\begin{array}{lcllcl} |\tau \rhd \gamma| & = & |\tau| & |\gamma(\gamma_1, \gamma_2)| & = & |\gamma|(\bullet_{co}, \bullet_{co}) \\ |\tau\, \gamma| & = & |\tau| \bullet_{co} & |\gamma \rhd \gamma'| & = & |\gamma| \\ & & & |\gamma @ (\gamma', \gamma'')| & = & |\gamma| @ (\bullet_{co}, \bullet_{co}) \end{array}$$

All other cases follow by simply propagating the $|\cdot|$ operation down the abstract syntax tree. (The full definition of this operation appears in the extended version of this paper.)

We further define the erasure *of a context Γ, denoted $|\Gamma|$, by erasing the types and equality propositions of each binding.*

Lemma 6.4 (Erasure is type preserving)**.** *If a judgement holds in the explicit system, the judgement with coercions erased throughout the context, types and coercions is derivable in the implicit system.*

We define a nondeterministic rewrite relation on open implicit types in Figure 7. We say that σ_1 is *joinable* with σ_2, written $\Gamma \models \sigma_1 \Leftrightarrow \sigma_2$, when both can multi-rewrite to a common reduct.

Consistency does not hold in arbitrary contexts, and it is difficult in general to check whether a context is inconsistent. Therefore, like in previous work (Weirich et al. 2010), we give sufficient conditions written $\mathbf{Good}\,\Gamma$, for a context to be consistent. Since we are working with the implicit language, these conditions are actually for the *erased* context.

Definition 6.5 (Good contexts)**.** *We have $\mathbf{Good}\,\Gamma$ when the following conditions hold:*

1. *All coercion assumptions and axioms in Γ are of the form $C{:}\, \forall\, \Delta.\, (F\, \overline{\tau} \sim \tau')$ or of the form $c{:}\, a_1 \sim a_2$. In the first form, the arguments to the type function must behave like patterns: for all $\overline{\rho}$, every $\tau_i \in \overline{\tau}$ and every τ_i' such that $\Gamma \models \tau_i[\overline{\rho}/\Delta] \rightsquigarrow \tau_i'$, there exists $\overline{\rho}'$ such that $\tau_i' = \tau_i[\overline{\rho'}/\Delta]$ and $\Gamma \models \sigma_m \rightsquigarrow \sigma_m'$ for each $\sigma_m \in \overline{\rho}$ and $\sigma_m' \in \overline{\rho}'$.*
2. *Axioms and coercion assumptions don't overlap. For each $F\, \overline{\tau}$, there exists at most one prefix $\overline{\tau_1}$ of $\overline{\tau}$ such that there exist C and $\overline{\rho}$ where $C{:}\, \forall\, \Delta.\, F\, \overline{\sigma_0} \sim \sigma_1 \in \Gamma$ and $\overline{\tau_1} = \overline{\sigma_0[\overline{\rho}/\Delta]}$. These C and $\overline{\rho}$ are unique for every matching $F\, \overline{\tau_1}$.*
3. *For each a, there is at most one assumption of the form $c{:}\, a \sim a'$ or $c{:}\, a' \sim a$, and $a \neq a'$.*
4. *Axioms equate types of the same kind. For each $C{:}\, \forall\, \Delta.\, (F\, \overline{\tau} \sim \tau')$ in Γ, the kinds of each side must equal: for some κ, $\Gamma, \Delta \models F\, \overline{\tau} : \kappa$ and $\Gamma, \Delta \models \tau' : \kappa$ and that kind must not mention bindings in the telescope, $\Gamma \models \kappa : \star$.*

The main lemma required for consistency is the completeness of joinability. Here, we write $fcv(\gamma) \subseteq dom\,\Gamma'$ to indicate that all coercion variables and axioms used in γ are in the domain of Γ'.

Lemma 6.6 (Completeness)**.** *Suppose that $\Gamma \models \gamma : \sigma_1 \sim \sigma_2$, and $fcv(\gamma) \subseteq dom\,\Gamma'$ for some subcontext Γ' satisfying $\mathbf{Good}\,\Gamma'$. Then $\Gamma \models \sigma_1 \Leftrightarrow \sigma_2$.*

The proof of this theorem appears in the extended version of this paper. Here, we highlight a technical point about coercions between coercion abstractions. The completeness proof requires that all coercion variables in a coercion γ must satisfy the requirements of Good contexts. As a result, we need to restrict the coercion abstraction equality rule in both the implicit and explicit systems.

$$\frac{\begin{array}{l} \Gamma \models \eta_1 : \sigma_1 \sim \sigma_1' \quad \phi_1 = \sigma_1 \sim \sigma_2 \\ \Gamma \models \eta_2 : \sigma_2 \sim \sigma_2' \quad \phi_2 = \sigma_1' \sim \sigma_2' \\ c_1 \# \gamma \quad c_2 \# \gamma \\ \Gamma,\ c_1{:}\, \phi_1,\ c_2{:}\, \phi_2 \models \gamma : \tau_1 \sim \tau_2 \\ \Gamma \models \forall\, c_1{:}\, \phi_1.\, \tau_1 : \star \quad \Gamma \models \forall\, c_2{:}\, \phi_2.\, \tau_2 : \star \end{array}}{\Gamma \models \forall_{(\eta_1, \eta_2)}(c_1, c_2).\gamma : (\forall\, c_1{:}\, \phi_1.\, \tau_1) \sim (\forall\, c_2{:}\, \phi_2.\, \tau_2)}$$

In this rule, the variables c_1 and c_2 cannot be used in γ due to the premises $c_1 \# \gamma$ and $c_2 \# \gamma$. (The analogous rule in the explicit system includes the premises $c_1 \# |\gamma|$ and $c_2 \# |\gamma|$.) This restriction is because c_1 and c_2 may be inconsistent assumptions: perhaps $c_1{:}\, \mathsf{Int} \sim \mathsf{Bool}$. If we were to introduce these into the context, induction would fail.

The consequence of these restrictions is that there are some types that cannot be shown equivalent, even though they are intuitively equivalent. For example, there is no proof of equivalence between the types $\forall\, c_1{:}\, \mathsf{Int} \sim b.\, \mathsf{Int}$ and $\forall\, c_2{:}\, \mathsf{Int} \sim b.\, b$—a coercion between these two types would need to use c_1 or c_2. However, this lack of expressiveness is not significant—in source Haskell, it could only be observed through exotic uses of first-class polymorphism, which are already rare in general. Furthermore, this restriction already exists in GHC[7] and other dependently-typed languages such as Agda and Coq. It is possible that a different consistency proof would validate a rule that does not restrict the use of these variables. However, we leave this possibility to future work.

7. Discussion and related work

Collapsing kinds and types Blurring the distinction between types and kinds is convenient, but is it wise? It is well known that type systems that include the $\Gamma \vdash_{\mathsf{ty}} \star : \star$ rule are inconsistent logics (Girard 1972). Does that cause trouble? For FC the answer is no—inconsistency here means that all kinds are inhabited, but even without our extensions, all kinds are already inhabited.

The $\Gamma \vdash_{\mathsf{ty}} \star : \star$ rule often causes type checking to be undecidable in dependently typed languages (Cardelli 1986; Augustsson 1998). This axiom permits the expression of divergent terms—if the type checker tries to reduce them it will loop. However, type checking in FC is decidable—all type equalities are witnessed by finite equality proofs, not potentially infinite reductions.

[7] Currently, coercions between the types $(\mathsf{Int} \sim b) \Rightarrow \mathsf{Int}$ and $(\mathsf{Int} \sim b) \Rightarrow b$ are disallowed

At the source language level, which does reduce type expressions, it is not clear whether adding the $\Gamma \vdash_{\mathsf{ty}} \star : \star$ rule could cause type inference to loop (in the absence of language extensions such as `UndecidableInstances` which already make divergence possible). However, even though this version of FC combines types and kinds, the Haskell source language need not do so (predictable type inference algorithms may require more traditional stratification). This gap would not be new—differing requirements for the core and surface languages have already led FC to be more expressive than source Haskell.

Heterogeneous equality Heterogeneous equality is an essential part of this system. It is primarily motivated by the presence of dependent application (such as rules K_INST and K_CAPP), where the kind of the result depends on the value of the argument. We would like type equivalence to be congruent with respect to application, as is demonstrated by rule CT_APP. However, if all equalities are required to be homogeneous, then not all uses of the rule are valid because the result kinds may differ.

For example, consider the datatype $\mathsf{TyRep} : \forall\, k{:}\, \star\, .\, \forall\, b{:}\, k.\, \star$. If we have coercions $\Gamma \vdash_{\mathsf{co}} \gamma_1 : \star \sim \kappa$ and $\Gamma \vdash_{\mathsf{co}} \gamma_2 : \mathsf{Int} \sim \tau$ (with $\Gamma \vdash_{\mathsf{ty}} \tau : \kappa$), then we can construct the proof

$$\Gamma \vdash_{\mathsf{co}} \langle \mathsf{TyRep} \rangle\, \gamma_1\, \gamma_2 \;:\; \mathsf{TyRep}\, \star\, \mathsf{Int} \sim \mathsf{TyRep}\, \kappa\, \tau$$

However, this proof requires heterogeneity because the first part ($\langle \mathsf{TyRep} \rangle\, \gamma_1$) creates an equality between types of different kinds: $\mathsf{TyRep}\, \star$ and $\mathsf{TyRep}\, \kappa$. The first has kind $\star \to \star$, whereas the second has kind $\kappa \to \star$.

The coherence rule (CT_COH) also requires that equality be heterogeneous because it equates types that almost certainly have different kinds. This rule, inspired by Observational Type Theory (Altenkirch et al. 2007), provides a simple way of ensuring that proofs do not interfere with equality. Without it, we would need coercions analogous to the many "push" rules of the operational semantics.

There are several choices in the semantics of heterogeneous equality. We have chosen the most popular, where a proposition $\sigma_1 \sim \sigma_2$ is interpreted as a conjunction: "the types are equal and their kinds are equal". This semantics is similar to Epigram 1 (McBride 2002), the `HeterogeneousEquality` module in the Agda standard library,[8] and the treatment in Coq.[9] Epigram 2 (Altenkirch et al. 2007) uses an alternative semantics, interpreted as "if the kinds are equal then the types are equal". (This relation requires a proof of kind equality before coercing types.) Guru (Stump et al. 2008) and Trellys (Kimmell et al. 2012; Sjöberg et al. 2012) use yet another interpretation which says nothing about the kinds. These differences reflect the design of the type systems—the syntax-directed type system of FC makes the conjunctive interpretation the most reasonable, whereas the bidirectional type system of Epigram 2 makes the implicational version more convenient. As Guru/Trellys demonstrate, it is also reasonable to not require kind equality. We conjecture that without the $\mathbf{kind}\,\gamma$ coercion form, it would be sound to drop the fourth condition from $\mathbf{Good}\,\Gamma$.

Unlike higher-dimensional type theory (Licata and Harper 2012), equality in this language has no computational content. Because of the separation between objects and proofs, FC is resolutely one-dimensional—we do not define what it means for proofs to be equivalent. Instead, we ensure that in any context the identity of equality proofs is unimportant.

The implicit language Our proof technique for consistency, based on erasing explicit type conversions, is inspired by ICC (Miquel 2001). Coercion proofs are irrelevant to the definition of type equality, so to reason about type equality it is convenient to ignore them entirely. Following ICC* (Barras and Bernardo 2008), we could also view the implicit language as the "real" semantics for FC, and consider the language of this paper as an adaptation of that semantics with annotations to make typing decidable. Furthermore, the implicit language is interesting in its own right as it is closer to source Haskell, which also makes implicit use of type equalities.

However, although the implicit language allows type equality assumptions to be used implicitly, it is not extensional type theory (ETT) (Martin-Löf 1984): it separates proofs from programs so that it can weaken the former (ensuring consistency) while enriching the latter (with "type-in-type"). As a result, the proof language of FC is not as expressive as ETT; besides the limitations on equalities between coercion abstractions in Section 6, FC lacks η-equivalence or extensional reasoning for type-level functions.

Explicit equality proofs In concurrent related work, van Doorn, Geuvers and Wiedijk (Geuvers and Wiedijk 2004; van Doorn et al. 2013) develop a variant of pure type systems that replaces implicit conversions with explicit convertibility proofs. There are strong connections to this paper: they too use heterogeneous equality and must significantly generalize the statement of a lifting lemma (which they call "equality of substitutions"). However, there are differences. Their work is based on Pure Type Systems, which generalize over sorts, rules and axioms; we only consider a single instance here. They also show that the system with explicit equalities is equivalent to the system with implicit equalities; we only show one direction. Finally, as their work is based on intensional type theory, it does not address coercion abstraction. Consequently, their analogue to rule CT_ALLT is the following asymmetric rule.

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{co}} \eta : \kappa_1 \sim \kappa_2 \\ \Gamma,\, a_1{:}\,\kappa_1 \vdash_{\mathsf{co}} \gamma : \tau_1 \sim \tau_2[a_1 \triangleright \eta / a_2] \\ \Gamma \vdash_{\mathsf{ty}} \forall\, a_1{:}\,\kappa_1.\, \tau_1 : \star \\ \Gamma \vdash_{\mathsf{ty}} \forall\, a_2{:}\,\kappa_2.\, \tau_2 : \star\end{array}}{\Gamma \vdash_{\mathsf{co}} \eta\; a_1{:}\,\kappa_1.\, \gamma : (\forall\, a_1{:}\,\kappa_1.\, \tau_1) \sim (\forall\, a_1{:}\,\kappa_2.\, \tau_2)}\;\text{CT\_ALLTA}$$

We conjecture that in our system, the above rule is equivalent to CT_ALLT.

8. Conclusions and future work

This work provides the basis for the practical extension of a popular programming language implementation. It does so without sacrificing any important metatheoretic properties. This extension is a necessary step towards making Haskell more dependently typed.

The next step in this research plan is to lift these extensions to the source language, incorporating these features within GHC's constraint solving algorithm. In particular, we plan future language extensions in support of type- and kind-level programming, such as datakinds (datatypes that exist only at the kind-level), kind synonyms and kind families. Although GHC already infers kinds, we will need to extend this mechanism to generate kind coercions and take advantage of these new features.

Going further, we would like to also like to support a true "dependent type" in Haskell, which would allow types to mention expressions directly, instead of requiring singleton encodings. One way to extend Haskell in this way is through elaboration: we believe that the translation between source Haskell and FC could automatically insert the appropriate singleton arguments (Eisenberg and Weirich 2012), perhaps using the class system to determine where they are necessary. This approach would not require further extension to FC. Alternatively, Adam Gundry's forthcoming dissertation[10] includes Π-types in a version of System FC that is

[8] `http://wiki.portal.chalmers.se/agda/agda.php?n=Libraries.StandardLibrary`

[9] `http://coq.inria.fr/stdlib/Coq.Logic.JMeq.html`

[10] Personal communication

strongly influenced by an early draft of this work. If elaboration does not prove to be sufficiently expressive, Gundry's work provides a blueprint for future core language extension.

In either case the interaction between dependent types and type inference brings new research challenges. However, the results in this paper mean that these challenges can be addressed in the context of a firm semantic basis.

Acknowledgments

Thanks to Simon Peyton Jones, Dimitrios Vytiniotis, Iavor Diatchki, José Pedro Magalhães, Adam Gundry and Conor McBride for discussion. This material is based upon work supported by the National Science Foundation under Grant Nos. 0910500 and 1116620.

References

T. Altenkirch and C. McBride. Generic programming within dependently typed programming. In J. Gibbons and J. Jeuring, editors, *Generic Programming*, volume 243 of *IFIP Conference Proceedings*, pages 1–20. Kluwer, 2002.

T. Altenkirch, C. McBride, and W. Swierstra. Observational equality, now! In *Proceedings of the 2007 workshop on Programming Languages meets Program Verification*, PLPV '07, pages 57–68, New York, 2007. ACM.

L. Augustsson. Cayenne—a language with dependent types. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, pages 239–250, New York, 1998. ACM. .

H. P. Barendregt. Handbook of logic in computer science (vol. 2). chapter Lambda calculi with types, pages 117–309. Oxford University Press, New York, 1992.

B. Barras and B. Bernardo. The implicit calculus of constructions as a programming language with dependent types. In *Foundations of Software Science and Computational Structures*, volume 4962 of *Lecture Notes in Computer Science*, pages 365–379. Springer, Berlin, Heidelberg, 2008.

L. Cardelli. A polymorphic lambda calculus with type:type. Technical Report 10, Digital Equipment Corporation Systems Research Center, 1986.

M. M. T. Chakravarty, G. Keller, and S. Peyon Jones. Associated type synonyms. In *Proceedings of the tenth ACM SIGPLAN International Conference on Functional Programming*, ICFP '05, pages 241–253, New York, 2005. ACM.

K. Crary, S. Weirich, and G. Morrisett. Intensional polymorphism in type erasure semantics. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, pages 301–313, Baltimore, MD, USA, 1998. ACM.

R. A. Eisenberg and S. Weirich. Dependently typed programming with singletons. In *Proceedings of the 2012 Haskell Symposium*, Haskell '12, pages 117–130, New York, 2012. ACM.

H. Geuvers and F. Wiedijk. A logical framework with explicit conversions. In C. Schuermann, editor, *LFM'04, Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages, Cork, Ireland*, pages 32–45, 2004.

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

M. P. Jones. Type classes with functional dependencies. In *9th European Symposium on Programming, ESOP '00*, volume 1782 of *Lecture Notes in Computer Science*, pages 230–244. Springer, 2000.

G. Kimmell, A. Stump, H. D. Eades III, P. Fu, T. Sheard, S. Weirich, C. Casinghino, V. Sjöberg, N. Collins, and K. Y. Ahn. Equational reasoning about programs with general recursion and call-by-value semantics. In *Sixth ACM SIGPLAN Workshop Programming Languages meets Program Verification (PLPV '12)*, 2012.

D. R. Licata and R. Harper. Canonicity for 2-dimensional type theory. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, pages 337–348, New York, NY, USA, 2012. ACM.

Z. Luo. *Computation and reasoning: a type theory for computer science*. Oxford University Press, Inc., New York, NY, USA, 1994.

J. P. Magalhães. The right kind of generic programming. In *8th ACM SIGPLAN Workshop on Generic Programming, WGP 2012, Copenhagen, Denmark*, New York, NY, USA, 2012. ACM.

P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

C. McBride. Elimination with a motive. In *Selected papers from the International Workshop on Types for Proofs and Programs*, TYPES '00, pages 197–216, London, UK, UK, 2002. Springer-Verlag.

C. T. McBride. Agda-curious?: an exploration of programming with dependent types. In P. Thiemann and R. B. Findler, editors, *ICFP*, pages 1–2. ACM, 2012. ISBN 978-1-4503-1054-3.

A. Miquel. The implicit calculus of constructions: extending pure type systems with an intersection type binder and subtyping. In *Proceedings of the 5th international conference on Typed lambda calculi and applications*, TLCA'01, pages 344–359, Berlin, Heidelberg, 2001. Springer-Verlag.

U. Norell. Functional generic programming and type theory, 2002. MSc thesis.

N. Oury and W. Swierstra. The power of Pi. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*, ICFP '08, pages 39–50, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-919-7.

S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *Proceedings of the eleventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '06, pages 50–61, New York, NY, USA, 2006. ACM.

T. Schrijvers, S. Peyton Jones, M. Sulzmann, and D. Vytiniotis. Complete and decidable type inference for GADTs. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 341–352, New York, NY, USA, 2009. ACM.

V. Sjöberg, C. Casinghino, K. Y. Ahn, N. Collins, H. D. Eades III, P. Fu, G. Kimmell, T. Sheard, A. Stump, and S. Weirich. Irrelevance, heterogenous equality, and call-by-value dependent type systems. In *Fourth workshop on Mathematically Structured Functional Programming (MSFP '12)*, 2012.

A. Stump, M. Deters, A. Petcher, T. Schiller, and T. Simpson. Verified programming in Guru. In *Proceedings of the 3rd workshop on Programming languages meets program verification*, PLPV '09, pages 49–58, New York, NY, USA, 2008. ACM.

M. Sulzmann, M. M. T. Chakravarty, S. P. Jones, and K. Donnelly. System F with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, TLDI '07, pages 53–66, New York, NY, USA, 2007. ACM.

F. van Doorn, H. Geuvers, and F. Wiedijk. Explicit convertibility proofs in pure type systems. Draft paper, in submission, `http://www.cs.ru.nl/~herman/PUBS/ExplicitPTS.pdf`, 2013.

D. Vytiniotis, S. Peyton Jones, and J. P. Magalhães. Equality proofs and deferred type errors: A compiler pearl. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming, ICFP 2012, Copenhagen, Denmark*, New York, NY, USA, 2012. ACM.

S. Weirich, D. Vytiniotis, S. Peyton Jones, and S. Zdancewic. Generative type abstraction and type-level computation (extended version). Technical report, University of Pennsylvania, Nov. 2010.

S. Weirich, D. Vytiniotis, S. Peyton Jones, and S. Zdancewic. Generative type abstraction and type-level computation. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 227–240, New York, NY, USA, 2011. ACM.

Z. Yang. Encoding types in ML-like languages. In *Proceedings of the Third International Conference on Functional Programming (ICFP)*, pages 289–300, Baltimore, Maryland, USA, 1998. ACM Press.

B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving Haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN workshop on Types in Language Design and Implementation*, TLDI '12, pages 53–66, New York, NY, USA, 2012. ACM.