System FC with Explicit Kind Equality (Extended Version)

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 branch1 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/.
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.

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:

```haskell
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.

```haskell
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 *. 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?

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.

```haskell
data Proxy (a :: : κ) = P
```

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

```haskell
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.

```haskell
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 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-kindled 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.

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

We can use this datatype to index the representation type.

```haskell
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).

```haskell
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.

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

---

2 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.

3 with GHC 7.6 and the language extensions PolyKinds, DataKinds, GADTs, ExplicitForAll and TypeFamilies.
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 *. 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-kindred 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
TMaybe :: TyRep (Arr Star Star) TMaybe
TApp :: 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

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

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 FC2 (Weirich et al. 2011), and F_C (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.

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

\[ Γ ⊢ \gamma : τ_1 \sim τ_2 \]

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 τ_1 \sim τ_2, and the expression e has type τ_1, then the expression e ▷ γ (pronounced "e casted by γ") has type τ_2.

Making type conversion explicit ensures that the FC typing relation 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_\omega (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

\[
\text{type family } F \ a :: *
\text{type instance } F \text{ Bool } = \text{Int}
\]

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 True 3 translates to the FC expression

\[
g \text{Bool True} (3 \triangleright \text{sym } axF)
\]
that instantiates \( g \) at type \( \text{Bool} \) and coerces \( 3 \) to have type \( \text{F Bool} \).

The coercion \( \text{sym} \ x \mathbf{F} \) 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:

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

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

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

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 \( a \).

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

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

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

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

\[
f = \lambda a : * . \lambda x : T \ a. \text{case } x \text{ of } \\
T\text{Int} ((c : a \sim \text{Int}) \rightarrow (3 \mathbf{♯} \text{sym} \, c))
\]

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 \( \tau_1' \sim \tau_2' \), 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 affect 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.

\[
\Gamma |- \gamma : \sigma_1 \rightarrow \sigma_2 \sim \tau_1 \sim \tau_2 \\
\frac{(v \triangleright \gamma) \ e \rightarrow (v (e \triangleright \text{sym} (\text{nth} \gamma) \rightarrow \text{nth} \gamma)) \text{S_PUSH}}{}
\]

In this rule, if the expression \( (v \triangleright \gamma) e \) is well typed, then \( \gamma \) must be a proof of the equality \( \sigma_1 \rightarrow \sigma_2 \sim \tau_1 \sim \tau_2 \). The coercions \( \text{sym} (\text{nth} \gamma) \) and \( \text{nth} \gamma \) 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 \( \tau \) casted 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).

\[
H ::= Type \ constants \\
\begin{array}{l}
\text{Arrow} \\
\text{Type/Kind} \\
\text{Type constructor} \\
\text{Promoted data constructor}
\end{array}
\]

\[
w ::= Type-level \ names \\
\begin{array}{l}
\text{Names} \\
\text{Polymorphic types} \\
\text{Coelection abstraction} \\
\text{Type/kind application} \\
\text{Casting} \\
\text{Coelection application}
\end{array}
\]

\[
\sigma, \tau, \kappa ::= Types \ and \ Kinds \\
\begin{array}{l}
\text{Types} \ and \ Kinds
\end{array}
\]

\[
\phi ::= \sigma \sim \tau
\]

\[
\gamma, \eta ::= Propositions \ (coercion \ kinds)
\]

\[
\rho ::= \tau \mid \gamma
\]

\[
e, u ::= Expressions \\
\begin{array}{l}
\text{Variables} \\
\text{Abstraction} \\
\text{Application} \\
\text{Type/kind abstraction} \\
\text{Type/kind application} \\
\text{Coelection abstraction} \\
\text{Coelection application} \\
\text{Casting} \\
\text{Data constructors} \\
\text{Case analysis} \\
\text{Absurdity}
\end{array}
\]

\[
p ::= K \Delta \tau \tau
\]

\[
\Delta ::= Telescopes \\
\begin{array}{l}
\emptyset \\
\text{Type \ variable \ binding} \\
\text{Coelection \ variable \ binding}
\end{array}
\]

\begin{tabular}{c}
Figure 1. Basic Grammar
\end{tabular}
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.\footnote{GHC already uses a shared datatype for types and kinds, so this merge brings the formalism closer to the actual implementation.} 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 (metavaraible $\kappa$) 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 $\vdash$ 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.\footnote{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.}

- **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 $\kappa_1$ and $\kappa_2$ that have no syntactic relation to each other. A proof $\gamma$ of $\tau_1 \sim \tau_2$ 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 $\kappa_1 \sim \kappa_2$ 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 $\forall \ c : \phi, \tau, \text{name}$ 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.

\[
\text{data } T :: : \forall \ k. \ k \rightarrow \star \ where \\
\text{K} :: : \forall \ (b :: \star). \ b \rightarrow T \ b
\]

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 $k$.

\[
K : : \forall k : \star, \ b. \ k \vdash \ c : (k \sim \star). \ (b \triangleright c) \rightarrow T \ k \ b
\]

### 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 $\kappa$ 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 $\Gamma$ is a list of assumptions for term variables ($x$), type variables/datatypes/data constructors ($w$), coercion variables ($c$), and coercion axioms ($C'$).

\[
\Gamma ::= \emptyset \mid \Gamma, x : \tau \mid \Gamma, w : \kappa \mid \Gamma, c : \phi \vdash \Gamma, C \vdash \forall \Delta, \phi
\]

The type system includes the following judgements:

- $\Gamma \vdash \Delta$ Context validity (Figure 5)
- $\Gamma \vdash \tau : \kappa$ Type/kind validity (Figure 2)
- $\Gamma \vdash \phi \text{ ok}$ Proposition validity (Figure 3)
- $\Gamma \vdash \kappa \vdash \tau : \sigma$ Expression typing (appendix)
- $\Gamma \vdash \gamma : \phi$ Coercion validity (Figure 4)
- $\Gamma \vdash \varphi \equiv \Delta$ Telescope arg. validity (appendix)

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 $K_{\text{VAR}}$ 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 $\text{Tnat}$, 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 $K_{\text{ARROW}}$ gives the expected kind for the arrow type constructor. We use the usual syntactic sugar for arrow types, writing $\tau_1 \rightarrow \tau_2$ for $(\rightarrow) \tau_1 \tau_2$. 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 $K_{\text{ALL}}$ 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 $\sigma_1 \rightarrow \sigma_2$ and polymorphic types $\forall \alpha : \kappa. \sigma$ 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
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 rule for $\phi$ appearing in Figure 3. These rules establish properties of the type equality relation:

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 rule for $\phi$ appearing in Figure 3. These rules establish properties of the type equality relation:
Equality is an equivalence relation, as seen in rules CT\_REPL, 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 \( \tau \vdash \gamma \), 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, \( C : \forall \alpha \phi \). \( \phi \) 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 \( \vec{\rho} \), a mixed list of types and coercions. When type checking an axiom application, we must type check its list of arguments \( \vec{\beta} \) against the given telescope. The judgement form \( \Gamma \vdash \alpha \vec{\rho} \in \Delta \) (presented in the appendix) checks each argument \( \rho \) in turn against the binding in the telescope, scopes 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:

\[
\frac{\Gamma \vdash \beta : \kappa \quad \Gamma, a : \kappa \vdash \gamma_1 : \tau_1 \sim \tau_2}{\Gamma \vdash \gamma_a : \forall a : \kappa. \gamma_1 : \tau_1 \sim \tau_2}
\]

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 \( \Gamma \vdash a : \kappa \vdash \gamma : \tau_1 \sim \tau_2 \), we must be able to derive a proof between \( \sigma[\gamma_1/a] \) and \( \sigma[\gamma_2/a] \). 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 \( \forall b : \tau_1 \sim \forall b : \tau_2 \).

Rule CT\_ALLTX 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, \( a_1 \) and \( a_2 \), one of each kind. The 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 \( \epsilon \) 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 abstraction, \( \forall \gamma \). \( \gamma \) is a kind coercion \( \tau \vdash \gamma \), 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 \( \eta_1 \) 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 \( \gamma \). The premises \( c_1 \not\equiv \gamma \) and \( c_2 \not\equiv \gamma \) 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 erasure operation \( \gamma \downarrow \) 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 \( \gamma_1 \) 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.

\[
\frac{\Gamma \vdash_0 \gamma : \tau_1 \sim \tau_2 \quad \Gamma, \beta \vdash_1 \eta_1 : \kappa_1 \quad \Gamma, \beta \vdash_2 \eta_2 : \kappa_2}{\Gamma \vdash_0 \text{sym}(\langle \langle \text{sym} \gamma \rangle \rangle \beta) \sim \tau_1 \sim \tau_2 \sim \tau_2 \sim \tau_2}
\]

(Again, note that there is no relation required between \( \eta_1 \) and \( \eta_2 \).)

We use the syntactic sugar \( \gamma \sim \eta \) to abbreviate the coercion \( \text{sym}(\langle \langle \text{sym} \gamma \rangle \rangle \beta) \sim \eta_1 \sim \eta_2 \). Likewise, coherence derives a proof term for decomposing equalities between coerced types.

\[
\frac{\Gamma \vdash_0 \gamma : \tau_1 \sim \tau_2 \sim \tau_2 \sim \tau_2}{\Gamma \vdash_0 \text{sym}(\langle \langle \gamma \rangle \rangle \beta) \sim \tau_1 \sim \tau_2}
\]

#### 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 \( \forall \pi. \kappa. \star \). We call the variables \( \pi \) 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

\[
\forall \pi. \forall \Delta. (\pi \rightarrow T \pi) \tag{5}
\]

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

The telescope \( \Delta \) 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 \( \text{TyInt} \) from Section 2 (a data constructor belonging to \( \text{TyRep} \)): \( \forall k; \star, t; k; \star \) includes two coercions in its telescope, one asserting that the kind parameter \( k \) is \( \star \), the second asserting that the type parameter \( t \) is \( \text{Int} \):

\[
\text{TyInt} : \forall k; \star, t; k; \forall c_1; k \sim \star, c_2; t \sim \text{Int}. \quad \text{TyRep} k t
\]

Likewise, the data constructor \( \text{TyApp} \) existentially binds \( k \), \( a \), \( b \), and \( c \)—one kind and two type variables followed by a coercion:

\[
\text{TyApp} : \forall k; \star, t; k; \forall k'; \star, a; k' \rightarrow k, b; k', c; t \sim a b.
\]

A datatype value is of the form \( K \tau \pi \pi \), where \( \pi \) denotes the parameters (which cannot include coercions), \( \pi \) instantiate the existential arguments, and \( \pi \) 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 \( S_{\text{KPUSH}} \) rule.

5.1 Pushing coercions through constructors

When pattern matching on a coerced datatype value of the form \( K \tau \pi \pi \pi \gamma \), the coercion must be distributed over all of the arguments of the data constructor, producing a new scrutinee \( K \tau' \pi' \pi' \gamma' \) 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 \( \tau', \pi' \) and \( \gamma' \).

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

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 \( \Psi \), 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 \( \eta_1 \) refer to the original, uncoerced parameters to the data constructor (\( \text{Int} \) in our example). The forms \( \tau_2 \) and \( \eta_2 \) refer to the new, coerced parameters to the data constructor (like \( \pi' \) 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: \( \forall a; \kappa, \tau \) and \( \forall c; \phi, \tau \). Lifting over these types introduces new mappings in the lifting context, marked with \( \hookrightarrow \),

\[
\Psi := \emptyset \quad | \quad \Psi, a; \kappa \mapsto (\tau_1, \tau_2, \gamma) \quad | \quad \Psi, c; \phi \mapsto (\gamma_1, \gamma_2)
\]

(We use the notation \( \hookrightarrow \) to refer to a mapping created either with \( \mapsto \) or with \( \mapsto \).) 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, telescopes, typing contexts, and even other lifting contexts.

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

Now, we can now state the judgement form \( \Gamma \vdash \Delta \hookrightarrow \Psi \), shown in Figure 7, which defines when a lifting context is valid and compatible with a given telescope.

The two substitution operations satisfy straightforward substitution lemmas, defined and proved in the appendix. The usual substitution lemmas, which substitute a single type or coercion, are a direct corollary of these lemmas.
Lifting context validity

\[
\Gamma \vdash \Delta \rightsquigarrow \Psi \\
\frac{\vdash \Gamma \quad \Gamma \ni \sigma \Delta \rightsquigarrow \Psi \quad \sigma \ni \# \Gamma, \Delta \quad a \ni \# \Gamma, \Delta}{\Gamma \ni a \sigma \Delta \rightsquigarrow \Psi} \\
\frac{\Gamma \ni \sigma \Delta \rightsquigarrow \Psi \quad \sigma \ni \# \Gamma, \Delta}{\Gamma \ni \sigma \Delta \rightsquigarrow \Psi} \\
\frac{\Gamma \ni \sigma \Delta \rightsquigarrow \Psi \quad \sigma \ni \# \Gamma, \Delta}{\Gamma \ni \sigma \Delta \rightsquigarrow \Psi}
\]

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.)

\[
\begin{align*}
\Psi(a) &= \gamma \text{ when } a : \kappa \ni (t_1, t_2, \gamma) \in \Psi \\
\Psi(\tau) &= \langle \tau \rangle \text{ when } \tau \# \text{ dom}(\Psi) \\
\Psi(\tau_1 \tau_2) &= \Psi(\tau_1)(\tau_2) \\
\Psi(\gamma) &= \Psi(\langle \gamma \rangle) = \Psi(\psi(\gamma), \Psi(\gamma)) \\
\Psi(\forall \alpha : \kappa. \tau) &= \forall \Psi(\psi(\alpha_1, \alpha_2, c)), \Psi(\tau) \\
& \quad \text{ where } \Psi' = \Psi, a : \kappa \ni (a_1, a_2, c) \\
& \quad \text{ and } a_1, a_2, c \text{ are fresh} \\
\Psi(\forall c : \sigma_1 \rightsquigarrow \sigma_2. \tau) &= \forall (\psi(\psi(\sigma_1), \sigma_2)), \Psi(\tau) \\
& \quad \text{ where } \Psi' = \Psi, c : \sigma_1 \rightsquigarrow \sigma_2 \ni (c_1, c_2) \\
& \quad \text{ and } c_1, c_2 \text{ are fresh} \\
\Psi(\tau \triangleright \gamma) &= \Psi(\tau) \triangleright \Psi(\langle \gamma \rangle) \rightsquigarrow \Psi(\gamma)
\end{align*}
\]

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 topology and \( \Delta, \Delta \ni \gamma : \kappa \), then

\[
\Gamma \ni \Psi(\tau) \rightsquigarrow \Psi(\tau)
\]

Lifting context generation In the \( \text{K} \text{PUSH} \) 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 \( \Gamma \vdash \gamma : T \rightarrow T \rightarrow T \), and \( T : \forall \pi \forall \pi, \pi \ni \Gamma, \) where the lists \( \pi, \pi, \pi \) are all of length \( n \), then define context(\( \gamma \)) as

\[
\text{context}(\gamma) = (a_1 : \kappa_1 \ni (a_1, a_2, c_1, c_2))^{(1 \leq i \leq n)}
\]

Intuitively, \((\text{context}(\gamma))_i(\tau)\) replaces all parameters \( a_i \) in \( \tau \) with the corresponding type on the left of \( \ni \) in the type of \( \gamma \). Similarly, \((\text{context}(\gamma))_i(\tau)\) replaces \( a_i \) with the corresponding type on the right of \( \ni \).

\[\text{This definition is not just for the proof—it is implemented in GHC as part of the optimizer to reduce case expressions.}\]
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 \( \xi \), with the following grammars:

\[
\begin{align*}
v & ::= \lambda \alpha : \sigma . e \mid \Delta : \alpha . \xi \mid \lambda : \alpha . \xi \mid \epsilon \mid K \prod \pi \tau \\
\xi & ::= \sigma_1 \to \sigma_2 \mid \forall \alpha : \sigma . \xi \mid \forall \epsilon : \sigma . \xi \mid T \pi \tau
\end{align*}
\]

The canonical forms lemma tells us that the shape of a value is determined by its type:

**Lemma 6.2 (Canonical Forms).** Say \( \Sigma \vdash_m v : \sigma \). Then \( \sigma \) is a value type. Furthermore,

1. If \( \sigma = \sigma_1 \to \sigma_2 \) then \( v = \lambda x : \sigma_1 . e \) or \( K \pi \pi \pi \).
2. If \( \sigma = \forall \alpha : \kappa . \sigma \) then \( v = \lambda \alpha : \kappa . e \) or \( K \pi \pi \pi \).
3. If \( \sigma = \forall \epsilon : \sigma \) then \( v = \lambda \epsilon : \tau_1 \sim \tau_2 . e \) or \( K \pi \pi \pi \).
4. If \( \sigma = T \pi \tau \) then \( v = K \pi \pi \pi \).

**Definition 6.3 (Consistency).** A context \( \Gamma \) is consistent if \( \xi_1 \) and \( \xi_2 \) have the same head form whenever \( \Gamma \vdash_{1o} \gamma : \xi_1 \sim \xi_2 \).

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 6.4) 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 8).
3. We specify a sufficient condition, which we write Good \( \Gamma \) (Definition 6.6), 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 \( \sim \)) analogous to the explicit language but for a few key differences where coercions are dropped from types.

First, the use of kind coercions are no longer explicitly marked in types.

\[
\Gamma \vdash \tau : \kappa \quad \Gamma \vdash \gamma : \kappa \sim \kappa' \quad \Gamma \vdash \kappa' : * \quad IT_{\text{CAST}}
\]

Note that this system is no longer syntax directed—a type may have several syntactically different kinds. This is not a problem, as we use this system as a proof device for progress only.

Second, the coercion in an application is erased to \( \bullet_{\text{co}} \).

\[
\Gamma \vdash \tau : \forall \epsilon : \phi . \kappa \quad \Gamma \vdash \gamma : \phi \quad IT_{\text{CAPP}}
\]

(In general, we use \( \bullet_{\text{co}} \) to represent an elided coercion proof.)

We define coercion proofs between erased types in a similar fashion. Most of the rules carry over from the explicitly typed system, but there are three major differences. First, the implicit language does not include a coherence rule as there are no explicit casts. In the explicit language, given a coercion proof \( \Gamma \vdash_{1o} \gamma : \tau \sim \gamma' \), the coherence rule was used to construct a proof \( \gamma \vdash \gamma' \) where the kind of the first type \( \tau \) is changed, by applying a cast \( \tau \vdash \gamma' \). However, we can accomplish this in the implicit language, by using IT\(_{\text{CAST}} \) to implicitly cast the kind of \( \tau \) using coercion \( \gamma' \).

Second, the coercion application congruence rule is modified in accordance with coercion erasure.

\[
\Gamma \vdash \gamma : \tau \sim \tau' \\
\Gamma \vdash \tau \bullet_{\text{co}} : \kappa \quad \Gamma \vdash \tau' \bullet_{\text{co}} : \kappa' \quad ICT_{\text{CAPP}}
\]

This rule says that if two erased coercion applications are well formed, then if the two erased coercion abstractions are equal, there is a proof that the two applications are equal.

The final difference is in the rule for coercion abstractions: The requirement that \( e_1 \) and \( e_2 \) not appear in the (erased) coercion proof \( \gamma \) is for purely technical reasons (see below).

To connect the explicit and implicit systems, we define an erasure operation:

**Definition 6.4 (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:

\[
| \tau \circ \gamma | = | \tau | \bullet_{\text{co}} : | \gamma | (\tau_1, \gamma_2) = | \gamma | (\bullet_{\text{co}}, \bullet_{\text{co}})
\]

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

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

**Lemma 6.5 (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.

1. If \( \Gamma \vdash_{1e} \Gamma \) then \( | \Gamma | \).
2. If \( \Gamma \vdash \tau : \kappa \) then \( | \Gamma | \vdash | \tau | : | \kappa | \).
3. If \( \Gamma \vdash \phi \) ok then \( | \Gamma | \vdash | \phi | \) ok.
4. If \( \Gamma \vdash_{1o} \gamma : \phi \) then \( | \Gamma | \vdash | \gamma | : | \phi | \).
5. If \( \Gamma \vdash_{1e} \pi \Rightarrow \Delta \) then \( | \Gamma | \vdash | \pi | : | \Delta | \).

We define a nondeterministic rewrite relation on open implicit types in Figure 8. We say that \( \sigma_1 \) is joinable with \( \sigma_2 \), written \( \Gamma \vdash \sigma_1 \Rightarrow \sigma_2 \), 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 6.6 (Good contexts).** We have Good \( \Gamma \) when the following conditions hold:

1. All coercion assumptions and axioms in \( \Gamma \) are of the form \( C : \forall \Delta . (\forall \pi \sim \pi') \) or of the form of the kind \( \alpha i_1 \sim \alpha_2 \). In the first form, the arguments to the type function must behave like patterns: for all \( \pi \), every \( \tau_i \in \pi \) and every \( \tau'_i \) such that \( \Gamma \vdash \tau_i \pi \pi \pi \sim \tau'_i \), there exists \( \pi' \) such that \( \tau'_i = \tau_i | \pi' \pi \pi \Delta \) and \( \Gamma \vdash \sigma_{\pi'} \sim \sigma'_{\pi'} \) for each \( \sigma_{\epsilon_{\pi}} \in \pi' \) and \( \sigma'_{\epsilon_{\pi}} \in \pi' \).
2. Axioms and coercion assumptions don’t overlap. For each \( F \), \( F \pi \pi \pi \sim \sigma_{\pi} \in \Gamma \) and \( F \pi \pi \pi \sim \sigma'_{\pi} \in \Gamma \). These \( C \) and \( \pi \) are unique for every matching \( F \pi \pi \pi \).
3. For each \( a \), there is at most one assumption of the form \( c : a \sim a' \) or \( c : a' \sim a \), and \( a \neq a' \).
The completeness proof requires that all coercion variables in the telescope, $\Gamma$, in $\Gamma'$, the kinds of each side must equal: for some $\kappa$, $\Gamma, \Delta \vdash \forall \kappa. \tau \sim \tau'$ and that kind must not mention bindings in the telescope, $\Gamma \vdash \kappa \equiv \ast$.  

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

**Lemma 6.7 (Completeness).** Suppose that $\Gamma \vdash \gamma : \sigma_1 \sim \sigma_2$, and $f_{\text{ev}}(\gamma) \subseteq \text{dom} \Gamma'$ for some subcontext $\Gamma'$ satisfying Good $\Gamma'$. Then $\Gamma \vdash \sigma_1 \sim \sigma_2$.

The proof of this theorem appears in the appendix. 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 implicit and explicit systems.

$$\Gamma \vdash \tau \sim \tau'$$

$$\Gamma, \Gamma' \vdash \kappa \sim \kappa' \quad \Gamma, c : a_1 \sim a_2, \Gamma' \vdash \sigma \sim \sigma' \quad \text{TS}_{\text{REFL}}$$

$$\Gamma, \Gamma' \vdash \forall c : a_1 \sim a_2, \sigma \sim \sigma' \quad \text{TS}_{\text{ALLT}}$$

$$\Gamma \vdash \tau_1 \sim \tau_2' \quad \Gamma, \Gamma' \vdash \tau_2 \sim \tau_2' \quad \Gamma \vdash \sigma \sim \sigma' \quad \text{TS}_{\text{ALLC}}$$

$$\Gamma \vdash \forall \sigma \vdash \tau_1 \sim \tau_2, \tau_2 \sim \tau_2' \quad \Gamma \vdash \sigma \sim \sigma' \quad \text{TS}_{\text{RED}}$$

$$\Gamma \vdash \forall \sigma \vdash \tau \sim \tau' \quad \Gamma \vdash \sigma \sim \sigma' \quad \text{TS}_{\text{VARRED}}$$

$$\Gamma \vdash \forall \sigma \vdash \tau \sim \tau' \quad \Gamma \vdash \sigma \sim \sigma' \quad \text{TS}_{\text{APP}}$$

$$\Gamma \vdash \forall \sigma \vdash \tau \sim \tau' \quad \Gamma \vdash \sigma \sim \tau' \quad \text{TS}_{\text{CAPP}}$$

**Figure 8. Rewrite relation**

4. Axioms equate types of the same kind. For each $\Gamma : \forall \Delta, (F \tau \sim \tau') \in \Gamma$, the kinds of each side must equal: for some $\kappa$, $\Gamma, \Delta \vdash \forall \kappa. \tau \sim \tau'$ and $\kappa$ and $\Gamma$ and $\Delta$ have kind $\kappa$ and that kind must not mention bindings in the telescope, $\Gamma \vdash \kappa \equiv \ast$.  

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 $\forall c : \text{Int} \sim \text{Int}$ and $\forall c_2 : \text{Int} \sim \text{Int}$—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 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 $\Gamma \vdash \gamma : \tau. \tau$ 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.

If we were to introduce these into the type checker, they would make 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.

At the source language level, which does reduce type expressions, it is not clear whether adding the $\Gamma \vdash \gamma : \tau. \tau$ 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 $\text{K}_{\text{INST}}$ and $\text{K}_{\text{CAPP}}$), where the kind of the result depends on the value of the argument. We would like such a rule to be congruent with respect to application, as is demonstrated by rule $\text{CT}_{\text{CAPP}}$. 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 $\text{TyRep} : \forall \kappa. \ast \sim \ast$. $\forall \kappa. \ast \sim \ast$. If we have coercions $\Gamma \vdash \gamma_1 : \ast \sim \kappa$ and $\Gamma \vdash \gamma_2 : \text{Int} \sim \tau$ (with $\Gamma \vdash \gamma_7 : \tau. \kappa$), then we can construct the proof $\Gamma \vdash (\text{TyRep}) \gamma_1 \gamma_2 : \text{TyRep} \ast \sim \text{TyRep} \kappa \tau$.

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

The coherence rule $\text{CT}_{\text{COH}}$ 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) currently supports a variant, where a proposition $\sigma_1 \sim \sigma_2$ is interpreted as a disjunction: “the types are equal or their kinds are equal”. This semantics is similar to Epigram 1 (McBride 2002), the HeterogeneousEquality module in the Agda standard library.

---

9 http://coq.inria.fr/stdlib/Cof.Logic.JMeq.html
tenkirch 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 kind γ coercion form, it would be sound to drop the fourth condition from Good 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 based on erasing explicit type conversions, is inspired by ICC

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_AllLT is the following asymmetric rule.

$$\begin{align*}
\Gamma \vdash_2 \eta : \kappa_1 &\rightsquivalent \kappa_2 \\
\Gamma, a_1 : \kappa_1 \vdash_2 \gamma : \tau_1 &\rightsquivalent \tau_2[a_1 \triangleright \eta \mid a_2] \\
\Gamma \vdash_2 \forall a_1 : \kappa_1, \tau_1 & : * \\
\Gamma \vdash_2 \forall a_2 : \kappa_2, \tau_2 & : * \\
\Gamma \vdash_2 \forall a_1 : \kappa_1, \gamma : \forall a_1 : \kappa_1, \tau_1 & \rightsquivalent \forall a_1 : \kappa_2, \tau_2 \quad \text{CT_AllLT} 
\end{align*}$$

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

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 datatypes (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 includes II-types in a version of System FC that is 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 Dimitraki, José Pedro Magalhães, Adam Gundry and Conor McBride. This material is based upon work supported by the National Science Foundation under Grant Nos. 0910500 and 1116620.

References


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.


A. Additional semantics
Below, we list a few rules and definitions not included in the main discussion.

A.1 Context well-formedness
These rules ensure that all assumptions in the context are well formed and unique. They additionally constrain the form of the kinds of datatypes and the types of data constructors.

\[ \Gamma \vdash \emptyset ~ \text{GWF\_EMPTY} \]
\[ \Gamma \vdash \phi ~ \kappa \rightarrow \alpha ~ \# ~ \Gamma ~ \text{GWF\_TYVAR} \]
\[ \Gamma \vdash \phi ~ \kappa ~ \# ~ \Gamma ~ \text{GWF\_TYFUN} \]
\[ \Gamma \vdash \phi ~ \forall \pi : \kappa \rightarrow T ~ \# ~ \Gamma \rightarrow \# \Gamma ~ \text{GWF\_DATA} \]
\[ \Gamma \vdash \phi ~ \# ~ \# \Gamma ~ \text{GWF\_VAR} \]
\[ \Gamma \vdash \phi ~ \# ~ \# \Gamma ~ \text{GWF\_CON} \]
\[ \Gamma \vdash \phi ~ \# ~ \# \Gamma ~ \text{GWF\_VAR} \]
\[ \Gamma \vdash \phi ~ \# ~ \# \Gamma ~ \text{GWF\_AX} \]
A.3 Expression typing and operational semantics

Expression typing

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\Gamma \vdash \text{var} : \tau)</td>
<td>(\text{T_VAR})</td>
</tr>
<tr>
<td>(\Gamma, x: \tau, \tau_1 \vdash \text{app} : \tau_2)</td>
<td>(\text{T_ABS})</td>
</tr>
<tr>
<td>(\Gamma \vdash \text{app} : \tau_2)</td>
<td>(\text{T_APP})</td>
</tr>
<tr>
<td>(\Gamma, \phi \vdash \text{app} : \tau)</td>
<td>(\text{T_CABS})</td>
</tr>
<tr>
<td>(\Gamma \vdash \text{app} : \tau, \tau_1 \vdash \text{app} : \tau_2)</td>
<td>(\text{T_TABS})</td>
</tr>
<tr>
<td>(\Gamma \vdash \text{var} : \tau)</td>
<td>(\text{T_CAST})</td>
</tr>
<tr>
<td>(\Gamma \vdash \text{fun} : \tau)</td>
<td>(\text{T_CON})</td>
</tr>
</tbody>
</table>

Case K is exhaustive for T for each i

\(K_i: \forall \alpha. \pi. \forall \Delta_i. \pi, T \gamma \rightarrow (T \pi) \in \Gamma\)

\(\Delta_i \rightarrow \Delta_i[\tau/\alpha]\)

\(\sigma_i \rightarrow \sigma_i[\tau/\alpha]\)

\(\Gamma, \Delta_i, \xi_i: \sigma_i \vdash \text{app} : \tau\)

\(\text{T_CASE}\)

Case e of K_i \(\Delta_i[\tau/\alpha] \rightarrow \text{app} : \tau\)

\(\Gamma \vdash \text{case} : \tau_i \rightarrow \tau_i \rightarrow \tau_i \rightarrow \tau_i\)

\(\Gamma \vdash \text{app} : \tau\)

Step reduction, parameterized by toplevel context

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>((\lambda x: \tau. \text{app}) \rightarrow e)</td>
<td>(\text{S_BETA})</td>
</tr>
<tr>
<td>(e_1 \rightarrow e'_1)</td>
<td>(\text{S_EAPP})</td>
</tr>
<tr>
<td>((\text{app} \circ \tau) \rightarrow (\text{app} \circ \sigma))</td>
<td>(\text{S_PUSH})</td>
</tr>
<tr>
<td>((\text{app} \circ \tau) \rightarrow (\text{app} \circ \sigma))</td>
<td>(\text{S_PUSH})</td>
</tr>
<tr>
<td>((\lambda x: \tau. \text{app}) \rightarrow e)</td>
<td>(\text{S_TBETA})</td>
</tr>
<tr>
<td>(e_1 \rightarrow e'_1)</td>
<td>(\text{S_TAPP})</td>
</tr>
<tr>
<td>(\Gamma \vdash \text{case} : \tau_i \rightarrow \tau_i \rightarrow \tau_i \rightarrow \tau_i)</td>
<td>(\text{S_CASE})</td>
</tr>
<tr>
<td>(\text{case} e \rightarrow e')</td>
<td>(\text{S_CASE})</td>
</tr>
<tr>
<td>(\text{case} e \rightarrow e')</td>
<td>(\text{S_PUSH})</td>
</tr>
<tr>
<td>(\text{case} e \rightarrow e')</td>
<td>(\text{S_PUSH})</td>
</tr>
<tr>
<td>(\text{case} e \rightarrow e')</td>
<td>(\text{S_PUSH})</td>
</tr>
</tbody>
</table>

Case \(K_1\) of \(p \rightarrow u\)

\(\text{case} (K_1 \circ \tau) \rightarrow (K_1 \circ \sigma)\) | \(\text{S_KPUSH}\)
A.4 Erasure operation

\[\Gamma \Rightarrow \Delta, (\pi \rightarrow T \pi) : \ast \ K \# \Gamma\]
\[\Rightarrow \Gamma, K : \forall a: \pi, \forall \Delta, (\pi \rightarrow T \pi)\]

\[\Gamma \Rightarrow \phi \ ok \ c \# \Gamma\]
\[\Rightarrow \Gamma, c : \phi\]

\[\Gamma \Rightarrow \Delta \Rightarrow \phi \ ok \ C \# \Gamma\]
\[\Rightarrow \Gamma, C : \forall \Delta, \phi\]

\[\Gamma \Rightarrow \phi \ ok\]

Implicit coercion kind well-formedness

\[\Gamma \Rightarrow \sigma_1 : \kappa_1\]
\[\Gamma \Rightarrow \sigma_2 : \kappa_2\]
\[\Gamma \Rightarrow \sigma_1 \sim \sigma_2\]

\[\Gamma \Rightarrow \tau : \kappa\]

Implict Kinding

\[\Gamma \Rightarrow \Gamma\]
\[\Gamma \Rightarrow \ast : \ast\]

IT_STAR_INST

\[\Gamma \Rightarrow \Gamma\]
\[\Gamma \Rightarrow (\rightarrow) : \ast \rightarrow \ast \rightarrow \ast\]

IT_ARROW

\[\Gamma \Rightarrow \Gamma\]
\[\Gamma \Rightarrow \Gamma \rightarrow \Gamma \rightarrow \Gamma \rightarrow \Gamma\]

IT_VAR

\[\Gamma \Rightarrow \sigma_1 : \kappa_1 \rightarrow \kappa_2\]
\[\Gamma \Rightarrow \sigma_2 : \kappa_2\]

IT_APP

\[\Gamma \Rightarrow \sigma_1 : \forall \kappa_1 : \sigma_1, \Gamma \Rightarrow \sigma_2 : \kappa_2\]
\[\Gamma \Rightarrow \sigma_2 : \forall \kappa_2 : \sigma_2\]

IT_INST

\[\Gamma \Rightarrow \tau : \forall \phi, \kappa. \phi \Rightarrow \gamma : \phi\]
\[\Gamma \Rightarrow \tau : \kappa\]

IT_CAPP

\[\Gamma, \phi, \kappa : \tau \Rightarrow \tau : \ast\]
\[\Gamma \Rightarrow \ast : \forall \phi, \kappa. \ast\]

IT_ALL_T

\[\Gamma, \phi, \kappa : \tau : \ast\]
\[\Gamma \Rightarrow \phi \ ok\]

IT_ALL_C

\[\Gamma \Rightarrow \tau : \kappa\]
\[\Gamma \Rightarrow \gamma : \kappa \sim \kappa'\]
\[\Gamma \Rightarrow \kappa' : \ast\]

IT_CAST

\[\Gamma \Rightarrow \gamma : \phi\]

Implicit Coercion Typing

\[\Gamma \Rightarrow \gamma : \tau \sim \tau'\]
\[\Gamma \Rightarrow \tau : \phi, \kappa \Rightarrow \tau' : \phi, \kappa'\]

ICT_APP

\[\Gamma \Rightarrow \ast : \phi, \kappa \Rightarrow \ast : \kappa'\]
\[\Gamma \Rightarrow \ast : \phi, \kappa \Rightarrow \ast : \kappa'\]

ICT_CAPP

\[\Gamma \Rightarrow \gamma : \ast : \phi, \kappa \Rightarrow \gamma : \ast : \kappa'\]
\[\Gamma \Rightarrow \gamma : \ast : \phi, \kappa \Rightarrow \gamma : \ast : \kappa'\]

ICT_INST

\[\Gamma \Rightarrow \gamma : \ast : \phi, \kappa \Rightarrow \gamma : \ast : \kappa'\]
\[\Gamma \Rightarrow \gamma : \ast : \phi, \kappa \Rightarrow \gamma : \ast : \kappa'\]

ICT_INSTC

A.5 Implicit Language Typing

\[\Gamma \Rightarrow \Gamma\]

Implicit Validity

\[\Gamma \Rightarrow \emptyset\]

IV_EMPTY

\[\Gamma \Rightarrow \ast \ k \ 
\[\Rightarrow \Gamma, \ k : \ast \ k\]

IV_TY_VAR

\[\Gamma \Rightarrow \forall \pi, x : \ast \ T \ # \Gamma\]
\[\Rightarrow \Gamma, T : \forall \pi, x : \ast \ T\ # \Gamma\]

IV_TY_FUN

\[\Gamma \Rightarrow \ast \ T \ # \Gamma\]
\[\Rightarrow \Gamma, x : \ast \ T \ # \Gamma\]

IV_TY_DATA

\[\Gamma \Rightarrow \tau : \ k \ x \ # \Gamma\]

IV_VAR

\[\Gamma \Rightarrow \forall \pi, x : \ast \ T \ # \Gamma\]
\[\Rightarrow \Gamma, x : \ast \ T \ # \Gamma\]

IV_TY_DATA

\[\Gamma \Rightarrow \ast \ T \ # \Gamma\]
\[\Rightarrow \Gamma, x : \ast \ T \ # \Gamma\]

IV_VAR
\[ \Gamma \vdash \tau : \kappa \quad \text{ICT_REFL} \]
\[ \Gamma \vdash (\gamma) : \tau \quad \text{ICT_SYM} \]
\[ \Gamma \vdash \gamma : \tau_1 \sim \tau_2 \quad \text{ICT_TRANS} \]
\[ \Gamma \vdash \text{sym} \gamma : \tau_2 \sim \tau_1 \quad \text{ICT_APP} \]
\[ \Gamma \vdash \eta : \kappa_1 \sim \kappa_2 \quad \text{ICT_VAR} \]
\[ \Gamma \vdash \text{V}_{\eta}(a_1, a_2, c) : \gamma_1 \sim (\text{V}_{\eta}(a_1; k_1, \tau_1)) \sim \text{V}_{\eta}(a_2; \kappa_2, \tau_2) \quad \text{ICT_ALLT} \]
\[ \Gamma \vdash \gamma : \tau_1 \sim \tau_2 \quad \text{ICT_NTH} \]
\[ \Gamma \vdash \text{nth} \gamma : \tau_1 \sim \tau_2 \quad \text{ICT_NTHITA} \]
\[ \Gamma \vdash \gamma : (\text{V}_{\eta}(c; k_1, \kappa_2, \tau)) \sim (\text{V}_{\eta}(c; k_1, \kappa_2, \tau)) \quad \text{ICT_NTH1CA} \]
\[ \Gamma \vdash \gamma : (\text{V}_{\eta}(c; k_1, \kappa_2, \tau)) \sim (\text{V}_{\eta}(c; k_1, \kappa_2, \tau)) \quad \text{ICT_NTH2CA} \]
\[ \Gamma \vdash \gamma : \kappa_1 \sim \kappa_2 \quad \text{ICT_NTH3CA} \]
\[ \Gamma \vdash \phi \circ \kappa : \tau = \gamma \quad \text{ICT_EXT} \]

**C. Preservation**

This section presents the necessary details for the proof of the preservation theorem. The theorem is itself proved by induction on the typing derivation with a case analysis of the rule used in the operational semantics. Below, we present only three cases, those for S.KPUSH, S.TPUSH, and S.CPUSH. These are the only cases that differ from the proof described in previous work (Weirich et al. 2010). We also present many supporting lemmas needed for these cases, particularly regarding the treatment of lifting contexts.

### C.1 Lifting contexts

A lifting context contains two types of mappings: those labeled with \( \rightarrow \) that denote a mapping from type or coercion variables to existing types and coercions, and those labeled with \( \leftarrow \) that denote a mapping from type or coercion variables to fresh type and coercion variables.

A lifting context is considered valid with respect to a context \( \Gamma \) and telescope \( \Delta \) when the following judgement holds:

\[ \Gamma \vdash \Delta \sim \Psi \quad \text{Lifting context validity} \]

\[ \vdash_{\text{def}} \Gamma \quad \text{LC_EMPTY} \]

\[ \Gamma \vdash \Delta \sim \Psi \quad \text{LC_TY} \]

\[ \Gamma \vdash \Delta \sim \Psi \quad \text{LC_CO} \]

\[ \vdash_{\text{ty}} \Gamma \quad \text{LC_TYFRESH} \]

\[ \Gamma \vdash \Delta \sim \Psi \quad \text{LC_COFRESH} \]

We can view the fresh bindings of a lifting context as a typing context:

**Definition C.1** (Single flattening). The operation \( \Psi \) has a lifting context into a typing context.

1. For each \( a : \kappa \rightarrow \bullet (a_1, a_2, c) \in \Psi \), the context \( \Psi \) includes the binding \( a_1 : \Psi_j(\kappa) \).
2. For each \( c : \phi \rightarrow \bullet (c_1, c_2) \), the context includes the binding \( c_1 : \Psi_j(\phi) \).

**Definition C.2** (Flattening). The operation \( \Psi \) has a lifting context into a typing context.

1. For each \( a : \kappa \rightarrow \bullet (a_1, a_2, c) \in \Psi \), the context \( \Psi \) includes the bindings \( a_1 : \Psi_j(\kappa), a_2 : \Psi_j(\kappa), c_1 : \kappa \rightarrow a_2 \).
2. For each \( c : \phi \rightarrow \bullet (c_1, c_2) \), the context includes the bindings \( c_1 : \Psi_j(\phi), c_2 : \Psi_j(\phi) \).

**Lemma C.3** (Lifting context domains). If \( \Gamma \vdash \Delta \sim \Psi \), then the domain of \( \Delta \) equals the set of variables mapped in \( \Psi \).

**Proof.** Straightforward induction on the derivation of \( \Gamma \vdash \Delta \sim \Psi \).

**Lemma C.4** (Lifting context substitution). Suppose \( \Gamma \vdash \Delta \sim \Psi \).

**Proof.**
Proof. We proceed by mutual induction.

There are many cases to consider. We consider the interesting ones here:

Case K_VAR: We know $\Gamma, \Delta, \Gamma' \vdash_\omega w : \kappa$, and by inversion, $\vdash_\omega \Gamma, \Delta, \Gamma' \vdash w : \kappa$. Furthermore, because $\kappa$ appears in $\Gamma, \Delta, \Gamma'$ before any element in $\Delta$ is declared, we know that $\kappa$ cannot refer to any variable declared in $\Delta$. Therefore, $\Psi_j(\kappa) = \kappa$. By the induction hypothesis, $\vdash_\omega \Gamma, \Delta, \Gamma' \vdash \Psi_j(\Gamma')$, and we can use rule K_VAR to get $\Gamma, \Delta, \Gamma' \vdash w : \kappa$ as desired.

Case CT_ALLC: We know $\Gamma, \Delta, \Gamma' \vdash \forall \psi \phi \tau : \ast$, and by inversion, $\Gamma, \Delta, \Gamma' \vdash \psi \phi \tau : \ast$ and $\Gamma, \Delta, \Gamma' \vdash \psi \phi \tau : \ast$. The induction hypothesis gives us $\vdash_\omega \Gamma, \Delta, \Gamma' \vdash \psi \phi \tau : \ast$ and $\Gamma, \Delta, \Gamma' \vdash \psi \phi \tau : \ast$. Thus, by rule K_ALLC, $\Gamma, \Delta, \Gamma' \vdash \forall \psi \phi \tau : \ast$ and we are done.

Case CT_ALLC: We know

$\Gamma, \Delta, \Gamma' \vdash \forall (r_1, r_2)(c_1, c_2). \gamma : (\forall c_1 : \phi_1, \gamma_1 \sim (\forall c_2 : \phi_2, \gamma_2)$

and by inversion

$\Gamma, \Delta, \Gamma' \vdash_\omega \eta_1 : \sigma_1 \sim \sigma_1' \Gamma, \Delta, \Gamma' \vdash_\omega \eta_2 : \sigma_2 \sim \sigma_2' \phi_1 = \sigma_1 \sim \sigma_2 \phi_2 = \sigma_1 \sim \sigma_2 c_1 \# \gamma c_2 \# \gamma \Gamma, \Delta, \Gamma' \vdash c_1 : \phi_1, c_2 : \phi_2 \vdash_\omega \gamma : \tau_1 \sim \tau_2 \Gamma, \Delta, \Gamma' \vdash_\omega \forall c_1 : \phi_1. \gamma \sim \tau_1 \Gamma, \Delta, \Gamma' \vdash_\omega \forall c_2 : \phi_2. \gamma \sim \tau_2$

We wish to show

$\Gamma, \Delta, \Gamma' \vdash_\omega \forall (\psi_j(\gamma_1), \psi_j(\gamma_2))(c_1, c_2). \psi_j(\gamma) : (\forall c_1 : \psi_j(\phi_1), \psi_j(\gamma_1) \sim (\forall c_2 : \psi_j(\phi_2), \psi_j(\gamma_2))$.

To use rule CT_ALLC, we need to show

$\Gamma, \Delta, \Gamma' \vdash_\omega \forall (\psi_j(\gamma_1), \psi_j(\gamma_2))(c_1, c_2). \psi_j(\gamma) : (\forall c_1 : \psi_j(\phi_1), \psi_j(\gamma_1) \sim (\forall c_2 : \psi_j(\phi_2), \psi_j(\gamma_2))$.

We know (1), (2), (5), (6), and (7) by the induction hypothesis. We can derive (3) and (4) by noting that $\psi_j(\gamma)$ and $\psi_j(\phi)$ commute with each other and that $c_1, c_2$ do not appear in $\psi_j$. Therefore, if $c_1 \# \gamma$, then $c_1 \# \psi_j(\gamma)$ and likewise for $c_2$. Now, we can apply CT_ALLC and we are done.
We will need the following lemmas to prove the lifting lemma:

**Lemma C.5** (Lifting context coercions). If \( \Gamma \vdash_\kappa \Delta \rightsquigarrow \Psi \) and \( \Psi \) contains the mapping \( a: \kappa \rightarrow (\tau_1, \tau_2, \gamma) \), then \( \Gamma, \Psi \vdash_{\kappa} \gamma : \tau_1 \rightsquigarrow \tau_2 \).

**Proof.** Straightforward induction on \( \Gamma \vdash_\kappa \Delta \rightsquigarrow \Psi \).

**Lemma C.6** (Weakened lifting context substitution). Suppose \( \Gamma \vdash_\kappa \Delta \rightsquigarrow \Psi \).

1. If \( \Gamma, \Delta \vdash_{\kappa} \gamma : \kappa \) then \( \Gamma, \Psi \vdash_{\kappa} \Psi_j(\gamma) : \Psi_j(\kappa) \).
2. If \( \Gamma, \Delta \vdash_{\kappa} \gamma : \phi \) then \( \Gamma, \Psi \vdash_{\kappa} \Psi_j(\gamma) : \Psi_j(\phi) \).
3. If \( \Gamma, \Delta \vdash_{\kappa} \Delta' \rightsquigarrow \Psi' \) then \( \Gamma, \Psi \vdash_{\kappa} \Psi_j(\Delta') \rightsquigarrow \Psi_j(\Psi') \).
4. If \( \Gamma, \Delta \vdash_{\kappa} \Psi \) then \( \Gamma, \Psi \vdash_{\kappa} \Psi_j(\gamma) \Rightarrow \Psi_j(\Delta') \).

(Include a note about the case in which \( \gamma \) is a type variable, so \( \Psi \) and \( \Psi_j \) are guaranteed to be fresh bindings.)

**Proof.** Immediate from Lemma C.4 and weakening, noting that any difference between \( \Psi \) and \( \Psi_j \) is preserved).

**Lemma C.7** (Erased lifted coercions). Let \( \Psi \) contain the mapping \( c: \phi \rightarrow (c_1, c_2). \) If \( \Gamma, \Delta \vdash_{\kappa} \Psi \) and \( \Delta \vdash_{\kappa} \gamma : \kappa \), then \( c_1 \not\in \Psi(\gamma) \) and \( c_2 \not\in \Psi(\gamma) \).

**Proof.** We proceed by induction on the typing derivation for \( \gamma \).

**Cases K_StarInst and K_Arrow:** Trivial.

**Case K_Var:** \( \tau = \omega \), and we know \( \vdash_{\kappa} \Gamma, \Delta \) and \( \omega \not\in \Delta \).

Here we have two cases:

- If \( \omega \in \text{dom} \Delta \): We know \( \omega \) is a type variable, so \( \omega \not\in \Delta \). Thus, \( \omega \) appears either after or before \( c \) in \( \Psi \). If \( \omega \) appears after \( c \), then, by the fact that all mappings with \( \rightarrow \) precede all mappings with \( \rightarrow \) in \( \Psi \), \( \Psi(\omega) \) is some fresh variable \( \gamma \), and thus \( c_1 \not\in \Psi(\gamma) \) and \( c_2 \not\in \Psi(\gamma) \) as desired. Going forward, we can assume \( \omega \) occurs before \( c \). Now, the mapping from \( \omega \) may be built with \( \rightarrow \) or \( \rightarrow \). We have already handled the latter case, so going forward, we can assume that the mapping is built with \( \rightarrow \). \( \Psi(\omega) = \gamma \) for some \( \gamma \). However, this \( \gamma \) is chosen from components all of which are out of scope of \( c_1 \) and \( c_2 \). Thus, neither \( c_1 \) nor \( c_2 \) appear in \( \gamma \) and thus do not appear in \( \gamma \). Thus, \( c_1 \not\in \Psi(\gamma) \) and \( c_2 \not\in \Psi(\gamma) \) as desired.

- If \( \omega \not\in \text{dom} \Delta \): In this case \( \Psi(\omega) = \omega \). Because the spaces of type variables and coercion variables are distinct, we know that \( \omega \not\in \Delta \) and \( \omega \not\in \Delta \), as desired.

**Case K_App:** \( \tau = \tau_1 \tau_2 \), and we know \( \Delta \vdash_{\kappa} \tau_1 : \kappa_1 \rightarrow \kappa_2 \) and \( \Delta \vdash_{\kappa} \tau_2 : \kappa_1 \). Here, \( \Psi(\tau_1 \tau_2) = \Psi(\tau_1) \Psi(\tau_2) \).

The induction hypothesis tells us that \( \kappa_1, \kappa_2 \) do not appear in \( \Psi(\tau_1) \) or \( \Psi(\tau_2) \). Since \( \tau_1 \tau_2 = \tau_1 \) and \( \tau_2 \), the desired result follows directly from this result.

**Case K_Inst:** Analogous to K_App.

**Case K_Comp:** \( \tau = \tau_1 \gamma_1 \gamma_2 \), and we know \( \Gamma, \Delta \vdash_{\kappa} \tau_1 : \forall \phi \rightarrow (a_1, a_2, c) \) for (fresh) \( a_1, a_2, c \), we have \( \Psi(\forall \phi : a_1, a_2, c) = \Psi(\forall \phi : a_1, a_2, c) = \forall \phi \Psi(\forall \phi : a_1, a_2, c) \).

The induction hypothesis tells us that \( \kappa_1, \kappa_2 \) do not appear in \( \Psi(\tau_1) \). By the definition of \( \forall \phi \), \( \Psi(\tau_1) \gamma_1 \gamma_2 = \Psi(\tau_1) \gamma_1 \gamma_2 \). Thus, \( c_1, c_2 \) do not appear in \( \Psi(\tau_1) \gamma_1 \gamma_2 \) as desired.

**Case K_AllT:** \( \tau = \forall \phi : a_1, a_2, c \), and we know \( \Gamma, \Delta \vdash_{\kappa} \tau_1 : \forall \phi : a_1, a_2, c \) and \( \Gamma, \Delta \vdash_{\kappa} \tau_1 \Delta : \kappa \rightarrow \gamma_1 \gamma_2 \). Letting \( \Psi = \forall \phi : a_1, a_2, c \rightarrow (a_1, a_2, c) \) for (fresh) \( a_1, a_2, c \), we have \( \Psi(\forall \phi : a_1, a_2, c) = \forall \phi \Psi(\forall \phi : a_1, a_2, c) \).

The induction hypothesis tells us that \( \kappa_1, \kappa_2 \) do not appear in \( \Psi(\tau_1) \). By the definition of \( \forall \phi \), \( \Psi(\tau_1) \gamma_1 \gamma_2 = \Psi(\tau_1) \gamma_1 \gamma_2 \). Thus, \( c_1, c_2 \) do not appear in \( \Psi(\tau_1) \gamma_1 \gamma_2 \) as desired.
for some types \( \kappa \) and \( \kappa' \). Lemma C.6 applied to the types of \( \tau_1 \) and \( \gamma_1 \), along with straightforward typing rule applications, gives us exactly these facts. Thus,

\[
\Gamma, \psi \vdash_\alpha \psi(\tau_1 \gamma_1) \quad : \quad \psi_1(\gamma_1) \sim \psi_2(\tau_1) \psi_2(\gamma_1)
\]
or

\[
\Gamma, \psi \vdash_\alpha \psi(\tau_1 \gamma_1) \quad : \quad \psi_1(\gamma_1) \sim \psi_2(\tau_1 \gamma_1)
\]
as desired.

**Case K.ALLT**: \( \tau = \forall a : \kappa \cdot \tau' \) and we know \( \Gamma, \Delta \vdash a \kappa \vdash \tau' : * \) and \( \Gamma, \Delta \vdash \psi \kappa \vdash * \). We can use LC_TyFresh to derive \( \Gamma \vdash_{\kappa} \Delta, \psi \kappa \vdash (a_1, a_2, c) \) for fresh \( a_1, a_2, c \). We write \( \psi' \) for this extended lifting context.

We wish to show

\[
\Gamma, \psi \vdash_\alpha \psi(\tau_1 \gamma_1) \quad : \quad \psi_1(\forall a : \kappa \cdot \tau') \sim \psi_2(\forall a : \kappa \cdot \tau')
\]
or, equivalently,

\[
\Gamma, \psi \vdash_\alpha \forall \phi(\psi(\tau)): \quad \forall \phi : \psi_1(\psi(\tau')) \sim \forall \phi : \psi_2(\psi(\tau'))
\]

By the induction hypothesis, we have

\[
\Gamma, \psi \vdash_\alpha \psi(\psi(\tau')) \quad : \quad \psi_1(\psi(\tau')) \sim \psi_2(\psi(\tau'))
\]

We wish to use CT_ALLT. The first two premises are already satisfied, noting that \( \psi' \) contains the extra bindings for the second premise. We must show

\[
\Gamma, \psi \vdash_\alpha \forall \phi : \psi_1(\psi(\tau')) \quad : \quad *
\]

This fact comes directly from the use of Lemma C.6 applied to the type of \( \forall a : \kappa \cdot \tau' \).

Thus, we can apply CT_ALLT, and we are done.

**Case K.Cast**: \( \tau = \tau' \circ \eta \) and we know \( \Gamma, \Delta \vdash \psi(\tau') : \eta \) and \( \Gamma, \Delta \vdash \psi(\eta) : \kappa_1 \). We wish to show

\[
\Gamma, \psi \vdash_\alpha \psi(\tau' \circ \eta) \quad : \quad \psi_1(\tau' \circ \eta) \sim \psi_2(\tau' \circ \eta)
\]
or, equivalently,

\[
\Gamma, \psi \vdash_\alpha \psi(\tau') \quad : \quad \psi_1(\tau') \sim \psi_2(\tau')
\]

Using this fact with straightforward application of typing rules gives us the desired result.

**C.2 Metatheory for S,KPush Preservation**

Having proved the lifting lemma, we still must present and prove a number of other lemmas before proving that the types are preserved in the S,KPush case.

**Lemma C.8** (Telescope substitution). If \( \Gamma \vdash_{\kappa} \Delta \iff \psi \) and \( \Gamma_{\psi} \vdash \Delta \), then \( \psi_{\eta}(\Delta) = \Delta \).

**Proof.** By Lemma C.3, the domain of \( \psi \) equals the domain of \( \Delta \). Furthermore, \( \Gamma_{\psi} \vdash \Delta \) implies that all kinds in \( \Delta \) (constructs to the right of a colon) are well-scoped—that is, no variable is mentioned before it is declared. Because the \( \psi_{\eta}(\Delta) \) operation is defined only to substitute in kinds and to not substitute a variable after it is locally bound, it is impossible for the substitution to change \( \Delta \). Thus, \( \psi_{\eta}(\Delta) = \Delta \), as desired.

**Lemma C.9** (\( \psi_{\eta} \)-consistency). If \( \Gamma \vdash_{\kappa} \Delta \iff \psi \) then \( \Gamma_{\psi_{\eta}} \vdash (dom \Delta) \iff \Delta \).

**Proof.** We wish to use clause 5 of the lifting context substitution lemma (Lemma C.4), with \( \overline{a} = dom \Delta \) and \( \Delta' = \Delta \). We must show \( \Gamma_{\psi_{\eta}} \vdash dom \Delta \iff \Delta \). This is true by straightforward induction on the length of \( \Delta \). Then, we apply Lemma C.4 to get \( \Gamma_{\psi_{\eta}} \vdash (dom \Delta) \iff \psi_{\eta}(\Delta) \). By Lemma C.8, this can be rewritten as \( \Gamma_{\psi_{\eta}} \vdash (dom \Delta) \iff \Delta, \) as desired.

**Lemma C.10** (Lifting context extension consistency). If \( \Gamma \vdash_{\kappa} \Delta \iff \psi \), \( \overline{a} \subseteq \Gamma, \Delta_{1}, \Delta_{2}, \Gamma_{\psi_{\eta}} \vdash_{\psi_{\eta}} \psi_{\eta}(\Delta_{2}), \) and \( \psi' = extend(\psi; \Delta_{2}) \), then \( \Gamma_{\psi_{\eta}} \vdash (\Delta_{1} : \Delta_{2} \iff \psi'_{\eta}) \).

**Proof.** We proceed by induction on the derivation of \( \Gamma_{\psi_{\eta}} \vdash_{\psi_{\eta}} \psi_{\eta}(\Delta_{2}) \).

- **Case** \( \overline{a} = \emptyset ; \Delta_{2} = \emptyset \): In this case \( \psi' = \psi \), and thus we must show \( \Gamma_{\psi_{\eta}} \vdash \Delta_{1} \iff \psi \), which we know by assumption.

- **Case** \( \overline{a} = \emptyset , \Delta_{2} = \emptyset : \Delta_{2} : a \kappa : \psi' = extend(\psi; \Delta_{2}) \): The inductive hypothesis is: if \( \Gamma_{\psi_{\eta}} \vdash \Delta_{1} \iff \psi_{\eta}(\Delta_{2}) \), and \( \psi'' = extend(\psi; \Delta_{2}), \) then \( \Gamma_{\psi_{\eta}} \vdash \Delta_{1} \iff \psi''_{\eta} \). We must show \( \Gamma_{\psi_{\eta}} \vdash \Delta_{1} \iff \psi''_{\eta}, \) where \( \psi'' = extend(\psi; \Delta_{2} : \Delta_{2} : a \kappa : \psi' \) by the definition of extend, we know that we will have to use rule LC_Ty. It is easy to see from the definition of extend that \( \psi'' \) is \( \psi' \) with an additional mapping from \( a \). Thus, \( \Gamma_{\psi_{\eta}} \vdash \Delta_{1} \iff \psi'' \) fulfills the first premise of LC_Ty. To use LC_Ty, we must show the following:
1. Γ ⊢ τ : Ψ(κ)
   We know Γ ⊢ ρ/τ, τ ∈ Ψ(κ). Inverting gives us Γ ⊢ ρ/Ψ(κ) ⊢ τ. Because we care only about the names of the variables in the substitution expression, we can rewrite this as Γ ⊢ ρ/Ψ(κ) ⊢ τ. From the definition of extend, we can see that all of the substitutions performed by Ψ(κ) that are not in Ψ map a domain element of Δ₂ to its corresponding ρ ∈ P₂. Thus, we can rewrite the judgement above as Γ ⊢ γ : Ψ(κ) as desired.

2. Γ ⊢ τ ⊬ Ψ(κ) : Ψ(κ)
   We wish to use the lifting lemma (Lemma 5.3). We know Γ ⊢ Δ₁, Δ₂ ⊢ τ : Ψ(κ). We must show Δ₁, Δ₂ ⊢ τ : Ψ(κ) for some σ. This fact, for σ = *, comes directly from inversion on Γ ⊢ Δ₁, Δ₂ : a, κ. Now, we apply the lifting lemma to get Γ ⊢ Ψ(κ) : Ψ(κ). As shown in the previous case, Γ ⊢ τ : Ψ(κ). Therefore, by simple application of typing rules, we can derive Γ ⊢ τ : Ψ(κ) as desired.

3. Γ ⊢ sym(τ → Ψ(κ)) : τ ⊬ Ψ(κ)
   Straightforward application of typing rules.

• Case 1: Γ ⊢ ρ/τ, γ : Ψ(κ) ⊢ τ, γ : Ψ(κ)
   The inductive hypothesis is the same as in the previous case. We must prove Γ ⊢ Δ₁, Δ₂, κ : Ψ(κ) ⊢ τ, γ : Ψ(κ). By the definition of the extend operation, we know we will have to use rule LC. It is easy to see from the definition of extend that Ψ(κ) is Ψ(κ) with an additional mapping from c. Thus, Γ ⊢ Δ₁, Δ₂ ⊢ Ψ(κ) fulfills the first premise of LC. To use LC, we must show the following:

1. Γ ⊢ ρ/τ, γ : Ψ(κ) ⊢ τ, γ : Ψ(κ)
   We know Γ ⊢ ρ/τ, γ : Ψ(κ) ⊢ τ, γ : Ψ(κ). Inverting gives us Γ ⊢ ρ/τ, γ : Ψ(κ) ⊢ τ, γ : Ψ(κ). Because we care only about the names of the variables in the substitution expression, we can rewrite this as Γ ⊢ ρ/τ, γ : Ψ(κ) ⊢ τ, γ : Ψ(κ). From the definition of extend, we can see that all of the substitutions performed by Ψ(κ) that are not in Ψ map a domain element of Δ₂ to its corresponding ρ ∈ P₂. Thus, we can rewrite the judgement above as Γ ⊢ γ : Ψ(κ) as desired.

2. Γ ⊢ sym(Ψ(κ)(sn)) ⊢ γ ⊢ Ψ(κ)(sn) : Ψ(κ)(sn) ⊢ Ψ(κ)(sn)
   We wish to use the lifting lemma (Lemma 5.3) twice to get the types of Ψ(κ)(sn) and Ψ(κ)(sn). We know Γ ⊢ Δ₁, Δ₂ : Ψ(κ)(sn). We must show Γ ⊢ Δ₁, Δ₂ : Ψ(κ)(sn). Inversion on Γ ⊢ Δ₁, Δ₂ : a, κ gives us Γ ⊢ Δ₁, Δ₂ : a, κ, σ₁, σ₂, σ₃, σ₄. Furthermore, we get Γ ⊢ Δ₁, Δ₂ : a, κ, σ₁, σ₂, σ₃, σ₄, κ₁, κ₂, κ₃. Now, we apply the lifting lemma to get Γ ⊢ Ψ(κ)(sn) : Ψ(κ)(sn) ⊢ Ψ(κ)(sn). As shown in the previous case, Γ ⊢ τ : Ψ(κ)(sn). Therefore, by simple application of typing rules, we can derive Γ ⊢ τ : Ψ(κ)(sn) as desired.

2. case (K p p p) of p → u → case (K p p p) of p → u, then

Γ ⊢ case (K p p p) of p → u, σ

Proof Sketch. By induction on the length of p₂.

Lemma C.11 (Telescope composition). If Γ ⊢ p₁ ⊢ Δ₁ and Γ ⊢ p₂ ⊢ Δ₂, then Γ ⊢ p₂ ⊢ Δ₂[p₁/Δ₂].

Proof Sketch. By induction on the length of p₂.

Lemma C.12 (S_KPUSH preservation). If

1. Γ ⊢ p₁ ⊢ τ → γ, then Γ ⊢ p₁ ⊢ τ → γ, by reflexivity and coherence and Γ ⊢ p₁ ⊢ γ, by type application.

 Furthermore, we have

• Γ ⊢ p₁ ⊢ γ, by reflexivity and coherence and Γ ⊢ p₁ ⊢ γ, by type application.

Thus the final term has the desired type by casting.
Lemma C.14 (S_C Push Preservation). If

1. \( \Gamma \vdash \text{CP} \ (v \triangleright \gamma) \gamma' : \sigma \) and
2. \( (v \triangleright \gamma) \gamma' : \nu \gamma'' \triangleright \chi @ (\gamma'', \gamma') \), where
3. \( \gamma'' = \text{nth}^1 \nu \gamma \triangleright \gamma' ; \text{sym} (\text{nth}^2 \nu \gamma) \) and
4. \( \Gamma \vdash \text{CP} \ (\forall \nu c. \chi \triangleright \gamma' ; \lambda) \sim (\forall \nu c'. \chi ; \tau') \),
then \( \Gamma \vdash \text{CP} \nu v \gamma'' \triangleright \chi @ (\gamma'', \gamma') : \sigma \).

Proof. By inversion, we have

- \( \Gamma \vdash \text{CP} \nu v \gamma' : \forall v' : \chi. \rho' \)
- \( \Gamma \vdash \text{CP} \nu v : \forall \nu c. \chi. \tau \)
- \( \Gamma \vdash \text{CP} \nu \gamma' : \forall v' : \chi. \rho' \)
- \( \sigma = \tau'[\gamma'/\gamma''] \).

Let \( \phi = \sigma_1 \sim \sigma_2 \) and \( \phi' = \sigma_1' \sim \sigma_2' \). We can show

- \( \Gamma \vdash \text{CP} \text{nth}^1 \nu \gamma : \sigma_1 \sim \sigma_1' \), by CT_NT H1CA
- \( \Gamma \vdash \text{CP} \text{nth}^2 \nu \gamma : \sigma_2 \sim \sigma_2' \), by CT_NT H2CA
- \( \Gamma \vdash \text{CP} \text{sym} (\text{nth}^2 \nu \gamma) : \sigma_2' \sim \sigma_2 \), by symmetry
- \( \Gamma \vdash \text{CP} \nu \gamma'' : \forall \nu v' : \chi. \rho' \), by definition of transitivity
- \( \Gamma \vdash \text{CP} \nu \gamma' : \nu [\gamma'/\gamma''] \), by coercion instantiation
- \( \Gamma \vdash \text{CP} \nu \gamma^0 (\nu \gamma', \gamma'') : \nu [\gamma'/\gamma''] \sim [\nu \gamma'/\gamma''] \), by CT_INST C.

The final term has the desired type by casting.

\( \square \)

D. Type Erasure

We need the following small lemma before we can prove type erasure:

Lemma D.1 (Type erasure erases types). No type variable \( a \) or coercion variable \( c \) appears free in \(|e|\), for any expression \( e \).

Proof. Straightforward inspection of definition of \(|e|\).

\( \square \)

Corollary D.2 (Substitution in erased expressions). For all expressions \( e \), types \( \tau \), coercions \( \gamma \), and variables \( a \) and \( c \),

1. \(|e[\tau/a]| = |e|\)
2. \(|e[\gamma/c]| = |e|\)

We can now present the proof for the type erasure theorem: If \( e \rightarrow e' \), then either \(|e| = |e'|\) or \(|e| \rightarrow |e'|\).

Proof. We proceed by induction on \( e \rightarrow e' \).

Cases S_BETA, S_TBETA, S_CBETA, S_CASEMATCH: The erased expression steps by the same rule as the unerased expression, appealing to Corollary D.2 in the S_TBETA and S_CBETA cases.

Cases S_EAPP, S_TAPP, S_CAPP, S_CASE: We appeal to the induction hypothesis. If \(|e| = |e'|\), we are done. If \(|e| \rightarrow |e'|\), then we use the same stepping rule as the unerased expression used.

Cases S_PUSH, S_TPUSH, S_C_PUSH, S_COMB: Straightforward application of the definition of erasure yields \(|e| = |e'|\).

Case S_COERC: \( e \rightarrow e' \rightarrow e' \triangleright \gamma \rightarrow e' \triangleright \gamma' \)

By the definition of erasure, \(|e \triangleright \gamma| = |e|\) and \(|e' \triangleright \gamma'| = |e'|\). By the induction hypothesis, either \(|e| = |e'|\) or \(|e| \rightarrow |e'|\).

In either case, we are done.

Case S_KPUSH:

\[
\begin{align*}
K & : \forall \nu \tau. \forall \Delta. \sigma \rightarrow (T \pi) \in \Gamma \\
\Psi & : \text{extend(context}(\gamma); \nu \chi; \Delta) \\
\tau' & : \Psi (\nu \pi) \\
\phi & : \Psi \nu (\nu \chi) ; \Delta \\
& \text{for each } \nu e_i \in \tau, \\
& e_i' : e_i \triangleright \Psi (\sigma_i) \\
& \text{case } ((K \tau \pi \nu \pi) \triangleright \gamma) \rightarrow \nu \pi \rightarrow \nu \pi \\
& \text{case } (K \tau \pi \nu \pi) \triangleright \gamma \rightarrow \nu \pi \rightarrow \nu \pi \\
\end{align*}
\]

Here, \( e \) is case \((K \tau \pi \nu \pi) \triangleright \gamma\) of \( \nu \pi \rightarrow \nu \pi \) and \( e' \) is case \((K \tau \pi \nu \pi) \triangleright \gamma\) of \( \nu \pi \rightarrow \nu \pi \). Thus, by the definition of coercion, \(|e| = |e'|\) of \( \nu \pi \rightarrow \nu \pi \). To show that \(|e| = |e'|\), we must show only that \(|e| = |e'|\). From the definition of S_KPUSH, \( e_i' = e_i \triangleright \Psi (\sigma_i) \) and thus \(|e_i'| = |e_i|\) as desired.

\( \square \)

E. Metatheory for Consistency

In this section, we show that good contexts are consistent contexts following the plan laid out in Section 6. Recall the conditions of a good context:

We have Good \( \Gamma \) when the following conditions hold:

1. All coercion assumptions and axioms in \( \Gamma \) are of the form \( C : \forall \Delta. (F \tau \sim \tau') \) or of the form \( a : a \sim a \). In the first form, the arguments to the type function must behave like patterns: for all \( \nu \pi \), every \( \nu \tau_i \in \tau \) and every \( \nu \tau'_i \) such that \( \Gamma \vdash \nu \tau_i = \nu \tau'_i \), there exists \( \nu \phi \) such that \( \nu \tau_i = \nu \phi \), and \( \Gamma \vdash \nu \sigma_m \sim \nu \sigma_m' \) for each \( \nu e_i \in \tau \) and \( \nu e_i' \in \tau' \).

2. Axioms and coercion assumptions don’t overlap. For each \( \nu \pi \), there exists at most one prefix \( \nu \pi \) such that there exist \( \nu C \) and \( \nu \pi \) where \( C : \forall \Delta. F \pi \pi_0 \sim \nu \sigma_1 \in \Gamma \) and \( \nu \pi \pi_0 = \nu \pi_0 \). These \( \nu C \) and \( \nu \pi \) are unique for each matching \( \nu \pi \).

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 \( \nu \Delta. (F \pi \pi \sim \nu \tau) \) in \( \Gamma \), the kinds of each side must equal: for some \( \nu \kappa \), \( \nu \Delta \vdash \nu \tau \sim \nu \tau \) and \( \nu \Delta \vdash \nu \kappa \). And that kind must not mention bindings in the telescope, \( \nu \kappa \sim \nu \kappa \).

Showing that these conditions ensure that the context cannot prove two value types equal requires a number of auxiliary lemmas.

Lemma E.1 (No free coercion variables in erased types). If \( \Gamma \vdash \nu \tau \ : \ \nu \kappa \), then \( \forall \nu \pi \ : \ \nu \kappa \).

Proof. Proof is by inspection of the erasure function. All coercions are removed from types.

\( \square \)

Proof of Lemma 6.5 (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.

1. If \( \Gamma \vdash \nu \tau \) then \( |\Gamma| \vdash |\nu \tau| \).
2. If \( \Gamma \vdash \nu \tau \ : \ \kappa \) then \( |\Gamma| \vdash |\nu \tau| \ : \ \kappa| \).
3. If \( \Gamma \vdash \phi \) then \( |\Gamma| \vdash |\phi| \).
4. If \( \Gamma \vdash \nu \tau \ : \ \phi \) then \( |\Gamma| \vdash |\nu \tau| \ : \ |\phi| \).
5. If \( \Gamma \vdash \nu \tau \ : \ \kappa \) then \( |\Gamma| \vdash |\nu \tau| \ : \ |\kappa| \).

Proof. By simultaneous induction on the length of the explicit typing derivation. We present a few representative cases.
Case K\_CAST: Given rule:
\[\Gamma \vdash \tau : \kappa_1, \Gamma \vdash \eta : \kappa_1 \sim \kappa_2, \Gamma \vdash \kappa_2 : \ast \quad \text{K\_CAST}\]

By induction, we have \(|\Gamma| : |\tau| : |\kappa_1|\) and \(|\Gamma| : |\eta| : |\kappa_2|\). For all coercions within \(|\kappa_1| \sim |\kappa_2|\) and \(|\Gamma| = |\kappa_2| = |\ast|\). By the rule K\_CAST, we have \(|\tau \triangleright \eta| = |\kappa_2|\). Finally, by definition of erasure, we have \(|\tau \triangleright \eta| = |\tau_1|\) and we are done.

Case K\_CAPP: Given rule:
\[\Gamma \vdash \tau_1 : \forall \zeta, \phi, \kappa, \Gamma \vdash \gamma_1 : \phi \quad \text{K\_CAPP}\]

By induction and definition of erasure, we have \(|\Gamma| \vdash |\tau_1| : \forall \zeta, |\phi|, |\kappa|\), and \(|\Gamma| = |\gamma_1| = |\phi|\). Hence, by rule K\_CAPP, we have \(|\Gamma| \vdash |\tau_1| \circ \zeta : |\kappa|\), and by erasure \(|\tau_1| \sim |\kappa| \circ \zeta|\). Finally, we have \(|\kappa| |\gamma_1| / |\zeta|) = |\kappa|\), as the erasure operation erases all coercions within \(\kappa\).

Case CT\_COH: Given rule:
\[\Gamma \vdash \gamma : \tau \sim \tau_1, \Gamma \vdash \tau \triangleright \gamma' : \kappa \quad \text{CT\_COH}\]

By induction and erasure, we have \(|\Gamma| \vdash \gamma : |\tau| : |\gamma_1| \sim |\tau_1|\) and \(|\tau| \triangleright \gamma' : |\kappa|\). But also by erasure, we have \(|\gamma| \triangleright \gamma' : |\gamma|\) and \(|\tau_1| \sim \tau_2| : |\kappa|\) so we are done.

Case CT\_CAPP: Given rule:
\[\Gamma \vdash \gamma_1 : \tau_1 \sim \tau_1', \Gamma \vdash \tau_1 \triangleright \gamma' : \kappa' \quad \text{CT\_CAPP}\]

By induction and definition of erasure, we have \(|\Gamma| \vdash |\gamma_1| : |\tau_1| : |\tau_1'|\) and \(|\gamma' : |\kappa'|\). Hence, by rule CT\_CAPP, we have \(|\Gamma| : |\gamma_1| \circ \zeta_1 : |\tau_1' \circ \zeta_2|\). Finally, we have \(|\tau_1| \sim \tau_2\) and \(|\tau_1'| \sim \tau_2\), so we are done.

Case CT\_ALL\_C: Given rule:
\[\Gamma \vdash \gamma : \tau_1 \sim \tau_2, \Gamma \vdash \gamma_1 : \tau_1 \sim \tau_1' \quad \text{CT\_ALL\_C}\]

By induction and definition of erasure, we have \(|\Gamma| : |\gamma| : |\tau_1| : |\tau_2|\), \(|\gamma| = |\tau_1'|\), \(|\gamma_1| = |\tau_1|\), \(|\gamma_1| = |\tau_1'|\), \(|\gamma_1| = |\tau_1|\), and \(|\gamma_1| = |\tau_1'|\). Hence, by rule CT\_ALL\_C, we have \(|\Gamma| : |\gamma_1| \circ \zeta_1 : |\tau_1' \circ \zeta_2|\). Finally, we have \(|\gamma_1| \sim \gamma_2\) and \(|\gamma_1| \sim \gamma_2\), so we are done.

Lemma E.4 (Unique type function reduction). If Good \(\Gamma, \Gamma \vdash F \triangleright \sigma_1\) by TS\_RED and \(\Gamma \vdash F \triangleright \sigma_2\) by TS\_APP, then \(\sigma_2 = F \triangleright \sigma_1\).

Proof. Since \(\Gamma \vdash \tau \sim \sigma_1\) by TS\_APP, we must have \(\tau = \tau_1 \circ \sigma_2\) and \(\sigma_2 = \sigma_1 \circ \sigma_1\) with \(\Gamma \vdash \tau_1 \sim \sigma_1\) and \(\Gamma \vdash \tau_2 \sim \sigma_2\). By Lemma E.4, we know that \(\sigma_1\) must have the form \(F \triangleright \sigma_1\). Thus, we have the desired result.

Lemma E.5 (Single Step Substitution). If \(\Gamma \vdash \tau \sim \tau'\), \(\phi\) well-formed and a not free in \(\Gamma\), then \(\Gamma \vdash \tau[a] \sim \tau'[a]\).

Proof. By induction on \(\tau\). For instance, if \(\sigma = \forall \phi, \tau, \sigma_1 \sim \sigma_2, \tau', \) by induction,
\(\Gamma \vdash \tau[a] \sim \tau'[a]\).

Thus also
\(\Gamma \vdash \sigma_i[t/a] \sim \sigma_i[t'/a]\).

The other cases are similar.

Here, we prove completeness of the rewrite reduction with respect to the coercion relation. The two key lemmas of the completeness proof are that joinability is preserved under substitution, and a local diamond property of rewriting.

Lemma E.6 (Local diamond property). If Good \(\Gamma, \Gamma \vdash \tau \sim \sigma_1\) and \(\Gamma \vdash \tau \sim \sigma_2\) then there exists \(\tau_1\) such that \(\Gamma \vdash \sigma_1 \sim \sigma_1\) and \(\Gamma \vdash \sigma_2 \sim \sigma_2\).

Proof. Induction on the length of the two step derivations with a case analysis on the last rule used in each.

The overlapping cases are TS\_REFL and anything else, TS\_APP\_TS\_RED (and symmetric), and all instances with the same final rule on both sides. The reflexivity overlaps are trivial. All other pairs of rules apply to types with different head forms. Of the same-same overlaps, most follow by induction. We demonstrate an example of this pattern with case TS\_APP\_TS\_APP below. The exception is TS\_RED\_TS\_RED and TS\_VAR\_TS\_VAR \_TS\_RED which are both deterministic. Below, we complete the proof with the TS\_APP\_TS\_RED case.
Case TS_APP-TS_APP: Concretely, we have a type $\tau \sigma$ with reductions:

$$
\Gamma \vdash \tau \sigma \leadsto \tau' \sigma', \quad \Gamma \vdash \tau \sigma \leadsto \tau'' \sigma''
$$

Now, we can deduce:

$$
\Gamma \vdash \sigma \leadsto \sigma' \quad \Gamma \vdash \sigma \leadsto \sigma''
$$

So by induction, we can find $\sigma'''$ that is a common reduct. We also know

$$
\Gamma \vdash \tau \leadsto \tau' \quad \Gamma \vdash \tau \leadsto \tau''
$$

So, also by induction, we can find $\tau'''$ that is a common reduct of the two. Hence, by TS_TAPP,

$$
\Gamma \vdash \tau' \sigma' \leadsto \tau'' \sigma'' \quad \Gamma \vdash \tau'' \sigma'' \leadsto \tau''' \sigma'''
$$

Case TS_RED-TS_APP: Concretely, we have a type $F \overline{\tau}$, with reductions:

$$
\Gamma \vdash (F \overline{\tau}) \leadsto \sigma_1 \quad \Gamma \vdash (F \overline{\tau}) \leadsto \sigma'_2 \sigma'
$$

Where the first reduction is a type function reduction. Now note that, since context is good, type function axioms are nonoverlapping. Now say that $\overline{\tau} = \overline{\pi}_0 \sigma$. We have by inversion, $\Gamma \vdash F \overline{\pi}_0 \leadsto \sigma'_2 \sigma'$. By Lemma E.4, we have that $\sigma'_2 = F \overline{\tau}_0$, such that $\Gamma \vdash \overline{\pi}_0 \leadsto \sigma'_2 \sigma'$ and so that $\Gamma \vdash \overline{\pi}_0 \sigma \leadsto \sigma'_2 \sigma'$. We have that if the coercion for $F$ is $\overline{\tau} : C : \Delta$. ($F \overline{\tau} \leadsto \overline{\tau}_1$), then we have $\overline{\pi}_0 \sigma = \overline{\tau}_1[\overline{\pi}_1/\Delta]$ (for some $\overline{\pi}_1$, and now by the first condition of good contexts, we have a $\overline{\pi}_1$, such that

$$
\overline{\tau}_0, \sigma' = \overline{\tau}_1[\overline{\pi}_1/\Delta] \quad \Gamma \vdash \overline{\pi}_1 \leadsto \overline{\pi}_1
$$

In which case we have a reduction $\Gamma \vdash F \overline{\tau}_0 \sigma' \leadsto \tau_1[\overline{\pi}_1/\Delta]$. But, by an extension of Lemma E.10 for telescopes, for good contexts, we have that

$$
\sigma_1 = \tau_1[\overline{\pi}_1/\Delta] \quad \Gamma \vdash \sigma_1 \leadsto \tau_1[\overline{\pi}_1/\Delta]
$$

As desired.

Case TS_RED-TS_RED: Concretely, we have a type $F \overline{\pi}_1 \overline{\pi}_2$, which can also be written as $F \overline{\pi}_1 \overline{\pi}_2$, such that we have reductions:

$$
\Gamma \vdash F \overline{\pi}_1 \sigma' \quad \Gamma \vdash F \overline{\pi}_1 \sigma''
$$

But since good contexts have nonoverlapping axioms, we have that only one axiom applies. Hence, we are done: $\sigma' = \sigma''$.

Lemma E.7 (Transitivity of Rewriting). If Good $\Gamma$ and $\Gamma \vdash \sigma_1 \Leftrightarrow \sigma_2$ and $\Gamma \vdash \sigma_2 \Leftrightarrow \sigma_3$, then $\Gamma \vdash \sigma_1 \Leftrightarrow \sigma_3$.

Proof. Appeal to the local diamond property. Suppose $\sigma_{12}$ is a join point for $\sigma_1, \sigma_2$ and $\sigma_{23}$ is a join point for $\sigma_2, \sigma_3$. By Lemma E.6, there is a join point $\sigma_0$ for $\sigma_{12}, \sigma_{23}$, and hence is a join point for $\sigma_1, \sigma_3$.

Lemma E.8 (Multistep Substitution). If $\Gamma \vdash \tau \leadsto^* \tau'$, $\sigma$ well-formed and a not free in $\Gamma$, then $\Gamma \vdash \sigma[\tau/a] \leadsto^* \sigma[\tau'/a]$.

Proof. By induction on the length of the reduction $\Gamma \vdash \tau \leadsto^* \tau'$. The base case is trivial, and the inductive step follows by Lemma E.5.

Lemma E.9 (Single Substitution). If $\Gamma \vdash \tau \leftrightarrow \tau'$, and $\Gamma \vdash \sigma \leadsto \sigma'$, with $\sigma, \sigma'$ well-formed and a not free in $\Gamma$, then $\Gamma \vdash \sigma[\tau/a] \Leftrightarrow \sigma'[\tau'/a]$.

Proof. There is a join point $\tau''$ of $\tau$ and $\tau'$, we can apply Lemma E.8 to the reductions $\Gamma \vdash \tau \leadsto^* \tau''$ and $\Gamma \vdash \tau' \leadsto^* \tau''$, and connect two reductions with Lemma E.7.

Lemma E.10 (Single Step Double Substitution). Suppose Good $\Gamma$ and $\Gamma \vdash \sigma \leadsto \sigma'$, with a free in $\sigma$ and $\sigma'$ in $\sigma'$ well-formed, and $\Gamma' = \Gamma, c : a \leadsto a', \Gamma''$ or $\Gamma = \Gamma', c : a \leadsto a, \Gamma''$. Then Good $(\Gamma', \Gamma'')[\sigma'/a][\sigma'/a']$, and if $(\Gamma', \Gamma'')[\sigma'/a] \vdash \tau \leftrightarrow \tau'$, then $(\Gamma', \Gamma'')[\sigma'/a][\sigma'/a'] \vdash \sigma[\tau/a] \Leftrightarrow \sigma'[\tau'/a']$.

Proof. Note first that since Good $\Gamma$, the only axiom mentioning $a, a'$ is $c$. Hence, Good $(\Gamma', \Gamma'')[\sigma'/a][\sigma'/a']$ is immediate. The rest follows by induction on the derivation of $\Gamma \vdash \sigma \leadsto \sigma'$.

Case TS_REFL: Follows from Lemma E.9.

Case TS_ALLT: The rule is

$$
\begin{array}{c}
\Gamma, \Gamma' \vdash \kappa \leadsto \kappa' \quad \Gamma, \Gamma' \vdash \forall a_1 : \kappa, \sigma \leadsto \forall a_2 : \kappa', \sigma' \\
\text{TS_ALLT}
\end{array}
$$

where $\sigma_1 \neq \sigma_2 \neq \sigma_3$, and $\Gamma, \Gamma' = \Gamma''$, $c : a \leadsto a', \Gamma''$. By induction, we have both $(\Gamma, \Gamma'')[\sigma'/a][\sigma'/a'] \vdash \kappa[\tau/a] \Rightarrow \kappa[\tau'/a']$. And $(\Gamma[\tau/a][\tau'/a'], \sigma_1 \leadsto \sigma_2, \Gamma[\tau/a][\tau'/a'] \Rightarrow \sigma[\tau/a] \Leftrightarrow \sigma'[\tau'/a']$. But now, we can put these transitions together with rule TS_ALLT: first, we have

$$
(\Gamma, \Gamma')[\tau/a][\tau'/a'] \Rightarrow \forall a_1 : \kappa[\tau'/a'], \sigma[\tau/a] \Rightarrow \forall a_2 : \kappa[\tau'/a'], \sigma[\tau'/a']
$$

By $\alpha$-renaming, the right hand side is $\forall a_2 : \kappa'[\tau'/a'], \sigma[\tau/a]$. Now, we have

$$
(\Gamma, \Gamma')[\tau/a][\tau'/a'] \Rightarrow \forall a_2 : \kappa'[\tau'/a'], \sigma[\tau/a] \Rightarrow \forall a_2 : \kappa'[\tau'/a'], \sigma'[\tau'/a']
$$

and we are done by Lemma E.7.

Case TS_ALLC: Almost identical to the previous case.

Case TS_RED: Follows from Lemma E.9.

Case TS_VARRED: If the coercion in question isn’t $c : a \leadsto a'$, then this case is trivial. Otherwise, we have by assumption that $(\Gamma', \Gamma'')[\tau/a][\tau'/a'] \vdash \tau \leftrightarrow \tau'$, which is exactly what we need to prove.

Case TS_APP: Similar to the previous cases. By induction (suppose we are substituting $\sigma', \sigma''$):

$$
(\Gamma', \Gamma'')[\sigma'/a][\sigma''/a'] \vdash \tau[\sigma''/a] \Leftrightarrow \tau'[\sigma''/a']
$$

and

$$
(\Gamma', \Gamma'')[\sigma'/a][\sigma''/a'] \vdash \sigma[\sigma'/a] \Leftrightarrow \sigma'[\sigma''/a'].
$$

So, by applying TS_APP, we have

$$
\begin{array}{c}
(\Gamma', \Gamma'')[\sigma'/a][\sigma''/a'] \vdash \tau[\sigma'/a] \Leftrightarrow \tau'[\sigma''/a']
\end{array}
$$

and

$$
\begin{array}{c}
(\Gamma', \Gamma'')[\sigma'/a][\sigma''/a'] \vdash \tau'[\sigma''/a'] \Leftrightarrow \tau'[\sigma''/a']
\end{array}
$$

so we are done by Lemma E.7.

Case TS_CAPP: Immediate, by induction.

Lemma E.11 (Substitution). Suppose Good $\Gamma$ and $\Gamma \vdash \sigma \leadsto \sigma'$, with $a, a'$ free in $\sigma, \sigma'$, and $\Gamma = \Gamma', c : a \leadsto a', \Gamma''$. Suppose also that $(\Gamma', \Gamma'')[\tau/a][\tau'/a'] \vdash \tau \Leftrightarrow \tau'$. Then, $(\Gamma', \Gamma'')[\tau/a][\tau'/a'] \vdash \sigma[\tau/a] \Leftrightarrow \sigma'[\tau'/a']$.

Proof. Induction on the length of reduction $\Gamma \vdash \sigma \leadsto \sigma'$. The base case is trivial. The inductive step follows by Lemma E.10 and Lemma E.7.
Lemma E.13 (Joinability strengthening). If Good $\Gamma$ and $\Gamma$, $\alpha, \kappa, \Gamma'$, and $\Gamma, \alpha, \kappa, \Gamma', \tau_1 \Rightarrow \tau_2$, then Good $(\Gamma, \Gamma')$, $\Gamma' \Rightarrow \tau_1 \Rightarrow \tau_2$.

Proof. By inspection on the rewrite relation. The rewrite relation does not depend on any type bindings in the context, only axioms.

Lemma E.14 (Basic implicit substitution). 1. If $\Gamma, \alpha : \kappa_1, \Gamma' \Rightarrow \tau : \kappa_2$ and $\Gamma \Rightarrow \sigma : \kappa_1$, then $\Gamma, \Gamma'[\sigma/\alpha] \Rightarrow \tau[\sigma/\alpha] : \kappa_2$. 2. If $\Gamma = \Gamma, \alpha : \kappa_1, \Gamma'$ and $\Gamma \Rightarrow \sigma : \kappa_1$, then $\Gamma, \Gamma'[\sigma/\alpha]$. 3. If $\Gamma, \alpha : \kappa_1, \Gamma' \Rightarrow \phi \text{ ok}$ and $\Gamma \Rightarrow \sigma : \kappa_1$, then $\Gamma, \Gamma'[\sigma/\alpha] \Rightarrow \phi[\sigma/\alpha]$ ok.

Proof. Straightforward mutual induction.

Lemma E.15 (Implicit regularity/generation). If $\Gamma \Rightarrow \tau : \kappa$, then $\Gamma \Rightarrow * : \text{ the height of this derivation is at most the height of } \Gamma \Rightarrow \tau : \kappa$.

Proof. Straightforward induction, appealing to Lemma E.14 in the IT,TAPP case. The base cases appeal to rules IT,STAR,INSTAR and IT,ARROW.

Lemma E.16 (Weakening for implicit system). If $\Gamma \Rightarrow \tau : \kappa$, then $\Gamma, \Gamma' \Rightarrow \tau : \kappa$ for any $\Gamma'$ such that $\Gamma = \Gamma$, $\Gamma'$, and there exists a derivation of $\Gamma \Rightarrow \tau : \kappa$ with height at most the height of the derivation of $\Gamma, \Gamma' \Rightarrow \tau : \kappa$.

Proof. Straightforward induction.

We need a lemma to deal with the kind $\gamma$ construct. Essentially, this lemma states that we don’t need the kind $\gamma$ construct, as it is already internalized in our system.

Lemma E.17 (Admissibility of “kind”). Suppose we have a derivation $\Gamma \Rightarrow \gamma : \tau_1 \Rightarrow \tau_2$ such that $\Gamma \Rightarrow \gamma : \kappa_1$ and $\Gamma \Rightarrow \tau_2 : \kappa_2$ and $\text{fcv}(\gamma) \subseteq \text{dom } \Gamma'$ for some subcontext $\Gamma'$ satisfying Good $\Gamma'$. Then, there exists a derivation $\Gamma \Rightarrow \eta : \kappa_1 \Rightarrow \kappa_2$ at strictly lower height, for some $\eta$, such that $\text{fcv}(\eta) \subseteq \text{dom } \Gamma'$.

Proof Sketch. By induction on the derivation $\Gamma \Rightarrow \gamma : \tau_1 \Rightarrow \tau_2$. Most cases are straightforward. We consider two here.

Case ICT_AXIOM: Given rule:
\[
C : \forall \Delta. (\tau_1 \Rightarrow \tau_2) \in \Gamma \Rightarrow \pi : \Delta
\]
\[
\Gamma \Rightarrow \pi : \Delta
\]
Note that the free coercion variables of $\tau_1 \Rightarrow \tau_2$ lie in a good context, so the same is true of $\tau_1$ and $\tau_2$. Hence, by induction, we are able to find derivations of $\Gamma \Rightarrow \eta_1 : \kappa_1 \Rightarrow \kappa_2$ and $\Gamma \Rightarrow \eta_2 : \kappa_2 \Rightarrow \kappa_3$, both of height less that than of any premises of ICT_TRANS. Now, by ICT_TRANS, we are able to create a proof $\Gamma \Rightarrow \eta_1 \eta_2 : \kappa_1 \Rightarrow \kappa_3$ at height strictly less than that of the conclusion, and we are done.

Case ICT_TRANS: Given rule:
\[
\Gamma \Rightarrow \tau_1 : \gamma \Rightarrow \tau_2 \Rightarrow \gamma \Rightarrow \tau_3 \Rightarrow \gamma \Rightarrow \Gamma
\]
\[
\text{ICT_TRANS}
\]
Note that the free coercion variables of $\tau_1 \Rightarrow \tau_2 \Rightarrow \gamma \Rightarrow \tau_3$ lie in a good context, so the same is true of $\gamma$. Hence, by induction, we are able to find derivations of $\Gamma \Rightarrow \eta_1 : \kappa_1 \Rightarrow \kappa_2$ and $\Gamma \Rightarrow \eta_2 : \kappa_2 \Rightarrow \kappa_3$, both of height less than that of any premises of ICT_TRANS. Now, by ICT_TRANS, we are able to create a proof $\Gamma \Rightarrow \eta_1 \eta_2 : \kappa_1 \Rightarrow \kappa_3$ at height strictly less than that of the conclusion, and we are done.

We have rule:
\[
\Gamma \Rightarrow \gamma : \tau_1 \Rightarrow \tau_2 \Rightarrow \gamma \Rightarrow \tau_3 \Rightarrow \gamma \Rightarrow \Gamma
\]
\[
\text{ICT_TRANS}
\]
Note that the free coercion variables of $\gamma \Rightarrow \tau_1 \Rightarrow \tau_2 \Rightarrow \gamma \Rightarrow \tau_3$ lie in a good context, so the same is true of $\gamma$. Hence, by induction, we are able to find derivations of $\Gamma \Rightarrow \eta_1 : \kappa_1 \Rightarrow \kappa_2$ and $\Gamma \Rightarrow \eta_2 : \kappa_2 \Rightarrow \kappa_3$, both of height less than that of any premises of ICT_TRANS. Now, by ICT_TRANS, we are able to create a proof $\Gamma \Rightarrow \eta_1 \eta_2 : \kappa_1 \Rightarrow \kappa_3$ at height strictly less than that of the conclusion, and we are done.
by induction, there is a join point $\sigma''_2$ for $\sigma_1$ and $\sigma'_1$, and there is a join point $\sigma''_2$ for $\sigma_2$ and $\sigma'_2$. Let $\phi = \sigma''_1 \sim \sigma''_2$. Also by induction, there is a join point $\tau$ for $\tau_1, \tau_2$. By rule TS.ALLC, we have that

$$\Gamma \vdash \forall c_1 : \phi_1, \gamma_1 : \gamma_1 \sim^* \forall c_1 : \phi, \tau$$

and

$$\Gamma \vdash \forall c_2 : \phi_2, \gamma_2 : \gamma_2 \sim^* \forall c_2 : \phi, \tau$$

and hence they are joinable.

Case ICT.INST:

$$\Gamma |\gamma_1 : \langle \forall a_1 : \kappa_1, \gamma_1 \rangle \sim \langle \forall a_2 : \kappa_2, \gamma_2 \rangle$$

$$\Gamma |\gamma_2 : \sigma_1 \sim \tau_1$$

$$\Gamma |\gamma_3 : \sigma_2 \sim \tau_2$$

$$\Gamma |\gamma_4 : \sigma_3 \sim \kappa_2$$

$$\Gamma |\gamma_4 : \tau_1[\sigma_1/a_1] \sim \tau_2[\sigma_2/a_2]$$

Note that the free coercion variables of $\gamma_4\phi_\gamma$ lie in a good context, so the same is true of $\gamma$ and $\gamma_4$. Hence, by induction, $\Gamma |\gamma_4 : \tau_1[\sigma_1/a_1] \sim \tau_2[\sigma_2/a_2]$. Now, by inversion on the step relation for quantified types, we find that $\Gamma |\gamma_4 : \gamma_1 \sim \gamma_2$. By induction, the two quantified types have a join point. By inversion on the rewrite relation, both sides must step via TS.ALLT. Hence, we can find a join point for the kinds, and $\Gamma |\kappa_1 \sim \kappa_2$ as desired.

Case ICT.INST:

$$\Gamma |\gamma_1 : \langle \forall c_1 : \kappa_1, \gamma_1 \rangle \sim \langle \forall c_2 : \kappa_2, \gamma_2 \rangle$$

$$\Gamma |\gamma_2 : \sigma_1 \sim \tau_1$$

$$\Gamma |\gamma_3 : \sigma_2 \sim \tau_2$$

$$\Gamma |\gamma_4 : \sigma_3 \sim \kappa_2$$

$$\Gamma |\gamma_4 : \tau_1[\sigma_1/c_1] \sim \tau_2[\sigma_2/c_2]$$

Note that the free coercion variables of $\gamma_4\phi_\gamma$ lie in a good context, so the same is true of $\gamma$ and $\gamma_4$. Hence, by induction, $\Gamma |\gamma_4 : \tau_1[\sigma_1/c_1] \sim \tau_2[\sigma_2/c_2]$. Now, by inversion on the step relation for quantified types, we find that $\Gamma |\gamma_4 : \gamma_1 \sim \gamma_2$. By substitution (Lemma E.11), we have $\Gamma |\gamma_4 : \pi[\gamma_1] = \pi[\gamma_2]$. Furthermore, we have

$$\Gamma |\gamma_4 : \gamma_4 \sim \gamma_4$$

$$\Gamma |\gamma_4 : \gamma_4 \sim \gamma_4$$

$$\Gamma |\gamma_4 : \gamma_4 \sim \gamma_4$$

$$\Gamma |\gamma_4 : \gamma_4 \sim \gamma_4$$

$$\Gamma |\gamma_4 : \gamma_4 \sim \gamma_4$$

$$\Gamma |\gamma_4 : \gamma_4 \sim \gamma_4$$

$$\Gamma |\gamma_4 : \gamma_4 \sim \gamma_4$$

$$\Gamma |\gamma_4 : \gamma_4 \sim \gamma_4$$

By completeness, we have that those two kinds are joinable. There is some $\gamma$ such that $\Gamma |\gamma : \gamma_1 \sim \gamma_2$. However, by inversion on the rewrite relation, we see that it preserves the head forms of value types (since there exist no axioms for those by the first condition of Good $\Gamma$). Also, we know that erasure preserves head forms. Thus, $\xi_1$ and $\xi_2$ (and $\gamma$) have the same head form.

\[ \Box \]

F. Metatheory for Progress

Using the consistency lemma, it is straightforward to prove progress. We refer the reader to previous work (Weirich et al. 2010) for this proof, which requires only one more case for the current system:

Lemma E.1 (Progress for T.CONTRA). Assume $\Sigma$ is a closed, consistent context. If $\Sigma \vdash \text{contra}_\gamma \tau : \tau$, then there exists some $e$ such that $\text{contra}_\gamma \tau \rightarrow e$.

Proof. By inversion on $\Sigma \vdash \text{contra}_\gamma \tau : \tau$, we get that $\Sigma \vdash \gamma : H_t \gamma_1 \sim H_t \gamma_2$ and that $H_t \neq H_t$. However, these facts exactly contradict the fact that $\Sigma$ is consistent. Thus, our premises are absurd and we are done.

\[ \Box \]