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; Vytniots 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 branch1 of GHC to demonstrate that our modifications are compatible with the existing system, and do not invalidate existing Haskell
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.

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 indexed representations of types, and shows how they may be used.

We detail these comparisons in Section 7.

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.

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
zero TyBool = False
```

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 *. To represent type constructors with kind *, 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?

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 α 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 κ and a type of that kind (not named in the kind annotation). The data constructors constrain κ to a concrete kind. For the example to be well-formed, TyInt must constrain the kind parameter to *. Similarly, TyMaybe requires the kind parameter to be * → *. 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 κ, the type variables b :: κ → * and c :: κ, and the type equality a ~ b c. The TyMaybe pattern adds the kind equality κ ~ * 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.

3Note 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.

\[
\begin{align*}
\text{zero} &:: \forall (a :: Ty). \text{TyRep} a \rightarrow I a \\
\text{zero TyInt} &= 0 \\
\text{zero TyBool} &= \text{False}
\end{align*}
\]

Pattern matching \(\text{TyInt}\) refines \(a\) to \(\text{TyInt}\), which then uses the type family definition to show that the result type is equal to \(\text{Int}\).

Dependently typed languages do not require an argument like \(\text{TyRep}\) to implement operations such as \(\text{zero}\)—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 \(\text{TyRep}\) argument is an example of a \textit{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 \(\ast\). What if we wanted to encode \(T\text{Maybe}\) with \(Ty\)?

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

\[
\begin{align*}
data\text{ Kind} &= \text{Star} \mid \text{Arr Kind Kind} \\
\text{Kind} \text{ is a normal datatype that, when promoted, can be used to index the Ty datatype, making it a (standard) GADT.}
\end{align*}
\]

\[
\begin{align*}
data\text{ Ty} ::& \text{ Kind} \rightarrow * \text{ where} \\
\text{TInt} &= \text{Ty Star} \\
\text{TBool} &= \text{Ty Star} \\
\text{TMaybe} &= \text{Ty (Arr Star Star)} \\
\text{TApp} &= \text{Ty (Arr k1 k2)} \rightarrow \text{Ty k1} \rightarrow \text{Ty k2}
\end{align*}
\]

This indexing means that \(Ty\) can only represent well-formed types. For example \(T\text{Maybe}\) has type \(Ty\ (\text{Arr Star Star})\) and \(\text{TApp}\) \(\text{TMaybe}\) \(\text{TBool}\) has type \(Ty\ Star\), while the value \(\text{TApp}\) \(\text{TInt}\) would be rejected. Although this \(GADT\) can be expressed in GHC, the corresponding \(\text{TyRep}\) type requires two new extensions: \textit{promoted GADTs} and \textit{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 \(\text{TyRep}\) or to be interpreted by the type family \(I\), as shown below.

\[
\begin{align*}
data\text{ TyRep } (k :: \text{Kind}) (t :: Ty k) \text{ where} \\
\text{TyInt} &= \text{TyRep Star TInt} \\
\text{TyBool} &= \text{TyRep Star TBool} \\
\text{TMaybe} &= \text{TyRep (Arr Star Star)} \text{TMaybe} \\
\text{TApp} &= \text{TyRep (Arr k1 k2) a} \rightarrow \text{TyRep k1 b} \\
&\rightarrow \text{TyRep k2 (TApp a b)}
\end{align*}
\]

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 \textit{kind family}—a function that produces a kind by pattern matching a type or kind argument. For example, we can interpret values of the \textit{Kind} datatype as Haskell kinds like so:

\[
\begin{align*}
\text{kind family } IK (k :: \text{Kind}) &= I k \\
\text{kind instance } IK \text{ Star} &= * \\
\text{kind instance } IK (\text{Arr k1 k2}) &= I k 1 \rightarrow IK k2
\end{align*}
\]

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

\[
\begin{align*}
type\text{ family } I (t :: Ty k) &= IK k \\
\text{type instance } I \text{ TInt} &= \text{Int} \\
\text{type instance } I \text{ TBool} &= \text{Bool} \\
\text{type instance } I \text{ TMaybe} &= \text{Maybe} \\
\text{type instance } I (\text{TApp a b}) &= (I a) (I b)
\end{align*}
\]

However, once \(I\) has been defined, \(Ty\) and \(\text{TyRep}\) can be used in type-indexed operations as before.

\[
\begin{align*}
\text{zero} &:: \forall (a :: Ty Star). \text{TyRep Star a} \rightarrow I a \\
\text{zero TyInt} &= 0 \\
\text{zero TyBool} &= \text{False} \\
\text{zero (TApp TMaybe _)} &= \text{Nothing}
\end{align*}
\]

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.

\section{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 \(FC_3\) (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, \(FC_3\).

Along with the usual kinds \((\kappa)\), types \((\tau)\) and expressions \((e)\), FC contains coercions \((\gamma)\) that are proofs of type equality. The judgement

\[\Gamma \triangleright_\kappa \gamma : \tau_1 \sim \tau_2\]

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

Making type conversion explicit ensures that the FC typing relation \(\Gamma \triangleright_\kappa e : \tau\) 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 \texttt{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\(_\omega\) (Girard 1972)) as well as explicit proofs of type equality.

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

\[
\begin{align*}
type\text{ family } F (a :: *) &= \text{Int} \\
\text{type instance } F \text{ Bool} &= \text{Int}
\end{align*}
\]

generates the following FC axiom declaration:

\[
\text{axF} : F\text{Bool} \sim \text{Int}
\]

When given a source language function of type

\[
g :: \forall a. \ a \rightarrow F\ a \rightarrow \text{Char}
\]
the expression \( g \) is equal to 3 translates to the FC expression
\[
g \text{ True 3} \text{ translates to } (3 \triangleright \text{sym axF})
\]
that instantiates \( g \) at type Bool and coerces 3 to have type F Bool.
The coercion \( \text{sym axF} \) is a proof that \( \text{Int} \sim \text{F Bool} \).

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

\[
data T :: * \rightarrow * \text{ where} 
TInt :: T \text{ Int}
\]

This declaration could have also been written as a normal datatype where the type parameter is constrained to be equal to \( \text{Int} \).

\[
data T a = (a \sim \text{Int}) \Rightarrow T \text{ Int}
\]

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 \( \text{Int} \) is known to be equal to \( a \).

\[
f :: T a \rightarrow a 
f \text{TInt } = 3
\]

In the translation to FC, the \( \text{TInt} \) data constructor takes this equality constraint as an explicit argument.

\[
\text{TInt} : \forall a \cdot \bullet \cdot (a \sim \text{Int}) \Rightarrow T \text{ a}
\]

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

\[
f = \Lambda a : \bullet \cdot \lambda x: T \ a \cdot \text{ case } x \text{ of} 
\text{TInt } (\langle c : a \sim \text{Int} \rangle ) \rightarrow (3 \triangleright \text{sym } c)
\]

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 \( \tau_2 \sim \tau_3 \), then we can form a proof of the equality \( \tau_1 \sim \tau_3 \). 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 \( \beta \)-reduction can occur.

\[
\Gamma \vdash_\tau \gamma : \sigma_1 \rightarrow \sigma_2 \sim \tau_1 \rightarrow \tau_2 
(v \triangleright \gamma) e \longrightarrow (v (e \triangleright \text{sym (nth}^1 \gamma))) \triangleright \text{nth}^2 \gamma 
\text{S-PUSH}
\]

In this rule, if the expression \( (v \triangleright \gamma) e \) is well typed, then \( \gamma \) must be a proof of the equality \( \sigma_1 \rightarrow \sigma_2 \sim \tau_1 \rightarrow \tau_2 \). The coercions \( \text{sym (nth}^1 \gamma) \) and \( \text{nth}^2 \gamma \) 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 \( \tau \) has kind \( \kappa_1 \), and \( \gamma \) is a proof that kind \( \kappa_1 \) equals kind \( \kappa_2 \), then \( \tau \triangleright \gamma \) is type \( \tau \) cast to kind \( \kappa_2 \). There are several challenges to this extension, which we address with the following technical solutions.
• 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. Although there is no syntactic distinction between types and kinds, we informally use the word type (metavariables \( \tau \) and \( \sigma \)) for those members that classify runtime expressions, and kind (metavariable \( k \)) 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 \(*:*\) 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 \(*:*\) 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.

• 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 \( \tau_1 \) and \( \tau_2 \) could have kinds \( k_1 \) and \( k_2 \) that have no syntactic relation to each other. A proof of \( \gamma \) for \( \tau_1 \sim \tau_2 \) implies not only that \( \tau_1 \) and \( \tau_2 \) are equal, but also that their kinds are equal. The new coercion form \( \text{kind} \gamma \) extracts the proof of \( k_1 \sim k_2 \) from \( \gamma \).

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 \( \forall c : \phi, \tau, \text{names} \) the abstracted proof with the variable \( c \) and allows the type \( \tau \) 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.

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 \(*\) into universe levels (Luo 1994), as is done in Coq and Agda.
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 \( K_{\text{STARINSTAR}}, K_{\text{CAST}} \) and \( K_{\text{CAPP}} \) and \( K_{\text{ALLC}} \) check the new type forms. The first says that \( \ast \) has kind \( \ast \).

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 \( \tau \triangleright \gamma \) of kind casts: when given a type \( \tau \) of kind \( \kappa_1 \) and a proof \( \gamma \) that kind \( \kappa_1 \) equals kind \( \kappa_2 \), the cast produces a kind of type \( \kappa_2 \). Because equality is heterogeneous, the \( K_{\text{CAST}} \) rule requires a third premise to ensure that the new kind has the correct classification, so that inhabited types have kind \( \ast \).

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 \( \text{TInt} \) from Section 3 requires a type argument \( \tau \) and a proof that \( \tau \sim \text{Int} \). Note that there is no type-level abstraction over coercion—the form \( \tau \gamma \) can only appear when the head of \( \tau \) is a promoted datatype constructor.

### 4.3 Coercions

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

- \( K_{\text{VAR}} \)
- \( K_{\text{ARROW}} \)
- \( K_{\text{ALLT}} \)
- \( K_{\text{APP}} \)
- \( K_{\text{INST}} \)
- \( K_{\text{STARINSTAR}} \)
- \( K_{\text{CAPP}} \)
- \( K_{\text{ALLC}} \)
- \( K_{\text{CAST}} \)

Figure 2. Kind and type formation rules

<table>
<thead>
<tr>
<th>[ \Gamma \vdash \phi \circ ] ok</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \Gamma \vdash \sigma_1 : \kappa_1 ]</td>
</tr>
<tr>
<td>[ \Gamma \vdash \sigma_2 : \kappa_2 ]</td>
</tr>
<tr>
<td>[ \Gamma \vdash \sigma_1 \sim \sigma_2 \circ ] ok</td>
</tr>
</tbody>
</table>

Figure 3. Proposition formation rule

- \( CT_{\text{REFL}} \)
- \( CT_{\text{SYM}} \)
- \( CT_{\text{TRANS}} \)
- \( CT_{\text{APPL}} \)
- \( CT_{\text{CAPP}} \)

Figure 4. Coercion formation rules
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 \( \tau \to \gamma \), 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, \( \forall \Delta. \phi \) gathers multiple parameters in a telescope, a context denoted with \( \Delta \) 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 \( \overline{p} \), a mixed list of types and coercions. When type checking an axiom application, we must type check its list of arguments \( \overline{p} \) against the given telescope. The judgement form \( \Gamma \vdash \overline{p} \models \Delta \) (presented in the extended version of this paper) checks each argument \( \rho \) 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 \( \gamma \) is a proof of the equality \( \tau_1 \sim \tau_2 \), then kind \( \gamma \) extracts a proof of equality between the kinds of \( \tau_1 \) and \( \tau_2 \).

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:

\[
\Gamma \vdash \kappa : \cdot \quad \Gamma, a: \kappa \vdash_0 \gamma : \tau_1 \sim \tau_2 \\
\Gamma_0 \vdash a: \kappa, \gamma : (\forall a:\kappa.\tau_1) \sim (\forall a:\kappa.\tau_2) \quad \text{CT_ALLTX}
\]

This rule sufficed because the only quantified types that could be shown equal had the same syntactic kinds \( \kappa \) 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 \( \kappa_1 \) is not syntactically equal to \( \kappa_2 \).

Without this generality, the language does not satisfy the preservation theorem, which requires that the equality relation be substitutive—given a valid type \( \sigma \) where \( a \) appears free, and a proof \( \Gamma \vdash_0 \sigma \gamma : \tau_1 \sim \tau_2 \), 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 \( \forall b: a.\gamma \), then we must be able to derive \( \forall b: \tau_1, \tau_2: \tau \sim \forall b: \tau_1, \tau_2: \tau \).

Rule CT_ALLT shows when two polytypes are equal. The first premise requires a proof \( \eta \) 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, \( \alpha_1 \) and \( \alpha_2 \), one of each kind. The second premise of the rule adds both bindings \( \alpha_1: \kappa_1 \) and \( \alpha_2: \kappa_2 \) to the context as well as an assertion \( \varepsilon \) that \( \alpha_1 \) and \( \alpha_2 \) 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 abstraction, \( \forall c: \phi.\tau \). 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 corresponding equal types, as witnessed by proofs \( \eta_1 \) and \( \eta_2 \). 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 \( \eta_1 \): the kind of \( c_1 \) is built from types in both \( \eta_1 \) and \( \eta_2 \).

The rule CT_ALLC also restricts how the variables \( c_1 \) and \( c_2 \) can be used in \( c \). The premises \( c_1 \not\# |\gamma| \) and \( c_2 \not\# |\gamma| \) prevent these variables from appearing in the relevant parts of \( \gamma \). (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 erase operation \( |= \) 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 \( \gamma_2 \) and \( \gamma_2 \) other than well-formedness. Another example is rule CT_INSTC—again, no relation is required between the coercions \( \gamma_1 \) and \( \gamma_2 \).

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.

\[
\Gamma_0 \vdash \gamma : \tau_1 \sim \tau_2 \\
\Gamma_0 \vdash \eta_1 : \kappa_1 \\
\Gamma_0 \vdash \eta_2 : \kappa_2 \\
\Gamma_0 \vdash (\text{sym } \gamma) \sim (\text{sym } \eta_2) \sim \eta_1 \\
\Gamma_0 \vdash (\text{sym } \gamma) \sim \eta_2 \sim \eta_1 \\
\text{(Again, note that there is no relation required between } \eta_1 \text{ and } \eta_2.)
\]

We use the syntactic sugar \( \gamma \sim \eta_1 \sim \eta_2 \) to abbreviate the coercion \( \text{sym } (\text{sym } \gamma) \sim \eta_2 \) \sim \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
The S_KPUSH rule

\[ K; \forall \overline{\alpha}; \forall \Delta, \overline{\sigma} \rightarrow (T \overline{\pi}) \in \Gamma \]
\[ \Psi = \text{extend}(\text{context}(\gamma); \overline{\pi}; \Delta) \]
\[ \overline{\rho} = \Psi_2(\Delta) \]
\[ \text{case } (K \overline{\pi} \overline{\rho} \overline{\sigma}) \text{ of } \overline{\rho} \rightarrow \overline{\tau} \rightarrow \overline{\nu} \]
\[ \text{case } (K \overline{\pi} \overline{\rho} \overline{\sigma}) \text{ of } \overline{\rho} \rightarrow \overline{\nu} \]

Figure 6. The S_KPUSH rule

The "push" rules and the preservation theorem

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 \( K \overline{\pi} \overline{\rho} \overline{\sigma} \rightarrow \gamma \), the coercion must be distributed over all of the arguments of the data constructor, producing a new scrutinee \( K \overline{\pi} \overline{\rho} \overline{\sigma} \rightarrow \gamma \) as shown in Figure 6. In the rest of this section, we explain the rule by describing the formation of the lifting context \( \Psi \) and its use in the definition of \( \overline{\rho} \), \( \overline{\pi} \) and \( \overline{\sigma} \).

The S_KPUSH rule uses a lifting operation \( \Psi(\cdot) \) on expressions which coheres the type of its argument (\( \overline{\pi} \) in Figure 6). For example, suppose we have a data constructor \( K \) of type \( \forall a: \ast . F a \rightarrow T a \) for some type function \( F \) and some type constructor \( T \). Consider what happens when a case expression scrutinee \( (K \text{ Int } e) \rightarrow \gamma \), where \( \gamma \) is a coercion of type \( T \text{ Int } \sim T \tau' \). The push rule should convert this expression to \( K \tau' (e \triangleright \gamma) \) for some coercion \( \gamma' \) showing \( F \text{ Int } \sim F \tau' \). To produce \( \gamma' \), we need to lift the type \( F a \) to a coercion along the coercion \( \text{nth}^1 \gamma \), which shows \( \text{Int } \sim \tau' \).

In previous work, lifting was written \( \sigma[a \rightarrow \gamma] \), 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 \( \Psi \), which maps type variables to triples \( (\tau_1, \tau_2, \gamma) \) and coercion variables to pairs \( (\eta_1, \eta_2) \). The forms \( \tau_1 \) and \( \tau_2 \) refer to the original, uncoerced parameters to the data constructor (Int in our example). The forms \( \eta_1 \) and \( \eta_2 \) refer to the new, coerced parameters to the data constructor (like \( \tau' \) in our example). Finally, the coercion \( \gamma \) witnesses the equality of \( \tau_1 \) and \( \tau_2 \). No witness is needed for the equality between \( \eta_1 \) and \( \eta_2 \)—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: \( \forall a: \kappa, \tau \text{ and } \forall c: \phi, \tau \). Lifting over these types introduces new mappings in the lifting context, marked with \( \uparrow \).

\[
\Psi ::= \emptyset \mid \Psi, a: \kappa \rightarrow (\tau_1, \tau_2, \gamma) \mid (\Psi, c: \phi \rightarrow (\gamma_1, \gamma_2))
\]

We use the notation \( \uparrow \) to refer to a mapping created either with \( \rightarrow \) or with \( \uparrow \). A lifting context \( \Psi \) 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, types, and even other lifting contexts.

1. For each \( a: \kappa \rightarrow (\tau_1, \tau_2, \gamma) \) in \( \Psi \), \( \Psi_1(\cdot) \) maps \( a \) to \( \tau_1 \) and \( \Psi_2(\cdot) \) maps \( a \) to \( \tau_2 \).
2. For each \( c: \phi \rightarrow (\gamma_1, \gamma_2) \) in \( \Psi \), \( \Psi_1(\cdot) \) maps \( c \) to \( \gamma_1 \) and \( \Psi_2(\cdot) \) maps \( c \) to \( \gamma_2 \).

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:

**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 substitutions introduced in Section 4.3.2.)

\[ \Psi(\tau) = \emptyset \]
\[ \Psi(\kappa) = a: \kappa \rightarrow (\tau_1, \tau_2, \gamma) \]
\[ \Psi(\kappa) = (\Psi, c: \phi \rightarrow (\gamma_1, \gamma_2)) \]

\[ \Psi(c: \phi : \tau) = (\Psi, c: \phi \rightarrow (\gamma_1, \gamma_2)) \]

\[ \Psi(\kappa) = (\Psi, \kappa : \tau_1, \tau_2, \gamma) \]

\[ \Psi(\kappa) = (\Psi, \kappa : \tau_1, \tau_2, \gamma) \]

\[ \Psi(\tau) = (\Psi, \tau : \gamma_1, \gamma_2) \]
\[\Psi(a) = \gamma \text{ when } a: \kappa \to (\tau_1, \tau_2, \gamma) \in \Psi\]
\[\Psi(\tau) = \langle \tau \rangle \text{ when } \tau \notin \text{ dom}(\Psi)\]
\[\Psi(\pi_1, \pi_2) = \Psi(\pi_1, \pi_2)\]
\[\Psi(\tau \gamma) = \Psi(\tau)(\Psi(\gamma), \Psi(\delta(\gamma)))\]
\[\Psi(\forall a: \kappa, \tau) = \forall \Psi(c_{\pi_1}, (a_1, a_2, c), \Psi)(\tau)\]
\[\Psi(\forall c: \sigma_1 \sim \sigma_2, \tau) = \forall \Psi(c, \sigma_1 \sim \sigma_2, \Psi)(\tau)\]
\[\Psi(\tau \triangleright \gamma) = \Psi(\tau) \triangleright \Psi(\gamma) \sim \Psi(\sigma_2)(\gamma)\]

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

**Lemma 5.3 (Lifting Lemma).** If \(\Psi\) is a valid lifting context for context \(\Gamma\) and the telescope \(\Delta_1, \Delta_1 \vdash \tau: \kappa\), then
\[\Gamma \vdash \Psi(\tau) : \forall \Psi(\tau) \sim \Psi(\tau)\]

**Lifting context generation** In the \(\Sigma\)_\text{K Push} rule, the actual context \(\Psi\) used for lifting is built in two stages. First, context(\(\gamma\)) defines a lifting context with coercions for the parameters to the datatype.

**Definition 5.4 (Lifting context generation).** If \(\Gamma \vdash \Psi(\tau) : \Gamma \vdash \tau: \kappa\) and \(\Psi: \forall \Psi(\tau) \sim \Psi(\tau)\), and the telescope \(\Delta_1, \Delta_1 \vdash \tau: \kappa\), then define context(\(\gamma\)) as
\[\text{context}(\gamma) = a_1: \kappa, \rightarrow (\sigma_1, \sigma_2, \text{nth}(\gamma))\]

Intuitively, (context(\(\gamma\))\(\langle i \rangle(\tau)\) replaces all parameters \(a\) in \(\tau\) with the corresponding type on the left of \(\sim\) in the type of \(\gamma\). Similarly, (context(\(\gamma\))\(\langle 2 \rangle(\tau)\) replaces \(a\) with the corresponding type on the right of \(\sim\).

Next, this initial lifting context is extended with coercions using the operation extend(\(\cdot\)), which adds mappings for the variables in \(\Delta_1\), the existential parameters to the data constructor \(K\). Due to the dependency, we define the operation recursively. The intuition still holds: (extend(\(\Psi, \tau: \kappa\))\(\langle i \rangle(\tau)\) replaces free variables in \(\tau\) with their corresponding “from” types, while (extend(\(\Psi, \tau: \kappa\))\(\langle 2 \rangle(\tau)\) replaces a variables with their corresponding “to” types.

**Definition 5.5 (Lifting context extension).** Define the operation of lifting context extension, written extend(\(\Psi, \tau: \kappa\)), as:
\[\text{extend}(\Psi, \tau: \kappa) = \Psi\]
\[\text{extend}(\Psi, \tau: \kappa, \Delta_1, a: \kappa) = \Psi, a: \kappa \mapsto (\tau, \tau \triangleright \Psi'(\kappa), \text{sym}(\langle \tau \rangle \triangleright \Psi'(\kappa)))\]
\[\text{extend}(\Psi, \tau: \kappa, \Delta_1, c: \sigma_1 \sim \sigma_2) = \Psi', c: \sigma_1 \sim \sigma_2 \leftrightarrow (\gamma, \text{sym}(\Psi(\sigma_1) \triangleright \gamma \triangleright \Psi'(\sigma_2)))\]

**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 \(\Gamma \vdash \tau: \tau\) and \(e \to e'\) then \(\Gamma \vdash \tau(\Psi(\tau),)\),

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 \(|\cdot|\) on expressions. This operation erases types, coercions, and equality propositions to trivial forms \(\_\gamma\) and \(\_\text{prop}\) 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:
\[|e| = |e| \text{ if } e|\] and \(\_\text{prop}\)

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 \(e \to e'\), then either \(|e| = |e'|\) or \(|e| \to |e'|\).

**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 \(\text{closed, consistent}\) 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 \(\Sigma\) to denote closed contexts.

**Theorem 6.1 (Progress).** Assume \(\Sigma\) is a closed, consistent context. If \(\Gamma \vdash \tau : \tau_1\) and \(\tau\) is not a value or a coerced value \(\nu \triangleright \gamma\), then there exists an \(e_2\) such that \(e_1 \to e_2\).

The definition of consistent contexts is stated using the notions of uncoerced values and their types, \(\text{value types}\). Formally, we define values \(v\) and value types \(\xi\), with the following grammars:
\[v ::= \lambda x. \sigma. e \mid \Delta a: \kappa, e | K \tau \triangleright \tau\]
\[\xi ::= \sigma_1 \to \sigma_2 | \forall \sigma: \kappa, \xi | \forall c: \phi, \sigma | \tau\]

**Definition 6.2 (Consistency).** A context \(\Gamma\) is consistent if \(\xi_1\) and \(\xi_2\) have the same head form whenever \(\Gamma \vdash \xi_1 \sim \xi_2\).

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 a 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 \textbf{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 \(\vdash\)) analogous to the explicit language but for a few key
\[
\begin{align*}
\Gamma &\Rightarrow \tau \leadsto \tau' \\
\Gamma &\Rightarrow \tau \leadsto \tau' & \text{TS_REFL} \\
\Gamma, \Gamma' \Rightarrow \kappa \leadsto \kappa' &\Rightarrow c: a_1 \leadsto a_2, \Gamma \Rightarrow \sigma \leadsto \sigma' & \text{TS_ALLT} \\
\Gamma, \Gamma' \Rightarrow \forall a: \kappa, \sigma \leadsto \forall a: \kappa', \sigma' &\Rightarrow \Gamma \Rightarrow \tau \leadsto \tau' & \text{TS_ALLC} \\
\Gamma \Rightarrow \tau_1 \leadsto \tau_1' &\Rightarrow \forall c: \tau_1 \leadsto \tau_2, \sigma \leadsto \forall c: \tau_1 \leadsto \tau_2, \sigma' & \text{TS_RED} \\
\Gamma \Rightarrow \tau_1 \leadsto \tau_2 &\Rightarrow \tau \Rightarrow \sigma \leadsto \tau' \leadsto \sigma' & \text{TS_APP} \\
\Gamma \Rightarrow \tau \leadsto \tau' &\Rightarrow \tau \leadsto \tau' & \text{TS_CAPP}
\end{align*}
\]

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 \(\tau\) or coercion \(\gamma\), we define its erasure, denoted \(|\tau|\) or \(|\gamma|\), by induction on its structure. The interesting cases follow:

\[
\begin{align*}
|\tau| &\Rightarrow |\tau| & |\gamma| &\Rightarrow |\gamma| & |\tau \circ \gamma| &\Rightarrow |\tau| \bullet |\gamma| & |\gamma \circ \tau| &\Rightarrow |\gamma| \bullet |\tau| \\
|\gamma| &\Rightarrow |\gamma| \bullet \ast & |\gamma| &\Rightarrow |\gamma|
\end{align*}
\]

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 \(\Gamma\), denoted \(|\Gamma|\), by erasing the types and equivalence 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 \(\sigma_1\) is joinable with \(\sigma_2\), written \(\Gamma \Rightarrow \sigma_1 \Leftrightarrow \sigma_2\), 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 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 Good \(\Gamma\) when the following conditions hold:

1. All coercion assumptions and axioms in \(\Gamma\) are of the form \(C: \forall \Delta, (F \leadsto \tau')\) or of the form \(c: a_1 \leadsto a_2\). In the first form, the arguments to the type function must behave like patterns: for all \(\overline{p}\), every \(\tau_i \in \tau\) and every \(\tau_i'\) such that \(\Gamma \Rightarrow \tau_i[\overline{p}/\Delta] \leadsto \tau_i'\), there exists \(\overline{p}'\) such that \(\tau_i' = \tau_i[\overline{p}'/\Delta]\) and \(\Gamma \Rightarrow \sigma_m \leadsto \sigma_m'\) for each \(\sigma_m \in \overline{p}\) and \(\sigma_m' \in \overline{p}'\).

2. Axioms and coercion assumptions don’t overlap. For each \(F \tau\), there exists at most one prefix \(\tau'\) of \(\tau\) such that there exist \(C\) and \(\overline{p}\) where \(C: \forall \Delta, F \overline{\sigma} \leadsto \sigma_1 \in \Gamma\) and \(\overline{\pi} = \overline{\sigma_0[\overline{p}/\Delta]}\). These \(C\) and \(\overline{p}\) are unique for every matching \(\overline{\pi}\).

3. For each \(a\), there is at most one assumption of the form \(c: a \leadsto a'\) or \(c: a' \leadsto a\) and \(a \neq a'\).

4. Axioms equate types of the same kind. For each \(C: \forall \Delta, (F \tau \leadsto \tau')\) in \(\Gamma\), the kinds of each side must equal: for some \(\kappa\), \(\Delta = \kappa\) and \(\Delta = \kappa\) and that kind must not mention bindings in the telescope, \(\Gamma \Rightarrow \kappa \Rightarrow \kappa\).

The main lemma required for consistency is the completeness of joinability. Here, we write \(\text{fev}(\gamma) \subseteq \text{dom} \Gamma\) to indicate that all coercion variables and axioms used in \(\gamma\) are in the domain of \(\Gamma\).

**Lemma 6.6** (Completeness). Suppose that \(\Gamma \Rightarrow \gamma \leadsto \sigma_1 \leadsto \sigma_2\), and \(\text{fev}(\gamma) \subseteq \text{dom} \Gamma\) for some subcontext \(\Gamma'\) satisfying Good \(\Gamma'\). Then \(\Gamma \Rightarrow \sigma_1 \Leftrightarrow \sigma_2\).

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 \(\gamma\) must satisfy the requirements of Good contexts. As a result, we need to restrict the coercion abstraction equality rule in both the explicit and explicit systems.

\[
\begin{align*}
\Gamma &\Rightarrow \eta_1 \leadsto \sigma_1 \leadsto \sigma_2 \leadsto \phi_1 = \sigma_1 \leadsto \sigma_2 \\
\Gamma &\Rightarrow \eta_2 \leadsto \sigma_2 \leadsto \phi_2 = \sigma_1 \leadsto \sigma_2 \\
\text{c_1} &\Leftrightarrow \gamma & \text{c_2} &\Leftrightarrow \gamma \\
\Gamma &\Rightarrow \forall \gamma(c_1: \eta_1, \eta_2) \leadsto \forall \gamma(c_2: \eta_1, \eta_2) \\
\Gamma &\Rightarrow \forall \gamma(c_1: \eta_1, \eta_2) \leadsto \forall \gamma(c_2: \eta_1, \eta_2)
\end{align*}
\]

In this rule, the variables \(c_1\) and \(c_2\) cannot be used in \(\gamma\) due to the premises \(c_1 \neq \gamma\) and \(c_2 \neq \gamma\). (The analogous rule in the explicit system includes the premises \(c_1 \neq \gamma\) and \(c_2 \neq \gamma\).) This restriction is because \(c_1\) and \(c_2\) may be inconsistent assumptions: perhaps \(c_1: \text{Int} \leadsto \text{Bool}\). 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 \(\forall c_1: \text{Int} \leadsto b. \text{Int}\) and \(\forall c_2: \text{Int} \leadsto b. \text{a}\) — 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 GHC7 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 Currently, coercions between the types \((\text{Int} \leadsto b) \Rightarrow \text{Int}\) and \((\text{Int} \leadsto b) \Rightarrow b\) are disallowed.

## 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 \(\Gamma \vdash \ast : \ast\) rule are inconsistent logics (Girard 1972). Does that cause trouble? For \(\text{FC}\) the answer is no— inconsistency here means that all kinds are inhabited, but even without our extensions, all kinds are already inhabited.

The \(\Gamma \vdash \ast : \ast\) 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 \(\text{FC}\) is decidable—all type equalities are witnessed by finite equality proofs, not potentially infinite reductions.
Heterogeneous equality  Heterogeneous equality is an essential part of this system. It is primarily motivated by the presence of dependent application (such as rules \( K_{\text{INST}} \) and \( K_{\text{CAPP}} \)), 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 \( CT_{\text{APP}} \). 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 \( \text{TyRep} := \forall \ k \ : \ * \cdot \forall b : k \cdot * \). If we have coercions \( \Gamma \vdash \alpha \gamma_1 : * \sim \kappa \) and \( \Gamma \vdash \alpha \gamma_2 : \text{Int} \sim \tau \) (with \( \Gamma \eta \gamma : \kappa \)), then we can construct the proof

\[
\Gamma \vdash \alpha (\text{TyRep}) \gamma_1 \gamma_2 : \text{TyRep} \star \text{Int} \sim \text{TyRep} \kappa \tau
\]

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

The coherence rule \( CT_{\text{C0H}} \) 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,\(^3\) and the treatment in Coq.\(^7\) 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 Epigram 2 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 kind \( \gamma \) coercion form, it would be sound to drop the fourth condition from 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\(^8\) (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 \( \eta \)-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 \( CT_{\text{ALLT}} \) is the following asymmetric rule.

\[
\begin{align*}
\Gamma \vdash_0 \eta : k_1 & \sim k_2 \\
\Gamma, a_1 : k_1 \vdash_0 \gamma_1 & : \tau_1 \sim \tau_2[a_1 \triangleright \eta/a_2] \\
\Gamma \vdash_0 \forall a_1 : k_1, \gamma_1 : \star & \\
\Gamma \vdash_0 \forall a_2 : k_2, \tau_2 : \star
\end{align*}
\]

\[\text{CT}_{\text{ALLT}}\]

We conjecture that in our system, the above rule is equivalent to \( CT_{\text{ALLT}} \).

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 “de-\(^9\)pendent 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 II-types in a version of System FC that is

---

\(^3\)http://wiki.portal.chalmers.se/agda/agda.php
\(^4\)http://coq.inria.fr/stdlib/Coq.Logic.JMeq.html
\(^5\)http://coq.inria.fr/stdlib/Coq.Logic.JMeq.html
\(^6\)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


H. P. Barendregt. Handbook of logic in computer science (vol. 2). chapter 1116620.


