Down with kinds: adding dependent heterogeneous equality to FC (Extended Version)

Stephanie Weirich Justin Hsu Richard A. Eisenberg

University of Pennsylvania
Philadelphia, PA, USA
{sweirich,justhsu,eir}@cis.upenn.edu

Abstract

FC, the core language of the Glasgow Haskell Compiler, is an explicitly-typed variant of System F with first-class type equality proofs (i.e. 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, especially 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 unifying types and kinds. 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.

1. Introduction

Is Haskell a dependently-typed programming language? That is a difficult question. For more than a decade, clever Haskellers have encoded many programs that were reputed to need dependent types. At the same time, GHC, Haskell's primary implementation, has augmented its type system with many new features inspired by dependently-typed languages, such as GADTs [Peyton Jones et al. 2006; Schrijvers et al. 2009], type families [Chakravarty et al. 2005], and datatype promotion with kind polymorphism [Yorgey et al. 2012]. These features have caused excitement among Haskell programmers eager to take advantage of the new expressiveness.

However, these extensions do not compose well. On the one hand, GADTs allow the programmer to exploit type equalities to write richer *terms*. On the other, datatype promotion and kind polymorphism have opened the door to much more expressive types. But GADTs cannot currently be promoted, so the useful *type* equalities available in *terms* cannot be lifted to useful *kind* equalities available in *types*. (We give examples in Section 2.)

Our goal in this paper is to eliminate such nonuniformities, and to do so with a single blow by unifying types and kinds. Specifically, we make the following contributions:

- We describe an explicitly-typed intermediate language, based directly on dependent type theory, in which we eliminate the distinction between types and kinds by adding the "Type-in-Type" axiom (Section 3). The language is no toy: it is an extension of the System FC intermediate language used by GHC today [Sulzmann et al. 2007; Vytiniotis et al. 2012; Weirich et al. 2011; Yorgey et al. 2012].
- The extended language uses explicit equality proofs at both the type and the kind level. This means that it enjoys a simple, fast, syntax-directed algorithm to determine the type of any term or the kind of any type (Section 3.5).
- We extend the *type preservation* proof of FC to the new constructs (Section 4.3). The treatment of datatypes requires an important result—that the equational theory is *congruent*. That is, we can derive a proof of equality for any form of type or kind, given proofs of equalities of its subcomponents. The computational content of this theorem, called *lifting*, generalizes a standard substitution operation. This operation is required in the operational semantics for datatypes.
- We also prove the *progress* theorem, which is true in the presence of a consistent set of axioms. Extending this theorem in the presence of kind coercions and dependent coercion abstraction requires two significant changes to the current proof. We discuss these changes and how they affect the design of the system in Section 5.

As far as we are aware, this type/kind system is the first dependently-typed language to include explicit (and irrelevant) equality proofs. Unlike other dependently-typed languages, definitional equality in FC is *only alpha-equivalence*. All of the action is in the provable equality. Furthermore, the provable equality supports *coherence*, which means that type and kind coercions can be ignored when showing equality. We discuss these differences in more detail in Section 6, as well as comparisons with other systems.

2. Why kind equalities?

Kind equalities enable new, useful features in GHC.

First, kind equalities are necessary for *kind-indexed GADTs*. Normal GADTs are nonuniform in their type parameters. For example, a representation type reflects the type structure as a datatype value that can be examined at runtime.

```
data TypeRep :: * → * where
  TyInt  :: Type Int
  TyBool :: Type Bool
```

Because of the nonuniform type index, pattern matching the data constructors of `TypeRep a` determines the identity of the type variable `a`. For example:

```
zero :: TypeRep a → a
zero TyInt  = 0        -- here we know that a is Int
zero TyBool = False    -- here we know that a is Bool
```

Type representations are useful for generic programming [Magalhães 2012; Weirich 2006]. However, the GADT above can only be used to represent types of kind $\star$. To represent type constructors with higher kinds, such as `Maybe` or `[]`, requires a separate data structure (perhaps called `TypeRep1`, indexed by types of kind $\star \to \star$). However, this approach is unsustainable. What about tuple types? Do we need a `TypeRep2`, `TypeRep3`, etc? (This situation is similar to that of GHC's `Data.Typeable` library, with its `Typeable`, `Typeable1`, `Typeable2`, etc. type classes.)

Kind polymorphism alone will not allow us to collapse these representations into a uniform datatype. Instead, we require a kind-indexed GADT as shown below.

```
data TypeRep :: forall k. k -> * where
   TyInt    :: TypeRep Int
   TyBool   :: TypeRep Bool
   TyMaybe  :: TypeRep Maybe
   TyApp    :: TypeRep a -> TypeRep b
            -> TypeRep (a b)
```

The data constructors of this datatype determine both the type and the (implicit) kind parameters. For example, `TyInt` is not kind polymorphic—the hidden kind parameter is $\star$. Pattern matching with this datatype refines kinds as well as types. For example, determining whether a type is of the form `Maybe b` makes new kind and type equalities available.

```
isMaybe :: forall k (a::k). TypeRep a -> ...
isMaybe rep = case rep of
  TyApp TyMaybe rb -> ..
     -- here we have k ~ * and
     -- a ~ Maybe b, where b :: *
```

Kind equalities also enable *datatype promotion for GADTs*. Currently GHC may only promote a subset of Haskell 98 datatypes [Yorgey et al. 2012]. There are some datatypes that are not available in the type level.

For example, Oury and Swierstra [2008] present a technique for embedding Cryptol [Galois, Inc. 2002] as a domain-specific language in Agda [Norell 2007], a full-spectrum dependently-typed language. In Agda, the definitions relevant to this discussion are as follows:

```
data Vec (A : Set) : Nat → Set where
  Nil : Vec A Zero
  _ :: _ : {n : Nat} → A → Vec A n → Vec A (Succ n)
data SplitView {A : Set} : {n : Nat} → (m : Nat)
    → Vec A (m * n) → Set where ...
```

The rest of the `SplitView` definition and its use is unimportant for our discussion; we are concerned solely with its type. It is straightforward to translate the definition for `Vec` into Haskell. However, it is impossible to translate the type of `SplitView` while retaining the type of the original. This is because `SplitView` uses `Vec` as a kind. To translate this usage into Haskell, Haskell's `Vec` datatype—a GADT—would have be promoted.

Finally, kind equalities are necessary for the definition and use of *kind families*. A kind family is a function that takes either types or kinds as arguments and returns a kind. Promoting datatypes that use type families requires kind families.

3. FC with kind equalities

FC is a language that has evolved over time, from its initial definition [Sulzmann et al. 2007], to the extensions FC_2 [Weirich et al. 2011], and $F_C^{\uparrow}$ [Yorgey et al. 2012].[1] However, its design has always been motivated by a desire to maintain the following properties.

- Type checking is syntax directed (and decidable). Although type inference for source Haskell programs may not terminate in the presences of certain flags (such as `UndecidableInstances`) once a core language term has been constructed, it can always be checked simply and quickly. This capability is necessary to ensure that transformation and optimization during compilation preserves typability.
- Types and equality proofs may be erased prior to runtime. As a result, the operational semantics includes a number of "push rules" that ensure that coercions do not suspend computation. The push rule for data constructors requires the ability to congruently lift equalities through types.
- Because type families are open, the soundness of the type system is parameterized by a *consistent* set of type equality axioms. More specifically, while preservation holds for any set of axioms, progress is ensured only in contexts where equalities between disjoint types cannot be proved.

The extensions that we describe to FC in this paper, even though they introduce a bit of "dependency" to the type/kind language, do not invalidate those properties. Furthermore, note that FC is the core language for a significant compiler, GHC. Therefore we are also constrained in that our extensions must be compatible with prior versions of the system and not require significant modification to the current implementation.

Type-in-Type A language with kind polymorphism, kind equalities, kind coercions, type polymorphism, type equalities and type coercions quickly becomes redundant (and somewhat overwhelming). Therefore, in this paper we follow pure type systems [Barendregt 1992] and unify the syntax of types and kinds. This compression allows us to reuse the syntax of type coercions as kind coercions. Furthermore, GHC already uses a shared datatype for types and kinds so this merging brings the formalism more in line with the actual implementation.

Following pure type systems, we could generalize over the sorts, axioms and rules. However, for simplicity we do not do so. Instead we combine types and kinds together semantically, by including only a single sort $\star$, and the $\star{:}\star$ axiom. Because of this axiom, the system does not distinguish between types and kinds.

However, we continue to use both of the words *type* and *kind* informally. In particular, we use the word *type* for those members that classify runtime expressions, and *kind* for those members that classify expressions of type language. Due to datatype promotion, which makes data constructors available to the type language, some objects may be both types and kinds.

Even though we have the type-in-type rule, we do not have a full-spectrum dependently-typed language. There is a still an important distinction between *expressions* e and *types* τ. This distinction reflects a phase distinction between compile-time and runtime terms. All types and coercions will be erased prior to execution.

3.1 Syntax

The basic syntactic classes appear in Figure 1. In this figure, types and kinds are drawn from the same syntactic class. By convention,

[1] We use the name FC for the language and all of its variants. In the technical discussion below, we contrast our new extensions with the most recent prior version, $F_C^{\uparrow}$.

we will use the metavariables τ and σ when treating an element of this class as a type and use the metavariable κ for kinds.

Types and kinds There are three important differences between types in this language and prior versions of FC:

1. Because we would like to preserve the syntax-directed nature of FC, we must make the use of kind equality proofs explicit. We do so via new form $\tau \rhd \gamma$ of kind coercions, that, when given a type τ, of kind κ_1, and a proof γ that kind κ_1 equals kind κ_2, produces a type of kind κ_2.
2. As in older versions of FC, coercions can be passed as arguments (using coercion abstractions $\lambda c{:}\, \phi.\ e$) and stored in data structures (as the arguments to data constructors of GADTs). This system deviates from earlier versions in that the types for these objects, written $\forall\, c{:}\, \phi.\, \tau$, *name* the abstracted proof with the coercion variable c and allow the body of the type τ to depend on this proof.
3. Finally, to promote GADTs, we must be able to promote data constructors that take coercions as arguments. This requires the new application form $\tau\, \gamma$. Note that there is no type-level abstraction over coercions. The form $\tau\, \gamma$ can only appear when the head of τ is a promoted datatype constructor.

Coercions Coercions γ are proof terms that witness the equality between types (or kinds). The modifications to the type language require analogous modifications to the coercions that reason about them. However, we defer the explanation of the syntax of these coercions to Section 3.5, when we discuss their formation rules.

Expressions 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. Therefore, we focus our discussion on types and coercions in the remainder of the paper. The exception is the treatment of datatypes, covered in Section 4.

Telescopes The bottom of the syntax figure displays the syntax for telescopes Δ, nested bindings of type and coercion variables. We describe the usage of telescopes in more detail in Section 3.8 and Section 4.

3.2 Syntactic sugar

To simplify the formalization, we rely on syntactic sugar listed in Figure 2. For example, we treat the function type constructor $(\to)$ as a right-associative, infix operator. Therefore, the type $\sigma_1 \to \sigma_2 \to \sigma_3$ is an abbreviation for $(\to)\, \sigma_1\, ((\to)\, \sigma_2\, \sigma_3)$.

Likewise, we use the metanotation $\sigma_1 \sim \sigma_2$ for *equality propositions*. This notation stands for the operator $(\sim)$ applied to the arguments κ_1, κ_2, σ_1 and σ_2. This operator is kind-polymorphic: the first two arguments κ_1 and κ_2 are the kinds of the second two arguments σ_1 and σ_2. Because the type system is syntax-directed, we can always recover these kinds, so we can safely omit them from the notation. We use the the metavariable ϕ to refer to types that are of this form.

The last line of the figure defines a derived coercion proof. We discuss this definition in Section 3.6.

3.3 Type System

The type system for this language includes the following judgements which check the validity of types, expressions, contexts, coercion proofs, and telescoped coercion proofs (the last is introduced in Section 3.8).

H	$::=$	Type constants
	$\mid\ (\to)$	Arrow
	$\mid\ (\sim)$	Equality proposition
	$\mid\ \star$	Type/Kind
w	$::=$	Type-level names
	$\mid\ a$	Type variables
	$\mid\ T$	Datatype constructors
	$\mid\ F$	Type functions
	$\mid\ K$	Data constructors
$\sigma,\ \tau,\ \kappa$	$::=$	Types and Kinds
	$\mid\ w$	Names
	$\mid\ H$	Constants
	$\mid\ \forall\, a{:}\, \kappa.\, \tau$	Polymorphic types
	$\mid\ \forall\, c{:}\, \phi.\, \tau$	Coercion abstr. type
	$\mid\ \tau_1\, \tau_2$	Type/kind application
	$\mid\ \tau_1 \rhd \gamma$	Casting
	$\mid\ \tau_1\, \gamma$	Coercion application
$\gamma,\ \eta$	$::=$	Coercions
	$\mid\ c$	Variables
	$\mid\ C\ \Theta$	Axiom application
	$\mid\ \langle\tau\rangle$	Reflexivity
	$\mid\ \mathbf{sym}\, \gamma$	Symmetry
	$\mid\ \gamma_1 \fatsemi \gamma_2$	Transitivity
	$\mid\ \forall\, a{:}\, \eta.\, \gamma$	Type/kind abstr. cong.
	$\mid\ \forall\, c{:}\, \eta.\, \gamma$	Coercion abstr. cong.
	$\mid\ \gamma_1\, \gamma_2$	Type/kind app. cong.
	$\mid\ \gamma(\gamma_2, \gamma_2')$	Coercion app. cong.
	$\mid\ \gamma \rhd \gamma'$	Coherence
	$\mid\ \gamma@\gamma'$	Type/kind instantiation
	$\mid\ \gamma@(\gamma_1, \gamma_2)$	Coercion instantiation
	$\mid\ \mathbf{nth}^i\, \gamma$	nth argument projection
	$\mid\ \mathbf{kind}\, \gamma$	Kind equality extraction
$e,\ u$	$::=$	Expressions
	$\mid\ x$	Variables
	$\mid\ \lambda x{:}\, \tau.\ e$	Abstraction
	$\mid\ e_1\, e_2$	Application
	$\mid\ \Lambda a{:}\, \kappa.\ e$	Type/kind abstraction
	$\mid\ e\, \tau$	Type/kind application
	$\mid\ \lambda c{:}\, \phi.\ e$	Coercion abstraction
	$\mid\ e\, \gamma$	Coercion application
	$\mid\ e \rhd \gamma$	Casting
	$\mid\ K$	Data constructors
	$\mid\ \mathbf{case}\ e\ \mathbf{of}\ \overline{p \to u}$	Case analysis
p	$::=$	Patterns
	$\mid\ K\ \Delta\ \overline{x{:}\, \tau}$	Data constructor pattern
Δ	$::=$	Telescopes
	$\mid\ \varnothing$	Empty
	$\mid\ \Delta,\ a{:}\, \kappa$	Type variable binding
	$\mid\ \Delta,\ c{:}\, \phi$	Coercion variable binding

Note: this figure refers to ϕ (described in Section 3.2) and Θ (described in Section 3.8).

Figure 1. Basic Grammar

Function type/kind	$\sigma_1 \to \sigma_2$	$\triangleq$	$(\to)\,\sigma_1\,\sigma_2$
Equality proposition, ϕ	$\sigma_1 \sim \sigma_2$	$\triangleq$	$(\sim)\,\kappa_1\,\kappa_2\,\sigma_1\,\sigma_2$
Coercion compatibility	$\gamma \triangleright \eta_1 \sim \eta_2$	$\triangleq$	$(\mathbf{sym}\,((\mathbf{sym}\,\gamma) \triangleright \eta_2)) \triangleright \eta_1$

Figure 2. Syntactic sugar

$\Gamma \vdash_{\mathsf{ty}} \tau : \kappa$	Type/kind validity	(Figure 3)
$\Gamma \vdash_{\mathsf{tm}} e : \tau$	Expression typing	(the appendix)
$\vdash_{\mathsf{wf}} \Gamma$	Context validity	(Figure 4)
$\Gamma \vdash_{\mathsf{co}} \gamma : \phi$	Coercion validity	(Figure 5)
$\Gamma \vdash_{\mathsf{tc}} \Delta \leftrightsquigarrow \Theta$	Telescoped coercion validity	(Figure 6)

These judgements refer to contexts Γ which are lists of assumptions for term variables, type variables, datatype constants, data constructors, and coercion variables and axioms.

$asn ::=$	Assumptions
$\mid\ x{:}\,\tau$	Term variables
$\mid\ w{:}\,\kappa$	Type variables and constants
$\mid\ c{:}\,\phi$	Coercion variables
$\mid\ C{:}\ \forall\,\Delta.\,\phi$	Coercion axioms
$\Gamma\ ::=$	Contexts
$\mid\ \varnothing$	Empty context
$\mid\ \Gamma,\ bnd$	Binding

Before explaining these judgements, we briefly review two of their properties. First, each of these judgements is syntax directed. Given the information before the colon (if present) there is a simple algorithm that determines if the judgement holds (and produces the appropriate kind, type or proposition). For the judgements without a colon, all the arguments are considered inputs; the algorithm simply determines whether or not the judgement holds. Second, these typing judgements are designed to satisfy the following generation properties that ensure that the subcomponents of each judgement are valid. Furthermore, the produced derivations are always *smaller* than the provided derivation.

Lemma 3.1 (Regularity/Generation)**.**

1. *If* $\Gamma \vdash_{\mathsf{ty}} \tau : \kappa$ *then* $\Gamma \vdash_{\mathsf{ty}} \kappa : \star$ *and* $\vdash_{\mathsf{wf}} \Gamma$.
2. *If* $\Gamma \vdash_{\mathsf{tm}} e : \tau$ *then* $\Gamma \vdash_{\mathsf{ty}} \tau : \star$ *and* $\vdash_{\mathsf{wf}} \Gamma$.
3. *If* $\Gamma \vdash_{\mathsf{co}} \gamma : \sigma_1 \sim \sigma_2$ *then* $\Gamma \vdash_{\mathsf{ty}} \sigma_1 : \kappa_1$ *and* $\Gamma \vdash_{\mathsf{ty}} \sigma_2 : \kappa_2$ *and* $\vdash_{\mathsf{wf}} \Gamma$.

Proof. The proof of this lemma is a straightforward induction on typing derivations, appealing to substitution Lemma 3.4. ☐

3.4 Type, expression and context validity

The rules for type formation appear in Figure 3. These rules are fairly standard. The first three rules declare the kinds of the constants: the sort $\star$, the function type constructor $(\to)$ and the equality proposition constructor $(\sim)$. The kinds of the first two of these mention themselves, but that does not cause difficulties. The kind of the third operator shows that equality is *heterogeneous*. As described above, in an equality proposition, the first two arguments are the kinds of the second two arguments and may differ.

The next rule applies to any name declared in the context that is valid in the type language. These names w include type variables, type constants, type function names and promoted data constructors. Datatype promotion allows data constructors, such as `Nothing` and `Just`, to appear in types and be the arguments of type

$\boxed{\Gamma \vdash_{\mathsf{ty}} \tau : \kappa}$ Kind and type validity

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

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

$$\frac{\vdash_{\mathsf{wf}} \Gamma}{\Gamma \vdash_{\mathsf{ty}} (\sim) : \forall\, a{:}\,\star.\,\forall\, b{:}\,\star.\, a \to b \to \star}\ \textsc{K\_Equal}$$

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

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

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

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

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

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

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

Figure 3. Type formation rules

$\boxed{\vdash_{\mathsf{wf}} \Gamma}$ Context well-formedness

$$\frac{}{\vdash_{\mathsf{wf}} \varnothing}\ \textsc{GWF\_Empty}$$

$$\frac{\Gamma \vdash_{\mathsf{ty}} \kappa : \star \quad a \# \Gamma}{\vdash_{\mathsf{wf}} \Gamma,\ a{:}\,\kappa}\ \textsc{GWF\_TyVar}$$

$$\frac{\Gamma \vdash_{\mathsf{ty}} \kappa : \star \quad F \# \Gamma}{\vdash_{\mathsf{wf}} \Gamma,\ F{:}\,\kappa}\ \textsc{GWF\_TyFun}$$

$$\frac{\Gamma \vdash_{\mathsf{ty}} \forall\,\overline{a{:}\,\kappa}.\star : \star \quad T \# \Gamma}{\vdash_{\mathsf{wf}} \Gamma,\ T{:}\forall\,\overline{a{:}\,\kappa}.\star}\ \textsc{GWF\_TyData}$$

$$\frac{\Gamma \vdash_{\mathsf{ty}} \tau : \kappa \quad x \# \Gamma}{\vdash_{\mathsf{wf}} \Gamma,\ x{:}\,\tau}\ \textsc{GWF\_Var}$$

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

$$\frac{\Gamma \vdash_{\mathsf{ty}} \phi : \star \quad c \# \Gamma}{\vdash_{\mathsf{wf}} \Gamma,\ c{:}\,\phi}\ \textsc{GWF\_CVar}$$

$$\frac{\Gamma, \Delta \vdash_{\mathsf{ty}} \phi : \star \quad C \# \Gamma}{\vdash_{\mathsf{wf}} \Gamma,\ C{:}\ \forall\,\Delta.\,\phi}\ \textsc{GWF\_Ax}$$

Figure 4. Context formation rules

functions. An advantage of combining types and kinds together is that here data constructors have the same "kinds" when they are used in the type language as their "types" when they are used in the expression language. (Previously, the types of data constructors had to be translated to kinds [Yorgey et al. 2012].) Because of this uniformity, *any* data constructor can be promoted.

The next two rules describe when type application is well-formed. Application is overloaded in the rules K_APP and K_TINST. However, this system is still syntax-directed because the type of the first component determines which rule applies. We do not combine function types $\sigma_1 \to \sigma_2$ and polymorphic types $\forall\, a{:}\,\kappa.\,\sigma$ into a single form because of type erasure. In the expression language, we must distinguish between term arguments, which are necessary at runtime, and type arguments, which may be erased. In kinds, the difference between nondependent and dependent arguments is not meaningful. However, when data constructors are promoted to the type level, their types maintain this distinction.

Because equality is heterogeneous, the casting rule K_CAST requires a third premise which ensures that the new kind has the correct classification. This premise ensures the invariant (Lemma 3.1) that everything to the right of the colon has kind $\star$.

Expression typing is unchanged from prior work so we do not discuss it here. For reasons of space, the elided rules and many of the proofs appear in the Appendix.

The rules for context formation appear in Figure 4. 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. We discuss these forms in more detail in Section 4.

3.5 Coercion proofs

Coercions γ are proof terms witnessing the equality between types (and kinds). The rules under which the proofs can be derived appear in Figure 5. These rules establish properties of the type equality relation.

- Equality is an *equivalence relation*, so the rules CT_REFL, CT_SYM, and CT_TRANS show that this relation is reflexive, symmetric and transitive.
- Equality is *compatible*, meaning that any pair of types can be shown equal by showing that their subcomponents are also equal. Every type formation rule (except for the base cases like variables and constants) has an associated compatibility rule. The exception is kind coercion $\tau \rhd \gamma$, where the compatibility rule is derivable (see Section 3.6). The compatibility rules are mostly straightforward; we discuss the rules for quantified types (rules CT_ALLT and CT_ALLC) in Section 3.7.
- Equality is *hypothetical*. Coercion variables and axioms add assumptions about equality to the context and appear in proofs (using rules CT_VAR and CT_VARAX respectively). We describe the use of parameterized axioms in Section 3.8. Such assumptions can be decomposed using the next five rules. For example, because we know that datatypes are injective type functions, we can decompose a proof of the equivalence of of two datatypes into equivalence proofs for each of the 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 and CT_INSTC).
- Equality is *heterogeneous*. The equality proposition $\sigma_1 \sim \sigma_2$ corresponds to McBride's "John Major" equality [McBride 2002]. Any two types can be declared to be equivalent even if they have different kinds. The proposition asserts both that the

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

$$\frac{\Gamma \vdash_{\mathsf{ty}} \tau \ :\ \kappa}{\Gamma \vdash_{\mathsf{co}} \langle\tau\rangle \ :\ \tau \sim \tau} \quad \text{CT\_REFL}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma \ :\ \tau_1 \sim \tau_2}{\Gamma \vdash_{\mathsf{co}} \mathbf{sym}\,\gamma \ :\ \tau_2 \sim \tau_1} \quad \text{CT\_SYM}$$

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

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

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

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{co}} \eta \ :\ \kappa_1 \sim \kappa_2 \quad a \overset{\bullet}{\mapsto} (a_1, a_2, c) \\ \Gamma,\ a_1{:}\,\kappa_1,\ a_2{:}\,\kappa_2,\ c{:}\,a_1 \sim a_2 \vdash_{\mathsf{co}} \gamma \ :\ \tau_1 \sim \tau_2 \\ \Gamma \vdash_{\mathsf{ty}} \forall\, a_1{:}\,\kappa_1.\,\tau_1 \ :\ \star \quad \Gamma \vdash_{\mathsf{ty}} \forall\, a_2{:}\,\kappa_2.\,\tau_2 \ :\ \star\end{array}}{\Gamma \vdash_{\mathsf{co}} \forall\, a{:}\,\eta.\,\gamma \ :\ (\forall\, a_1{:}\,\kappa_1.\,\tau_1) \sim (\forall\, a_2{:}\,\kappa_2.\,\tau_2)} \quad \text{CT\_ALLT}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{co}} \eta \ :\ \phi_1 \sim \phi_2 \quad c \overset{\bullet}{\mapsto} (c_1, c_2) \\ c_1 \mathrel{\#} |\gamma| \quad c_2 \mathrel{\#} |\gamma| \\ \Gamma,\ c_1{:}\,\phi_1,\ c_2{:}\,\phi_2 \vdash_{\mathsf{co}} \gamma \ :\ \tau_1 \sim \tau_2 \\ \Gamma \vdash_{\mathsf{ty}} \forall\, c_1{:}\,\phi_1.\,\tau_1 \ :\ \star \quad \Gamma \vdash_{\mathsf{ty}} \forall\, c_2{:}\,\phi_2.\,\tau_2 \ :\ \star\end{array}}{\Gamma \vdash_{\mathsf{co}} \forall\, c{:}\,\eta.\,\gamma \ :\ (\forall\, c_1{:}\,\phi_1.\,\tau_1) \sim (\forall\, c_2{:}\,\phi_2.\,\tau_2)} \quad \text{CT\_ALLC}$$

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

$$\frac{c{:}\,\phi \in \Gamma \quad \vdash_{\mathsf{wf}} \Gamma}{\Gamma \vdash_{\mathsf{co}} c \ :\ \phi} \quad \text{CT\_VAR}$$

$$\frac{C{:}\,\forall\,\Delta.\,(\tau_1 \sim \tau_2) \in \Gamma \quad \Gamma \vdash_{\mathsf{tc}} \Delta \leftrightsquigarrow \Theta}{\Gamma \vdash_{\mathsf{co}} C\,\Theta \ :\ \Theta_1(\tau_1) \sim \Theta_2(\tau_2)} \quad \text{CT\_VARAX}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma \ :\ H\,\overline{\tau} \sim H\,\overline{\tau'}}{\Gamma \vdash_{\mathsf{co}} \mathbf{nth}^i\,\gamma \ :\ \tau_i \sim \tau_i'} \quad \text{CT\_NTH}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma_1 \ :\ (\forall\, a_1{:}\,\kappa_1.\,\tau_1) \sim (\forall\, a_2{:}\,\kappa_2.\,\tau_2)}{\Gamma \vdash_{\mathsf{co}} \mathbf{nth}^1\,\gamma_1 \ :\ \kappa_1 \sim \kappa_2} \quad \text{CT\_NTH1TA}$$

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

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma \ :\ (\forall\, c{:}\,\phi.\,\tau) \sim (\forall\, c'{:}\,\phi'.\,\tau')}{\Gamma \vdash_{\mathsf{co}} \mathbf{nth}^1\,\gamma \ :\ \phi \sim \phi'} \quad \text{CT\_NTH1CA}$$

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

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

Figure 5. Coercion proofs

types are equal *and* that their kinds are also equal. Therefore, given a proof of an equality between two types, we can extract from it a proof of equality between their kinds.

3.6 Coercion irrelevance and coherence

Although the type system includes a judgement that decides whether two types are equal, and types may include explicit coercion proofs, the system does not include a judgement that states when two coercions are equal. The reason is that this relation is trivial. All coercions between equivalent proofs can be considered equivalent. Coercion proofs are irrelevant to type equality. As a result, FC is open to extension by new, consistent coercion axioms.

This "proof irrelevance" is reflected in the compatibility rule for coercion application, CT_CAPP. In this rule, note that there are no restrictions on γ_2 and γ_2' other than ensuring that the applications are well-formed, which indirectly implies that they prove equivalent equalities. Another example of irrelevance is in rule CT_INSTC. Again, the rule requires no relation between the two coercions γ_1 and γ_2.

Not only is the identity of coercion proofs irrelevant, but type equivalence also ignores their uses. The coherence rule, CT_COH, essentially says that the use of kind coercions can be ignored when proving type equalities. Although this rule is asymmetric, it is powerful. In particular, it can derive the compatibility and elimination rules for coerced types. The derived compatibility rule is below (note that the rule requires no explicit relation between η_1 and η_2):

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

For convenience, we define notation for this coercion.

Definition 3.2 (Coercion compatibility)**.** *The notation* $\gamma \rhd \eta_1 \sim \eta_2$ *abbreviates the coercion* $(\mathbf{sym}\,((\mathbf{sym}\,\gamma) \rhd \eta_2)) \rhd \eta_1$.

Likewise, coherence derives a proof term for decomposing equalities between coerced types.

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma \;:\; \tau_1 \rhd \gamma_1 \sim \tau_2 \rhd \gamma_2}{\Gamma \vdash_{\mathsf{co}} \mathbf{sym}\,(\langle \tau_1 \rangle \rhd \gamma_1) \,⨾\, \gamma \,⨾\, \langle \tau_2 \rangle \rhd \gamma_2 \;:\; \tau_1 \sim \tau_2}$$

3.7 Compatibility rules for quantified types

The compatibility rules for the two type forms with quantifiers, $\forall\, a{:}\, \kappa.\, \sigma$ and $\forall\, c{:}\, \phi.\, \sigma$, require explanation.

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

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

This rule sufficed in those systems because the only quantified types that could be shown equal had the same syntactic kinds κ for the bound variable. However, here we have a nontrivial equality between kinds. That means that we need to show a more general proposition: $\forall\, a{:}\,\kappa_1.\, \tau_1 \sim \forall\, a{:}\,\kappa_2.\, \tau_2$, even when κ_1 is not syntactically equal to κ_2. Without this generality, the language does not satisfy the preservation lemma, which requires that the quality relation be substitutive (see Section 4). In other words, given a valid type σ where a appears free, and a proof $\Gamma \vdash_{\mathsf{co}} \gamma \;:\; \tau_1 \sim \tau_2$, we must be able to derive a proof between $\sigma[\tau_1/a]$ and $\sigma[\tau_2/a]$. For this property to hold, if a occurs in the bound of a quantified type $\forall\, b{:}\, a.\, \tau$, then we must be able to derive $\forall\, b{:}\, \tau_1.\, \tau \sim \forall\, b{:}\, \tau_2.\, \tau$.

Rule CT_ALLT shows when two polytypes are equal. The first premise requires a proof η that the kinds of the bound variables are equal. The syntax of the proof term for this rule $\forall\, a{:}\, \eta.\, \gamma$ uses a single variable a to abbreviate the three variables (a_1, a_2, c). We assume the presence of a bijective function $\overset{\bullet}{\mapsto}$ to map between a and this triple in the second premise of the rule. Because the kinds of the bound variables are not syntactically equal, the third premise of the rule adds both bindings $a_1{:}\,\kappa_1$ and $a_2{:}\,\kappa_2$ to the context as well as an assertion c that a_1 and a_2 are equal. Both a_1 and a_2 are available for γ, the proof that the bodies of the polytypes are equal. However, 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 type of coercion abstractions, $\forall\, c{:}\, \phi.\, \tau$. The rule CT_ALLC constructs a proof that two such types of this form are equal. We can only construct such proofs when the abstracted propositions are equal. The proof term introduces two coercion variables into the context, similar to the two type variables above. However, because of proof irrelevance, there is no need for a proof of equality between coercions—so there is no analogue to the c variable in the rule CT_ALLT. Similarly, we assume the presence of a bijective function $\overset{\bullet}{\mapsto}$ between c and the pair (c_1, c_2).

The rule CT_ALLC also restricts how the variables c_1 and c_2 can be used in γ. The premises $c_1 \# |\gamma|$ and $c_2 \# |\gamma|$ prevent these variables from appearing in the relevant parts of γ. The reason for this restriction comes from our proof technique for the consistency of this proof system. We define the erasure operation $|\cdot|$ and discuss this issue in more detail in Section 5 and Section 6.

3.8 Axioms

In FC, top-level axioms for type equality are allowed to be *axiom schemes*—they may be parameterized and must be instantiated when used. For example, a type family declaration and instance

```
type family F a :: *
type instance F [a] = Maybe a
```

generates the following parameterized axiom

$$\mathsf{axF} : \forall\, a{:}\, \star\,.\, F\,(\mathbf{List}\; a) \sim \mathbf{Maybe}\; a$$

When we use an axiom we must fill in its parameters. For example, instantiating the above with the type $\mathbf{Int}$ produces a proof of the equality $F\,(\mathbf{List}\,\mathbf{Int}) \;\sim\; \mathbf{Maybe}\,\mathbf{Int}$. However, FC allows a slightly more general instantiation via coercions.[2] Given a coercion

$$\Gamma \vdash_{\mathsf{co}} \gamma \;:\; \mathbf{Int} \sim b$$

we can use it to instantiate the axiom above as follows

$$\Gamma \vdash_{\mathsf{ty}} \mathsf{axF}\,\gamma \;:\; F\,(\mathbf{List}\,\mathbf{Int}) \sim \mathbf{Maybe}\; b$$

The general form of an axiom gathers multiple parameters in a *telescope*, a context 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 Θ, a structure that we call a *telescoped coercion*. This structure is like a list, and includes a coercion for each type parameter, and a pair of coercions for each proof parameters. (We also use this structure for the semantics of datatypes, see Section 4.)

$$\Theta ::= \emptyset \mid \Theta,\, a{:}\,\kappa \mapsto (\tau_1, \tau_2, \gamma) \mid \Theta,\, c{:}\,\phi \mapsto (\gamma_1, \gamma_2)$$

The judgement form $\Gamma \vdash_{\mathsf{tc}} \Delta \leftrightsquigarrow \Theta$, shown in Figure 6, defines when a telescoped coercion is compatible with a given telescope.

The general rule for axiom schemes requires a valid telescoped coercion.

$$\frac{C{:}\; \forall\, \Delta.\, (\tau_1 \sim \tau_2) \;\in\; \Gamma \qquad \Gamma \vdash_{\mathsf{tc}} \Delta \leftrightsquigarrow \Theta}{\Gamma \vdash_{\mathsf{co}} C\; \Theta \;:\; \Theta_1(\tau_1) \sim \Theta_2(\tau_2)} \quad \text{CT\_VARAX}$$

Given a telescoped coercion, Θ, we must compute its effect on the left and right hand sides of the axiom. We do so by defining two different multisubstitutions, written $\Theta_1(\cdot)$ and $\Theta_2(\cdot)$, based on Θ.

[2] This generality helps GHC with coercion simplification.

$\boxed{\Gamma \vdash_{\mathsf{tc}} \Delta \leftrightsquigarrow \Theta}$ Telescoped coercion compatibility

$$\frac{\vdash_{\mathsf{wf}} \Gamma}{\Gamma \vdash_{\mathsf{tc}} \varnothing \leftrightsquigarrow \varnothing} \quad \text{TELCO\_EMPTY}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{tc}} \Delta \leftrightsquigarrow \Theta \\ \Gamma \vdash_{\mathsf{ty}} \sigma_1 \ : \ \Theta_1(\kappa) \quad \Gamma \vdash_{\mathsf{ty}} \sigma_2 \ : \ \Theta_2(\kappa) \\ \Gamma \vdash_{\mathsf{co}} \gamma \ : \ \sigma_1 \sim \sigma_2\end{array}}{\Gamma \vdash_{\mathsf{tc}} (\Delta,\ a{:}\kappa) \leftrightsquigarrow (\Theta, a{:}\kappa \mapsto (\sigma_1, \sigma_2, \gamma))} \quad \text{TELCO\_TY}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{tc}} \Delta \leftrightsquigarrow \Theta \\ \Gamma \vdash_{\mathsf{co}} \eta_1 \ : \ \Theta_1(\phi) \quad \Gamma \vdash_{\mathsf{co}} \eta_2 \ : \ \Theta_2(\phi)\end{array}}{\Gamma \vdash_{\mathsf{tc}} (\Delta,\ c{:}\phi) \leftrightsquigarrow (\Theta, c{:}\phi \mapsto (\eta_1, \eta_2))} \quad \text{TELCO\_CO}$$

Figure 6. Telescoped coercion validity

Definition 3.3 (Telescoped coercion substitution). *$\Theta_1(\cdot)$ and $\Theta_2(\cdot)$ are multisubstitutions, applicable to types, coercions, telescopes, typing contexts, and even other telescoped coercions.*

1. *For each $a{:}\,\kappa \mapsto (\tau_1, \tau_2, \gamma)$ in Θ, $\Theta_1(\cdot)$ maps a to τ_1 and $\Theta_2(\cdot)$ maps a to τ_2.*
2. *For each $c{:}\,\sigma_1 \sim \sigma_2 \mapsto (\gamma_1, \gamma_2)$ in Θ, $\Theta_1(\cdot)$ maps c to γ_1 and $\Theta_2(\cdot)$ maps c to γ_2.*

These two substitution operations satisfy the usual substitution lemmas. When a telescoped coercion is compatible with a given telescope, its substitution preserves types in all of our other judgements (and produces a derivation of smaller height than the sum of the original derivation plus that of the telescoped coercion validity judgement).

Lemma 3.4 (Telescoped coercion substitution).
Suppose $\Gamma \vdash_{\mathsf{tc}} \Delta \leftrightsquigarrow \Theta$.

1. *If $\Gamma, \Delta \vdash_{\mathsf{ty}} \tau \ : \ \kappa$ then $\Gamma \vdash_{\mathsf{ty}} \Theta_j(\tau) \ : \ \Theta_j(\kappa)$*
2. *If $\Gamma, \Delta \vdash_{\mathsf{co}} \gamma \ : \ \phi$ then $\Gamma \vdash_{\mathsf{co}} \Theta_j(\gamma) \ : \ \Theta_j(\phi)$*
3. *If $\Gamma, \Delta \vdash_{\mathsf{tc}} \Delta' \leftrightsquigarrow \Theta'$ then $\Gamma \vdash_{\mathsf{tc}} \Theta_j(\Delta') \leftrightsquigarrow \Theta_j(\Theta')$*

For space reasons, the proof of this lemma appears in the appendix. Note that the usual substitution lemmas, which substitute a single type or coercion, are a corollary of this lemma.

4. Datatypes

Because the focus of this paper is on the treatment of equality in the type language, we omit most of the discussion of the expression language and its operational semantics. However, because we have combined the semantics of types and kinds together, we must revise the treatment of datatypes. Previously, the arguments to datatype constructors could be stratified by dependency, 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 use the technique of telescopes to describe arbitrary dependency between arguments.

4.1 Telescopes and Datatypes

The validity rules for contexts (see Figure 4) restricts datatype constants T to have a kind of the form $\forall\, \overline{a{:}\,\kappa}.\, \star$. We call the variables $\overline{a}$ the *parameters* of the datatype. For simplicity, the type system requires the datatype parameters to be named even when they are not mentioned in later kinds. For example, the kind of the datatype List is $\forall\, a{:}\ \star\,.\, \star$ and the kind of the datatype $\mathsf{TypeRep}$ (from Section 2) is $\forall\, a{:}\star,\ b{:}\ a.\, \star$. Furthermore, datatypes can only be parameterized by types and kinds; no coercion parameters are allowed.

$\boxed{\Gamma \vdash_{\mathsf{tel}} \overline{\rho} : \Delta}$

$$\frac{\vdash_{\mathsf{wf}} \Gamma}{\Gamma \vdash_{\mathsf{tel}} \varnothing : \varnothing} \quad \text{T2\_EMPTY}$$

$$\frac{\Gamma \vdash_{\mathsf{ty}} \tau \ : \ \kappa[\overline{\rho}/\Delta] \quad \Gamma \vdash_{\mathsf{tel}} \overline{\rho} : \Delta}{\Gamma \vdash_{\mathsf{tel}} \overline{\rho}, \tau : (\Delta,\ a{:}\kappa)} \quad \text{T2\_CONST}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma \ : \ \phi[\overline{\rho}/\Delta] \quad \Gamma \vdash_{\mathsf{tel}} \overline{\rho} : \Delta}{\Gamma \vdash_{\mathsf{tel}} \overline{\rho}, \gamma : (\Delta,\ c{:}\phi)} \quad \text{T2\_CONSG}$$

Figure 7. Telescope arguments

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

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

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

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

$$\mathsf{TyInt} : \forall\, a{:}\ \star\,.\, \forall\, b{:}\, a.\, \forall\, c{:}\, a \sim \star.\, \forall\, c'{:}\, b \sim \mathbf{Int}.\ \mathsf{TypeRep}\ a\ b$$

Alternatively, the data constructor TyApp existentially binds a_1,b_1, b_2, and c—one kind and two type variables followed by a coercion.

$$\begin{array}{l}\mathsf{TyApp} : \forall a{:}\ \star\,.\forall b{:}\ a.\\ \quad \forall a_1{:}\ \star\,.\forall b_1{:}\ a_1 \to a.\forall b_2{:}\ a_1.\forall c{:}\ b \sim b_1\ b_2.\\ \qquad \mathsf{TypeRep}\ (a_1 \to a)\ b_1 \to \mathsf{TypeRep}\ a_1\ b_2 \to \mathsf{TypeRep}\ a\ b\end{array}$$

We use the metavariable ρ to stand for either a type τ or coercion γ. Then, a datatype value is of the form $K\ \overline{\tau}\ \overline{\rho}\ \overline{e}$, where the $\overline{\tau}$ denote the parameters (which cannot include coercions), the $\overline{\rho}$ instantiate the existential arguments, and $\overline{e}$ is the list of usual expression arguments to the data constructor. When reasoning about datatype values, we must type check its list of datatype arguments $\overline{\rho}$ against the given telescope. The judgement form $\Gamma \vdash_{\mathsf{tel}} \overline{\rho} : \Delta$, in Figure 7 performs this check.

4.2 Pushing datatypes

The most intricate part of the semantics of FC are the "push rules". These rules ensure that coercions do not interfere with the small step semantics by "pushing" coercions into the subcomponents of values whenever a coerced value appears in an elimination context. In the case of datatypes, the coercion must be distributed to all of the arguments of the data constructor, as shown in Figure 8. In the rest of this section, we explain the rule by describing the formation of the telescoped coercion Θ and its use in the operation of lifting.

The S_KPUSH rule uses a *lifting* operation, written $\Theta(\cdot)$, to coerce the types of its expression arguments. The intuition behind this operation is best found in an example: Say we have a data constructor K of type $\forall\, a{:}\ \star\,.\, F\ a \to T\ a$ for some type function F and some type constructor T. We then wish to cast an expression $K\ \mathbf{Int}\ e$ (for some appropriate term e) of type $T\,\mathbf{Int}$ to type $T\ a$. This will be done with a coercion γ of type $T\,\mathbf{Int} \sim T\ a$. In order

$$\frac{\begin{array}{l} K{:}\,\forall\,\overline{a{:}\,\kappa}.\,\forall\,\Delta.\,\overline{\sigma} \to (T\ \overline{a})\ \in\ \Gamma \\ \Theta = \{\gamma\} \prec \overline{\rho} : \Delta \\ \overline{\tau'} = \Theta_2(\overline{a}) \\ \overline{\rho}' = \Theta_2(\mathit{dom}\,\Delta) \\ \text{for each } e_i \in \overline{e}, \\ \qquad e_i' = e_i \rhd \Theta(\sigma_i) \end{array}}{\begin{array}{c} \textbf{case}\ ((K\ \overline{\tau}\ \overline{\rho}\ \overline{e}) \rhd \gamma)\ \textbf{of}\ \overline{p \to u} \to \\ \textbf{case}\ (K\ \overline{\tau'}\ \overline{\rho}'\ \overline{e'})\ \textbf{of}\ \overline{p \to u} \end{array}}\quad \text{S\_KPUSH}$$

Figure 8. The S_KPUSH rule

to pattern-match against $(K\,\mathbf{Int}\ e) \rhd \gamma$, we must "push" γ into the arguments of K so that we get an uncoerced datatype value as the scrutinee of the pattern match. Finding the right coercion to cast $\mathbf{Int}$ to a is easy enough: we use $\mathbf{nth}^1\ \gamma$. However, to find the right coercion for e we need to *lift* the type $F\ a$ into a coercion with respect to the original coercion γ.

In previous work, lifting was written $\sigma[a \mapsto \gamma]$, defined by analogy with substitution—because of the close syntax between 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 requires more machinery to describe precisely.

Lifting contexts We define lifting with respect to a *lifting context*, denoted with Ψ. A lifting context is a generalized form of a telescoped coercion. (A full treatment of lifting contexts appears in the appendix.) We base lifting on telescoped coercions because datatypes may have multiple, dependent parameters. However, the definition of lifting is also complicated by the two type productions that bind fresh variables: $\forall\, a{:}\,\kappa.\,\tau$ and $\forall\, c{:}\,\phi.\,\tau$. To be able to define lifting over these types, we need to be able to extend the mapping in a telescoped coercion with the fresh names for a and c used in proofs. The new mappings are marked with $\overset{\bullet}{\mapsto}$ as they are analogous to the bijective function $\overset{\bullet}{\mapsto}$ used previously.

$$\Psi ::= \Theta \mid \Psi, a{:}\,\kappa \overset{\bullet}{\mapsto} (a_1, a_2, c) \mid \Psi, c{:}\,\phi \overset{\bullet}{\mapsto} (c_1, c_2)$$

Definition 4.1 (Lifting). *We define the lifting of types to coercions, written $\Psi(\tau)$, by induction on the type structure. (Note that the cases with binding structure add new fresh variables to the lifting context, and that the last line uses Notation 3.2. The notation $\Psi_j(\cdot)$ indicates a multi-substitution analogous to $\Theta_j(\cdot)$.)*

$$\begin{array}{lcl} \Psi(a) & = & \gamma\ \textit{when} \\ & & \quad a{:}\,\kappa \mapsto (\tau_1, \tau_2, \gamma) \in \Psi \\ \Psi(a) & = & c\ \textit{when} \\ & & \quad a{:}\,\kappa \overset{\bullet}{\mapsto} (a_1, a_2, c) \in \Psi \\ \Psi(\tau) & = & \langle\tau\rangle\ \textit{when} \\ & & \quad \mathit{dom}\,(\Psi)\ \#\ \mathbf{fv}\,(\tau) \\ \Psi(\tau_1\,\tau_2) & = & \Psi(\tau_1)\,\Psi(\tau_2) \\ \Psi(\tau\,\gamma) & = & \Psi(\tau)(\Psi_1(\gamma), \Psi_2(\gamma)) \\ \Psi(\forall\, a{:}\,\kappa.\,\tau) & = & \forall\, a{:}\,\Psi(\kappa).\,(\Psi, a{:}\,\kappa \overset{\bullet}{\mapsto} (a_1, a_2, c))(\tau) \\ & & \quad \textit{when}\ a \overset{\bullet}{\mapsto} (a_1, a_2, c) \\ \Psi(\forall\, c{:}\,\phi.\,\tau) & = & \forall c : \Psi(\phi).(\Psi, c{:}\,\phi \overset{\bullet}{\mapsto} (c_1, c_2))(\tau) \\ & & \quad \textit{when}\ c \overset{\bullet}{\mapsto} (c_1, c_2) \\ \Psi(\tau \rhd \gamma) & = & \Psi(\tau) \rhd \Psi_1(\gamma) \sim \Psi_2(\gamma) \end{array}$$

Note that, because a lifting context Ψ generalizes a telescoped coercion, it is acceptable to use the notation $\Theta(\cdot)$ to refer to the lifting operation. Now we can state the lifting lemma which states that $\Theta(\tau)$ is a coercion between $\Theta_1(\tau)$ and $\Theta_2(\tau)$.

Lemma 4.2 (Lifting). *If $\Gamma \vdash_{\mathsf{tc}} \Delta \leftrightsquigarrow \Theta$ and $\Gamma, \Delta \vdash_{\mathsf{ty}} \tau\ :\ \kappa$ then*

$$\Gamma \vdash_{\mathsf{co}} \Theta(\tau)\ :\ \Theta_1(\tau) \sim \Theta_2(\tau)$$

Lifting context generation and extension In the S_KPUSH rule, the telescoped coercion Θ that is used for lifting is built in two stages. We use γ to build a telescoped coercion $\{\gamma\}$ that contains coercions for the parameters to the datatype (the operation $\{\cdot\}$ is defined below) and then extend this telescoped coercion with coercions for the existential arguments to the data constructor using the operation $\prec$, also defined below.

The definitions for these operations are motivated by the construction of Θ. This telescoped coercion contains mappings to the new $\overline{\tau'}$ and $\overline{\rho}'$ that are the "pushed" versions of the original $\overline{\tau}$ and $\overline{\rho}$ arguments. It is also used by lifting to construct each $\Theta(\sigma_i)$, the coercion for each expression argument e_i in $\overline{e}$.

Definition 4.3 (Lifting context generation). *If $\Gamma \vdash_{\mathsf{co}} \gamma\ :\ T\ \overline{\sigma} \sim T\ \overline{\sigma'}$, and $T{:}\,\forall\,\overline{a{:}\,\kappa}.\star\ \in\ \Gamma$, then define the telescoped coercion $\{\gamma\}$ as*

$$\{\gamma\} = \overline{a_i{:}\,\kappa_i \mapsto (\sigma_i, \sigma_i', \mathbf{nth}^i\ \gamma)}$$

Intuitively, $\{\gamma\}_1(\tau)$ replaces all parameters a in τ with the corresponding type on the left of the $\sim$ in the type of γ. Similarly, $\{\gamma\}_2(\tau)$ replaces with the corresponding type on the right of the $\sim$. We can think of $\{\gamma\}_1(\tau)$ as a "from" type and $\{\gamma\}_2(\tau)$ as a "to" type. Note that $\Gamma \vdash_{\mathsf{co}} \gamma\ :\ \{\gamma\}_1(T\ \overline{a}) \sim \{\gamma\}_2(T\ \overline{a})$.

The telescoped coercion that results from this coercion is compatible with the parameters of the datatype. More precisely:

Lemma 4.4 (Lifting context specification). *If $\Gamma \vdash_{\mathsf{co}} \gamma\ :\ T\ \overline{\sigma} \sim T\ \overline{\tau}$, and $T{:}\,\forall\,\overline{a{:}\,\kappa}.\star\ \in\ \Gamma$ then $\Gamma \vdash_{\mathsf{tc}} \overline{a{:}\,\kappa} \leftrightsquigarrow \{\gamma\}$.*

Proof. Straightforward induction. □

We now must define the $\prec$ operation that extends the telescoped coercion $\{\gamma\}$ with mappings for the variables in Δ, the existential parameters to the data constructor K. Because these arguments are dependent, we must define the operation recursively. The result of the operation is a new telescoped coercion that extends the input one. The intuition presented before extends as well: $\Theta_1(\tau)$ replaces any parameter or existential argument in τ with its corresponding "from" type and $\Theta_2(\tau)$ replaces a variable with its corresponding "to" type. The definition here is more complicated because of the dependent nature of the substitution.

Definition 4.5 (Telescoped coercion extension). *Define the operation of telescoped coercion extension $\Theta' = \Theta \prec \overline{\rho} : \Delta$ as:*

$$\begin{array}{lcl} \Theta \prec \varnothing : \varnothing & = & \Theta \\ \Theta \prec \overline{\rho}, \tau : \Delta,\ a{:}\,\kappa & = & \\ \quad \Theta', a{:}\,\kappa \mapsto (\tau, \tau \rhd \Theta'(\kappa), \mathbf{sym}\,(\langle\tau\rangle \rhd \Theta'(\kappa))) & & \\ \quad \textit{where}\ \Theta' = \Theta \prec \overline{\rho} : \Delta & & \\ \Theta \prec \overline{\rho}, \gamma : \Delta,\ c{:}\,\sigma_1 \sim \sigma_2 & = & \\ \quad \Theta', c{:}\,\sigma_1 \sim \sigma_2 \mapsto (\gamma, \mathbf{sym}\,(\Theta'(\sigma_1)) \fatsemi \gamma \fatsemi \Theta'(\sigma_2)) & & \\ \quad \textit{where}\ \Theta' = \Theta \prec \overline{\rho} : \Delta & & \end{array}$$

4.3 Type preservation

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

Lemma 4.6 (Preservation). *If $\Gamma \vdash_{\mathsf{tm}} e\ :\ \tau$ and $e \longrightarrow e'$ then $\Gamma \vdash_{\mathsf{tm}} e'\ :\ \tau$.*

The proof of this theorem is by induction on the typing derivation, with a case analysis on the rule used by the operational semantics. Most of the rules are straightforward, following directly by induction or by substitution (using a corollary of Lemma 3.4). The "push" rules require reasoning about coercion propagation. We include the details of the rules that differ from previous work [Weirich et al. 2010] in the appendix.

5. Consistency

The progress theorem holds only for *closed*, *consistent* contexts. A context is *closed* if it does not contain any expression variable bindings—as usual, open expressions could be stuck. We use the metavariable Σ to denote closed contexts.

The definition of consistency and the canonical forms lemma (necessary to show the progress theorem) are both stated using the notions of uncoerced *values* and their types, *value types*. Formally, we define values v and value types ξ, with the following grammars:

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

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

Lemma 5.1 (Canonical Forms)**.** *Say $\Sigma \vdash_{\mathsf{tm}} v \ : \ \sigma$. Then σ is a value type. Furthermore,*

1. *If $\sigma = \sigma_1 \to \sigma_2$ then v is $\lambda x{:}\,\sigma_1.\,e$ or $K\,\overline{\tau}\,\overline{\rho}\,\overline{e}$.*
2. *If $\sigma = \forall\, a{:}\,\kappa.\,\sigma'$ then v is $\Lambda a{:}\,\kappa.\,e$ or $K\,\overline{\tau}\,\overline{\rho}\,\overline{e}$.*
3. *If $\sigma = \forall\, c{:}\,\phi.\,\sigma'$ then v is $\lambda c{:}\,\tau_1 \sim \tau_2.\,e$ or $K\,\overline{\tau}\,\overline{\rho}\,\overline{e}$.*
4. *If $\sigma = T\,\overline{\tau}$ then v is $K\,\overline{\tau}\,\overline{\rho}\,\overline{e}$.*

Definition 5.2 (Consistency)**.** *A context Γ is consistent if whenever $\Gamma \vdash_{\mathsf{co}} \gamma \ : \ \xi_1 \sim \xi_2$ it is the case that*

1. *If ξ_1 is $T\,\overline{\sigma_1}$ then ξ_2 is $T\,\overline{\sigma_2}$.*
2. *If ξ_1 is $\sigma_1 \to \sigma_1'$ then ξ_2 is $\sigma_2 \to \sigma_2'$.*
3. *If ξ_1 is $\forall\, a{:}\,\kappa_1.\,\sigma_1$ then ξ_2 is $\forall\, a{:}\,\kappa_2.\,\sigma_2$.*
4. *If ξ_1 is $\forall\, c{:}\,\phi_1.\,\sigma_1$ then ξ_2 is $\forall\, c{:}\,\phi_2.\,\sigma_2$.*

Our approach to prove consistency is similar to previous versions of FC, with two key differences: we work in an implicitly typed version of our system, and we have had to restrict the rule CT_ALLC for forming coercions between types that abstract coercions.

At a high level, our consistency argument proceeds in four steps.

1. We define an *implicitly coerced* version of the language, where coercion proofs have been erased. By working with the implicit language, our results don't depend on specific proofs. Derivations in the explicit language can be translated to derivations in the implicit language.
2. We define a *rewrite relation* that reduces types in the implicit system by firing axioms in the context.
3. We give a sufficient condition, which we write $\mathbf{Good}\,\Gamma$, for a context to be consistent.
4. We argue that in good contexts, joinability of the rewrite relation is complete with respect to the implicit coercion proof system. Since the rewrite relation preserves the head form of value types, this gives consistency for both the implicit and explicit systems.

5.1 Implicit language

Similar to surface Haskell, the implicitly typed language elides coercion proofs and casts from the type language. Concretely, we have judgements:

$\models \Gamma$	Implicit context validity
$\Gamma \models \tau \ : \ \kappa$	Implicit type/kind validity
$\Gamma \models \gamma \ : \ \phi$	Implicit coercion validity
$\Gamma \models \Delta \rightsquigarrow \Theta$	Implicit telescoped coercion validity

These judgements apply to the same forms as their explicit analogues. However, in the implicit system, we add a new a new form $\bullet$ to the syntax of coercions (γ), to mark an elided coercion proof. Another difference between the syntax is that since the implicit system erases casts, the system is not syntax directed—a given type may have several syntactically different kinds.

The main differences between the types in the implicit and explicit systems are the following two rules for type formation. In the former, the kinds of types may be coerced at any time. In the latter, the coercion in an application is erased to $\bullet$.

$$\frac{\Gamma \models \tau \ : \ \kappa \quad \Gamma \models \gamma \ : \ \kappa \sim \kappa' \quad \Gamma \models \kappa' \ : \ \star}{\Gamma \models \tau \ : \ \kappa'} \ \text{IT\_CAST}$$

$$\frac{\Gamma \models \tau \ : \ \forall\, c{:}\,\phi.\,\kappa \quad \Gamma \models \gamma \ : \ \phi}{\Gamma \models \tau\,\bullet \ : \ \kappa} \ \text{IT\_CAPP}$$

We define coercion proofs between erased types in a similar fashion. Most of the rules are the same as their counterparts in the explicitly typed system, but here there are three major differences.

The first is that the implicit language does not include a coherence rule. In the explicit language, given a coercion proof $\Gamma \vdash_{\mathsf{co}} \gamma \ : \ \tau \sim \tau'$, the coherence rule was used to construct a proof $\gamma \rhd \gamma'$ where the kind of the first type τ, had been changed, by applying a cast $\tau \rhd \gamma'$. However, in the implicit language, we can change the kind of τ by using IT_CAST to *implicitly* cast the kind of τ using coercion γ'. Therefore, we don't need a coherence form in the implicit coercion language.

The second difference is in the rule for coercion application compatibility:

$$\frac{\begin{array}{l}\Gamma \models \gamma \ : \ \tau \sim \tau' \\ \Gamma \models \tau\,\bullet \ : \ \kappa \quad \Gamma \models \tau'\,\bullet \ : \ \kappa'\end{array}}{\Gamma \models \gamma(\bullet, \bullet) \ : \ \tau\,\bullet \sim \tau'\,\bullet} \ \text{ICT\_CAPP}$$

This rule says that if two coercion applications (with proofs erased) are well formed, then if the two coercion abstractions are equal (in the implicit language), there is a proof that the two applications are equal.

The final difference is in the rule for coercions between coercion abstractions:

$$\frac{\begin{array}{l}\Gamma \models \eta \ : \ \phi_1 \sim \phi_2 \quad c \overset{\bullet}{\mapsto} (c_1, c_2) \\ c_1 \,\#\, \gamma \quad c_2 \,\#\, \gamma \\ \Gamma,\ c_1{:}\,\phi_1,\ c_2{:}\,\phi_2 \models \gamma \ : \ \tau_1 \sim \tau_2 \\ \Gamma \models \forall\, c_1{:}\,\phi_1.\,\tau_1 \ : \ \star \quad \Gamma \models \forall\, c_2{:}\,\phi_2.\,\tau_2 \ : \ \star\end{array}}{\Gamma \models \forall\, c{:}\,\eta.\,\gamma \ : \ (\forall\, c_1{:}\,\phi_1.\,\tau_1) \sim (\forall\, c_2{:}\,\phi_2.\,\tau_2)} \ \text{ICT\_ALLC}$$

Note that we continue to require that the variables c_1 and c_2 not be used in the (erased) coercion proof γ. The motivation for this restriction is that when we introduce coercions into the context for the coercion abstraction rule, they may assert bogus equalities—we may assume a proof $c{:}\,\mathbf{Int} \sim \mathbf{Bool}$. Coercions that mention these spurious assumptions may equate types with different head forms. Therefore, we require that the coercions we are adding to the context, c_1 and c_2, do not show up in γ. This is the primary motivation for restricting the coercion abstraction equality rule in the explicit system as well.

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{co}} \eta \ : \ \phi_1 \sim \phi_2 \quad c \overset{\bullet}{\mapsto} (c_1, c_2) \\ c_1 \,\#\, |\gamma| \quad c_2 \,\#\, |\gamma| \\ \Gamma,\ c_1{:}\,\phi_1,\ c_2{:}\,\phi_2 \vdash_{\mathsf{co}} \gamma \ : \ \tau_1 \sim \tau_2 \\ \Gamma \vdash_{\mathsf{ty}} \forall\, c_1{:}\,\phi_1.\,\tau_1 \ : \ \star \quad \Gamma \vdash_{\mathsf{ty}} \forall\, c_2{:}\,\phi_2.\,\tau_2 \ : \ \star\end{array}}{\Gamma \vdash_{\mathsf{co}} \forall\, c{:}\,\eta.\,\gamma \ : \ (\forall\, c_1{:}\,\phi_1.\,\tau_1) \sim (\forall\, c_2{:}\,\phi_2.\,\tau_2)} \ \text{CT\_ALLC}$$

We use the following definition to produce an erased type or coercion:

Definition 5.3 (Coercion Erasure)**.** *Given an explicitly typed term σ, we define the* erasure *of σ, denoted $|\sigma|$, by induction on the form of σ. The interesting cases are casting ($\tau \rhd \gamma$) and coercion*

$\boxed{\Gamma \models \tau \rightsquigarrow \tau'}$ Type parallel reduction

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

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

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

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

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

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

$$\frac{\Gamma \models \tau \rightsquigarrow \tau'}{\Gamma \models \tau \bullet \rightsquigarrow \tau' \bullet} \quad \text{TS\_CAPP}$$

Figure 9. Rewrite relation

application $(\tau_1\, \gamma)$.

$$\begin{array}{lcl} |\tau \rhd \gamma| & = & |\tau| \\ |\tau\, \gamma| & = & |\tau| \bullet \end{array}$$

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

Likewise, given an explicitly typed coercion proof γ, we define the erasure *of γ, denoted $|\gamma|$, by induction on the form of γ. The interesting cases are coercion application, coherence and coercion instantiation.*

$$\begin{array}{lcl} |\gamma(\gamma_1, \gamma_2)| & = & |\gamma|(\bullet, \bullet) \\ |\gamma \rhd \gamma'| & = & |\gamma| \\ |\gamma @(\gamma', \gamma'')| & = & |\gamma| @(\bullet, \bullet) \end{array}$$

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

Lemma 5.4 (Erasure is type preserving)**.**

1. *If $\vdash_{\mathsf{wf}} \Gamma$ then $\models |\Gamma|$.*
2. *If $\Gamma \vdash_{\mathsf{ty}} \tau \;:\; \kappa$ then $|\Gamma| \models |\tau| \;:\; |\kappa|$.*
3. *If $\Gamma \vdash_{\mathsf{co}} \gamma \;:\; \phi$ then $|\Gamma| \models |\gamma| \;:\; |\phi|$.*
4. *If $\Gamma \vdash_{\mathsf{tc}} \Delta \leftrightsquigarrow \Theta$ then $|\Gamma| \models |\Delta| \leftrightsquigarrow |\Theta|$.*

5.2 Rewrite relation

Next, we give a non-deterministic rewrite relation on types in Figure 9. Rewriting works with open terms in the implicit language, and it preserves the head form of value types. From this rewrite relation, we define a joinability relation, written $\Gamma \models \sigma_1 \Leftrightarrow \sigma_2$, if both σ_1 and σ_2 can multistep to a common reduct.

5.3 Good contexts

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 for an *erased* context to be consistent, written $\mathbf{Good}\,\Gamma$.

Definition 5.5 (Good contexts)**.** *We have $\mathbf{Good}\,\Gamma$ when the following conditions hold:*

1. *All coercion assumptions and axioms in Γ are of the form $C{:}\, \forall\, \Delta.\, (F\, \overline{\tau} \sim \tau')$ or of the form $c{:}\, a \sim \tau$.*
In the first form, the arguments to the type function must behave like patterns. For every well kinded $\overline{\rho}$, every $\tau_i \in \overline{\tau}$ and every $\tau_i' \in \overline{\tau'}$ such that $\Gamma \models \tau_i[\overline{\rho}/\Delta] \rightsquigarrow \tau_i'$, it must be $\tau_i' = \tau_i[\overline{\rho'}/\Delta]$ for some $\overline{\rho}'$ with $\Gamma \models \tau_m \rightsquigarrow \tau_m'$ for each $\tau_m \in \overline{\rho}$.
2. *There is no overlap between axioms and coercion assumptions. For each a, there is at most one assumption of the form $c{:}\, a \sim \tau$ in the context. For each $F\, \overline{\rho}$ there exists at most one prefix $\overline{\rho}_1$ of $\overline{\rho}$ such that there exist C, σ and Θ where $\Gamma \models C\, \Theta \;:\; (F\, \overline{\rho_1} \sim \sigma)$. This C is unique for every matching $F\, \overline{\tau_1}$.*
3. *Axioms equate types of the same kind. For each $C{:}\, \forall\, \Delta.\, (F\, \overline{\tau} \sim \tau')$ in Γ, the kinds of each side must match i.e. $\Gamma, \Delta \models F\, \overline{\tau} \;:\; \kappa$ and $\Gamma, \Delta \models \tau' \;:\; \kappa$ and that kind must not mention bindings in the telescope, $\Gamma \models \kappa \;:\; \star$.*

5.4 Consistency

In the rest of this section, we sketch the proof that good contexts are consistent. Our approach is similar to previous work [Weirich et al. 2010], but differs in two ways. First, the rewrite relation works on types in the implicit language. Second, the rewrite relation is not type directed: we maintain only the set of top level axioms in the context while rewriting.

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. The proofs of these lemmas as well as the completeness theorem appear in the appendix.

Theorem 5.6 (Local diamond property)**.** *If $\mathbf{Good}\,\Gamma$, $\Gamma \models \sigma \rightsquigarrow \sigma_1$, and $\Gamma \models \sigma \rightsquigarrow \sigma_2$ then there exists a σ_3 such that $\Gamma \models \sigma_1 \rightsquigarrow \sigma_3$ and $\Gamma \models \sigma_2 \rightsquigarrow \sigma_3$.*

Lemma 5.7 (Substitution)**.** *If $\mathbf{Good}\,\Gamma$, $\Gamma \models \sigma \rightsquigarrow^* \sigma'$, and $\Gamma \models \tau \rightsquigarrow^* \tau'$, then if a appears free in σ and σ', we have $\Gamma \models \sigma[\tau/a] \Leftrightarrow \sigma'[\tau'/a]$.*

From these lemmas we see that joinability is complete. In the following, the proposition $\mathit{fcv}(\gamma) \subseteq \mathit{dom}\,\Gamma'$ indicates that all coercion variables and axioms used in γ are in the domain of Γ'. The similar proposition $\mathit{fcv}(\Theta) \subseteq \mathit{dom}\,\Gamma'$ indicates the same for a telescoped coercion Θ.

Theorem 5.8 (Completeness)**.**

1. *Suppose that $\Gamma \models \gamma \;:\; \sigma_1 \sim \sigma_2$, and $\mathit{fcv}(\gamma) \subseteq \mathit{dom}\,\Gamma'$ for some subcontext Γ' satisfying $\mathbf{Good}\,\Gamma'$. Then $\Gamma \models \sigma_1 \Leftrightarrow \sigma_2$.*
2. *Suppose that $\Gamma \models \Delta \leftrightsquigarrow \Theta$, and $\mathit{fcv}(\Theta) \subseteq \mathit{dom}\,\Gamma'$ for some subcontext Γ' satisfying $\mathbf{Good}\,\Gamma'$. Then for each $a{:}\, \kappa \mapsto (\tau_1, \tau_2, \gamma) \in \Theta$, we have $\Gamma \vdash \tau_1 \Leftrightarrow \tau_2$.*

Theorem 5.9 (Consistency)**.** *If $\mathbf{Good}\,|\Gamma|$ then Γ is consistent.*

Proof. Suppose $\Gamma \vdash_{\mathsf{co}} \gamma \;:\; \xi_1 \sim \xi_2$. Then, we have that $\Gamma \models |\gamma| \;:\; |\xi_1| \sim |\xi_2|$. By completeness, we have that those two types are joinable. There is some σ such that $\Gamma \models |\xi_1| \rightsquigarrow^* \sigma$ and $\Gamma \models |\xi_2| \rightsquigarrow^* \sigma$. However, by inversion on the rewriting relation, we see that it preserves the head forms of value types (since there exist no axioms for those by the first condition of $\mathbf{Good}\,|\Gamma|$). Also, we know that erasure preserves head forms. Thus, ξ_1 and ξ_2 (and σ) have the same head form. □

6. Discussion

In this section we discuss aspect of the design and relate it to existing systems.

Collapsing kinds and types Blurring the distinction between types and kinds is convenient, but is it wise? It is well known that type systems that include the $\Gamma \vdash_{\mathsf{ty}} \star : \star$ rule are inconsistent logics [Girard 1972]. Does that cause trouble here?

The answer is no because FC, even without these extensions, is already inconsistent. Inconsistency means that all kinds (of type $\star$) are inhabited. FC is extensible, so datatypes and type functions can be declared at any such kind. Furthermore, with the addition of kind polymorphism, the GHC standard library defines the kind-polymorphic type constant Any, which can be used at any kind.

It is not clear whether adding the $\Gamma \vdash_{\mathsf{ty}} \star : \star$ rule to source Haskell would cause type inference to loop, as the type language does not include anonymous abstractions. However, even in the presence of nonterminating type expressions, the only danger is to the decidability of type inference. Once a type equality has been discovered by the constraint solver, it is elaborated to a finite equality proof. In FC, type checking is always decidable.

Even though the FC language combines types and kinds, the Haskell source language need not do so. Even if predictable type inference algorithms require more traditional stratification, a distinction between types can kinds can be eliminated in the translation to the core language. This situation is not new—the desires of a simple core language have already lead FC to be more expressive than source Haskell.

Other languages that adopt dependent types and the "type-in-type" axiom [Augustsson 1998; Cardelli 1986] do not have decidable type checking. These languages do not make same distinctions as FC does between run-time expressions and compile-time types, separating logically inconsistent types from logically consistent equality proofs. The coercion language is limited in expressive power and the consistency of this language (i.e. that there are equalities that cannot be derived) is a consequence of this limitation. This interplay between an inconsistent programming language and a consistent metalogic is also the subject of current research in the Trellys project [Casinghino et al. 2012; Kimmell et al. 2012].

Heterogeneous equality Heterogeneous equality is a necessary part of this system. Even though equality proofs may only used for casting when both sides have kind $\star$, heterogeneous equalities are needed for intermediate results.

One motivation for heterogeneous equality is the coherence rule (CT_Coh), which 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 equivalence rules analogous to the many "push" rules of the operational semantics.

Heterogeneous equality is also motivated by the presence of dependent application (such as rules K_Inst and K_Capp), where the kind of the result depends on the value of the argument. We would like type equivalence to be compatible with respect to application, as is demonstrated by rule CT_App. However, if all equalities are required to be homogeneous, then not all uses of rule are valid because the result kinds may differ.

For example, consider the datatype TypeRep of kind $\forall\, a{:}\ \star\ .\ \forall\, b{:}\ \star\ .\ \star$. If we have coercions $\Gamma \vdash_{\mathsf{co}} \gamma_1 : \star \sim \kappa$ and $\Gamma \vdash_{\mathsf{co}} \gamma_2 : \mathbf{Int} \sim \tau$, then we can construct the proof

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

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

There are several choices in the semantics of heterogeneous equality. We have chosen the most popular, where a proposition $\sigma_1 \sim \sigma_2$ is interpreted as a conjunction: "the types are equal and their kinds are equal". This semantics is similar to Epigram 1 [McBride 2002], the `HeterogeneousEquality` module in the Agda standard library [3], and the treatment in Coq [4]. Epigram 2 [Altenkirch et al. 2007] uses an alternative semantics, interpreted as "if the kinds are equal than the types are equal". 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 arise from differences in the overall type system. The syntax-directed types system of FC make the conjunctive interpretation the most reasonable, whereas the bidirectional type system of Epigram 2 makes the implicational version more convenient. Trellys terms can be given many different, inequivalent types, so that language uses a type-independent equality.

There is also a choice to be made about whether equality is relevant. The coherence axiom is inspired by observational type theory. Unlike higher-dimensional type theory [Licata and Harper 2012], equality in this language has no computational content. Because of the separation between objects and proofs, FC is resolutely one-dimensional. We do not define what it means for proofs to be equivalent. Instead, we ensure that in any context the identity of equality proofs is unimportant.

The* CT_AllC *rule and consistency proof The rule CT_AllC restricts how the coercion variables c_1 and c_2 can be used in a proof that $\forall\, c_1{:}\, \phi_1.\, \tau_1$ is equal to $\forall\, c_2{:}\, \phi_2.\, \tau_2$. This restriction is motivated by our consistency proof. The proof first defines what it means for a set of assumptions to be good, and then defines a rewriting system that is complete for good sets of assumptions. However, this rule causes trouble for that plan—we do not know whether ϕ_1 and ϕ_2 can be added to the current set of good assumptions. Our solution is to revise the statement of completeness so that not all coercion assumptions need to be good. If the assumptions are not needed for rewriting then we do not need any restrictions on them. The ones that are not needed for rewriting are the ones that do not show up in the erased coercion. (They may be used implicitly to verify that types are valid.)

The consequence of these restrictions is that there are some types that cannot be shown equivalent. For example, there is no proof of equivalence between the types $\forall\, c_1{:}\, \mathbf{Int} \sim b.\, \mathbf{Int}$ and $\forall\, c_2{:}\, \mathbf{Int} \sim b.\, b$. A coercion between these two types would need to use c_1 or c_2. However, this lack of expressiveness is not significant. In source Haskell, it would show up only through uses of first-class polymorphism. Furthermore, this restriction already exists in GHC. GHC currently does not allow coercions between the types $(\mathbf{Int} \sim b) \Rightarrow \mathbf{Int}$ and $(\mathbf{Int} \sim b) \Rightarrow b$.

Nevertheless, the restrictions on c_1 and c_2 in this version of FC are due to the proof technique that we have employed. It is possible that a completely different consistency proof would validate a rule that does not restrict the use of these variables. However, we leave this alternative proof to future work.

The implicit language Our proof technique for the consistency proof, which is 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 eliminate them entirely. Following ICC* [Barras and Bernardo 2008], we could alternatively view the implicit language as the "real" semantics for FC, and then 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.

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

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

However, although the implicit language allows type equality assumptions to be used implicitly, it is not the same as extensional type theory (ETT) [Martin-Löf 1984]. Foremost, it separates proofs from programs so that it can weaken one (ensuring consistency) while enriching the other (with "type-in-type"). The proof language is not as expressive as that of ETT, but it is expressive enough for Haskell. We have discussed the limitations on equalities between coercion abstractions above. Another way in which the proof language is weaker than ETT is the lack of η-equivalence or extensional reasoning for type-level functions.

7. 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 with GHC's constraint solving algorithm. Although the interaction between dependent types and type inference brings new research challenges, these challenges can be addressed in the context of a firm semantic basis.

Acknowledgements

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

References

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

L. Augustsson. Cayenne—a language with dependent types. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, ICFP '98, pages 239–250, New York, NY, USA, 1998. ACM. doi: 10.1145/289423.289451.

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

B. Barras and B. Bernardo. The implicit calculus of constructions as a programming language with dependent types. In *Proceedings of the Theory and practice of software, 11th international conference on Foundations of software science and computational structures*, FOSSACS'08/ETAPS'08, pages 365–379, Berlin, Heidelberg, 2008. Springer-Verlag.

L. Cardelli. A polymorphic lambda calculus with type:type. Technical Report 10, 1986.

C. Casinghino, V. Sjöberg, and S. Weirich. Step-indexed normalization for a language with general recursion. In *Fourth workshop on Mathematically Structured Functional Programming (MSFP '12)*, 2012.

M. M. T. Chakravarty, G. Keller, and S. Peyon Jones. Associated type synonyms. In *Proceedings of the tenth ACM SIGPLAN International Conference on Functional Programming*, ICFP '05, pages 241–253, New York, NY, USA, 2005. ACM. ISBN 1-59593-064-7.

Galois, Inc. *Cryptol Reference Manual*. 2002.

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

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

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

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. To appear.

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

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

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

U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

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

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

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

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

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

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

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

S. Weirich. RepLib: A library for derivable type classes. In *Haskell Workshop*, pages 1–12, Portland, OR, USA, Sept. 2006.

S. Weirich, D. Vytiniotis, S. Peyton Jones, and S. Zdancewic. Generative type abstraction and type-level computation (extended version). Technical report, University of Pennsylvania, Nov. 2010. URL `http://www.cis.upenn.edu/~sweirich/papers/newtypes.pdf`.

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

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

A. Additional semantics

Below, we list a few syntactic forms, rules and definitions not included in the main discussion.

A.1 Grammars

$$\begin{array}{lll} \rho & ::= & & \text{Telescope argument} \\ & \mid & \tau & \\ & \mid & \gamma & \end{array}$$

$$\begin{array}{lll} v & ::= & & \text{Values} \\ & \mid & \lambda x{:}\,\tau.\, e & \\ & \mid & \Lambda a{:}\,\kappa.\, e & \\ & \mid & \lambda c{:}\,\phi.\, e & \\ & \mid & K\,\overline{\tau}\,\overline{\rho}\,\overline{e} & \end{array}$$

$$\begin{array}{lll} cv & ::= & & \text{Coerced values} \\ & \mid & v \rhd \gamma & \\ & \mid & v & \end{array}$$

$$\begin{array}{lll} tbnd & ::= & & \text{Telescoped Coercion Binding} \\ & \mid & a{:}\,\kappa \mapsto (\tau_1, \tau_2, \gamma) & \\ & \mid & c{:}\,\phi \mapsto (\gamma_1, \gamma_2) & \end{array}$$

$$\begin{array}{lll} pbnd & ::= & & \text{Lifting Binding} \\ & \mid & a{:}\,\kappa \overset{\bullet}{\mapsto} (\tau_1, \tau_2, \gamma) & \\ & \mid & c{:}\,\phi \overset{\bullet}{\mapsto} (\gamma_1, \gamma_2) & \end{array}$$

$$\begin{array}{lll} \Theta & ::= & & \text{Telescoped Coercion} \\ & \mid & \varnothing & \\ & \mid & \Theta, tbnd & \end{array}$$

$$\begin{array}{lll} \Psi & ::= & & \text{Lifting Context} \\ & \mid & \varnothing & \\ & \mid & \Theta, tbnd & \\ & \mid & \Psi, pbnd & \end{array}$$

A.2 Expression typing and operational semantics

$\boxed{\Gamma \vdash_{\mathsf{tm}} e \;:\; \tau}$ Expression typing

$$\frac{\vdash_{\mathsf{wf}} \Gamma \quad x{:}\,\tau \in \Gamma}{\Gamma \vdash_{\mathsf{tm}} x \;:\; \tau}\ \textsc{T\_Var}$$

$$\frac{\Gamma, x{:}\,\tau_1 \vdash_{\mathsf{tm}} e \;:\; \tau_2}{\Gamma \vdash_{\mathsf{tm}} \lambda x{:}\,\tau_1.\, e \;:\; \tau_1 \to \tau_2}\ \textsc{T\_Abs}$$

$$\frac{\Gamma \vdash_{\mathsf{tm}} e \;:\; \tau_1 \to \tau_2 \quad \Gamma \vdash_{\mathsf{tm}} u \;:\; \tau_1}{\Gamma \vdash_{\mathsf{tm}} e\, u \;:\; \tau_2}\ \textsc{T\_App}$$

$$\frac{\Gamma, c{:}\,\phi \vdash_{\mathsf{tm}} e \;:\; \tau \quad \Gamma \vdash_{\mathsf{ty}} \phi \;:\; \star}{\Gamma \vdash_{\mathsf{tm}} \lambda c{:}\,\phi.\, e \;:\; \forall\, c{:}\,\phi.\, \tau}\ \textsc{T\_CAbs}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{tm}} e \;:\; \forall\, c{:}\,\phi.\, \tau \\ \Gamma \vdash_{\mathsf{co}} \gamma \;:\; \phi\end{array}}{\Gamma \vdash_{\mathsf{tm}} e\, \gamma \;:\; \tau[\gamma/c]}\ \textsc{T\_CApp}$$

$$\frac{\Gamma, a{:}\,\kappa \vdash_{\mathsf{tm}} e \;:\; \tau}{\Gamma \vdash_{\mathsf{tm}} \Lambda a{:}\,\kappa.\, e \;:\; \forall\, a{:}\,\kappa.\, \tau}\ \textsc{T\_TAbs}$$

$$\frac{\Gamma \vdash_{\mathsf{tm}} e \;:\; \forall\, a{:}\,\kappa.\, \tau \quad \Gamma \vdash_{\mathsf{ty}} \tau' \;:\; \kappa}{\Gamma \vdash_{\mathsf{tm}} e\, \tau' \;:\; \tau[\tau'/a]}\ \textsc{T\_TApp}$$

$$\frac{\Gamma \vdash_{\mathsf{tm}} e \;:\; \tau_1 \quad \Gamma \vdash_{\mathsf{co}} \gamma \;:\; \tau_1 \sim \tau_2 \quad \Gamma \vdash_{\mathsf{ty}} \tau_2 \;:\; \star}{\Gamma \vdash_{\mathsf{tm}} e \rhd \gamma \;:\; \tau_2}\ \textsc{T\_Cast}$$

$$\frac{\vdash_{\mathsf{wf}} \Gamma \quad K{:}\,\tau \in \Gamma}{\Gamma \vdash_{\mathsf{tm}} K \;:\; \tau}\ \textsc{T\_Con}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{tm}} e \;:\; T\,\overline{\tau'} \\ \text{for each } i \\ \quad K_i{:}\,\forall\,\overline{a{:}\,\kappa}.\,\forall\,\Delta_i.\,\overline{\sigma_i} \to (T\,\overline{a}) \in \Gamma \\ \quad \overline{\Delta_i' = \Delta_i[\overline{\tau'/a}]} \\ \quad \sigma_i' = \sigma_i[\overline{\tau'}/\overline{a}] \\ \quad \Gamma, \Delta_i', \overline{x_i{:}\,\sigma_i'} \vdash_{\mathsf{tm}} u_i \;:\; \tau\end{array}}{\Gamma \vdash_{\mathsf{tm}} \mathbf{case}\ e\ \mathbf{of}\ \overline{K_i\ \Delta_i'\ \overline{x_i{:}\,\sigma_i'} \to u_i} \;:\; \tau}\ \textsc{T\_Case}$$

$\boxed{e \longrightarrow e'}$ Step reduction, parameterized by toplevel context

$$\frac{}{(\lambda x{:}\,\tau.\, e)\, e' \longrightarrow e[e'/x]}\ \textsc{S\_Beta}$$

$$\frac{e_1 \longrightarrow e_1'}{e_1\, e_2 \longrightarrow e_1'\, e_2}\ \textsc{S\_EApp}$$

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

$$\frac{}{(\Lambda a{:}\,\kappa.\, e)\, \tau \longrightarrow e[\tau/a]}\ \textsc{S\_TBeta}$$

$$\frac{e_1 \longrightarrow e_1'}{e_1\, \sigma \longrightarrow e_1'\, \sigma}\ \textsc{S\_TApp}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{co}} \gamma \;:\; \forall\, a{:}\,\kappa_1.\, \sigma_1 \sim \forall\, a{:}\,\kappa_2.\, \sigma_2 \\ \gamma' = \mathbf{sym}\, (\mathbf{nth}^1\, \gamma) \\ \tau' = \tau \rhd \gamma'\end{array}}{(v \rhd \gamma)\, \tau \longrightarrow (v\, \tau') \rhd \gamma@(\langle\tau\rangle \rhd \gamma')}\ \textsc{S\_TPush}$$

$$\frac{}{(\lambda c{:}\,\sigma_1 \sim \sigma_2.\, e)\, \gamma \longrightarrow e[\gamma/c]}\ \textsc{S\_CBeta}$$

$$\frac{e_1 \longrightarrow e_1'}{e_1\, \gamma \longrightarrow e_1'\, \gamma}\ \textsc{S\_CApp}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{co}} \gamma \;:\; (\forall\, c{:}\,\phi.\, \tau) \sim (\forall\, c'{:}\,\phi'.\, \tau') \\ \gamma'' = (((\mathbf{nth}^3\, (\mathbf{nth}^1\, \gamma)) \fatsemi \gamma') \fatsemi (\mathbf{sym}\, (\mathbf{nth}^4\, (\mathbf{nth}^1\, \gamma))))\end{array}}{(v \rhd \gamma)\, \gamma' \longrightarrow v\, \gamma'' \rhd \gamma@(\gamma'', \gamma')}\ \textsc{S\_CPush}$$

$$\frac{}{(v \rhd \gamma_1) \rhd \gamma_2 \longrightarrow v \rhd (\gamma_1 \fatsemi \gamma_2)}\ \textsc{S\_Comb}$$

$$\frac{e \longrightarrow e'}{e \rhd \gamma \longrightarrow e' \rhd \gamma}\ \textsc{S\_Coerce}$$

$$\frac{K_i\ \Delta_i\ \overline{x_i{:}\,\sigma_i} \to u_i \in \overline{p \to u}}{\mathbf{case}\ K_i\,\overline{\tau}\,\overline{\rho}\,\overline{e}\ \mathbf{of}\ \overline{p \to u} \longrightarrow u_i\,[\overline{e}/\overline{x_i}]\,[\overline{\rho}/\Delta_i]}\ \textsc{S\_CaseMatch}$$

$$\frac{e \longrightarrow e'}{\mathbf{case}\ e\ \mathbf{of}\ \overline{p \to u} \longrightarrow \mathbf{case}\ e'\ \mathbf{of}\ \overline{p \to u}}\ \textsc{S\_Case}$$

$$\frac{\begin{array}{l}K{:}\,\forall\,\overline{a{:}\,\kappa}.\,\forall\,\Delta.\,\overline{\sigma} \to (T\,\overline{a}) \in \Gamma \\ \Theta = \{\gamma\} \prec \overline{\rho} : \Delta \\ \overline{\tau'} = \Theta_2(\overline{a}) \\ \overline{\rho}' = \Theta_2(dom\,\Delta) \\ \text{for each } e_i \in \overline{e}, \\ \quad e_i' = e_i \rhd \Theta(\sigma_i)\end{array}}{\begin{array}{c}\mathbf{case}\ ((K\,\overline{\tau}\,\overline{\rho}\,\overline{e}) \rhd \gamma)\ \mathbf{of}\ \overline{p \to u} \to \\ \mathbf{case}\ (K\,\overline{\tau'}\,\overline{\rho}'\,\overline{e'})\ \mathbf{of}\ \overline{p \to u}\end{array}}\ \textsc{S\_KPush}$$

A.3 Erasure operation

$$\begin{array}{lcl}
|a| & = & a \\
|H| & = & H \\
|F| & = & F \\
|K| & = & K \\
|\forall\, a{:}\,\kappa.\,\tau| & = & \forall\, a{:}\,|\kappa|.\,|\tau| \\
|\forall\, c{:}\,\phi.\,\tau| & = & \forall\, c{:}\,|\phi|.\,|\tau| \\
|\tau_1\,\tau_2| & = & |\tau_1|\,|\tau_2| \\
|\tau_1 \rhd \gamma| & = & |\tau_1| \\
|\tau_1\,\gamma| & = & |\tau_1|\,\bullet \\
\\
|c| & = & c \\
|C\,\Theta| & = & C\,|\Theta| \\
|\langle\tau\rangle| & = & \langle|\tau|\rangle \\
|\mathbf{sym}\,\gamma| & = & \mathbf{sym}\,|\gamma| \\
|\gamma_1 \,⨾\, \gamma_2| & = & |\gamma_1| \,⨾\, |\gamma_2| \\
|\forall\, a{:}\,\eta.\,\gamma| & = & \forall\, a{:}\,|\eta|.\,|\gamma| \\
|\forall\, c{:}\,\eta.\,\gamma| & = & \forall\, c{:}\,|\eta|.\,|\gamma| \\
|\gamma_1\,\gamma_2| & = & |\gamma_1|\,|\gamma_2| \\
|\gamma(\gamma_1,\gamma_2)| & = & |\gamma|(\bullet,\bullet) \\
|\gamma \rhd \gamma'| & = & |\gamma| \\
|\gamma @ \gamma'| & = & |\gamma| @ |\gamma'| \\
|\gamma @ (\gamma',\gamma'')| & = & |\gamma| @ (\bullet,\bullet) \\
|\mathbf{nth}^i\,\gamma| & = & \mathbf{nth}^i\,|\gamma| \\
|\mathbf{kind}\,\gamma| & = & \mathbf{kind}\,|\gamma| \\
\\
|\varnothing| & = & \varnothing \\
|\Gamma,\ a{:}\,\kappa| & = & |\Gamma|,\ a{:}\,|\kappa| \\
|\Gamma,\ c{:}\,\phi| & = & |\Gamma|,\ c{:}\,|\phi| \\
\\
|\varnothing| & = & \varnothing \\
|\Theta, a{:}\,\kappa \mapsto (\tau_1,\tau_2,\gamma)| & = & |\Theta|, a{:}\,|\kappa| \mapsto (|\tau_1|,|\tau_2|,|\gamma|) \\
|\Theta, c{:}\,\phi \mapsto (\gamma_1,\gamma_2)| & = & |\Theta|, c{:}\,|\phi| \mapsto (\bullet,\bullet)
\end{array}$$

A.4 Implicit Language Typing

$\boxed{\models \Gamma}$ Implicit Validity

$$\frac{}{\models \varnothing}\ \textsc{IV\_Empty}$$

$$\frac{\Gamma \models \kappa\ :\ \star \quad a \# \Gamma}{\models \Gamma,\ a{:}\,\kappa}\ \textsc{IV\_TyVar}$$

$$\frac{\Gamma \models \kappa\ :\ \star \quad F \# \Gamma}{\models \Gamma,\ F{:}\,\kappa}\ \textsc{IV\_TyFun}$$

$$\frac{\Gamma \models \forall\,\overline{a{:}\,\kappa}.\,\star\ :\ \star \quad T \# \Gamma}{\models \Gamma,\ T{:}\,\forall\,\overline{a{:}\,\kappa}.\,\star}\ \textsc{IV\_TyData}$$

$$\frac{\Gamma \models \tau\ :\ \kappa \quad x \# \Gamma}{\models \Gamma,\ x{:}\,\tau}\ \textsc{IV\_Var}$$

$$\frac{\Gamma \models \forall\,\overline{a{:}\,\kappa}.\,\forall\,\Delta.\,(\overline{\sigma} \to T\ \overline{a})\ :\ \star \quad K \# \Gamma}{\models \Gamma,\ K{:}\,\forall\,\overline{a{:}\,\kappa}.\,\forall\,\Delta.\,(\overline{\sigma} \to T\ \overline{a})}\ \textsc{IV\_Con}$$

$$\frac{\Gamma \models \phi\ :\ \star \quad c \# \Gamma}{\models \Gamma,\ c{:}\,\phi}\ \textsc{IV\_CVar}$$

$$\frac{\Gamma, \Delta \models \phi\ :\ \star \quad C \# \Gamma}{\models \Gamma,\ C{:}\,\forall\,\Delta.\,\phi}\ \textsc{IV\_Ax}$$

$\boxed{\Gamma \models \Delta \leftrightsquigarrow \Theta}$ Implicit Lifting

$$\frac{\models \Gamma}{\Gamma \models \varnothing \leftrightsquigarrow \varnothing}\ \textsc{IL\_empty}$$

$$\frac{\begin{array}{c}\Gamma \models \Delta \leftrightsquigarrow \Theta \\ \Gamma \models \sigma_1\ :\ \Theta_1(\kappa) \\ \Gamma \models \sigma_2\ :\ \Theta_2(\kappa) \\ \Gamma \models \gamma\ :\ \sigma_1 \sim \sigma_2\end{array}}{\Gamma \models (\Delta,\ a{:}\,\kappa) \leftrightsquigarrow (\Theta, a{:}\,\kappa \mapsto (\sigma_1,\sigma_2,\gamma))}\ \textsc{IL\_ty}$$

$$\frac{\begin{array}{c}\Gamma \models \Delta \leftrightsquigarrow \Theta \\ \Gamma \models \eta_1\ :\ \Theta_1(\phi) \\ \Gamma \models \eta_2\ :\ \Theta_2(\phi)\end{array}}{\Gamma \models (\Delta,\ c{:}\,\phi) \leftrightsquigarrow (\Theta, c{:}\,\phi \mapsto (\bullet,\bullet))}\ \textsc{IL\_co}$$

$\boxed{\Gamma \models \tau\ :\ \kappa}$ Implicit Kinding

$$\frac{\models \Gamma}{\Gamma \models \star\ :\ \star}\ \textsc{IT\_StarInStar}$$

$$\frac{\models \Gamma}{\Gamma \models (\to)\ :\ \star \to \star \to \star}\ \textsc{IT\_Arrow}$$

$$\frac{\models \Gamma}{\Gamma \models (\sim)\ :\ \forall\, a{:}\,\star.\,\forall\, b{:}\,\star.\ a \to b \to \star}\ \textsc{IT\_Equal}$$

$$\frac{\models \Gamma \quad w{:}\,\kappa \in \Gamma}{\Gamma \models w\ :\ \kappa}\ \textsc{IT\_Var}$$

$$\frac{\Gamma \models \tau_1\ :\ \kappa_1 \to \kappa_2 \quad \Gamma \models \tau_2\ :\ \kappa_1}{\Gamma \models \tau_1\,\tau_2\ :\ \kappa_2}\ \textsc{IT\_App}$$

$$\frac{\Gamma \models \tau_1\ :\ \forall\, a{:}\,\kappa_1.\,\kappa_2 \quad \Gamma \models \tau_2\ :\ \kappa_1}{\Gamma \models \tau_1\,\tau_2\ :\ \kappa_2[\tau_2/a]}\ \textsc{IT\_TInst}$$

$$\frac{\Gamma \models \tau\ :\ \forall\, c{:}\,\phi.\,\kappa \quad \Gamma \models \gamma\ :\ \phi}{\Gamma \models \tau\,\bullet\ :\ \kappa}\ \textsc{IT\_CApp}$$

$$\frac{\Gamma,\ a{:}\,\kappa \models \tau\ :\ \star \quad \Gamma \models \kappa\ :\ \star}{\Gamma \models \forall\, a{:}\,\kappa.\,\tau\ :\ \star}\ \textsc{IT\_AllT}$$

$$\frac{\Gamma,\ c{:}\,\phi \models \tau\ :\ \star \quad \Gamma \models \phi\ :\ \star}{\Gamma \models \forall\, c{:}\,\phi.\,\tau\ :\ \star}\ \textsc{IT\_AllC}$$

$$\frac{\Gamma \models \tau\ :\ \kappa \quad \Gamma \models \gamma\ :\ \kappa \sim \kappa' \quad \Gamma \models \kappa'\ :\ \star}{\Gamma \models \tau\ :\ \kappa'}\ \textsc{IT\_Cast}$$

$\boxed{\Gamma \models \gamma\ :\ \phi}$ Implicit Coercion Typing

$$\frac{\begin{array}{c}\Gamma \models \gamma\ :\ \tau \sim \tau' \\ \Gamma \models \tau\,\bullet\ :\ \kappa \quad \Gamma \models \tau'\,\bullet\ :\ \kappa'\end{array}}{\Gamma \models \gamma(\bullet,\bullet)\ :\ \tau\,\bullet \sim \tau'\,\bullet}\ \textsc{ICT\_CApp}$$

$$\frac{\begin{array}{l}\Gamma \models \eta\ :\ \phi_1 \sim \phi_2 \quad c \overset{\bullet}{\mapsto} (c_1,c_2) \\ c_1 \# \gamma \quad c_2 \# \gamma \\ \Gamma,\ c_1{:}\,\phi_1,\ c_2{:}\,\phi_2 \models \gamma\ :\ \tau_1 \sim \tau_2 \\ \Gamma \models \forall\, c_1{:}\,\phi_1.\,\tau_1\ :\ \star \quad \Gamma \models \forall\, c_2{:}\,\phi_2.\,\tau_2\ :\ \star\end{array}}{\Gamma \models \forall\, c{:}\,\eta.\,\gamma\ :\ (\forall\, c_1{:}\,\phi_1.\,\tau_1) \sim (\forall\, c_2{:}\,\phi_2.\,\tau_2)}\ \textsc{ICT\_AllC}$$

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

$$\frac{\begin{array}{l}\Gamma \models \gamma\ :\ (\forall\, c{:}\,\sigma_1 \sim \sigma_2.\,\tau) \sim (\forall\, c'{:}\,\sigma_1' \sim \sigma_2'.\,\tau') \\ \Gamma \models \gamma_1\ :\ \sigma_1 \sim \sigma_2 \quad \Gamma \models \gamma_2\ :\ \sigma_1' \sim \sigma_2'\end{array}}{\Gamma \models \gamma @ (\bullet,\bullet)\ :\ \tau \sim \tau'}\ \textsc{ICT\_InstC}$$

$$\frac{\Gamma \models \tau\ :\ \kappa}{\Gamma \models \langle\tau\rangle\ :\ \tau \sim \tau}\ \textsc{ICT\_Refl}$$

$$\frac{\Gamma \models \gamma\ :\ \tau_1 \sim \tau_2}{\Gamma \models \mathbf{sym}\,\gamma\ :\ \tau_2 \sim \tau_1}\ \textsc{ICT\_Sym}$$

$$\frac{\Gamma \models \gamma_1 \ :\ \tau_1 \sim \tau_2 \quad \Gamma \models \gamma_2 \ :\ \tau_2 \sim \tau_3}{\Gamma \models \gamma_1 \fatsemi \gamma_2 \ :\ \tau_1 \sim \tau_3} \quad \textsc{ICT\_Trans}$$

$$\frac{\begin{array}{c}\Gamma \models \gamma_1 \ :\ \tau_1' \sim \tau_2' \quad \Gamma \models \gamma_2 \ :\ \tau_1 \sim \tau_2 \\ \Gamma \models \tau_1' \, \tau_1 \ :\ \kappa_1 \quad \Gamma \models \tau_2' \, \tau_2 \ :\ \kappa_2\end{array}}{\Gamma \models \gamma_1 \, \gamma_2 \ :\ \tau_1' \, \tau_1 \sim \tau_2' \, \tau_2} \quad \textsc{ICT\_App}$$

$$\frac{\begin{array}{c}\Gamma \models \eta \ :\ \kappa_1 \sim \kappa_2 \quad a \overset{\bullet}{\mapsto} (a_1, a_2, c) \\ \Gamma,\ a_1{:}\,\kappa_1,\ a_2{:}\,\kappa_2,\ c{:}\ a_1 \sim a_2 \models \gamma \ :\ \tau_1 \sim \tau_2 \\ \Gamma \models \forall\, a_1{:}\,\kappa_1 .\, \tau_1 \ :\ \star \quad \Gamma \models \forall\, a_2{:}\,\kappa_2 .\, \tau_2 \ :\ \star\end{array}}{\Gamma \models \forall\, a{:}\,\eta .\, \gamma \ :\ (\forall\, a_1{:}\,\kappa_1 .\, \tau_1) \sim (\forall\, a_2{:}\,\kappa_2 .\, \tau_2)} \quad \textsc{ICT\_AllT}$$

$$\frac{c{:}\,\phi \in \Gamma \quad \models \Gamma}{\Gamma \models c \ :\ \phi} \quad \textsc{ICT\_Var}$$

$$\frac{C{:}\ \forall\, \Delta .\, (\tau_1 \sim \tau_2) \in \Gamma \quad \Gamma \models \Delta \leftrightsquigarrow \Theta}{\Gamma \models C\ \Theta \ :\ \Theta_1(\tau_1) \sim \Theta_2(\tau_2)} \quad \textsc{ICT\_VarAx}$$

$$\frac{\Gamma \models \gamma \ :\ H\ \overline{\tau} \sim H\ \overline{\tau'}}{\Gamma \models \mathbf{nth}^i\ \gamma \ :\ \tau_i \sim \tau_i'} \quad \textsc{ICT\_Nth}$$

$$\frac{\Gamma \models \gamma_1 \ :\ (\forall\, a_1{:}\,\kappa_1 .\, \tau_1) \sim (\forall\, a_2{:}\,\kappa_2 .\, \tau_2)}{\Gamma \models \mathbf{nth}^1\ \gamma_1 \ :\ \kappa_1 \sim \kappa_2} \quad \textsc{ICT\_Nth1TA}$$

$$\frac{\Gamma \models \gamma \ :\ (\forall\, c{:}\,\phi .\, \tau) \sim (\forall\, c'{:}\,\phi' .\, \tau')}{\Gamma \models \mathbf{nth}^1\ \gamma \ :\ \phi \sim \phi'} \quad \textsc{ICT\_Nth1CA}$$

$$\frac{\Gamma \models \gamma \ :\ \tau_1 \sim \tau_2 \quad \Gamma \models \tau_1 \ :\ \kappa_2 \quad \Gamma \models \tau_2 \ :\ \kappa_2}{\Gamma \models \mathbf{kind}\, \gamma \ :\ \kappa_1 \sim \kappa_2} \quad \textsc{ICT\_Ext}$$

A.5 Telescope reduction

$\boxed{\Gamma \models \overline{\rho} \rightsquigarrow \overline{\rho}'}$ Telescope reduction

$$\frac{}{\Gamma \models \varnothing \rightsquigarrow \varnothing} \quad \textsc{RNil}$$

$$\frac{\Gamma \models \tau \rightsquigarrow \tau' \quad \Gamma \models \overline{\rho} \rightsquigarrow \overline{\rho}'}{\Gamma \models \overline{\rho}, \tau \rightsquigarrow \overline{\rho}', \tau'} \quad \textsc{RCons}$$

B. Preservation

This section presents the necessary details for the proof of the preservation theorem. The theorem itself is 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.

B.1 Lifting Contexts

Section 4 describes telescoped coercions Θ and refers to lifting contexts Ψ. This section expands upon lifting contexts, which are required to prove the lifting lemma (Lemma 4.2).

A lifting context Ψ is a telescoped coercion Θ with special mappings appended on the end. These special mappings are denoted with $\overset{\bullet}{\mapsto}$ instead of $\mapsto$ and map a variable to 3 (for types) or 2 (for coercions) fresh variables, analogous to the bijective functions also denoted with $\overset{\bullet}{\mapsto}$.

Throughout this section, the notation $\overset{?}{\mapsto}$ means either $\mapsto$ or $\overset{\bullet}{\mapsto}$.

The use of Ψ as a multisubstitution is a straightforward extension of the use of Θ:

Definition B.1 (Lifting context substitution)**.**

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

We can view these fresh bindings as a typing context:

Definition B.2 (Single flattening)**.** *The operation* $\dot{\Psi}_j$ *turns a lifting context into a typing context.*

1. *For each* $a{:}\,\kappa \overset{\bullet}{\mapsto} (a_1, a_2, c)$*, the context includes the binding* $\mathsf{a}j{:}\ \Psi_j(\kappa)$.
2. *For each* $c{:}\,\phi \overset{\bullet}{\mapsto} (c_1, c_2)$*, the context includes the binding* $\mathsf{c}j{:}\ \Psi_j(\phi)$.

Definition B.3 (Flattening)**.** *The operation* $\dot{\Psi}$ *turns a lifting context into a typing context.*

1. *For each* $a{:}\,\kappa \overset{\bullet}{\mapsto} (a_1, a_2, c)$*, the context includes the bindings* $a_1{:}\ \Psi_1(\kappa)$, $a_2{:}\ \Psi_2(\kappa)$, $c{:}\ a_1 \sim a_2$.
2. *For each* $c{:}\,\phi \overset{\bullet}{\mapsto} (c_1, c_2)$*, the context includes the bindings* $c_1{:}\ \Psi_1(\phi)$, $c_2{:}\ \Psi_2(\phi)$.

Lemma B.4 (Telescoped coercion domains)**.** *If* $\Gamma \vdash_{\mathsf{tc}} \Delta \leftrightsquigarrow \Theta$*, then the domain of* Δ *equals the set of variables mapped in* Θ.

Proof. Straightforward induction on the derivation of $\Gamma \vdash_{\mathsf{tc}} \Delta \leftrightsquigarrow \Theta$. $\square$

Lemma B.5 (Lifting context domains)**.** *If* $\Gamma \vdash_{\mathsf{lc}} \Delta \leftrightsquigarrow \Psi$*, then the domain of* Δ *equals the set of variables mapped in* Ψ.

Proof. Straightforward induction on the derivation of $\Gamma \vdash_{\mathsf{lc}} \Delta \leftrightsquigarrow \Psi$, using Lemma B.4 in the LC_THETA case. $\square$

The following lemma is a generalization of Lemma 3.4 (Telescoped Coercion Substitution) from Section 3.8 to lifting contexts.

Lemma B.6 (Lifting context substitution)**.** *Suppose* $\Gamma \vdash_{\mathsf{lc}} \Delta \leftrightsquigarrow \Psi$.

1. *If* $\Gamma, \Delta \vdash_{\mathsf{ty}} \tau \ :\ \kappa$ *then* $\Gamma, \dot{\Psi}_j \vdash_{\mathsf{ty}} \Psi_j(\tau) \ :\ \Psi_j(\kappa)$
2. *If* $\Gamma, \Delta \vdash_{\mathsf{co}} \gamma \ :\ \phi$ *then* $\Gamma, \dot{\Psi}_j \vdash_{\mathsf{co}} \Psi_j(\gamma) \ :\ \Psi_j(\phi)$
3. *If* $\Gamma, \Delta \vdash_{\mathsf{lc}} \Delta' \leftrightsquigarrow \Psi'$ *then* $\Gamma, \dot{\Psi}_j \vdash_{\mathsf{lc}} \Psi_j(\Delta') \leftrightsquigarrow \Psi_j(\Psi')$
4. *If* $\Gamma, \Delta \vdash_{\mathsf{tel}} \overline{\rho} : \Delta'$ *then* $\Gamma, \dot{\Psi}_j \vdash_{\mathsf{tel}} \Psi_j(\overline{\rho}) : \Psi_j(\Delta')$
5. *If* $\vdash_{\mathsf{wf}} \Gamma, \Delta$*, then* $\vdash_{\mathsf{wf}} \Gamma, \dot{\Psi}_j$

Proof. We proceed by mutual induction. However, we will need to strengthen the lemma even more to get a usable induction hypothesis. The stronger version of the lemma replaces Γ, Δ in the *if* clauses with Γ, Δ, Γ' and replaces the $\Gamma, \dot{\Psi}_j$ in the conclusions with $\Gamma, \dot{\Psi}_j, \Psi_j(\Gamma')$.

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

Case K_VAR: We know $\Gamma, \Delta, \Gamma' \vdash_{\mathsf{ty}} w \ :\ \kappa$, and by inversion, $\vdash_{\mathsf{wf}} \Gamma, \Delta, \Gamma'$ and $w{:}\,\kappa \in \Gamma, \Delta, \Gamma'$. We must show $\Gamma, \dot{\Psi}_j, \Psi_j(\Gamma') \vdash_{\mathsf{ty}} \Psi_j(w) \ :\ \Psi_j(\kappa)$.

We have two cases:

$w \in \mathit{dom}\,\Gamma$**:** Because $w \notin \mathit{dom}\,\Delta$, $\Psi_j(w) = w$. Furthermore, because κ appears in Γ, Δ, Γ' before any element in Δ is declared, we know that κ cannot refer to any variable declared in Δ. Therefore, $\Psi_j(\kappa) = \kappa$. By the induction hypothesis, $\vdash_{\mathsf{wf}} \Gamma, \dot{\Psi}_j, \Psi_j(\Gamma')$, and we can use rule K_VAR to get $\Gamma, \dot{\Psi}_j, \Psi_j(\Gamma') \vdash_{\mathsf{ty}} w \ :\ \kappa$ as desired.

$w \in \mathit{dom}\,\Delta$**:** By Lemma B.5, a mapping $w{:}\,\kappa \overset{?}{\mapsto} (\tau_1, \tau_2, \gamma)$ must exist in Ψ. Here, we have two further cases, depending on the nature of the mapping:

$\mapsto$**:** Inverting $\Gamma \vdash_{\mathsf{lc}} \Delta \rightsquigarrow \Psi$ eventually gives us $\Gamma \vdash_{\mathsf{ty}} \Psi_j(w) : \Psi_j(\kappa)$ (from rule TELCO_TY). Weakening then gives us $\Gamma, \dot\Psi_j, \Psi_j(\Gamma') \vdash_{\mathsf{ty}} \Psi_j(w) : \Psi_j(\kappa)$ as desired.

$\overset{\bullet}{\mapsto}$**:** By the definition of $\dot\Psi_j$, $w{:}\,\Psi_j(\kappa) \in \dot\Psi_j$. By the induction hypothesis, we can derive $\vdash_{\mathsf{wf}} \Gamma, \dot\Psi_j, \Psi_j(\Gamma')$. Then, we apply rule K_VAR to get $\Gamma, \dot\Psi_j, \Psi_j(\Gamma') \vdash_{\mathsf{ty}} \Psi_j(w) : \Psi_j(\kappa)$ as desired.

$w \in \mathit{dom}\,\Gamma'$**:** Because $w \notin \mathit{dom}\,\Delta$, $\Psi_j(w) = w$. Furthermore, we know $w{:}\,\kappa \in \Gamma'$ and therefore $w{:}\,\Psi_j(\kappa) \in \Psi_j(\Gamma')$. The induction hypothesis gives us $\vdash_{\mathsf{wf}} \Gamma, \dot\Psi_j, \Psi_j(\Gamma')$ and we can use rule K_VAR to derive $\Gamma, \dot\Psi_j, \Psi_j(\Gamma') \vdash_{\mathsf{ty}} \Psi_j(w) : \Psi_j(\kappa)$ as desired.

$w \notin \mathit{dom}\,\Delta$**:**

Case K_ALLC: We know $\Gamma, \Delta, \Gamma' \vdash_{\mathsf{ty}} \forall\, c{:}\,\phi.\,\tau : \star$, and by inversion, $\Gamma, \Delta, \Gamma',\ c{:}\,\phi \vdash_{\mathsf{ty}} \tau : \star$ and $\Gamma, \Delta, \Gamma' \vdash_{\mathsf{ty}} \phi : \star$. We must show $\Gamma, \dot\Psi_j, \Psi_j(\Gamma') \vdash_{\mathsf{ty}} \forall\, c{:}\,\Psi_j(\phi).\,\Psi_j(\tau) : \star$.
The induction hypothesis gives us $\Gamma, \dot\Psi_j, \Psi_j(\Gamma'),\ c{:}\,\Psi_j(\phi) \vdash_{\mathsf{ty}} \Psi_j(\tau) : \star$ (letting Γ' in the inductive step include the binding for c) and $\Gamma, \dot\Psi_j, \Psi_j(\Gamma') \vdash_{\mathsf{ty}} \Psi_j(\phi) : \star$. Thus, by rule K_ALLC, $\Gamma, \dot\Psi_j, \Psi_j(\Gamma') \vdash_{\mathsf{ty}} \forall\, c{:}\,\Psi_j(\phi).\,\Psi_j(\tau) : \star$ and we are done.

Case CT_ALLC: We know

$$\Gamma, \Delta, \Gamma' \vdash_{\mathsf{co}} \forall\, c{:}\,\eta.\,\gamma : (\forall\, c_1{:}\,\phi_1.\,\tau_1) \sim (\forall\, c_2{:}\,\phi_2.\,\tau_2)$$

and by inversion

$$\Gamma, \Delta, \Gamma' \vdash_{\mathsf{co}} \eta : \phi_1 \sim \phi_2$$
$$c \overset{\bullet}{\mapsto} (c_1, c_2)$$
$$c_1 \# |\gamma|$$
$$c_2 \# |\gamma|$$
$$\Gamma, \Delta, \Gamma',\ c_1{:}\,\phi_1,\ c_2{:}\,\phi_2 \vdash_{\mathsf{co}} \gamma : \tau_1 \sim \tau_2$$
$$\Gamma, \Delta, \Gamma' \vdash_{\mathsf{ty}} \forall\, c_1{:}\,\phi_1.\,\tau_1 : \star$$
$$\Gamma, \Delta, \Gamma' \vdash_{\mathsf{ty}} \forall\, c_2{:}\,\phi_2.\,\tau_2 : \star$$

We wish to show

$$\Gamma, \dot\Psi_j, \Psi_j(\Gamma') \vdash \forall\, c{:}\,\Psi_j(\eta).\,\Psi_j(\gamma) : (\forall\, c_1{:}\,\Psi_j(\phi_1).\,\Psi_j(\tau_1)) \sim (\forall\, c_2{:}\,\Psi_j(\phi_2).\,\Psi_j(\tau_2)).$$

To use rule CT_ALLC, we need to show

$$\Gamma, \dot\Psi_j, \Psi_j(\Gamma') \vdash_{\mathsf{co}} \Psi_j(\eta) : \Psi_j(\phi_1) \sim \Psi_j(\phi_2) \quad (1)$$
$$c \overset{\bullet}{\mapsto} (c_1, c_2) \quad (2)$$
$$c_1 \# |\Psi_j(\gamma)| \quad (3)$$
$$c_2 \# |\Psi_j(\gamma)| \quad (4)$$
$$\Gamma, \dot\Psi_j, \Psi_j(\Gamma'),\ c_1{:}\,\Psi_j(\phi_1),\ c_2{:}\,\Psi_j(\phi_2) \vdash_{\mathsf{co}} \Psi_j(\gamma) : \Psi_j(\tau_1) \sim \Psi_j(\tau_2) \quad (5)$$
$$\Gamma, \dot\Psi_j, \Psi_j(\Gamma') \vdash_{\mathsf{ty}} \forall\, c_1{:}\,\Psi_j(\phi_1).\,\Psi_j(\tau_1) : \star \quad (6)$$
$$\Gamma, \dot\Psi_j, \Psi_j(\Gamma') \vdash_{\mathsf{ty}} \forall\, c_2{:}\,\Psi_j(\phi_2).\,\Psi_j(\tau_2) : \star \quad (7)$$

We know (1), (5), (6), and (7) by the induction hypothesis. (Note that we extend Γ' for induction with (5).) We know (2) by inversion, above. We can derive (3) and (4) by noting that $\Psi_j(\cdot)$ and $|\cdot|$ commute with each other and that c_1, c_2 do not appear in Ψ. 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.

Case CT_VARAX: We know $\Gamma, \Delta, \Gamma' \vdash_{\mathsf{co}} C\ \Theta : \Theta_1(\tau_1) \sim \Theta_2(\tau_2)$, and by inversion, $C{:}\,\forall\,\Delta'.\,(\tau_1 \sim \tau_2) \in \Gamma, \Delta, \Gamma'$ and $\Gamma, \Delta, \Gamma' \vdash_{\mathsf{tc}} \Delta' \rightsquigarrow \Theta$. We must show

$$\Gamma, \dot\Psi_j, \Psi_j(\Gamma') \vdash_{\mathsf{co}} \Psi_j(C\ \Theta) : \Psi_j(\Theta_1(\tau_1) \sim \Theta_2(\tau_2))$$

or, equivalently,

$$\Gamma, \dot\Psi_j, \Psi_j(\Gamma') \vdash_{\mathsf{co}} \Psi_j(C)\,\Psi_j(\Theta) : \Psi_j(\Theta)_1(\Psi_j(\tau_1)) \sim \Psi_j(\Theta)_2(\Psi_j(\tau_2))$$

To use CT_VARAX to prove this fact, we need, in turn

$$\Psi_j(C) : \forall \Delta''.(\Psi_j(\tau_1) \sim \Psi_j(\tau_2)) \in \Gamma, \dot\Psi_j, \Psi_j(\Gamma')$$
$$\Gamma, \dot\Psi_j, \Psi_j(\Gamma') \vdash_{\mathsf{tc}} \Delta'' \rightsquigarrow \Psi_j(\Theta)$$

Choose $\Delta'' = \Psi_j(\Delta')$. Then, the induction hypothesis gives us the second fact above.
By the definition of the form of Ψ, we can see that no axiom schemes (with type $\forall\,\Delta.\,\phi$) can be mapped from Ψ. We now have two cases:

$C \in \mathit{dom}\,\Gamma$**:** Because C appears in Γ, the type of C cannot mention any variables in Δ. Thus, $\Psi_j(\Delta') = \Delta'$, $\Psi_j(\tau_1) = \tau_1$ and $\Psi_j(\tau_2) = \tau_2$. Then, we can conclude that $C{:}\,\forall\,\Delta'.\,(\tau_1 \sim \tau_2) \in \Gamma, \dot\Psi$ and we are done.

$C \in \mathit{dom}\,\Gamma'$**:** In this case, we can conclude that

$$C{:}\,\forall\,\Psi_j(\Delta').\,(\Psi_j(\tau_1) \sim \Psi_j(\tau_2)) \in \Psi_j(\Gamma')$$

and we are done.

□

We will need the following lemmas to prove the lifting lemma:

Lemma B.7 (Telescoped coercion coercions)**.** *If* $\Gamma \vdash_{\mathsf{tc}} \Delta \rightsquigarrow \Theta$ *and* Θ *contains the mapping* $a{:}\,\kappa \mapsto (\tau_1, \tau_2, \gamma)$*, then* $\Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \sim \tau_2$.

Proof. Straightforward induction on $\Gamma \vdash_{\mathsf{tc}} \Delta \rightsquigarrow \Theta$. □

Lemma B.8 (Lifting context coercions)**.** *If* $\Gamma \vdash_{\mathsf{lc}} \Delta \rightsquigarrow \Psi$ *and* Ψ *contains the mapping* $a{:}\,\kappa \overset{?}{\mapsto} (\tau_1, \tau_2, \gamma)$*, then* $\Gamma, \dot\Psi \vdash_{\mathsf{co}} \gamma : \tau_1 \sim \tau_2$.

Proof. Straightforward induction on $\Gamma \vdash_{\mathsf{lc}} \Delta \rightsquigarrow \Psi$, using Lemma B.7 in the LC_THETA case. □

Lemma B.9 (Weakened lifting context substitution)**.** *Suppose* $\Gamma \vdash_{\mathsf{lc}} \Delta \rightsquigarrow \Psi$.

1. *If* $\Gamma, \Delta \vdash_{\mathsf{ty}} \tau : \kappa$ *then* $\Gamma, \dot\Psi \vdash_{\mathsf{ty}} \Psi_j(\tau) : \Psi_j(\kappa)$
2. *If* $\Gamma, \Delta \vdash_{\mathsf{co}} \gamma : \phi$ *then* $\Gamma, \dot\Psi \vdash_{\mathsf{co}} \Psi_j(\gamma) : \Psi_j(\phi)$
3. *If* $\Gamma, \Delta \vdash_{\mathsf{lc}} \Delta' \rightsquigarrow \Psi'$ *then* $\Gamma, \dot\Psi \vdash_{\mathsf{lc}} \Psi_j(\Delta') \rightsquigarrow \Psi_j(\Psi')$
4. *If* $\Gamma, \Delta \vdash_{\mathsf{tel}} \overline{\rho} : \Delta'$ *then* $\Gamma, \dot\Psi \vdash_{\mathsf{tel}} \Psi_j(\overline{\rho}) : \Psi_j(\Delta')$

(This lemma is the same as Lemma B.6, except the j subscripts in the conclusion contexts are removed.)

Proof. Immediate from Lemma B.6 and weakening, noting that any difference between $\dot\Psi_j$ and $\dot\Psi$ are guaranteed to be fresh bindings. □

Lemma B.10 (Fresh variables)**.**

1. *If* $a \# \Gamma$ *and* $a \overset{\bullet}{\mapsto} (a_1, a_2, c)$*, then* $a_1 \# \Gamma$*,* $a_2 \# \Gamma$*, and* $c \# \Gamma$.
2. *If* $c \# \Gamma$ *and* $c \overset{\bullet}{\mapsto} (c_1, c_2)$*, then* $c_1 \# \Gamma$ *and* $c_2 \# \Gamma$.

Proof. Immediate from the definition of $\overset{\bullet}{\mapsto}$. □

Lemma B.11 (Erased lifted coercions)**.** *Let* Ψ *contain the mapping* $c{:}\,\phi \overset{\bullet}{\mapsto} (c_1, c_2)$*. If* $\Gamma \vdash_{\mathsf{lc}} \Delta \rightsquigarrow \Psi$ *and* $\Gamma, \Delta \vdash_{\mathsf{ty}} \tau : \kappa$*, then* $c_1 \# |\Psi(\tau)|$ *and* $c_2 \# |\Psi(\tau)|$.

Proof. We proceed by induction on the typing derivation for τ.

Cases K_STARINSTAR, K_ARROW, and K_EQUAL: Trivial.

Case K_VAR: $\tau = w$, and we know $\vdash_{\mathsf{wf}} \Gamma, \Delta$ and $w{:}\,\kappa \in \Gamma, \Delta$. Here we have two cases:

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

$w \notin dom\,\Delta$**:** In this case $\Psi(w) = \langle w \rangle$. Because the spaces of type variables and coercion variables are distinct, we know that $w \neq c_1$ and $w \neq c_2$, as desired.

Case K_APP: $\tau = \tau_1\,\tau_2$, and we know $\Gamma, \Delta \vdash_{\mathsf{ty}} \tau_1 : \kappa_1 \to \kappa_2$ and $\Gamma, \Delta \vdash_{\mathsf{ty}} \tau_2 : \kappa_1$. Here, $\Psi(\tau_1\,\tau_2) = \Psi(\tau_1)\,\Psi(\tau_2)$. The induction hypothesis tells us that c_1, c_2 do not appear in $|\Psi(\tau_1)|, |\Psi(\tau_2)|$. Since $|\gamma_1\,\gamma_2| = |\gamma_1|\,|\gamma_2|$, the desired result follows directly from this result.

Case K_TINST: Analogous to K_APP.

Case K_CAPP: $\tau = \tau_1\,\gamma_1$, and we know $\Gamma, \Delta \vdash_{\mathsf{ty}} \tau_1 : \forall\, c{:}\,\phi.\,\kappa$ and $\Gamma, \Delta \vdash_{\mathsf{co}} \gamma_1 : \phi$. We have $\Psi(\tau_1\,\gamma_1) = \Psi(\tau_1)(\Psi_1(\gamma_1), \Psi_2(\gamma_1))$. The induction hypothesis tells us that c_1, c_2 do not appear in $|\Psi(\tau_1)|$. By the definition of $|\cdot|$, $|\Psi(\tau_1\,\gamma_1)| = |\Psi(\tau_1)|(\bullet, \bullet)$. Thus, c_1, c_2 do not appear in $|\Psi(\tau_1\,\gamma_1)|$ as desired.

Case K_ALLT: $\tau = \forall\, a{:}\,\kappa.\,\tau'$, and we know $\Gamma, \Delta, a{:}\,\kappa \vdash_{\mathsf{ty}} \tau' : \star$ and $\Gamma, \Delta \vdash_{\mathsf{ty}} \kappa : \star$. Letting $\Psi' = \Psi, a{:}\,\kappa \overset{\bullet}{\mapsto} (a_1, a_2, c)$, we have $|\Psi(\forall\, a{:}\,\kappa.\,\tau')| = |\forall\, a{:}\,\Psi(\kappa).\,\Psi'(\tau')| = \forall\, a{:}\,|\Psi(\kappa)|.\,|\Psi'(\tau')|$. The induction hypothesis tells us that c_1, c_2 do not appear in $|\Psi(\kappa)|$ and $|\Psi'(\tau')|$, so we are done.

Case K_ALLC: Analogous to K_ALLT.

Case K_CAST: $\tau = \tau' \triangleright \eta$ and we know $\Gamma, \Delta \vdash_{\mathsf{ty}} \tau' : \kappa_1, \Gamma, \Delta \vdash_{\mathsf{co}} \eta : \kappa_1 \sim \kappa_2$, and $\Gamma, \Delta \vdash_{\mathsf{ty}} \kappa_2 : \star$. We have $|\Psi(\tau' \triangleright \eta)| = |\Psi(\tau') \triangleright \Psi_1(\eta) \sim \Psi_2(\eta)| = |\mathbf{sym}\,((\mathbf{sym}\,\Psi(\tau')) \triangleright \Psi_2(\eta)) \triangleright \Psi_1(\eta)| = \mathbf{sym}\,(\mathbf{sym}\,|\Psi(\tau')|)$. The induction hypothesis tells us that c_1, c_2 do not appear in $|\Psi(\tau')|$, so we are done.

□

Proof of Lemma 4.2 (Lifting): This lemma is proved by generalizing it to the following lemma that applies to lifting contexts Ψ.

Lemma B.12 (Generalized Lifting)**.** *If* $\Gamma \vdash_{\mathsf{lc}} \Delta \leftrightsquigarrow \Psi$ *and* $\Gamma, \Delta \vdash_{\mathsf{ty}} \tau : \kappa$*, then*

$$\Gamma, \dot{\Psi} \vdash_{\mathsf{co}} \Psi(\tau) : \Psi_1(\tau) \sim \Psi_2(\tau)$$

Proof. We proceed by induction on the typing derivation for τ.

Case K_STARINSTAR: Trivial: $\Gamma, \dot{\Psi} \vdash_{\mathsf{co}} \langle\star\rangle : \star \sim \star$.

Case K_ARROW: Trivial: $\Gamma, \dot{\Psi} \vdash_{\mathsf{co}} \langle(\to)\rangle : (\to) \sim (\to)$.

Case K_EQUAL: Trivial: $\Gamma, \dot{\Psi} \vdash_{\mathsf{co}} \langle(\sim)\rangle : (\sim) \sim (\sim)$.

Case K_VAR: $\tau = w$, and we know $\vdash_{\mathsf{wf}} \Gamma, \Delta$ and $w{:}\,\kappa \in \Gamma, \Delta$. Here we have two cases:

$w \in dom\,\Delta$**:** By the definition of Δ, w must be a type variable a. Using Lemma B.5, there must exist a mapping $a{:}\,\kappa \overset{?}{\mapsto} (\tau_1, \tau_2, \gamma)$ in Ψ. Then, we know $\Psi(w) = \gamma$, $\Psi_1(w) = \tau_1$, and $\Psi_2(w) = \tau_2$. By Lemma B.8, we can get $\Gamma, \dot{\Psi} \vdash_{\mathsf{co}} \gamma : \tau_1 \sim \tau_2$, and thus $\Gamma, \dot{\Psi} \vdash_{\mathsf{co}} \Psi(w) : \Psi_1(w) \sim \Psi_2(w)$ as desired.

$w \notin dom\,\Delta$**:** Trivial: $\Gamma, \dot{\Psi} \vdash_{\mathsf{co}} \langle w \rangle : w \sim w$.

Case K_APP: $\tau = \tau_1\,\tau_2$, and we know $\Gamma, \Delta \vdash_{\mathsf{ty}} \tau_1 : \kappa_1 \to \kappa_2$ and $\Gamma, \Delta \vdash_{\mathsf{ty}} \tau_2 : \kappa_1$. $\Psi(\tau_1\,\tau_2) = \Psi(\tau_1)\,\Psi(\tau_2)$. The induction hypothesis gives us

$$\Gamma, \dot{\Psi} \vdash_{\mathsf{co}} \Psi(\tau_1) : \Psi_1(\tau_1) \sim \Psi_2(\tau_1)$$
$$\Gamma, \dot{\Psi} \vdash_{\mathsf{co}} \Psi(\tau_2) : \Psi_1(\tau_2) \sim \Psi_2(\tau_2).$$

We now wish to use rule CT_APP, but we need to know

$$\Gamma, \dot{\Psi} \vdash_{\mathsf{ty}} \Psi_1(\tau_1)\,\Psi_1(\tau_2) : \sigma_1$$
$$\Gamma, \dot{\Psi} \vdash_{\mathsf{ty}} \Psi_2(\tau_1)\,\Psi_2(\tau_2) : \sigma_2$$

for some types σ_1 and σ_2. Lemma B.9 applied to the types of τ_1 and τ_2, along with straightforward typing rule applications, gives us exactly these facts. Thus,

$$\Gamma, \dot{\Psi} \vdash_{\mathsf{co}} \Psi(\tau_1)\,\Psi(\tau_2) : \Psi_1(\tau_1)\,\Psi_1(\tau_2) \sim \Psi_2(\tau_1)\,\Psi_2(\tau_2)$$

or

$$\Gamma, \dot{\Psi} \vdash_{\mathsf{co}} \Psi(\tau_1\,\tau_2) : \Psi_1(\tau_1\,\tau_2) \sim \Psi_2(\tau_1\,\tau_2)$$

as desired.

Case K_TINST: $\tau = \tau_1\,\tau_2$, and we know $\Gamma, \Delta \vdash_{\mathsf{ty}} \tau_1 : \forall\, a{:}\,\kappa_1.\,\kappa_2$ and $\Gamma, \Delta \vdash_{\mathsf{ty}} \tau_2 : \kappa_1$. This case then proceeds identically to the previous case.

Case K_CAPP: $\tau = \tau_1\,\gamma_1$, and we know $\Gamma, \Delta \vdash_{\mathsf{ty}} \tau_1 : \forall\, c{:}\,\phi.\,\kappa$ and $\Gamma, \Delta \vdash_{\mathsf{co}} \gamma_1 : \phi$. $\Psi(\tau_1\,\gamma_1) = \Psi(\tau_1)(\Psi_1(\gamma_1), \Psi_2(\gamma_1))$. The induction hypothesis gives us

$$\Gamma, \dot{\Psi} \vdash_{\mathsf{co}} \Psi(\tau_1) : \Psi_1(\tau_1) \sim \Psi_2(\tau_1).$$

We now wish to use rule CT_CAPP, but we need to know

$$\Gamma, \Delta \vdash_{\mathsf{ty}} \Psi_1(\tau_1)\,\Psi_1(\gamma_1) : \kappa$$
$$\Gamma, \Delta \vdash_{\mathsf{ty}} \Psi_2(\tau_1)\,\Psi_2(\gamma_1) : \kappa'$$

for some types κ and κ'. Lemma B.9 applied to the types of τ_1 and γ_1, along with straightforward typing rule applications, gives us exactly these facts. Thus,

$$\Gamma, \dot{\Psi} \vdash_{\mathsf{co}} \Psi(\tau_1\,\gamma_1) : \Psi_1(\tau_1)\,\Psi_1(\gamma_1) \sim \Psi_2(\tau_1)\,\Psi_2(\gamma_1)$$

or

$$\Gamma, \dot{\Psi} \vdash_{\mathsf{co}} \Psi(\tau_1\,\gamma_1) : \Psi_1(\tau_1\,\gamma_1) \sim \Psi_2(\tau_1\,\gamma_1)$$

as desired.

Case K_ALLT: $\tau = \forall\, a{:}\,\kappa.\,\tau'$, and we know $\Gamma, \Delta, a{:}\,\kappa \vdash_{\mathsf{ty}} \tau' : \star$ and $\Gamma, \Delta \vdash_{\mathsf{ty}} \kappa : \star$. We can use LC_TY to derive $\Gamma \vdash_{\mathsf{lc}} \Delta, a{:}\,\kappa \leftrightsquigarrow \Psi, a{:}\,\kappa \overset{\bullet}{\mapsto} (a_1, a_2, c)$. Write Ψ' for this extended lifting context.

We wish to show

$$\Gamma, \dot{\Psi} \vdash_{\mathsf{co}} \Psi(\forall\, a{:}\,\kappa.\,\tau') : \Psi_1(\forall\, a{:}\,\kappa.\,\tau') \sim \Psi_2(\forall\, a{:}\,\kappa.\,\tau')$$

or, equivalently,

$$\Gamma, \dot{\Psi} \vdash_{\mathsf{co}} \forall\, a{:}\,\Psi(\kappa).\,\Psi'(\tau') :$$
$$\forall\, a_1{:}\,\Psi_1(\kappa).\,\Psi'_1(\tau') \sim \forall\, a_2{:}\,\Psi_2(\kappa).\,\Psi'_2(\tau')$$

By the induction hypothesis, we have

$$\Gamma, \dot{\Psi} \vdash_{\mathsf{co}} \Psi(\kappa) : \Psi_1(\kappa) \sim \Psi_2(\kappa)$$
$$\Gamma, \dot{\Psi}' \vdash_{\mathsf{co}} \Psi'(\tau') : \Psi'_1(\tau') \sim \Psi'_2(\tau')$$

We wish to use CT_ALLT. The first three premises are already satisfied. We must show

$$\Gamma, \dot{\Psi} \vdash_{\mathsf{ty}} \forall a_j : \Psi_j(\kappa).\Psi_j(\tau') : \star$$

This fact comes directly from the use of Lemma B.9 applied to the type of $\forall\, a{:}\,\kappa.\,\tau'$.

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

Case K_ALLC: $\tau = \forall\, c{:}\,\phi.\,\tau'$, and we know $\Gamma, \Delta,\ c{:}\,\phi \vdash_{\mathsf{ty}} \tau' : \star$ and $\Gamma, \Delta \vdash_{\mathsf{ty}} \phi : \star$. We can use LC_CO to derive $\Gamma \vdash_{\mathsf{lc}} \Delta,\ c{:}\,\phi \leftrightsquigarrow \Psi, c{:}\,\phi \overset{\bullet}{\mapsto} (c_1, c_2)$. Write Ψ' for this extended lifting context.

We wish to show

$$\Gamma, \dot\Psi \vdash_{\mathsf{co}} \Psi(\forall\, c{:}\,\phi.\,\tau') \ :\ \Psi_1(\forall\, c{:}\,\phi.\,\tau') \sim \Psi_2(\forall\, c{:}\,\phi.\,\tau')$$

or, equivalently,

$$\Gamma, \dot\Psi \vdash_{\mathsf{co}} \forall c : \Psi(\phi).\Psi'(\tau') : \\ \forall\, c_1{:}\,\Psi_1(\phi).\,\Psi_1'(\tau') \sim \forall\, c_2{:}\,\Psi_2(\phi).\,\Psi_2'(\tau')$$

By the induction hypothesis, we have

$$\Gamma, \dot\Psi \vdash_{\mathsf{co}} \Psi(\phi) \ :\ \Psi_1(\phi) \sim \Psi_2(\phi)$$
$$\Gamma, \dot\Psi' \vdash_{\mathsf{co}} \Psi'(\tau') \ :\ \Psi_1'(\tau') \sim \Psi_2'(\tau')$$

We wish to use CT_ALLC. The first, second, and fifth premises are already satisfied. The third and fourth premises are $c_1 \ \#\ |\Psi'(\tau')|$ and $c_2 \ \#\ |\Psi'(\tau')|$, respectively. We use Lemma B.11 to get these conditions. Now, it remains only to show

$$\Gamma, \dot\Psi \vdash_{\mathsf{ty}} \forall c_j : \Psi_j(\phi).\Psi_j'(\tau') \ :\ \star$$

This fact comes directly from the use of Lemma B.9 applied to the type of $\forall\, c{:}\,\phi.\,\tau'$.

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

Case K_CAST: $\tau = \tau' \rhd \eta$, and we know $\Gamma, \Delta \vdash_{\mathsf{ty}} \tau' : \kappa_1$, $\Gamma, \Delta \vdash_{\mathsf{co}} \eta : \kappa_1 \sim \kappa_2$, and $\Gamma, \Delta \vdash_{\mathsf{ty}} \kappa_2 : \star$. We wish to show

$$\Gamma, \dot\Psi \vdash_{\mathsf{co}} \Psi(\tau' \rhd \eta) \ :\ \Psi_1(\tau' \rhd \eta) \sim \Psi_2(\tau' \rhd \eta)$$

or, equivalently,

$$\Gamma, \dot\Psi \vdash_{\mathsf{co}} \mathbf{sym}\,((\mathbf{sym}\,\Psi(\tau')) \rhd \Psi_2(\eta)) \rhd \Psi_1(\eta) : \\ \Psi_1(\tau') \rhd \Psi_1(\eta) \sim \Psi_2(\tau') \rhd \Psi_2(\eta).$$

By the induction hypothesis, we have

$$\Gamma, \dot\Psi \vdash_{\mathsf{co}} \Psi(\tau') \ :\ \Psi_1(\tau') \sim \Psi_2(\tau').$$

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

□

B.2 Metatheory for S_KPUSH Preservation

Having defined and proved the generalized 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 B.13 (Telescope substitution)**.** *If* $\Gamma \vdash_{\mathsf{tc}} \Delta \leftrightsquigarrow \Theta$ *and* $\vdash_{\mathsf{wf}} \Delta$*, then* $\Theta_j(\Delta) = \Delta$*.*

Proof. By Lemma B.4, the domain of Θ equals the domain of Δ. Furthermore, $\vdash_{\mathsf{wf}} \Delta$ implies that all kinds in Δ (constructs to the right of a colon) are well-scoped—that is, no variable is mentioned before it is declared. Because the $\Theta_j(\Delta)$ 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 Δ. Thus, $\Theta_j(\Delta) = \Delta$, as desired. □

Lemma B.14 (Θ_j-consistency)**.** *If* $\Gamma \vdash_{\mathsf{tc}} \Delta \leftrightsquigarrow \Theta$*, then* $\Gamma \vdash_{\mathsf{tel}} \Theta_j(\mathit{dom}\,\Delta) : \Delta$*.*

Proof. We wish to use clause 5 of the lifting context substitution lemma (Lemma B.6), with $\overline{\rho} = \mathit{dom}\,\Delta$ and $\Delta' = \Delta$. We must show $\Gamma, \Delta \vdash_{\mathsf{tel}} \mathit{dom}\,\Delta : \Delta$. This is true by straightforward induction on the length of Δ. Then, we apply Lemma 3.4 to get $\Gamma \vdash_{\mathsf{tel}} \Theta_j(\mathit{dom}\,\Delta) : \Theta_j(\Delta)$. By Lemma B.13, this can be rewritten as $\Gamma \vdash_{\mathsf{tel}} \Theta_j(\mathit{dom}\,\Delta) : \Delta$, as desired. □

Lemma B.15 (Telescoped coercion extension consistency)**.** *If* $\Gamma \vdash_{\mathsf{tc}} \Delta_1 \leftrightsquigarrow \Theta$*,* $\vdash_{\mathsf{wf}} \Gamma, \Delta_1, \Delta_2$*,* $\Gamma \vdash_{\mathsf{tel}} \overline{\rho}_2 : \Theta_1(\Delta_2)$*, and* $\Theta' = \Theta \prec \overline{\rho}_2 : \Delta_2$*, then* $\Gamma \vdash_{\mathsf{tc}} \Delta_1, \Delta_2 \leftrightsquigarrow \Theta'$*.*

Proof. We proceed by induction on the derivation of $\Gamma \vdash_{\mathsf{tel}} \overline{\rho}_2 : \Theta_1(\Delta_2)$.

- Case $\overline{\rho}_2 = \varnothing$; $\Delta_2 = \varnothing$: In this case $\Theta' = \Theta$, and thus we must show $\Gamma \vdash_{\mathsf{tc}} \Delta_1 \leftrightsquigarrow \Theta$, which we know by assumption.
- Case $\overline{\rho}_2 = \overline{\rho}_2', \tau$; $\Delta_2 = \Delta_2',\ a{:}\,\kappa$:
  The inductive hypothesis is: if $\vdash_{\mathsf{wf}} \Gamma, \Delta_1, \Delta_2'$, $\Gamma \vdash_{\mathsf{tel}} \overline{\rho}_2' : \Theta_1(\Delta_2')$, and $\Theta'' = \Theta \prec \overline{\rho}_2' : \Delta_2'$, then $\Gamma \vdash_{\mathsf{tc}} \Delta_1, \Delta_2' \leftrightsquigarrow \Theta''$.
  We must show $\Gamma \vdash_{\mathsf{tc}} \Delta_1, \Delta_2',\ a{:}\,\kappa \leftrightsquigarrow \Theta'$, where $\Theta' = \Theta \prec \overline{\rho}_2', \tau : \Delta_2',\ a{:}\,\kappa$.
  By the definition of the $\prec$ operation, we know we will have to use rule TELCO_TY. It is easy to see from the definition of $\prec$ that Θ' is Θ'' with an additional mapping from a. Thus, $\Gamma \vdash_{\mathsf{tc}} \Delta_1, \Delta_2' \leftrightsquigarrow \Theta''$ fulfills the first premise of TELCO_TY. To use TELCO_TY, we must show the following:
  1. $\Gamma \vdash_{\mathsf{ty}} \tau : \ \Theta_1''(\kappa)$
     We know $\Gamma \vdash_{\mathsf{tel}} \overline{\rho}_2', \tau : \Theta_1(\Delta_2',\ a{:}\,\kappa)$. Inverting gives us $\Gamma \vdash_{\mathsf{ty}} \tau : \ \Theta_1(\kappa)[\overline{\rho}_2'/\Theta_1(\Delta_2')]$. Because we care only about the names of the variables in the substitution expression, we can rewrite this as $\Gamma \vdash_{\mathsf{ty}} \tau : \ \Theta_1(\kappa)[\overline{\rho}_2'/\Delta_2']$. From the definition of $\prec$, we can see that all of the substitutions performed by $\Theta_1''(\cdot)$ that are not in Θ map a domain element of Δ_2' to its corresponding $\rho \in \overline{\rho}_2'$. Thus, we can rewrite the judgement above as $\Gamma \vdash_{\mathsf{ty}} \tau : \ \Theta_1''(\kappa)$ as desired.
  2. $\Gamma \vdash_{\mathsf{ty}} \tau \rhd \Theta''(\kappa) : \ \Theta_2''(\kappa)$
     We wish to use the lifting lemma (Lemma 4.2). We know $\Gamma \vdash_{\mathsf{tc}} \Delta_1, \Delta_2' \leftrightsquigarrow \Theta''$. We must show $\Gamma, \Delta_1, \Delta_2' \vdash_{\mathsf{ty}} \kappa : \sigma$ for some σ. This fact, for $\sigma = \star$, comes directly from inversion on $\vdash_{\mathsf{wf}} \Gamma, \Delta_1, \Delta_2',\ a{:}\,\kappa$.
     Now, we apply the lifting lemma to get $\Gamma \vdash_{\mathsf{co}} \Theta''(\kappa) : \Theta_1''(\kappa) \sim \Theta_2''(\kappa)$. As shown in the previous case, $\Gamma \vdash_{\mathsf{ty}} \tau : \ \Theta_1''(\kappa)$. Therefore, by simple application of typing rules, we can derive $\Gamma \vdash_{\mathsf{ty}} \tau \rhd \Theta''(\kappa) : \ \Theta_2''(\kappa)$ as desired.
  3. $\Gamma \vdash_{\mathsf{co}} \mathbf{sym}\,(\langle\tau\rangle \rhd \Theta''(\kappa)) \ :\ \tau \sim (\tau \rhd \Theta''(\kappa))$
     Straightforward application of typing rules.
- Case $\overline{\rho}_2 = \overline{\rho}_2', \gamma$; $\Delta_2 = \Delta_2',\ c{:}\,\phi$:
  The inductive hypothesis is the same as in the previous case. We must show $\Gamma \vdash_{\mathsf{tc}} \Delta_1, \Delta_2',\ c{:}\,\phi \leftrightsquigarrow \Theta'$, where $\Theta' = \Theta \prec \overline{\rho}_2', \gamma : \Delta_2',\ c{:}\,\phi$.
  By the definition of the $\prec$ operation, we know we will have to use rule TELCO_CO. It is easy to see from the definition of $\prec$ that Θ' is Θ'' with an additional mapping from c. Thus, $\Gamma \vdash_{\mathsf{tc}} \Delta_1, \Delta_2' \leftrightsquigarrow \Theta''$ fulfills the first premise of TELCO_CO. To use TELCO_CO, we must show the following:
  1. $\Gamma \vdash_{\mathsf{co}} \gamma : \ \Theta_1''(\phi)$
     We know $\Gamma \vdash_{\mathsf{tel}} \overline{\rho}_2', \gamma : \Theta_1(\Delta_2',\ c{:}\,\phi)$. Inverting gives us $\Gamma \vdash_{\mathsf{co}} \gamma : \Theta_1(\phi)[\overline{\rho}_2'/\Theta_1(\Delta_2')]$. Because we care only about the names of the variables in the substitution expression, we can rewrite this as $\Gamma \vdash_{\mathsf{co}} \gamma : \ \Theta_1(\phi)[\overline{\rho}_2'/\Delta_2']$. From the definition of $\prec$, we can see that all of the substitutions performed by $\Theta_1''(\cdot)$ that are not in Θ map a domain element of Δ_2' to its corresponding $\rho \in \overline{\rho}_2'$. Thus, we can rewrite the judgement above as $\Gamma \vdash_{\mathsf{co}} \gamma : \ \Theta_1''(\phi)$, as desired.
  2. $\Gamma \vdash_{\mathsf{co}} \mathbf{sym}\,(\Theta''(\sigma_1)) \mathbin{⨾} \gamma \mathbin{⨾} \Theta''(\sigma_2) \ :\ \Theta_2''(\sigma_1) \sim \Theta_2''(\sigma_2)$, where $\phi = \sigma_1 \sim \sigma_2$
     We wish to use the lifting lemma (Lemma 4.2) twice to

get the types of $\Theta''(\sigma_1)$ and $\Theta''(\sigma_2)$.. We know $\Gamma \vdash_{\mathsf{tc}} \Delta_1, \Delta_2 \leftrightsquigarrow \Theta''$. We must show $\Gamma, \Delta_1, \Delta_2 \vdash_{\mathsf{ty}} \sigma_1 : \kappa_1$ for some κ_1 and $\Gamma, \Delta_1, \Delta_2 \vdash_{\mathsf{ty}} \sigma_2 : \kappa_2$ for some κ_2. Inversion on $\vdash_{\mathsf{wf}} \Gamma, \Delta_1, \Delta_2, c{:}\, \sigma_1 \sim \sigma_2$ gives us $\Gamma, \Delta_1, \Delta_2 \vdash_{\mathsf{ty}} \sigma_1 \sim \sigma_2 : \star$, which stands for $\Gamma, \Delta_1, \Delta_2 \vdash_{\mathsf{ty}} ((((\sim)\,\kappa_1)\,\kappa_2)\,\sigma_1)\,\sigma_2 : \star$ for some κ_1 and κ_2. Further inversion on this judgement gives us $\Gamma, \Delta_1, \Delta_2 \vdash_{\mathsf{ty}} \sigma_1 : \kappa_1$ and $\Gamma, \Delta_1, \Delta_2 \vdash_{\mathsf{ty}} \sigma_2 : \kappa_2$ as desired.

Now, we apply the lifting lemma to get $\Gamma \vdash_{\mathsf{co}} \Theta''(\sigma_i) : \Theta''_1(\sigma_i) \sim \Theta''_2(\sigma_i)$. As shown in the previous case, $\Gamma \vdash_{\mathsf{co}} \gamma : \Theta''_1(\sigma_1) \sim \Theta''_1(\sigma_2)$. Therefore, by simple application of typing rules, we can derive $\Gamma \vdash_{\mathsf{co}} \mathbf{sym}\,(\Theta''(\sigma_1)) \fatsemi \gamma \fatsemi \Theta''(\sigma_2) : \Theta''_2(\sigma_1) \sim \Theta''_2(\sigma_2)$ as desired.

□

Lemma B.16 (Telescope composition)**.** *If* $\Gamma \vdash_{\mathsf{tel}} \overline{\rho}_1 : \Delta_1$ *and* $\Gamma \vdash_{\mathsf{tel}} \overline{\rho}_1, \overline{\rho}_2 : \Delta_1, \Delta_2$*, then* $\Gamma \vdash_{\mathsf{tel}} \overline{\rho}_2 : \Delta_2[\overline{\rho}_1/\Delta_1]$.

Proof Sketch. By induction on the length of $\overline{\rho}_2$. □

Lemma B.17 (S_KPUSH preservation)**.** *If*

1. $\Gamma \vdash_{\mathsf{tm}} \mathbf{case}\ (K\,\overline{\tau}\,\overline{\rho}\,\overline{e} \triangleright \gamma)\ \mathbf{of}\ \overline{p \to u} \ :\ \sigma$ *and*
2. $\mathbf{case}\ (K\,\overline{\tau}\,\overline{\rho}\,\overline{e} \triangleright \gamma)\ \mathbf{of}\ \overline{p \to u} \longrightarrow \mathbf{case}\ (K\,\overline{\tau'}\,\overline{\rho'}\,\overline{e'})\ \mathbf{of}\ \overline{p \to u}$*, then*

$\Gamma \vdash_{\mathsf{tm}} \mathbf{case}\ (K\,\overline{\tau'}\,\overline{\rho'}\,\overline{e'})\ \mathbf{of}\ \overline{p \to u} \ :\ \sigma$

Proof. By inversion we know that:

- $K{:}\forall\,\overline{a{:}\kappa}.\,\forall\,\Delta.\,\overline{\sigma} \to (T\,\overline{a})$
- $\Theta = \{\gamma\} \prec \overline{\rho} : \Delta$
- $\overline{\tau'} = \Theta_2(\overline{a})$
- $\overline{\rho}' = \Theta_2(dom\,\Delta)$
- $e'_i = e_i \triangleright \Theta(\sigma_i)$
- $\Gamma \vdash_{\mathsf{tm}} e_i \ :\ \sigma_i[\overline{\tau}/\overline{a}][\overline{\rho}/\Delta]$
- $\Gamma \vdash_{\mathsf{tm}} (K\,\overline{\tau}\,\overline{\rho}\,\overline{e}) \triangleright \gamma \ :\ T\,\overline{\tau'}$.
- $\Gamma \vdash_{\mathsf{tm}} K\,\overline{\tau}\,\overline{\rho}\,\overline{e} \ :\ T\,\overline{\tau}$.

We will have to use rule T_CASE to get the desired result. Because the patterns are not changing, we need only show that $\Gamma \vdash_{\mathsf{tm}} K\,\overline{\tau'}\,\overline{\rho}'\,\overline{e'} \ :\ T\,\overline{\tau'}$.

By convention, we have chosen the length of the list $\overline{\tau}$ to be the same as that of the list $\overline{a{:}\kappa}$ in the type of K. Thus, we know that $\Gamma \vdash_{\mathsf{ty}} K\,\overline{\tau'} \ :\ \forall\Delta[\overline{\tau'}/\overline{a}].\,(\overline{\sigma}[\overline{\tau'}/\overline{a}] \to T\,\overline{\tau'})$.

Now, we must show that $\Gamma \vdash_{\mathsf{tel}} \overline{\rho'} \ :\ \Delta[\overline{\tau'}/\overline{a}]$. This can be rewritten as $\Gamma \vdash_{\mathsf{tel}} \Theta_2(dom\,\Delta) : \Delta[\overline{\tau'}/\overline{a}]$.

We know from Lemma B.15 that $\Gamma \vdash_{\mathsf{tc}} \overline{a{:}\kappa}, \Delta \leftrightsquigarrow \Theta$ (using Lemma 4.4 to get $\Gamma \vdash_{\mathsf{tc}} \overline{a{:}\kappa} \leftrightsquigarrow \{\gamma\}$). Lemma B.14 then gives us $\Gamma \vdash_{\mathsf{tel}} \Theta_2(\overline{a}, dom\,\Delta) : \overline{a{:}\kappa}, \Delta$. Invoking Lemma B.16 gives us $\Gamma \vdash_{\mathsf{tel}} \Theta_2(dom\,\Delta) : \Delta[\overline{\tau'}/\overline{a}]$ as desired.

We have now shown that $\Gamma \vdash_{\mathsf{ty}} K\,\overline{\tau'}\,\overline{\rho}' \ :\ \Theta_2(\overline{\sigma}) \to T\,\overline{\tau'}$. We need to show that $\Gamma \vdash_{\mathsf{tm}} e'_i \ :\ \Theta_2(\sigma_i)$, or equivalently, $\Gamma \vdash_{\mathsf{tm}} e_i \triangleright \Theta(\sigma_i) \ :\ \Theta_2(\sigma_i)$. We will need the lifting lemma (Lemma 4.2). We have already shown $\Gamma \vdash_{\mathsf{tc}} \overline{a{:}\kappa}, \Delta \leftrightsquigarrow \Theta$; we must show $\Gamma, \overline{a{:}\kappa}, \Delta \vdash_{\mathsf{ty}} \sigma_i \ :\ \kappa_i$ for some type κ_i. By repeated inversion on the typing judgement for K, we will get $\Gamma, \overline{a{:}\kappa}, \Delta \vdash_{\mathsf{ty}} \sigma_i \ :\ \kappa_i$ as desired. Thus, the lifting lemma gives us $\Gamma \vdash_{\mathsf{co}} \Theta(\sigma_i) \ :\ \Theta_1(\sigma_i) \sim \Theta_2(\sigma_i)$. We note that, by construction, $\Theta_1(\cdot)$ maps $\overline{a}$ to $\overline{\tau}$ and $dom\,\Delta$ to $\overline{\rho}$. Thus, $\sigma_i[\overline{\tau}/\overline{a}][\overline{\rho}/\Delta] = \Theta_1(\sigma_i)$. Now, by straightforward application of typing rules, we can see that $\Gamma \vdash_{\mathsf{tm}} e_i \triangleright \Theta(\sigma_i) \ :\ \Theta_2(\sigma_i)$ as desired.

Thus, $\Gamma \vdash_{\mathsf{tm}} K\,\overline{\tau'}\,\overline{\rho'}\,\overline{e'} \ :\ T\,\overline{\tau'}$ as desired, and we are done. □

B.3 Other preservation cases

Lemma B.18 (TPush Preservation)**.** *If*

1. $\Gamma \vdash_{\mathsf{tm}} (v \triangleright \gamma)\,\tau \ :\ \sigma_2[\tau/a_2]$ *and*
2. $\Gamma \vdash_{\mathsf{co}} \gamma \ :\ \forall\,a_1{:}\,\kappa_1.\,\sigma_1 \sim \forall\,a_2{:}\,\kappa_2.\,\sigma_2$
3. $(v \triangleright \gamma)\,\tau \longrightarrow e'$ *where*
4. $e' = v\,(\tau \triangleright \gamma') \triangleright \gamma@(\langle\tau\rangle \triangleright \gamma')$ *and*
5. $\gamma' = \mathbf{sym}\,(\mathbf{nth}^1\,\gamma)$,

then $\Gamma \vdash_{\mathsf{tm}} e' \ :\ \sigma_2[\tau/a_2]$.

Proof. By inversion of the typing derivation we know that $\Gamma \vdash_{\mathsf{tm}} v \triangleright \gamma \ :\ \forall\,a_2{:}\,\kappa_2.\,\sigma_2$ and $\Gamma \vdash_{\mathsf{ty}} \tau \ :\ \kappa_2$. An additional inversion gives us $\Gamma \vdash_{\mathsf{tm}} v \ :\ \forall\,a_1{:}\,\kappa_1.\,\sigma_1$. Therefore we can show that

- $\Gamma \vdash_{\mathsf{co}} \gamma' \ :\ \kappa_2 \sim \kappa_1$, by the rules for symmetry and nth and
- $\Gamma \vdash_{\mathsf{ty}} \tau \triangleright \gamma' \ :\ \kappa_1$, by casting and
- $\Gamma \vdash_{\mathsf{tm}} v\,(\tau \triangleright \gamma') \ :\ \sigma_1[\tau \triangleright \gamma'/a_1]$, by type application.

Furthermore, we have

- $\Gamma \vdash_{\mathsf{co}} \langle\tau\rangle \triangleright \gamma' \ :\ \tau \triangleright \gamma' \sim \tau$, by reflexivity and coherence and
- $\Gamma \vdash_{\mathsf{co}} \gamma@(\langle\tau\rangle \triangleright \gamma') \ :\ \sigma_1[\tau \triangleright \gamma'/a_1] \sim \sigma_2[\tau/a_2]$, by instantiation.

Thus the final term has the desired type by casting. □

Lemma B.19 (CPush Preservation)**.** *If*

1. $\Gamma \vdash_{\mathsf{tm}} (v \triangleright \gamma)\,\gamma' \ :\ \sigma$ *and*
2. $(v \triangleright \gamma)\,\gamma' \longrightarrow v\,\gamma'' \triangleright \gamma@(\gamma'', \gamma')$*, where*
3. $\gamma'' = (((\mathbf{nth}^3\,(\mathbf{nth}^1\,\gamma)) \fatsemi \gamma') \fatsemi (\mathbf{sym}\,(\mathbf{nth}^4\,(\mathbf{nth}^1\,\gamma))))$ *and*
4. $\Gamma \vdash_{\mathsf{co}} \gamma \ :\ (\forall\,c{:}\,\phi.\,\tau) \sim (\forall\,c'{:}\,\phi'.\,\tau')$,

then $\Gamma \vdash_{\mathsf{tm}} v\,\gamma'' \triangleright \gamma@(\gamma'', \gamma') \ :\ \sigma$.

Proof. By inversion, we have

- $\Gamma \vdash_{\mathsf{tm}} v \triangleright \gamma \ :\ \forall\,c'{:}\,\phi'.\,\tau'$
- $\Gamma \vdash_{\mathsf{tm}} v \ :\ \forall\,c{:}\,\phi.\,\tau$
- $\Gamma \vdash_{\mathsf{co}} \gamma' \ :\ \phi'$
- $\sigma = \tau'[\gamma'/c']$.

From these, we can show

- $\Gamma \vdash_{\mathsf{co}} \mathbf{nth}^1\,\gamma \ :\ \phi \sim \phi'$, by CT_NTH1CA
- $\phi = (\sim)\,\kappa_1\,\kappa_2\,\sigma_1\,\sigma_2$ and $\phi' = (\sim)\,\kappa'_1\,\kappa'_2\,\sigma'_1\,\sigma'_2$, by expanding notation.
- $\Gamma \vdash_{\mathsf{co}} \mathbf{nth}^3\,(\mathbf{nth}^1\,\gamma) \ :\ \sigma_1 \sim \sigma'_1$, by nth rule.
- $\Gamma \vdash_{\mathsf{co}} \mathbf{sym}\,(\mathbf{nth}^4\,(\mathbf{nth}^1\,\gamma)) \ :\ \sigma'_2 \sim \sigma_2$, by symmetry and nth.
- $\Gamma \vdash_{\mathsf{co}} \gamma'' \ :\ \sigma_1 \sim \sigma_2$, by definition of transitivity.
- $\Gamma \vdash_{\mathsf{tm}} v\,\gamma'' \ :\ \tau[\gamma''/c]$, by coercion instantiation.
- $\Gamma \vdash_{\mathsf{co}} \gamma@(\gamma'', \gamma') \ :\ \tau[\gamma''/c] \sim \tau'[\gamma'/c']$, by CT_INSTC.

The final term has the desired type by casting. □

C. Metatheory for Consistency

In this section, we show that good contexts are consistent contexts following the plan laid out in Section 5. Recall the conditions of a good context:

We have $\mathbf{Good}\,\Gamma$ when the following conditions hold:

1. All coercion assumptions and axioms in Γ are of the form $C{:}\,\forall\,\Delta.\,(F\,\overline{\tau} \sim \tau')$ or of the form $c{:}\,a \sim \tau$.

   In the first form, the arguments to the type function must behave like patterns. For every well kinded $\overline{\rho}$, every $\tau_i \in \overline{\tau}$ and every $\tau'_i \in \overline{\tau'}$ such that $\Gamma \models \tau_i[\overline{\rho}/\Delta] \rightsquigarrow \tau'_i$, it must be $\tau'_i = \tau_i[\overline{\rho'}/\Delta]$ for some $\overline{\rho}'$ with $\Gamma \models \tau_m \rightsquigarrow \tau'_m$ for each $\tau_m \in \overline{\rho}$.

2. There is no overlap between axioms and coercion assumptions. For each a, there is at most one assumption of the form $c\colon a \sim \tau$ in the context. For each $F\,\overline{\rho}$ there exists at most one prefix $\overline{\rho}_1$ of $\overline{\rho}$ such that there exist C, σ and Θ where $\Gamma \models C\ \Theta\ :\ (F\ \overline{\rho_1} \sim \sigma)$. This C is unique for every matching $F\ \overline{\tau_1}$.
3. Axioms equate types of the same kind. For each $C\colon \forall\,\Delta.\,(F\,\overline{\tau} \sim \tau')$ in Γ, the kinds of each side must match i.e. $\Gamma, \Delta \models F\ \overline{\tau}\ :\ \kappa$ and $\Gamma, \Delta \models \tau'\ :\ \kappa$ and that kind must not mention bindings in the telescope, $\Gamma \models \kappa\ :\ \star$.

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

Lemma C.1 (No free coercion variables in erased types)**.** *If* $\Gamma \vdash_{\mathsf{ty}} \tau\ :\ \kappa$*, then* $c \# |\tau|$.

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

Proof of Lemma 5.4 (Erasure is type preserving)

1. If $\vdash_{\mathsf{wf}} \Gamma$ then $\models |\Gamma|$.
2. If $\Gamma \vdash_{\mathsf{ty}} \tau\ :\ \kappa$ then $|\Gamma| \models |\tau|\ :\ |\kappa|$.
3. If $\Gamma \vdash_{\mathsf{co}} \gamma\ :\ \phi$ then $|\Gamma| \models |\gamma|\ :\ |\phi|$.
4. If $\Gamma \vdash_{\mathsf{c}} \Delta \leftrightsquigarrow \Theta$ then $|\Gamma| \models |\Delta| \leftrightsquigarrow |\Theta|$.

Proof. By simultaneous induction on the length of the explicit typing derivation. We present a few representative cases.

Case K_CAST: Given rule:

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

By induction, we have $|\Gamma| \models |\tau|\ :\ |\kappa_1|$ and $|\Gamma| \models |\eta|\ :\ |\kappa_1| \sim |\kappa_2|$ and $|\Gamma| \models |\kappa_2|\ :\ |\star|$. By the rule IT_CAST, we have $|\Gamma| \models |\tau|\ :\ |\kappa_2|$. Finally, by definition of erasure, we have $|\tau \rhd \eta| = |\tau|$, and we are done.

Case K_CAPP: Given rule:

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

By induction and definition of erasure, we have $|\Gamma| \models |\tau_1|\ :\ \forall\, c{:}\,|\phi|.\,|\kappa|$, and $|\Gamma| \models |\gamma_1|\ :\ |\phi|$. Hence, by rule IT_CAPP, we have $|\Gamma| \models |\tau_1|\,\bullet\ :\ |\kappa|$, and by erasure $|\tau_1\,\gamma_1| = |\tau_1|\,\bullet$. Finally, we have $|\kappa[\gamma/c]| = |\kappa|$, as the erasure operation erases all coercions within κ.

Case CT_COH: Given rule:

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

By induction and erasure, we have $|\Gamma| \models |\gamma|\ :\ |\tau_1| \sim |\tau_2|$. But also by erasure, we have $|\gamma \rhd \gamma'| = |\gamma|$ and $|\tau_1 \rhd \gamma' \sim \tau_2| = |\tau_1| \sim |\tau_2|$, so we are done.

Case CT_CAPP: Given rule:

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

By induction and definition of erasure, we have $|\Gamma| \models |\tau_1|\,\bullet\ :\ |\kappa|$, $|\Gamma| \models |\tau_1'|\,\bullet\ :\ |\kappa'|$, and $|\Gamma| \models |\gamma|\ :\ |\tau_1| \sim |\tau_1'|$. Hence, by rule ICT_CAPP, we have $|\Gamma| \models |\gamma|(\bullet, \bullet)\ :\ |\tau_1|\,\bullet \sim |\tau_1'|\,\bullet$, and we are done by erasure.

Case CT_ALLC: Given rule:

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{co}} \eta\ :\ \phi_1 \sim \phi_2 \quad c \overset{\bullet}{\mapsto} (c_1, c_2) \\ c_1 \# |\gamma| \quad c_2 \# |\gamma| \\ \Gamma,\ c_1{:}\,\phi_1,\ c_2{:}\,\phi_2 \vdash_{\mathsf{co}} \gamma\ :\ \tau_1 \sim \tau_2 \\ \Gamma \vdash_{\mathsf{ty}} \forall\, c_1{:}\,\phi_1.\,\tau_1\ :\ \star \quad \Gamma \vdash_{\mathsf{ty}} \forall\, c_2{:}\,\phi_2.\,\tau_2\ :\ \star\end{array}}{\Gamma \vdash_{\mathsf{co}} \forall\, c{:}\,\eta.\,\gamma\ :\ (\forall\, c_1{:}\,\phi_1.\,\tau_1) \sim (\forall\, c_2{:}\,\phi_2.\,\tau_2)}\ \text{CT\_ALLC}$$

By induction and definition of erasure, we have
- $|\Gamma| \models |\eta|\ :\ |\phi_1| \sim |\phi_2|$,
- $|\Gamma|,\ c_1{:}\,|\phi_1|,\ c_2{:}\,|\phi_2| \models |\gamma|\ :\ |\tau_1| \sim |\tau_2|$,
- $|\Gamma| \models \forall\, c_1{:}\,|\phi_1|.\,|\tau_1|\ :\ \star$, and
- $|\Gamma| \models \forall\, c_2{:}\,|\phi_2|.\,|\tau_2|\ :\ \star$.

Furthermore, the original rule restricted c_1 and c_2 from appearing in $|\gamma|$. Hence by, ICT_ALLC, we have $|\Gamma| \models \forall\, c{:}\,|\eta|.\,|\gamma|\ :\ (\forall\, c_1{:}\,|\phi_1|.\,|\tau_1|) \sim (\forall\, c_2{:}\,|\phi_2|.\,|\tau_2|)$ and we are done by erasure.

□

Lemma C.2 (Application)**.** *If* **Good** Γ *and* $\Gamma \models \sigma_1 \Leftrightarrow \sigma_1'$ *and* $\Gamma \models \sigma_2 \Leftrightarrow \sigma_2'$ *then* $\Gamma \models \sigma_1\,\sigma_2 \Leftrightarrow \sigma_1'\,\sigma_2'$.

Proof. Let τ_1 be a join point of σ_1, σ_1', and τ_2 a join point for σ_2, σ_2'. By repeatedly applying rule TS_APP and reflexivity of rewriting, we find that $\tau_1\,\tau_2$ is a join point for $\sigma_1\,\sigma_2$ and $\sigma_1'\,\sigma_2'$. □

Lemma C.3 (Type function preservation)**.** *Suppose that* $C\colon \forall\,\Delta.\,(F\,\overline{\tau} \sim \tau') \in \Gamma$*, and* **Good** Γ*. Now, suppose that* $\Gamma \models F\,\overline{\rho} \rightsquigarrow \sigma$*, where* $\overline{\rho}$ *has length strictly smaller than the size of the telescope* $\overline{\tau}$*. Then,* $\sigma = F\,\overline{\rho}'$*, such that* $\Gamma \models \overline{\rho} \rightsquigarrow \overline{\rho}'$.

Proof. By induction on the length of the telescope $\overline{\rho}$. The base case is trivial. For the induction step, note that the rule that applies in the given reduction cannot be TS_RED, as there aren't enough terms in the telescope to reduce. Thus, it must be TS_APP that applies. Therefore, if $\overline{\rho} = \overline{\rho}', \sigma'$, then $\Gamma \models F\,\overline{\rho}' \rightsquigarrow \sigma'$, and $\Gamma \models \sigma' \rightsquigarrow \sigma''$. By induction, $\sigma' = F\,\overline{\rho}''$, and by rule TS_CONS, $\Gamma \models \overline{\rho} \rightsquigarrow \overline{\rho}'', \sigma''$ as desired. □

Proof of Theorem 5.6 (Local diamond property) If **Good** Γ, $\Gamma \models \sigma \rightsquigarrow \sigma_1$, and $\Gamma \models \sigma \rightsquigarrow \sigma_2$ then there exists a σ_3 such that $\Gamma \models \sigma_1 \rightsquigarrow \sigma_3$ and $\Gamma \models \sigma_2 \rightsquigarrow \sigma_3$.

Proof. Induction on lengths 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_VARRED-TS_VARRED 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:

$$\Gamma \models \tau\,\sigma \rightsquigarrow \tau'\,\sigma', \quad \Gamma \models \tau\,\sigma \rightsquigarrow \tau''\,\sigma''$$

Now, we can deduce:

$$\Gamma \models \sigma \rightsquigarrow \sigma', \quad \Gamma \models \sigma \rightsquigarrow \sigma''$$

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

$$\Gamma \models \tau \rightsquigarrow \tau', \quad \Gamma \models \tau \rightsquigarrow \tau''$$

So, also by induction, we can find τ''' that is a common reduct of the two. Hence, by TS_TApp,

$$\Gamma \models \tau'\,\sigma' \rightsquigarrow \tau'''\,\sigma''' \qquad \Gamma \models \tau''\,\sigma'' \rightsquigarrow \tau'''\,\sigma'''$$

case TS_Red-TS_App Concretely, we have a type $F\,\overline{\rho}$, with reductions:

$$\Gamma \models (F\,\overline{\rho}) \rightsquigarrow \sigma_1, \quad \Gamma \models (F\,\overline{\rho}) \rightsquigarrow \sigma_2'\,\sigma'$$

where the first reduction is a type function reduction. Now note that, since context is good, type functions axioms are non-overlapping. Now say that $\overline{\rho} = \overline{\rho}_0, \sigma$ We have by inversion, $\Gamma \models F\,\overline{\rho}_0 \rightsquigarrow \sigma_2'$. By Lemma C.3, we have that $\sigma_2' = F\,\overline{\rho}_0'$, such that $\Gamma \models \overline{\rho}_0 \rightsquigarrow \overline{\rho}_0'$, and so that $\Gamma \models \overline{\rho}_0, \sigma \rightsquigarrow \overline{\rho}_0', \sigma'$. We have that if the coercion for F is $C\colon \forall\,\Delta.\,(F\,\overline{\tau} \sim \tau')$, then we have $\overline{\rho}_0, \sigma = \overline{\tau}[\overline{\rho_1}/\Delta]$, and now by the second condition of good contexts, we have a $\overline{\rho}_1'$, such that

$$\overline{\rho}_0', \sigma' = \overline{\tau}[\overline{\rho_1'}/\Delta] \qquad \Gamma \models \overline{\rho}_1 \rightsquigarrow \overline{\rho}_1'$$

In which case we have a reduction $\Gamma \models F\,\overline{\rho}_0'\,\sigma' \rightsquigarrow \tau'[\overline{\rho_1'}/\Delta]$. But, by an extension of Lemma C.5 for telescopes, we have that

$$\sigma_1 = \tau'[\overline{\rho_1}/\Delta] \qquad \Gamma \models \sigma_1 \rightsquigarrow \tau'[\overline{\rho_1'}/\Delta]$$

as desired.

case TS_Red-TS_Red Concretely, we have a type $F\;\overline{\sigma_1}\;\overline{\sigma_2}$, which can also be written as $F\;\overline{\sigma_3}\;\overline{\sigma_4}$, such that we have reductions:

$$\Gamma \models F\,\overline{\sigma} \rightsquigarrow \sigma', \quad \Gamma \models F\,\overline{\sigma} \rightsquigarrow \sigma''$$

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

□

Lemma C.4 (Transitivity of Rewriting)**.** *If* $\mathbf{Good}\,\Gamma$ *and* $\Gamma \models \sigma_1 \Leftrightarrow \sigma_2$ *and* $\Gamma \models \sigma_2 \Leftrightarrow \sigma_3$*, then* $\Gamma \models \sigma_1 \Leftrightarrow \sigma_3$*.*

Proof. Appeal to the local diamond property. Suppose σ_{12} is a join point for σ_1, σ_2 and σ_{23} is a join point for σ_2, σ_3. By 5.6, there is a join point σ_0 for σ_{12}, σ_{23}, and hence is a join point for σ_1, σ_3. □

Lemma C.5 (Single Step Substitution)**.** *If* $\mathbf{Good}\,\Gamma$ *and* $\Gamma \models \tau \rightsquigarrow \tau'$ *then for a free in* σ *and* σ *implicitly well-typed under* Γ*, we have* $\Gamma \models \sigma[\tau/a] \rightsquigarrow \sigma[\tau'/a]$*.*

Proof Sketch. By induction on the form of σ. For example, if σ is of the form $\forall\,a\colon\kappa.\,\sigma$, by induction, we have that $\Gamma \models \sigma[\tau/a'] \rightsquigarrow \sigma[\tau'/a']$, for $a \neq a'$. Hence, by rule TS_AllT, we have that $\Gamma \models (\forall\,a\colon\kappa.\,\sigma)[\tau/a'] \rightsquigarrow (\forall\,a\colon\kappa.\,\sigma)[\tau/a']$. □

Lemma C.6 (Multistep Substitution)**.** *If* $\mathbf{Good}\,\Gamma$ *and* $\Gamma \models \tau \rightsquigarrow^* \tau'$ *then for a free in* σ *and* σ *implicitly well-typed under* Γ*, we have* $\Gamma \models \sigma[\tau/a] \rightsquigarrow^* \sigma[\tau'/a]$*.*

Proof. By induction on the length of $\Gamma \models \tau \rightsquigarrow^* \tau'$. The base case is trivial. The induction case uses Lemma C.5. □

Lemma C.7 (Single Step Substitution 2)**.** *If* $\mathbf{Good}\,\Gamma$ *and* $\Gamma \models \sigma \rightsquigarrow \sigma'$*, then for a free in* σ, σ'*, we have* $\Gamma \models \sigma[\tau/a] \rightsquigarrow \sigma'[\tau/a]$

Proof Sketch. By induction on $\Gamma \models \sigma \rightsquigarrow \sigma'$. For example, for rule

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

Suppose we want to substitute τ'' for a free. Thus, we would continue to have $\overline{\sigma_1[\tau''/a] = \tau[\overline{\rho}/\Delta][\tau''/a]}$ and $\sigma_2[\tau''/a] = (\tau'[\overline{\rho}/\Delta][\tau''/a])$, so we conclude

$$\Gamma \models (F\;\overline{\sigma_1}\;\overline{\sigma_2})[\tau''/a] \rightsquigarrow (\tau'\;\overline{\sigma_2})[\tau''/a]$$

□

Proof of Lemma 5.7 (Substitution) If $\mathbf{Good}\,\Gamma$, $\Gamma \models \sigma \rightsquigarrow^* \sigma'$, and $\Gamma \models \tau \rightsquigarrow^* \tau'$, then if a appears free in σ and σ', we have $\Gamma \models \sigma[\tau/a] \Leftrightarrow \sigma'[\tau'/a]$.

Proof. Induction on the length of reduction $\Gamma \models \sigma \rightsquigarrow^* \sigma'$. The base case is Lemma C.6, while the induction step is first by Lemma C.6, and then by Lemma C.7 along with Lemma C.4. □

Corollary C.8 (Joinability substitution)**.** *If* $\mathbf{Good}\,\Gamma$*,* $\Gamma \models \sigma \Leftrightarrow \sigma'$*,* $\Gamma \models \tau \Leftrightarrow \tau'$*, then if a appears free in* σ *and* σ'*, then we have* $\Gamma \models \sigma[\tau/a] \Leftrightarrow \sigma'[\tau'/a]$*.*

Proof. By induction on the lengths of the derivations. The base case is trivial. For the induction step, we can use the induction hypothesis, combined with Lemma 5.7. □

Lemma C.9 (Joinability strengthening)**.** *If* $\mathbf{Good}\,\Gamma$ *and* $\Gamma,\ a\colon\kappa \models \tau_1 \Leftrightarrow \tau_2$*, then* $\Gamma \models \tau_1 \Leftrightarrow \tau_2$*.*

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

We need a lemma to deal with the $\mathbf{kind}\,\gamma$ construct. Essentially, this lemma states that we don't need the $\mathbf{kind}\,\gamma$ construct (it is already internalized in our system).

Lemma C.10 (Admissibility of kind)**.** *Suppose we have a derivation* $\Gamma \models \gamma\ :\ \sigma_1 \sim \sigma_2$*, such that* $\Gamma \models \tau_1\ :\ \kappa_1$ *and* $\Gamma \models \tau_2\ :\ \kappa_2$ *and* $fcv(\gamma) \subseteq dom\,\Gamma'$ *for some subcontext* Γ' *satisfying* $\mathbf{Good}\,\Gamma'$*. Then, there exists a derivation* $\Gamma \models \eta\ :\ \kappa_1 \sim \kappa_2$ *at strictly* lower *height, for some* η*, such that* $fcv(\eta) \subseteq dom\,\Gamma'$*.*

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

Case ICT_Trans: Given rule:

$$\frac{\Gamma \models \gamma_1\ :\ \tau_1 \sim \tau_2 \quad \Gamma \models \gamma_2\ :\ \tau_2 \sim \tau_3}{\Gamma \models \gamma_1 \mathbin{;} \gamma_2\ :\ \tau_1 \sim \tau_3}\ \text{ICT\_Trans}$$

Note that the free coercion variables of $\gamma_1 \mathbin{;} \gamma_2$ lie in a good context, so the same is true of γ_1 and γ_2. Hence, by induction, we are able to find derivations of $\Gamma, \Gamma' \models \eta_1\ :\ \kappa_1 \sim \kappa_2$ and $\Gamma, \Gamma' \models \eta_2\ :\ \kappa_2 \sim \kappa_3$ that conclude strictly above the premise of this rule, satisfying the required freshness condition. Now, by ICT_Trans, we are able to create a proof $\Gamma \models \eta_1 \mathbin{;} \eta_2\ :\ \kappa_1 \sim \kappa_3$ at height strictly above the conclusion, and we are done.

Case ICT_VarAx: Given rule:

$$\frac{C\colon \forall\,\Delta.\,(\tau_1 \sim \tau_2) \in \Gamma \quad \Gamma \models \Delta \leftrightsquigarrow \Theta}{\Gamma \models C\,\Theta\ :\ \Theta_1(\tau_1) \sim \Theta_2(\tau_2)}\ \text{ICT\_VarAx}$$

Note that the free coercion variables of $C\Theta$ lie in a good context, so the same is true of C and Θ. Thus, the axiom lies in a good subcontext. By definition of $\mathbf{Good}\,\Gamma'$, we have that both sides are kind κ for a closed kind. Hence, simply $\langle\kappa\rangle$ suffices. This derivation requires total height 2, and so works as long as $\Gamma \models \Delta \leftrightsquigarrow \Theta$ is of height at least 2, which is true by the rules defining the judgement. Evidently, as the coercion doesn't mention any coercion variables, the freshness condition is satisfied as well.

□

Lemma C.11 (Nth joinability)**.** *Suppose that* $\Gamma \models H\,\overline{\rho} \Leftrightarrow H\,\overline{\rho}'$, *and* $\mathbf{Good}\,\Gamma$. *Then,* $\Gamma \models \rho_i \Leftrightarrow \rho_i'$.

Proof. By induction on the length of the telescopes (by inversion, both have the same length). The base case is trivial. For induction, note that $H\,\overline{\rho}, H\,\overline{\rho}'$ must both step by TS_APP. Hence, by the form of that rewrite rule, say that $\overline{\rho} = \overline{\rho}_0, \rho_0$ and $\overline{\rho}' = \overline{\rho}_0', \rho_0'$, and the length of the telescopes are preserved. So, $\Gamma \models \rho_0 \Leftrightarrow \rho_0'$, and we want the last element in the telescope, we are done. Otherwise, $\Gamma \models H\,\overline{\rho}_0 \Leftrightarrow H\,\overline{\rho}_0'$. By induction, we have the result. □

From these lemmas we see that joinability is complete.

Proof of Theorem 5.8 (Completeness)

1. Suppose that $\Gamma \models \gamma : \sigma_1 \sim \sigma_2$, and $fcv(\gamma) \subseteq dom\,\Gamma'$ for some subcontext Γ' satisfying $\mathbf{Good}\,\Gamma'$. Then $\Gamma \models \sigma_1 \Leftrightarrow \sigma_2$.
2. Suppose that $\Gamma \models \Delta \leftrightsquigarrow \Theta$, and $fcv(\Theta) \subseteq dom\,\Gamma'$ for some subcontext Γ' satisfying $\mathbf{Good}\,\Gamma'$. Then for each $a{:}\,\kappa \mapsto (\tau_1, \tau_2, \gamma) \in \Theta$, we have $\Gamma \vdash \tau_1 \Leftrightarrow \tau_2$.

Proof. By joint induction on the structures of $\Gamma \models \gamma : \sigma_1 \sim \sigma_2$, and $\Gamma \models \Delta \leftrightsquigarrow \Theta$.

Case ICT_CAPP: We have rule:

$$\frac{\begin{array}{c}\Gamma \models \gamma : \tau \sim \tau' \\ \Gamma \models \tau\,\bullet : \kappa \quad \Gamma \models \tau'\,\bullet : \kappa'\end{array}}{\Gamma \models \gamma(\bullet,\bullet) : \tau\,\bullet \sim \tau'\,\bullet}\ \text{ICT\_CAPP}$$

Note that the free coercion variables of $\gamma(\bullet,\bullet)$ lie in a good context, so the same is true of γ. Hence, by induction, $\Gamma \models \tau \Leftrightarrow \tau'$. Then, by Lemma C.2, we are done.

Case ICT_ALLC:

$$\frac{\begin{array}{l}\Gamma \models \eta : \phi_1 \sim \phi_2 \quad c \overset{\bullet}{\mapsto} (c_1, c_2) \\ c_1 \# \gamma \quad c_2 \# \gamma \\ \Gamma,\ c_1{:}\,\phi_1,\ c_2{:}\,\phi_2 \models \gamma : \tau_1 \sim \tau_2 \\ \Gamma \models \forall\, c_1{:}\,\phi_1.\,\tau_1 : \star \quad \Gamma \models \forall\, c_2{:}\,\phi_2.\,\tau_2 : \star\end{array}}{\Gamma \models \forall\, c{:}\,\eta.\,\gamma : (\forall\, c_1{:}\,\phi_1.\,\tau_1) \sim (\forall\, c_2{:}\,\phi_2.\,\tau_2)}\ \text{ICT\_ALLC}$$

Note that the free coercion variables of $\forall\, c{:}\,\eta.\,\gamma$ lie in a good context, so the same is true of γ and η. Hence, by induction, there is a join point ϕ for ϕ_1, ϕ_2. Also by induction, there is a join point τ for τ_1, τ_2. By rule TS_ALLC, we have that

$$\Gamma \models \forall\, c_1{:}\,\phi_1.\,\tau_1 \rightsquigarrow^* \forall\, c_1{:}\,\phi.\,\tau$$

and

$$\Gamma \models \forall\, c_2{:}\,\phi_2.\,\tau_2 \rightsquigarrow^* \forall\, c_2{:}\,\phi.\,\tau$$

and hence they are joinable.

Case ICT_INST:

$$\frac{\begin{array}{l}\Gamma \models \gamma_1 : (\forall\, a_1{:}\,\kappa_1.\,\tau_1) \sim (\forall\, a_2{:}\,\kappa_2.\,\tau_2) \\ \Gamma \models \gamma_2 : \sigma_1 \sim \sigma_2 \\ \Gamma \models \sigma_1 : \kappa_1 \quad \Gamma \models \sigma_2 : \kappa_2\end{array}}{\Gamma \models \gamma_1 @ \gamma_2 : \tau_1[\sigma_1/a_1] \sim \tau_2[\sigma_2/a_2]}\ \text{ICT\_INST}$$

Note that the free coercion variables of $\gamma @ \gamma'$ lie in a good context, so the same is true of γ and γ. Hence, by induction, $\Gamma \models \sigma_1 \Leftrightarrow \sigma_2$, and $\Gamma \models (\forall\, a_1{:}\,\kappa_1.\,\tau_1) \Leftrightarrow (\forall\, a_2{:}\,\kappa_2.\,\tau_2)$. Now, by inversion on the step relation for quantified types, we find that $\Gamma \models \tau_1 \Leftrightarrow \tau_2$. Hence, by substitution (Lemma 5.7) and transitivity (Lemma C.4), we have that $\Gamma \models \tau_1[\sigma_1/a_1] \Leftrightarrow \tau_2[\sigma_2/a_2]$, as desired.

Case ICT_INSTC:

$$\frac{\begin{array}{l}\Gamma \models \gamma : (\forall\, c{:}\,\sigma_1 \sim \sigma_2.\,\tau) \sim (\forall\, c'{:}\,\sigma_1' \sim \sigma_2'.\,\tau') \\ \Gamma \models \gamma_1 : \sigma_1 \sim \sigma_2 \quad \Gamma \models \gamma_2 : \sigma_1' \sim \sigma_2'\end{array}}{\Gamma \models \gamma @ (\bullet,\bullet) : \tau \sim \tau'}\ \text{ICT\_INSTC}$$

Note that the free coercion variables of $\gamma @ (\bullet,\bullet)$ lie in a good context, so the same is true of γ. Hence, by induction, $\Gamma \models (\forall\, c{:}\,\sigma_1 \sim \sigma_2.\,\tau) \Leftrightarrow (\forall\, c{:}\,\sigma_1' \sim \sigma_2'.\,\tau')$. Now, by inversion on the step relation for quantified types, we find that $\Gamma \models \tau \Leftrightarrow \tau'$. Hence, by substitution (Lemma 5.7) and transitivity (Lemma C.4), we have that $\Gamma \models \tau \Leftrightarrow \tau'$, as desired ($\tau, \tau'$ have no free coercion variables) .

Case ICT_REFL: Trivial.

Case ICT_SYM: Trivial.

Case ICT_TRANS: Follows from Lemma C.4.

Case ICT_APP: Follows from Lemma C.2.

Case ICT_ALLT:

$$\frac{\begin{array}{l}\Gamma \models \eta : \kappa_1 \sim \kappa_2 \quad a \overset{\bullet}{\mapsto} (a_1, a_2, c) \\ \Gamma,\ a_1{:}\,\kappa_1,\ a_2{:}\,\kappa_2,\ c{:}\,a_1 \sim a_2 \models \gamma : \tau_1 \sim \tau_2 \\ \Gamma \models \forall\, a_1{:}\,\kappa_1.\,\tau_1 : \star \quad \Gamma \models \forall\, a_2{:}\,\kappa_2.\,\tau_2 : \star\end{array}}{\Gamma \models \forall\, a{:}\,\eta.\,\gamma : (\forall\, a_1{:}\,\kappa_1.\,\tau_1) \sim (\forall\, a_2{:}\,\kappa_2.\,\tau_2)}\ \text{ICT\_ALLT}$$

Note that the free coercion variables of $\forall\, a{:}\,\eta.\,\gamma$ lie in a good context, so the same is true of γ since $c{:}\,a_1 \sim a_2$ is a good assumption that doesn't overlap with the previous axioms, as the variables a_1, a_2 are fresh. Hence, by induction, we have $\Gamma,\ a_1{:}\,\kappa_1,\ a_2{:}\,\kappa_2,\ c{:}\,a_1 \sim a_2 \models \tau_1 \Leftrightarrow \tau_2$, which we can strengthen to $\Gamma,\ c{:}\,a_1 \sim a_2 \models \tau_1 \Leftrightarrow \tau_2$, by Lemma C.9. Also by induction, we have $\Gamma \models \kappa_1 \Leftrightarrow \kappa_2$, which allows us to finish the rule by TS_ALLT.

Case ICT_VAR: Trivial, all assumptions are rewrite rules in good contexts. Note that c must be a good assumption in the context.

Case ICT_VARAX: We have the rule:

$$\frac{C{:}\,\forall\,\Delta.\,(\tau_1 \sim \tau_2) \in \Gamma \quad \Gamma \models \Delta \leftrightsquigarrow \Theta}{\Gamma \models C\,\Theta : \Theta_1(\tau_1) \sim \Theta_2(\tau_2)}\ \text{ICT\_VARAX}$$

Note that the free coercion variables of $C\Theta$ lie in a good context, so the same is true of C and Θ. Hence, we may apply the induction hypothesis. Note that $\Theta_1(\tau), \Theta_2(\tau)$ are both substitutions, by definition. Further, note that by the induction hypothesis, we have that for every free variable a that is substituted for, with binding $a{:}\,\kappa \mapsto (\sigma_1, \sigma_2, \gamma)$ in Θ, we have that $\Gamma \models \sigma_1 \Leftrightarrow \sigma_2$. Hence, we can make the substitutions one by one, and using Corollary C.8 repeatedly, we have the desired result.

Case ICT_NTH: We have the rule:

$$\frac{\Gamma \models \gamma : H\,\overline{\tau} \sim H\,\overline{\tau'}}{\Gamma \models \mathbf{nth}^i\,\gamma : \tau_i \sim \tau_i'}\ \text{ICT\_NTH}$$

Note that the free coercion variables of $\mathbf{nth}^i\,\gamma$ lie in a good context, so the same is true of γ. Hence, by induction, then Lemma C.11, we are done.

Case ICT_NTH1TA: We have rule:

$$\frac{\Gamma \models \gamma_1 : (\forall\, a_1{:}\,\kappa_1.\,\tau_1) \sim (\forall\, a_2{:}\,\kappa_2.\,\tau_2)}{\Gamma \models \mathbf{nth}^1\,\gamma_1 : \kappa_1 \sim \kappa_2}\ \text{ICT\_NTH1TA}$$

Note that the free coercion variables of $\mathbf{nth}^1\,\gamma_1$ lie in a good context, so the same is true of γ_1. Hence, by induction on γ_1, 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 $\Gamma \models \kappa_1 \Leftrightarrow \kappa_2$ as desired.

Case ICT_NTH1CA: We have rule:

$$\frac{\Gamma \models \gamma : (\forall\, c{:}\,\phi.\,\tau) \sim (\forall\, c'{:}\,\phi'.\,\tau')}{\Gamma \models \mathbf{nth}^1\,\gamma : \phi \sim \phi'}\ \text{ICT\_NTH1CA}$$

Virtually identical to the previous case.

Case ICT_EXT: We have rule:

$$\frac{\Gamma \models \gamma : \tau_1 \sim \tau_2 \quad \Gamma \models \tau_1 : \kappa_2 \quad \Gamma \models \tau_2 : \kappa_2}{\Gamma \models \mathbf{kind}\,\gamma : \kappa_1 \sim \kappa_2}\ \text{ICT\_EXT}$$

By the admissibility of $\mathbf{kind}\,\gamma$ (Lemma C.10) we can construct a derivation of $\Gamma \models \eta \ : \ \kappa_1 \sim \kappa_2$ at strictly smaller height that proves the same equality, such that η has free variables in a good context. Then, we are done by induction.

For the derivation $\Gamma \models \Delta' \leftrightsquigarrow \Theta'$, there are two cases. Suppose we want to show that for $a_0{:}\,\kappa_0 \ \mapsto \ (\tau_1, \tau_2, \gamma) \ \in \ \Theta$, we have $\Gamma \models \tau_1 \Leftrightarrow \tau_2$.

Case IL_TY: We have rule:

$$\frac{\begin{array}{l}\Gamma \models \Delta \leftrightsquigarrow \Theta \\ \Gamma \models \sigma_1 \ : \ \Theta_1(\kappa) \\ \Gamma \models \sigma_2 \ : \ \Theta_2(\kappa) \\ \Gamma \models \gamma \ : \ \sigma_1 \sim \sigma_2\end{array}}{\Gamma \models (\Delta,\ a{:}\,\kappa) \leftrightsquigarrow (\Theta, a{:}\,\kappa \mapsto (\sigma_1, \sigma_2, \gamma))} \ \text{IL\_TY}$$

Note that the free coercion variables of Θ lie in a good context. Now, case on whether $a_0 = a$ or not. If so, we are done by induction on the coercion $\Gamma \models \gamma \ : \ \tau_1 \sim \tau_2$. If not, by induction on the judgement $\Gamma, \Gamma' \models \Delta \leftrightsquigarrow \Theta$, we are done.

Case IL_CO: Since we are only interested in the type bindings, we are done by induction.

$\square$