Type Normalization in R$\omega\mu$

ALEX HUBERS, The University of Iowa, USA

1 INTRODUCTION

We describe the normalization-by-evaluation (NBE) of types in R$\omega\mu$. Types are normalized modulo β- and η-equivalence—that is, to $\beta\eta$-long forms. Because the type system of R$\omega\mu$ is a strict extension of System F$\omega\mu$, type level computation for arrow kinds is isomorphic to reduction of arrow types in the STLC. Novel to this report are the reductions of Π, Σ, and row types.

2 THE R$\omega\mu$ CALCULUS

We present the type calculus of R$\omega\mu$. The syntax of kinds, types, and predicates are given in Figure 1 for reference. We describe the mechanized syntax in the section that follows.

$$\text{Type variables}\ \ \alpha \in \mathcal{A} \qquad \text{Labels}\ \ \ell \in \mathcal{L}$$

$$\begin{array}{llcl}
\text{Kinds} & \kappa & ::= & \star \mid \mathsf{L} \mid \mathsf{R}^{\kappa} \mid \kappa \to \kappa \\
\text{Predicates} & \pi, \psi & ::= & \rho \lesssim \rho \mid \rho \odot \rho \sim \rho \\
\text{Types}\ \ \mathcal{T} \ni & \phi, \tau, \upsilon, \rho, \xi & ::= & \alpha \mid \pi \Rightarrow \tau \mid \forall \alpha : \kappa . \tau \mid \lambda \alpha : \kappa . \tau \mid \tau\, \tau \\
 & & \mid & \{\xi_i \triangleright \tau_i\}_{i \in 0 \ldots m} \mid \ell \mid \# \tau \mid \phi \,\$\, \rho \mid \rho \setminus \rho \\
 & & \mid & \tau \to \tau \mid \Pi^{(\kappa)}\, \rho \mid \Sigma^{(\kappa)}\, \rho \mid \mu\, \phi
\end{array}$$

Fig. 1. Syntax

2.1 Kind syntax

Our formalization of R$\omega\mu$ types is *intrinsic*, meaning we define the syntax of *typing* and *kinding judgments*, foregoing any formalization of / indexing by untyped syntax. The syntax of types is indexed by kinding environments and kinds, defined below.

```
data Kind : Set where
  ★     : Kind
  L     : Kind
  _‘→_ : Kind → Kind → Kind
  R[_] : Kind → Kind

infixr 5 _‘→_
```

The kind system of R$\omega\mu$ defines $\star$ as the type of types; L as the type of labels; $(\to)$ as the type of type operators; and $R[\kappa]$ as the type of *rows* containing types at kind κ.

The syntax of kinding environments is given below. Kinding environments are isomorphic to lists of kinds.

Author's address: Alex Hubers, Department of Computer Science, The University of Iowa, 14 MacLean Hall, Iowa City, Iowa, USA, alexander-hubers@uiowa.edu.

```
data KEnv : Set where
  ϵ : KEnv
  _„_ : KEnv → Kind → KEnv
```

Let the metavariables Δ and κ range over kinding environments and kinds, respectively. Correspondingly, we define *generalized variables* in Agda at these names.

```
private
  variable
    Δ Δ₁ Δ₂ Δ₃ : KEnv
    κ κ₁ κ₂ : Kind
```

The syntax of intrinsically well-scoped De-Bruijn type variables is given below. We say that the type variable x is indexed by kinding environment Δ and kind κ to specify that x has kind κ in kinding environment Δ.

```
data TVar : KEnv → Kind → Set where
  Z : TVar (Δ „ κ) κ
  S : TVar Δ κ₁ → TVar (Δ „ κ₂) κ₁
```

2.2 Syntax of types

R$\omega\mu$ is a qualified type system with predicates of the form $\rho_1 \lesssim \rho_2$ and $\rho_1 \cdot \rho_2 \sim \rho_3$ for row-kinded types ρ_1, ρ_2, and ρ_3. Because predicates occur in types and types occur in predicates, the syntax of well-kinded types and well-kinded predicates are mutually recursive. The syntax for each is given below. we describe (in this order) the syntactic components belonging to System F$\omega\mu$, qualified type systems, and system Rω.

```
data Pred (Δ : KEnv) : Kind → Set
data Type Δ : Kind → Set
data Type Δ where

  ` :
    (α : TVar Δ κ) →
    Type Δ κ

  `λ :
    (τ : Type (Δ „ κ₁) κ₂) →
    Type Δ (κ₁ `→ κ₂)

  _·_ :
    (τ₁ : Type Δ (κ₁ `→ κ₂)) →
    (τ₂ : Type Δ κ₁) →
    Type Δ κ₂

  _`→_ :
    (τ₁ : Type Δ ★) →
    (τ₂ : Type Δ ★) →
    Type Δ ★

  `∀ :
```

$(\tau : \mathsf{Type}\ (\Delta\ ,,\ \kappa)\ \star) \to$
$\mathsf{Type}\ \Delta\ \star$

μ :
$(F : \mathsf{Type}\ \Delta\ (\star\ `\!\!\to \star)) \to$
$\mathsf{Type}\ \Delta\ \star$

The first three constructors are analogous to the terms of the STLC. the constructor $`\!\!\to$ classifies term functions; the constructor $`\forall$ classifies type-in-term quantification; and the constructor μ classifies recursive terms. Note that μ could be further generalized to kind $\kappa\ `\!\!\to\ \star$; however, we found that kind $\star\ `\!\!\to\ \star$ was sufficient for our needs while simplifying both presentation and mechanization.

The syntax of qualified types is given below.

$\_\!\Rightarrow\!\_$:
$(\pi : \mathsf{Pred}\ \Delta\ \mathsf{R}[\ \kappa_1\ ]) \to (\tau : \mathsf{Type}\ \Delta\ \star) \to$
$\mathsf{Type}\ \Delta\ \star$

The type $\pi\ \Rightarrow\ \tau$ states that τ is *qualified* by the predicate π—that is, the type variables bound in τ are restricted in instantiation to just those that satisfy the predicate π. This is completely analogous to identical syntax used in Haskell to introduce typeclass qualification. Predicates are defined below (after the presentation of type syntax).

We now describe the syntax exclusive to R$\omega\mu$, beginning with label kind introduction and elimination. Labels are first-class entities in R$\omega\mu$, and may be represented by both constants and variables.

lab :
$(l : \mathsf{Label}) \to$
$\mathsf{Type}\ \Delta\ \mathsf{L}$

$\lfloor\_\rfloor$:
$(\tau : \mathsf{Type}\ \Delta\ \mathsf{L}) \to$
$\mathsf{Type}\ \Delta\ \star$

Label constants in R$\omega\mu$ are constructed from the type `Label`; in our mechanization, `Label` is a type synonym for `String`, but one could choose any other candidate with decidable equality. Types at label kind L may be cast to *label singletons* by the $\lfloor\_\rfloor$ constructor. This makes labels first-class entities: for example, as the type $\lfloor$ `lab "l"` $\rfloor$ has kind $\star$, it can be inhabited by a term.

Types at row kind are constructed by one of the following three constructors.

ϵ :
$\mathsf{Type}\ \Delta\ \mathsf{R}[\ \kappa\ ]$

$\_\triangleright\_$:
$(l : \mathsf{Type}\ \Delta\ \mathsf{L}) \to (\tau : \mathsf{Type}\ \Delta\ \kappa) \to$
$\mathsf{Type}\ \Delta\ \mathsf{R}[\ \kappa\ ]$

_<\$>_ :

```
  (f : Type Δ (κ₁ `→ κ₂)) → (τ : Type Δ R[ κ₁ ]) →
  Type Δ R[ κ₂ ]
```

Rows in R$\omega\mu$ are either the empty row ϵ, a labeled row `(l ▹ τ)`, or a row mapping `f <$> τ`. The row mapping `f <$> (l ▹ τ)` describes the lifting of the function `f` over row `(l ▹ τ)`, which we will define to equal `(l ▹ f τ)` in the case where the right hand applicand is a labeled row. We will show that rows in Rome (that is, types at row kind) reduce to either the empty row ϵ or a labeled row `(l ▹ τ)` after normalization. There are two important consequences of this canonicity: firstly, we treat row mapping `_<$>_` as having latent computation to perform (there are no normal types with form `f <$> τ` except when τ is a neutral variable). The second consequence is that we do not permit the formation of rows with more than one label-type association. Such rows are instead formed as type variables with predicates specifying the shape of the row.

Rows in R$\omega\mu$ are eliminated by the Π and Σ constructors.

```
Π :
    Type Δ (R[ κ ] `→ κ)

Σ :
    Type Δ (R[ κ ] `→ κ)
```

Given a type ρ at row kind, $\Pi\rho$ constructs a record with label-type associations from ρ and $\Sigma\rho$ constructs a variant that has label and type from ρ. We choose to represent Π and Σ as type constants at kind `(R[ κ ] `→ κ)`; we will show that many applications of Π and Σ induce type reductions, and hence it is convenient to group such reductions with type application.

The syntax of predicates is given below. The predicate $\rho_1 \lesssim \rho_2$ states that label-to-type mappings in ρ_1 are a subset of those in ρ_2; the predicate $\rho_1 \odot \rho_2 \sim \rho_3$ states that the combination of mappings in ρ_1 and ρ_2 equals ρ_3.

```
data Pred Δ where
  _·_~_ :
    (ρ₁ ρ₂ ρ₃ : Type Δ R[ κ ]) →
    Pred Δ R[ κ ]

  _≲_ :
    (ρ₁ ρ₂ : Type Δ R[ κ ]) →
    Pred Δ R[ κ ]
```

Hubers and Morris [2023] implicitly define two type-level row lifting operators, *left mapping* <\$> and *right mapping* <?>, but the latter is superfluous. We appeal to the kinds of these operators for their intuition: left mapping `f <$> ρ` lifts a function at arrow kind `f : κ₁ → κ₂` into a function at kind `R[ κ₁ ] → R[ κ₂ ]` and then applies it to `ρ : R[ κ₂ ]`. We may define right mapping (named *flap* and written <?>, after similar Haskell operators) of row function $f : R[\kappa_1 \to \kappa_2]$ over type $\tau : \kappa_1$ using left mapping under the following identity:

$$f \mathbin{<?>} \tau = (\lambda\, g.\, g\, \tau) \mathbin{<\$>} f$$

which we encode in Agda as follows:

```
flap : Type Δ (R[ κ₁ `→ κ₂ ] `→ κ₁ `→ R[ κ₂ ])
flap = `λ (`λ ((`λ ((` Z) · (` (S Z)))) <$> (` (S Z))))

_<?>_ : Type Δ (R[ κ₁ `→ κ₂ ]) → Type Δ κ₁ → Type Δ R[ κ₂ ]
f <?> a = flap · f · a
```

(We choose to define `_<?>_` as the application of `flap` to inputs `f` and `a` so that we needn't pollute the definition with weakenings of its arguments.)

2.2.1 Type renaming. We closely follow Wadler et al. [2022] and Chapman et al. [2019] in defining a *type renaming* as a function from type variables in one kinding environment to type variables in another. This is the *parallel renaming and substitution* approach for which weakening and single variable substitution are special cases. The code we establish now will be mimicked again for both normal types and for terms; many names are reused, and so we find it helpful to index duplicate names by a suffix. The suffix $_k$ specifies that this definition describes the `Type` syntax.

```
Renamingₖ : KEnv → KEnv → Set
Renamingₖ Δ₁ Δ₂ = ∀ {κ} → TVar Δ₁ κ → TVar Δ₂ κ
```

We will let the metavariable ρ range over both renamings and types at row kind.

Lifting can be thought of as the weakening of a renaming, and permits renamings to be pushed under binders.

```
liftₖ : Renamingₖ Δ₁ Δ₂ → Renamingₖ (Δ₁ ,, κ) (Δ₂ ,, κ)
liftₖ ρ Z = Z
liftₖ ρ (S x) = S (ρ x)
```

We define renaming as a function that translates a kinding derivation in kinding environment Δ_1 to environment Δ_2 provided a renaming from Δ_1 to Δ_2. The definition proceeds by induction on the input kinding derivation. In the variable case, we use ρ to rename variable x. In the `` `λ `` and `` `∀ `` cases, we must lift the renaming ρ over the type variable introduced by these binders. The rest of the cases are effectively just congruence over the type structure.

```
renₖ : Renamingₖ Δ₁ Δ₂ → Type Δ₁ κ → Type Δ₂ κ
renPredₖ : Renamingₖ Δ₁ Δ₂ → Pred Δ₁ R[ κ ] → Pred Δ₂ R[ κ ]

renₖ ρ (` x)        = ` (ρ x)
renₖ ρ (`λ τ)       = `λ (renₖ (liftₖ ρ) τ)
renₖ ρ (π ⇒ τ)      = renPredₖ ρ π ⇒ renₖ ρ τ
renₖ ρ (`∀ τ)       = `∀ (renₖ (liftₖ ρ) τ)
renₖ ρ ϵ            = ϵ
renₖ ρ (τ₁ · τ₂)    = (renₖ ρ τ₁) · (renₖ ρ τ₂)
renₖ ρ (τ₁ `→ τ₂) = (renₖ ρ τ₁) `→ (renₖ ρ τ₂)
renₖ ρ (μ F)        = μ (renₖ ρ F)
renₖ ρ Π            = Π
renₖ ρ Σ            = Σ
renₖ ρ (lab x)      = lab x
renₖ ρ (l ▹ τ)      = renₖ ρ l ▹ renₖ ρ τ
```

```
renₖ ρ ⌊ ℓ ⌋          = ⌊ (renₖ ρ ℓ) ⌋
renₖ ρ (f <$> m) = renₖ ρ f <$> renₖ ρ m
```

As Type and Pred are mutually inductive, we must define $\mathtt{renPred}_k$ as mutually recursive to $\mathtt{ren}_k$. Its definition is completely unsuprising.

```
renPredₖ ρ (ρ₁ · ρ₂ ~ ρ₃) = renₖ ρ ρ₁ · renₖ ρ ρ₂ ~ renₖ ρ ρ₃
renPredₖ ρ (ρ₁ ≲ ρ₂) = (renₖ ρ ρ₁) ≲ (renₖ ρ ρ₂)
```

Finally, weakening is a special case of renaming.

```
weakenₖ : Type Δ κ₂ ⟶ Type (Δ ,, κ₁) κ₂
weakenₖ = renₖ S
```

2.2.2 Type substitution. We wish to give both a declarative and algorithmic treatment of type equivalence. For the latter, we will normalize types to normal forms, meaning types are equivalent iff their normal forms are definitionally equal. For the former, we must define β-substitution syntactically so that we can express β-equivalence of types declaratively. In our development, β-reduction is a special case of substitution.

We define a substitution as a function mapping type variables in context Δ_1 to types in context Δ_2.

```
Substitutionₖ : KEnv ⟶ KEnv ⟶ Set
Substitutionₖ Δ₁ Δ₂ = ∀ {κ} ⟶ TVar Δ₁ κ ⟶ Type Δ₂ κ
```

Substitutions must be lifted over binders, just as is done for renamings.

```
liftsₖ : Substitutionₖ Δ₁ Δ₂ ⟶ Substitutionₖ(Δ₁ ,, κ) (Δ₂ ,, κ)
liftsₖ σ Z = ‘ Z
liftsₖ σ (S x) = weakenₖ (σ x)
```

Substitution is defined inductively over types in a similar fashion to renaming. Note that this is *simultaneous* substitution and renaming—The variable case translates type variable x to the type $\sigma\,\tau$, for which the substitution σ also performs a renaming from environment Δ_1 to Δ_2. The rest of the cases (as with renaming) are either congruences over the type structure or congruences plus lifting of the substitution. Again, substitution over predicates is defined mutually recursively.

```
subₖ : Substitutionₖ Δ₁ Δ₂ ⟶ Type Δ₁ κ ⟶ Type Δ₂ κ
subPredₖ : Substitutionₖ Δ₁ Δ₂ ⟶ Pred Δ₁ κ ⟶ Pred Δ₂ κ
subₖ σ ε = ε
subₖ σ (‘ x) = σ x
subₖ σ (‘λ τ) = ‘λ (subₖ (liftsₖ σ) τ)
subₖ σ (τ₁ · τ₂) = (subₖ σ τ₁) · (subₖ σ τ₂)
subₖ σ (τ₁ ‘⟶ τ₂) = (subₖ σ τ₁) ‘⟶ (subₖ σ τ₂)
subₖ σ (π ⇒ τ) = subPredₖ σ π ⇒ subₖ σ τ
subₖ σ (‘∀ τ) = ‘∀ (subₖ (liftsₖ σ) τ)
subₖ σ (μ F) = μ (subₖ σ F)
subₖ σ (Π) = Π
subₖ σ Σ = Σ
subₖ σ (lab x) = lab x
```

```
subₖ σ (l ▹ τ) = subₖ σ l ▹ subₖ σ τ
subₖ σ ⌊ ℓ ⌋ = ⌊ (subₖ σ ℓ) ⌋
subₖ σ (f <$> a) = subₖ σ f <$> subₖ σ a

subPredₖ σ (ρ₁ · ρ₂ ~ ρ₃) = subₖ σ ρ₁ · subₖ σ ρ₂ ~ subₖ σ ρ₃
subPredₖ σ (ρ₁ ≲ ρ₂) = (subₖ σ ρ₁) ≲ (subₖ σ ρ₂)
```

We define the extension of a substitution σ by the type τ functionally. If we had chosen to represent a `Substitution`$_k$ as a list, extension would be done by the `cons` constructor. In a De-Bruijn representation, the most recently appended variable is zero—hence an extension here maps the zero variable to τ in the `Z` case and maps each variable `(S x)` to its value in σ at predecessor `x`.

```
extendₖ : Substitutionₖ Δ₁ Δ₂ → (τ : Type Δ₂ κ) → Substitutionₖ (Δ₁ ,, κ) Δ₂
extendₖ σ τ Z = τ
extendₖ σ τ (S x) = σ x
```

Finally, β-substitution is simply a special case of substitution. Note that the constructor ` has type `TVar Δ κ → Type Δ κ`, making it a substitution. It is in fact an identity substitution, which fixes the meaning of its type variables, hence it is the substitution we choose to extend when defining β-substitution.

```
_βₖ[_] : Type (Δ ,, κ₁) κ₂ → Type Δ κ₁ → Type Δ κ₂
τ₁ βₖ[ τ₂ ] = subₖ (extendₖ ` τ₂) τ₁
```

2.3 Type equivalence

We define type and predicate equivalence mutually recursively. You may think of type equivalence also as a sort of small-step relation on types, as we include rules to equate β-equivalent and η-equivalent types, as well as a number of computational steps a row kinded type may take.

```
infix 0 _≡t_
infix 0 _≡p_
data _≡p_ : Pred Δ R[ κ ] → Pred Δ R[ κ ] → Set
data _≡t_ : Type Δ κ → Type Δ κ → Set
```

Unless otherwise quantified, let the metavariable `l` range over types with label kind, let π range over predicates, and let τ and υ range over types:

```
private
  variable
    l l₁ l₂ l₃ : Type Δ L
    ρ₁ ρ₂ ρ₃ : Type Δ R[ κ ]
    π₁ π₂ : Pred Δ R[ κ ]
    τ τ₁ τ₂ τ₃ υ υ₁ υ₂ υ₃ : Type Δ κ
```

The rules for predicate equivalence are uninteresting: two predicates are considered equivalent when their component types are equivalent.

```
data _≡p_ where
  _eq-≲_ :
    τ₁ ≡t υ₁ → τ₂ ≡t υ₂ →
    τ₁ ≲ τ₂ ≡p υ₁ ≲ υ₂

  _eq-·_~_ :
    τ₁ ≡t υ₁ → τ₂ ≡t υ₂ → τ₃ ≡t υ₃ →
    τ₁ · τ₂ ~ τ₃ ≡p υ₁ · υ₂ ~ υ₃
```

The first three rules for type equivalence state that it is an equivalence relation.

```
data _≡t_ where
  eq-refl :
    τ ≡t τ

  eq-sym :
    τ₁ ≡t τ₂ →
    τ₂ ≡t τ₁

  eq-trans :
    τ₁ ≡t τ₂ → τ₂ ≡t τ₃ →
    τ₁ ≡t τ₃
```

Type equivalence is congruent over the total structure of types, including λ-bindings (hence you may view type normalization as being *call-by-value*). We omit the other eight congruence rules.

```
eq-λ : ∀ {τ υ : Type (Δ ,, κ₁) κ₂} →
  τ ≡t υ →
  `λ τ ≡t `λ υ
```

We have one η-equivalence rule. It is henceforth useful to view the following rules as directed left-to-right, as normal forms are produced on the right-hand side.

```
eq-η : ∀ {f : Type Δ (κ₁ `→ κ₂)} →
  f ≡t `λ (weakenₖ f · (` Z))
```

The rules that remain as *computational*—these are precisely the rules we would use to define small-step reduction of types. We begin with the β-equivalence rule, which states that lambda abstractions applied to arguments are equivalent to their beta reduction.

```
eq-β : ∀ {τ₁ : Type (Δ ,, κ₁) κ₂} {τ₂ : Type Δ κ₁} →
  ((`λ τ₁) · τ₂) ≡t (τ₁ βₖ[ τ₂ ])
```

The next two rules specify the computational behavior of mapping over rows. Rule (eq-<$>$\epsilon$) states that mapping over the empty row ϵ should yield the empty row; rule eq-▹$ states that mapping over a labeled row should push the left applicand into the body of the row.

```
eq-<$>ε : {F : Type Δ (κ₁ `→ κ₂)} →
  (F <$> ε) ≡t ε

eq-▹$ : ∀ {l} {τ : Type Δ κ₁} {F : Type Δ (κ₁ `→ κ₂)} →
  (F <$> (l ▹ τ)) ≡t (l ▹ (F · τ))
```

We wish to establish that normal forms of types at row kind are either the empty row ϵ or labeled rows. This is, of course, not the case for types in general. For example, the type `Π · (l ▹ τ)` has row kind when τ has row kind `R[ κ ]`. In this case, rule `eq-Π▹` pushes the Π over the label so that a canonical form is restored.

```
eq-Π▹ : ∀ {l} {τ : Type Δ R[ κ ]} →
  Π · (l ▹ τ) ≡t (l ▹ (Π · τ))
```

The application of Π and Σ to a type τ at nested-row kind is in fact just the mapping of Π and Σ over τ:

```
eq-Π : ∀ {τ : Type Δ R[ R[ κ ] ]} →
  Π · τ ≡t Π <$> τ
```

Likewise to rows, we wish to show that normal forms of types at arrow kind are canonically λ-bound. However, the type `Π · (l ▹ `λ τ)` has arrow kind! Rule `eq-Πλ` pushes the $\lambda$ outwards in order to restore canonicity and so that application of `Π · (l ▹ `λ τ)` to an applicand is simply β-reduction.

```
eq-Πλ : ∀ {l} {τ : Type (Δ ,, κ₁) κ₂} →
  Π · (l ▹ `λ τ) ≡t `λ (Π · (weakenₖ l ▹ τ))
```

Finally, in many cases (such as record concatenation and variant branching) it is necessary to reassociate the application $(\Pi\ \rho)\ \tau$ inward so that Π (or Σ) are the outermost syntax. We observe the following reassociation identity:

```
eq-Π-assoc : ∀ {ρ : Type Δ (R[ κ₁ `→ κ₂ ])} {τ : Type Δ κ₁} →
  (Π · ρ) · τ ≡t Π · (ρ <?> τ)
```

The definition of `_≡t_` concludes by repeating the last four rules, replacing each Π with Σ. As a final aside, it might be thought that we could have rid ourselves of the syntax for mapping by elaborating types at kind `R[ κ₁ → κ₂]`. For example, the type `(l ▹ λ x : κ₁. τ)` could perhaps have its λ binding pushed outside to yield `λ x : κ₁. (l ▹ τ)`. However, this would not be kind-preserving (the latter has kind `κ₁ → R[ κ₂ ]`), and therefore such a translation would induce a normalization that does not preserve kinds. We believe it would be possible but complicated to consider a kind-changing translation.

3 NORMAL TYPES

As is common in other *normalization by evaluation* approaches, we separate *neutral types* from *normal types*. These two definitions are defined mutually inductively with the data type for normal predicates:

```
data NormalType (Δ : KEnv) : Kind → Set
data NormalPred (Δ : KEnv) : Kind → Set
data NeutralType Δ : Kind → Set
```

A type is neutral if it is (respectively) (i) a variable, (ii) the application of a variable to an argument, or (iii) the mapping of a normal function type over a neutral row type. Intuitively, neutral forms are forms for which computation is "stuck" waiting on a variable to be substituted for a canonical form. Note that this third neutral form (row mapping) is novel to our development, and, in comparison to application, inverts the normal/neutral expectation of its arguments. It captures the stuck nature of a type such as `(l ▹ λ x. M) <$> ρ`—that is, we are unable to map a function over a type variable.

```
data NeutralType Δ where
  ` :
    (α : TVar Δ κ) →
    NeutralType Δ κ

  _·_ :
    (f : NeutralType Δ (κ₁ `→ κ)) →
    (τ : NormalType Δ κ₁) →
    NeutralType Δ κ

  _<$>_ :
    (F : NormalType Δ (κ₁ `→ κ₂)) → (τ : NeutralType Δ R[ κ₁ ]) →
    NeutralType Δ (R[ κ₂ ])
```

A predicate is normal if its component types are each normal.

```
data NormalPred Δ where
  _·_~_ :
    (ρ₁ ρ₂ ρ₃ : NormalType Δ R[ κ ]) →
    NormalPred Δ R[ κ ]

  _≲_ :
    (ρ₁ ρ₂ : NormalType Δ R[ κ ]) →
    NormalPred Δ R[ κ ]
```

Because we consider the normalization of types modulo η-equivalence, we wish to restrict our normal types to η-long form. This can be done by restricting the construction of normal-neutral types to just ground kind. This also ensures a canonical form for arrow-kinded normal types, as neutral types at arrow-kind cannot be promoted to normal types. We define a `Ground` predicate on types that maps all non-arrow kinds to the unit type ⊤ and maps the arrow kind to ⊥. (In other words, `Ground` κ is trivially inhabitable so long as $\kappa \neq \kappa_1 \to \kappa_2$.)

```
Ground : Kind → Set
Ground ★ = ⊤
Ground L = ⊤
Ground (κ `→ κ₁) = ⊥
Ground R[ κ ] = ⊤
```

It is easy to show that this predicate is decidable.

```
ground? : ∀ κ → Dec (Ground κ)
ground? ★ = yes tt
ground? L = yes tt
ground? (_ `→ _) = no (λ ())
ground? R[ _ ] = yes tt
```

Now we may restrict the ne constructor to promoting just neutral types at ground kind by adding the (implicit) requirement that ne only be used when `Ground` κ is satisfied. To make this evidence easy to populate when κ is known, we employ a well-known proof-by-reflection trick (see Wadler et al. [2022]) and require evidence of the form `True (ground?` κ`)`.

```
data NormalType Δ where
  ne :
    (x : NeutralType Δ κ) → {ground : True (ground? κ)} →
    NormalType Δ κ
```

Likewise, to ensure canonical forms of rows, we restrict Π and Σ to formation at kind ★ and `L`. The constructors for record types are given below.

```
Π :
  (ρ : NormalType Δ R[ ★ ]) →
  NormalType Δ ★

ΠL :
  (ρ : NormalType Δ R[ L ]) →
  NormalType Δ L
```

The rest of the `NormalType` syntax is identical to the `Type` syntax with the exception that we remove the ` constructor for variables and Π and Σ constructors at arbitrary kind. We choose not to omit this syntax, as our proofs of canonicity follow from knowing the totality of `NormalType` constructors.

```
-- Fω
‘λ :
  (τ : NormalType (Δ ,, κ₁) κ₂) →
  NormalType Δ (κ₁ ‘→ κ₂)

_‘→_ :
  (τ₁ τ₂ : NormalType Δ ★) →
  NormalType Δ ★

‘∀ :
  {κ : Kind} → (τ : NormalType (Δ ,, κ) ★) →
  NormalType Δ ★

μ :
  (F : NormalType Δ (★ ‘→ ★)) →
  NormalType Δ ★

-- Qualified types
_⇒_ :
  (π : NormalPred Δ R[ κ₁ ]) → (τ : NormalType Δ ★) →
  NormalType Δ ★

-- Rω
ε :
  NormalType Δ R[ κ ]

_▹_ :
  (l : NormalType Δ L) →
  (τ : NormalType Δ κ) →
  NormalType Δ R[ κ ]

lab :
  (l : Label) →
  NormalType Δ L

⌊_⌋ :
  (l : NormalType Δ L) →
  NormalType Δ ★
Σ :
  (ρ : NormalType Δ R[ ★ ]) →
  NormalType Δ ★

ΣL :
  (ρ : NormalType Δ R[ L ]) →
  NormalType Δ L
```

3.0.1 Renaming. We define renaming over `NormalTypes` in the same fashion as defined over `Types`. Note that we use the suffix $_k$`NF` now to denote functions which operate on `NormalType` syntax. Definitions are unsurprising and omitted.

```
renₖNE      : Renamingₖ Δ₁ Δ₂ → NeutralType Δ₁ κ → NeutralType Δ₂ κ
renₖNF      : Renamingₖ Δ₁ Δ₂ → NormalType Δ₁ κ → NormalType Δ₂ κ
weakenₖNF : NormalType Δ κ₂ → NormalType (Δ ,, κ₁) κ₂
weakenₖNE : NeutralType Δ κ₂ → NeutralType (Δ ,, κ₁) κ₂
```

3.1 Properties of normal types

We use Agda to confirm the desired canonicity properties. First, we wish for arrow kinds to be canonically formed by λ-abstractions. This can be shown easily by induction on arrow-kinded `f`.

```
arrow-canonicity : (f : NormalType Δ (κ₁ `→ κ₂)) → ∃[ τ ] (f ≡ `λ τ)
arrow-canonicity (`λ f) = f , refl
```

Second, we wish for types at row kind to be canonically either (i) a labeled type `(l ▹ τ)`, (ii) a neutral type, or (iii) the empty row ϵ. The `row-canonicity` lemma below states precisely this. Note that we permit row-kinded types to be neutral because we do not η-expand arrow-kinded rows. Recall our discussion above that such an expansion would not be kind-preserving. This means arrow-kinded rows must be permitted to be canonically neutral.

```
row-canonicity : (ρ : NormalType Δ R[ κ ]) →
  ∃[ l ] (Σ[ τ ∈ NormalType Δ κ ] ((ρ ≡ (l ▹ τ)))) or
  Σ[ τ ∈ NeutralType Δ R[ κ ] ] (ρ ≡ ne τ) or
  ρ ≡ ϵ
row-canonicity (l ▹ τ) = left (l , τ , refl)
row-canonicity (ne τ) = right (left (τ , refl))
row-canonicity ϵ = right (right refl)
```

3.2 Type embeddings

We establish an embedding back from normal types to types below. The embedding is written $\Uparrow$ because its type is converse to our definition of normalization, written $\Downarrow$. We will show in later sections precisely that $\Uparrow$ is right-inverse to $\Downarrow$.

$\Uparrow$: NormalType Δ κ $\to$ Type Δ κ
$\Uparrow$NE : NeutralType Δ κ $\to$ Type Δ κ
$\Uparrow$Pred : NormalPred Δ R[κ] $\to$ Pred Δ R[κ]

Much of the embedding is defined by using like-for-like constructors and recursing on the subdata.

$\Uparrow$NE (‘ x) = ‘ x
$\Uparrow$NE ($\tau_1 \cdot \tau_2$) = ($\Uparrow$NE τ_1) $\cdot$ ($\Uparrow$ τ_2)
$\Uparrow$NE (F <\$> τ) = ($\Uparrow$ F) <\$> ($\Uparrow$NE τ)

$\Uparrow$Pred ($\rho_1 \cdot \rho_2 \sim \rho_3$) = ($\Uparrow$ ρ_1) $\cdot$ ($\Uparrow$ ρ_2) $\sim$ ($\Uparrow$ ρ_3)
$\Uparrow$Pred ($\rho_1 \lesssim \rho_2$) = ($\Uparrow$ ρ_1) $\lesssim$ ($\Uparrow$ ρ_2)

$\Uparrow$ ϵ = ϵ
$\Uparrow$ (ne x) = $\Uparrow$NE x
$\Uparrow$ ($l \triangleright \tau$) = ($\Uparrow$ l) $\triangleright$ ($\Uparrow$ τ)
$\Uparrow$ (‘λ τ) = ‘λ ($\Uparrow$ τ)
$\Uparrow$ (τ_1 ‘$\to$ τ_2) = $\Uparrow$ τ_1 ‘$\to$ $\Uparrow$ τ_2
$\Uparrow$ (‘$\forall$ τ) = ‘$\forall$ ($\Uparrow$ τ)
$\Uparrow$ (μ τ) = μ ($\Uparrow$ τ)
$\Uparrow$ (lab l) = lab l
$\Uparrow$ $\lfloor \tau \rfloor$ = $\lfloor \Uparrow \tau \rfloor$
$\Uparrow$ ($\pi \Rightarrow \tau$) = ($\Uparrow$Pred π) $\Rightarrow$ ($\Uparrow$ τ)

An exception is made for record and variant constructors, which we must reconstruct as applications:

$\Uparrow$ (Π x) = Π $\cdot$ $\Uparrow$ x
$\Uparrow$ (ΠL x) = Π $\cdot$ $\Uparrow$ x
$\Uparrow$ (Σ x) = Σ $\cdot$ $\Uparrow$ x
$\Uparrow$ (ΣL x) = Σ $\cdot$ $\Uparrow$ x

4 SEMANTIC TYPES

We next define `SemType` Δ κ, the semantic interpretation of types. `SemType`s are defined by induction on the kind κ and mutually-recursively with `KripkeFunction`s, the interpretation of type functions.

SemType : KEnv $\to$ Kind $\to$ Set
KripkeFunction : KEnv $\to$ Kind $\to$ Kind $\to$ Set

Type functions are interpreted as Kripke function spaces because they must permit arbitrary and intermediate renaming. That is, they are functions at "any world."

KripkeFunction Δ_1 κ_1 κ_2 = ($\forall$ {Δ_2} $\to$ Renaming$_k$ Δ_1 Δ_2 $\to$ SemType Δ_2 κ_1 $\to$ SemType Δ_2 κ_2)
SemType Δ_1 (κ_1 ‘$\to$ κ_2) = KripkeFunction Δ_1 κ_1 κ_2

We interpret ⋆ and L kinded types as their normal forms.

$$\begin{aligned}
&\mathsf{SemType}\ \Delta\ \star = \mathsf{NormalType}\ \Delta\ \star\\
&\mathsf{SemType}\ \Delta\ \mathsf{L} = \mathsf{NormalType}\ \Delta\ \mathsf{L}
\end{aligned}$$

We interpret rows as either `nothing` (the empty row), `just (left x)` for neutral `x`, or `just (right (l , τ))` for normal `l` and τ. These cases correspond precisely to the three canonical forms of types with row kind.

$$\begin{aligned}
&\mathsf{SemType}\ \Delta\ \mathsf{R}[\ \kappa\ ] = \mathsf{Maybe}\\
&\quad ((\mathsf{NeutralType}\ \Delta\ \mathsf{R}[\ \kappa\ ])\ \mathsf{or}\\
&\quad (\mathsf{NormalType}\ \Delta\ \mathsf{L} \times \mathsf{SemType}\ \Delta\ \kappa))
\end{aligned}$$

4.1 Renaming & substitution

Renaming is defined over semantic types in an obvious fashion. Definitions are omitted except in the functional case.

$$\begin{aligned}
&\mathsf{renSem} : \mathsf{Renaming}_k\ \Delta_1\ \Delta_2 \to \mathsf{SemType}\ \Delta_1\ \kappa \to \mathsf{SemType}\ \Delta_2\ \kappa\\
&\mathsf{weakenSem} : \mathsf{SemType}\ \Delta\ \kappa_1 \to \mathsf{SemType}\ (\Delta\ ,,\ \kappa_2)\ \kappa_1
\end{aligned}$$

Because R$\omega\mu$ functions are interpreted into Kripke function spaces, renaming of arrow-kinded types is simply composition by the function's renaming.

$$\begin{aligned}
&\mathsf{renKripke} : \mathsf{Renaming}_k\ \Delta_1\ \Delta_2 \to \mathsf{KripkeFunction}\ \Delta_1\ \kappa_1\ \kappa_2 \to \mathsf{KripkeFunction}\ \Delta_2\ \kappa_1\ \kappa_2\\
&\mathsf{renKripke}\ \{\Delta_1\}\ \rho\ F\ \{\Delta_2\} = \lambda\ \rho' \to F\ (\rho' \circ \rho)
\end{aligned}$$

$$\mathsf{renSem}\ \{\kappa = \kappa\ `\!\!\to \kappa_1\}\ \rho\ F = \mathsf{renKripke}\ \rho\ F$$

4.2 Normalization by evaluation

Our *normalization by evaluation* proceeds in a standard fashion. We will define `reflect`, which maps neutral types to semantic types, and `reify`, which maps semantic types to normal types. We then write an evaluator that takes a `Type` into the semantic domain. During this process, function applications (and other forms of computation) are reduced. We finally reify the semantic type back to a normal form.

Reflection and reification are defined mutually recursively. We define the type synonym `reifyKripke` the reification of types at arrow kind, for repeated use later.

$$\begin{aligned}
&\mathsf{reflect} : \forall\ \{\kappa\} \to \mathsf{NeutralType}\ \Delta\ \kappa \to \mathsf{SemType}\ \Delta\ \kappa\\
&\mathsf{reify} : \forall\ \{\kappa\} \to \mathsf{SemType}\ \Delta\ \kappa \to \mathsf{NormalType}\ \Delta\ \kappa\\
&\mathsf{reifyKripke} : \mathsf{KripkeFunction}\ \Delta\ \kappa_1\ \kappa_2 \to \mathsf{NormalType}\ \Delta\ (\kappa_1\ `\!\!\to \kappa_2)
\end{aligned}$$

Reflection of neutral types at ground kind leaves the type undisturbed.

$$\begin{aligned}
&\mathsf{reflect}\ \{\kappa = \star\}\ \tau &&= \mathsf{ne}\ \tau\\
&\mathsf{reflect}\ \{\kappa = \mathsf{L}\}\ \tau &&= \mathsf{ne}\ \tau\\
&\mathsf{reflect}\ \{\kappa = \mathsf{R}[\ \kappa\ ]\}\ \tau &&= \mathsf{just}\ (\mathsf{left}\ \tau)
\end{aligned}$$

Reflection of neutral types at arrow kind must be η-expanded into a Kripke function. Note here that is necessary to reify the input v back to a normal type.

$$\mathsf{reflect}\ \{\kappa = \kappa_1\ `\!\!\to \kappa_2\}\ \tau = \lambda\ \rho\ v \to \mathsf{reflect}\ (\mathsf{ren}_k\mathsf{NE}\ \rho\ \tau \cdot \mathsf{reify}\ v)$$

Reification similarly leaves ground types undisturbed. Semantic types at ★ and label kind are already in normal form; semantic types at row kind must be translated from their semantic constructors to their `NormalType` constructors.

```
reify {κ = ★} τ = τ
reify {κ = L} τ = τ
reify {κ = R[ κ ]} (just (left x)) = ne x
reify {κ = R[ κ ]} (just (right (l , τ))) = l ▹ (reify τ)
reify {κ = R[ κ ]} nothing = ϵ
```

Semantic functions must be reified from Agda functions back into `NormalType` syntax. This is done by reifying the application of semantic function `F` to the reflection of the η-expanded variable ` Z.

```
reify {κ = κ₁ `→ κ₂} F = reifyKripke F
reifyKripke {κ₁ = κ₁} F = `λ (reify (F S (reflect {κ = κ₁} (` Z))))
```

Observe that neutral types can be forced into η-long form simply by composing reification and reflection. This will prove helpful later, as the neutral type former `ne` has the same type except restricted to ground kind, but we will need to be able to promote from neutral to normal type at *all* kinds.

```
η-norm : NeutralType Δ κ → NormalType Δ κ
η-norm = reify ∘ reflect
```

Towards writing an evaluator, we define a semantic environment as a function mapping type variables to semantic types.

```
Env : KEnv → KEnv → Set
Env Δ₁ Δ₂ = ∀ {κ} → TVar Δ₁ κ → SemType Δ₂ κ
```

Environment extension and lifting can be written in a straightforward manner.

```
extende : (η : Env Δ₁ Δ₂) → (V : SemType Δ₂ κ) → Env (Δ₁ ,, κ) Δ₂
lifte : Env Δ₁ Δ₂ → Env (Δ₁ ,, κ) (Δ₂ ,, κ)
```

The identity environment now maps type variables to semantic types. Unlike in Chapman et al. [2019], this environment can no longer be truly said to be an identity: type variables are de facto put into η-long form during reflection. However this change is mandatory for normalization, so we cannot define an environment that does not.

```
idEnv : Env Δ Δ
idEnv = reflect ∘ `
```

4.3 Helping evaluation

In aid of writing an evaluator, we found it helpful to develop *semantic* notions of the syntax introduced by R$\omega\mu$. For example, we define a type synonym for application, which is simply Agda application within the identity renaming.

```
_·V_ : SemType Δ (κ₁ `→ κ₂) → SemType Δ κ₁ → SemType Δ κ₂
F ·V V = F id V
```

We can further define the constructors of the three canonical forms of row-kinded types:

```
_▹V_ : SemType Δ L → SemType Δ κ → SemType Δ R[ κ ]
_▹V_ {κ = κ} ℓ τ = just (right (ℓ , τ))

ne-R : NeutralType Δ R[ κ ] → SemType Δ R[ κ ]
ne-R = just ∘ left

ϵV : SemType Δ R[ κ ]
ϵV = nothing
```

The definition of semantic row mapping varies by the shape of the row V over which we are lifting. If V is neutral, so too must the mapping of F over !V! be neutral. Hence we reify F to normal form and leave its mapping in neutral form. If V is a labeled row `(l ▹ τ)`, we push the application of F over τ. Finally, if V is the empty row, its mapping is empty.

```
_<$>V_ : SemType Δ (κ₁ `→ κ₂) → SemType Δ R[ κ₁ ] → SemType Δ R[ κ₂ ]
_<$>V_ {κ₁ = κ₁} {κ₂} F (just (left x)) = ne-R (reifyKripke F <$> x)
_<$>V_ {κ₁ = κ₁} {κ₂} F (just (right (l , τ))) = (l ▹V (F ·V τ))
_<$>V_ {κ₁ = κ₁} {κ₂} F nothing = ϵV
```

Although the flap operator _<?>_ is expressible as a special case of row mapping, we nevertheless find it a useful abstraction to express as a semantic function. It is defined below in terms of semantic row mapping; we find it likewise helpful to give a type synonym `apply` to the left hand side of this equation.

```
apply : SemType Δ κ₁ → SemType Δ ((κ₁ `→ κ₂) `→ κ₂)
apply a = λ ρ F → F ·V (renSem ρ a)

infixr 0 _<?>V_
_<?>V_ : SemType Δ R[ κ₁ `→ κ₂ ] → SemType Δ κ₁ → SemType Δ R[ κ₂ ]
f <?>V a = apply a <$>V f
```

Much of the latent computation in R$\omega\mu$ occurs under an outermost Π and Σ syntax. To this end, we chose to represent Π and Σ as arrow-kinded type-constants—meaning they will evaluate into Agda functions. This provides an opportunity to concisely abstract their reduction logic. We define a semantic combinator for the Π type constant below. The first two equations state that record types at ⋆ and label kind may be formed provided normal bodies; The third equation pushes the λ-binding of F outside of the record type; the fourth equation states that application *is* mapping at nested row kind.

```
ΠV : SemType Δ R[ κ ] → SemType Δ κ
ΠV {κ = ★} x = Π (reify x)
ΠV {κ = L} x = ΠL (reify x)
ΠV {κ = κ₁ `→ κ₂} F = λ ρ v → ΠV (renSem ρ F <?>V v)
ΠV {κ = R[ κ ]} x = (λ ρ v → ΠV v) <$>V x
```

We can turn the semantic helper ΠV into a true Kripke function easily:

```
Π-Kripke : KripkeFunction Δ R[ κ ] κ
Π-Kripke = λ ρ v → ΠV v
```

We omit the definitions of ΣV and Σ-Kripke, as they are identical modulo the use of Π constants.

4.4 Evaluation

We now write an evaluator that translates Types to semantic types; that is, translating syntactic forms to the semantic domain. A normalizer composes reification with evaluation. One can see this in the definition of evalPred, the predicate normalizer. (Predicates must be fully normalized as they do not have a semantic image.)

```
eval : Type Δ₁ κ → Env Δ₁ Δ₂ → SemType Δ₂ κ
evalPred : Pred Δ₁ R[ κ ] → Env Δ₁ Δ₂ → NormalPred Δ₂ R[ κ ]

evalPred (ρ₁ · ρ₂ ~ ρ₃) η = reify (eval ρ₁ η) · reify (eval ρ₂ η) ~ reify (eval ρ₃ η)
evalPred (ρ₁ ≲ ρ₂) η = reify (eval ρ₁ η) ≲ reify (eval ρ₂ η)
```

Evaluation is defined by induction over the type structure. The first three cases have types which may occur at any kind. The variable case simply uses the environment to perform a lookup; application defers to our semantic combinator _·V_; and evaluation of arrow types is defined recursively.

```
eval {κ = κ} (‘ x) η = η x
eval {κ = κ} (τ₁ · τ₂) η = (eval τ₁ η) ·V (eval τ₂ η)
eval {κ = κ} (τ₁ ‘→ τ₂) η = (eval τ₁ η) ‘→ (eval τ₂ η)
```

The next four cases are for types that only occur at kind ★. The qualified type and label singleton cases proceed by recursion over the type structure. For `∀-bound types, we must lift the environment η appropriately. In the μ case, τ has kind $\star \to \star$ and so its evaluation must be reified back to NormalType.

```
eval {κ = ★} (π ⇒ τ) η = evalPred π η ⇒ eval τ η
eval {κ = ★} ⌊ τ ⌋ η = ⌊ eval τ η ⌋
eval {κ = ★} (‘∀ τ) η = ‘∀ (eval τ (lifte η))
eval {κ = ★} (μ τ) η = μ (reify (eval τ η))
```

There is only one type with exclusively label kind. Its definition is unsurprising (it houses only a String label).

```
eval {κ = L} (lab l) η = lab l
```

We evaluate λ-bound functions by evaluating their bodies in environments extended by the meaning their input v. Note that we are building a Kripke function and so ρ is a renaming from Δ_1 to Δ_2 and v is an input of type SemType Δ_2 κ_1.

```
eval {κ = κ₁ ‘→ κ₂} (‘λ τ) η = λ ρ v → eval τ (extende (λ {κ} v’ → renSem {κ = κ} ρ (η v’)) v)
```

Lastly, we define evaluation over the row-kinded constants and operators. As Π and Σ are represented as type constants in the Type syntax, they translate directly to the Kripke functions we defined for Π and Σ as semantic helpers. Likewise, the row mapping and labeled-row cases are interpreted immediately and desirably by their semantic helpers.

```
eval {κ = R[ κ ] ‘→ κ} Π η = Π-Kripke
eval {κ = R[ κ ] ‘→ κ} Σ η = Σ-Kripke
eval {κ = R[ κ ]} (f <$> a) η = (eval f η) <$>V (eval a η)
eval {κ = _} (l ▹ τ) η = (eval l η) ▹V (eval τ η)
eval ϵ η = ϵV
```

Finally, we define a normalizer as the reification of evaluation.

```
⇓ : ∀ {Δ} → Type Δ κ → NormalType Δ κ
⇓ τ = reify (eval τ idEnv)

⇓NE : ∀ {Δ} → NeutralType Δ κ → NormalType Δ κ
⇓NE τ = reify (eval (⇑NE τ) idEnv)
```

5 METATHEORY

We now verify that ⇓ indeed behaves as a normalization function ought to. We first show that normalization is *stable*. Stability states that embedding ⇑ is a right-inverse to normalization ⇓, or, in categorical terms, that ⇓ is a split-monomorphism. The proof is by induction over τ.

```
stability : ∀ (τ : NormalType Δ κ) → ⇓ (⇑ τ) ≡ τ
```

It is desirable that a normalization algorithm adheres to this property, as it states effectively that there is "no more work" to be done by re-normalization. Both idempotency and surjectivity are implied.

```
idempotency : ∀ (τ : Type Δ κ) → (⇑ (⇓ (⇑ (⇓ τ)))) ≡ ⇑ (⇓ τ)
idempotency τ rewrite stability (⇓ τ) = refl

surjectivity : ∀ (τ : NormalType Δ κ) → ∃[ υ ] (⇓ υ ≡ τ)
surjectivity τ = ( ⇑ τ , stability τ )
```

It next falls upon us to verify that this normalization algorithm indeed respects our syntactic account of type equivalence. How we do so is fairly routine to other normalization-by-evaluation efforts. We show that the algorithm is complete with respect to syntactic type equivalence:

```
completeness : ∀ {τ₁ τ₂ : Type Δ κ} → τ₁ ≡t τ₂ → ⇓ τ₁ ≡ ⇓ τ₂
```

Completeness here states that equivalent types normalize to the same types. Soundness states that every type is equivalent to its normalization.

```
soundness : ∀ {Δ₁ κ} → (τ : Type Δ₁ κ) → τ ≡t ⇑ (⇓ τ)
```

Soundness implies the converse of `completeness`, hence we may conclude that τ_1 `≡t` τ_2 iff ⇓ τ_1 ≡ ⇓ τ_2.

```
completeness⁻¹ : ∀ {Δ κ} → (τ₁ τ₂ : Type Δ κ) → ⇓ τ₁ ≡ ⇓ τ₂ → τ₁ ≡t τ₂
completeness⁻¹ τ₁ τ₂ eq =
  eq-trans
    (soundness τ₁)
  (eq-trans
    (inst (cong ⇑ eq))
  (eq-sym (soundness τ₂)))
  where
    inst : ∀ {υ₁ υ₂ : Type Δ κ} → υ₁ ≡ υ₂ → υ₁ ≡t υ₂
    inst refl = eq-refl
```

5.1 A logical relation for completeness

We will prove completeness using a logical relation on semantic types. We would like to be able to equate semantic types, but they prove to be "too large": in particular, our definition of Kripke functions permit functions which may not respect composition of renaming. The solution is to reason about semantic types modulo a partial equivalence relation (PER) that both respects renamings (which we call *uniformity*) and also equates functions extensionally. We write $\tau_1 \approx \tau_2$ to denote that the semantic types τ_1 and τ_2 are equivalent modulo this relation. For clarity, we give names to the two properties (*uniformity* and *point equality*) we desire related types to hold, and define them mutually recursively.

```
_≈_ : SemType Δ κ → SemType Δ κ → Set
PointEqual-≈ : ∀ {Δ₁} {κ₁} {κ₂} (F G : KripkeFunction Δ₁ κ₁ κ₂) → Set
Uniform : ∀ {Δ} {κ₁} {κ₂} → KripkeFunction Δ κ₁ κ₂ → Set
```

We define _≈_ recursively over the kind of its equated types. In the first two cases, τ_1 and τ_2 are normal types, which we equate propositionally. In the third case, we assert that Kripke functions F and G are uniform and point-equal to one another. Uniformity asserts a certain commutativity of renaming: you may either rename the result of applying F to V_1, or you may rename F before applying it to a renamed input. point equality on Kripke functions F and G asserts that F and G take related inputs to related outputs. The latter property is what one should expect of a logical relation; the former property can be attributed to Chapman et al. [2019], who in turn attribute Allais et al. [2013].

```
Uniform {Δ₁} {κ₁} {κ₂} F =
  ∀ {Δ₂ Δ₃} (ρ₁ : Renamingₖ Δ₁ Δ₂) (ρ₂ : Renamingₖ Δ₂ Δ₃) (V₁ V₂ : SemType Δ₂ κ₁) →
  V₁ ≈ V₂ → (renSem ρ₂ (F ρ₁ V₁)) ≈ (renKripke ρ₁ F ρ₂ (renSem ρ₂ V₂))

PointEqual-≈ {Δ₁} {κ₁} {κ₂} F G =
  ∀ {Δ₂} (ρ : Renamingₖ Δ₁ Δ₂) {V₁ V₂ : SemType Δ₂ κ₁} →
  V₁ ≈ V₂ → F ρ V₁ ≈ G ρ V₂

_≈_ {κ = ★} τ₁ τ₂ = τ₁ ≡ τ₂
_≈_ {κ = L} τ₁ τ₂ = τ₁ ≡ τ₂
_≈_ {Δ₁} {κ = κ₁ ‘→ κ₂} F G =
  Uniform F × Uniform G × PointEqual-≈ {Δ₁} F G
```

The last six cases are over row kinded semantic types. The first case states that neutral rows must be propositionally equal; the second states that two rows of the form $(\{l_1\}\tau_1)$ and $(\{l_2\}\tau_2)$ are related iff their labels are equal and their types are related. The third case states that the empty row is related to itself (which is always true). All other cases are nonsensical, and so are set to ⊥.

```
_≈_ {κ = R[ κ ]} (just (left x)) (just (left y)) = x ≡ y
_≈_ {κ = R[ κ ]} (just (right (l₁ , τ₁))) (just (right (l₂ , τ₂))) = l₁ ≡ l₂ × τ₁ ≈ τ₂
_≈_ {κ = R[ κ ]} nothing nothing              = ⊤
_≈_ {κ = R[ κ ]} (just _) (just _)            = ⊥
_≈_ {κ = R[ κ ]} (just _) nothing             = ⊥
_≈_ {κ = R[ κ ]} nothing (just _)             = ⊥
```

5.1.1 Properties of the completeness relation. The completeness relation forms a *partial equivalence relation* (PER). As uniformity is a unary property, it follows quickly that `_≈_` cannot be reflexive, but a limited form of reflexivity does hold: provided that `V` is related to *some* other `V'`, it relates to itself. The other properties (symmetry and transitivity) are simple enough to show. We introduce two helpers, `refl-≈`$_l$ and `refl-≈`$_r$ to describe left and right reflexive projections.

```
refl-≈l : ∀ {V₁ V₂ : SemType Δ κ} → V₁ ≈ V₂ → V₁ ≈ V₁
refl-≈r : ∀ {V₁ V₂ : SemType Δ κ} → V₁ ≈ V₂ → V₂ ≈ V₂
sym-≈ : ∀ {τ₁ τ₂ : SemType Δ κ} → τ₁ ≈ τ₂ → τ₂ ≈ τ₁
trans-≈ : ∀ {τ₁ τ₂ τ₃ : SemType Δ κ} → τ₁ ≈ τ₂ → τ₂ ≈ τ₃ → τ₁ ≈ τ₃
```

we commonly invoke two main lemmas. `reflect-≈` reflects propositional equality to semantic equivalence, and `reify-≈` reifies equivalent semantic types to propositional equality. We make great use of the latter lemma, which states intuitively that related types should have the same reifications. One may alternatively think of this lemma as congruence of reification modulo semantic equivalence.

```
reflect-≈ : ∀ {τ₁ τ₂ : NeutralType Δ κ} → τ₁ ≡ τ₂ → reflect τ₁ ≈ reflect τ₂
reify-≈   : ∀ {τ₁ τ₂ : SemType Δ κ} → τ₁ ≈ τ₂ → reify τ₁ ≡ reify τ₂
```

5.2 The fundamental theorem & completeness

We would like to show that all well-kinded equivalent types have semantically equivalent evaluations. Completeness follows shortly thereafter. The fundamental theorem for completeness (`fundC`) states that equivalent types evaluate to related types under related environments. Towards this goal, we first define a point-wise equivalence on semantic environments.

```
Env-≈ : (η₁ η₂ : Env Δ₁ Δ₂) → Set
Env-≈ η₁ η₂ = ∀ {κ} (x : TVar _ κ) → (η₁ x) ≈ (η₂ x)
```

We show that related environments remain related when extended with related arguments.

```
extend-≈ : ∀ {η₁ η₂ : Env Δ₁ Δ₂} → Env-≈ η₁ η₂ →
              {V₁ V₂ : SemType Δ₂ κ} →
              V₁ ≈ V₂ →
              Env-≈ (extende η₁ V₁) (extende η₂ V₂)
```

It is easy to show as well that the identity environment relates to itself.

```
idEnv-≈ : ∀ {Δ} → Env-≈ (idEnv {Δ}) (idEnv {Δ})
idEnv-≈ x = reflect-≈ refl
```

We may now state the fundamental theorem for completeness. Again, as we have no semantic image of predicates, the fundamental theorem for predicates simply asserts that the evaluation of equivalent predicates are propositional equal.

```
fundC : ∀ {τ₁ τ₂ : Type Δ₁ κ} {η₁ η₂ : Env Δ₁ Δ₂} →
          Env-≈ η₁ η₂ → τ₁ ≡t τ₂ → eval τ₁ η₁ ≈ eval τ₂ η₂
fundC-pred : ∀ {π₁ π₂ : Pred Δ₁ R[ κ ]} {η₁ η₂ : Env Δ₁ Δ₂} →
               Env-≈ η₁ η₂ → π₁ ≡p π₂ → evalPred π₁ η₁ ≡ evalPred π₂ η₂
```

Completeness follows immediatelly as a special case of the fundamental theorem.

```
Completeness : ∀ {τ₁ τ₂ : Type Δ κ} → τ₁ ≡t τ₂ → ⇓ τ₁ ≡ ⇓ τ₂
Completeness eq = reify-≈ (fundC idEnv-≈ eq)
```

5.3 Soundness

Soundness states that every type is equivalent to its normalization. Intuitively, completeness tells us that all "computation" inherent in the equivalence relation is captured by normalization; coversely, soundness tells us that all computation inherent in the normalization algorithm is declared in the equivalence relation.

5.3.1 A logical relation. We prove soundness by a separate logical relation that relates (unnormalized) types to semantic types. We write ⟦ τ ⟧≈ V to denote that the type τ is related to the semantic type V. This syntax is inspired by the result we wish to show: that evaluating τ yields a semantic type V. We give the type synonym `SoundKripke` for the functional case.

```
infix 0 ⟦_⟧≈_
⟦_⟧≈_ : ∀ {κ} → Type Δ κ → SemType Δ κ → Set
SoundKripke : Type Δ₁ (κ₁ ‘→ κ₂) → KripkeFunction Δ₁ κ₁ κ₂ → Set
```

In two of the ground cases, V is a normal type, and so we simply assert type equivalence with the normal type's embedding.

```
⟦_⟧≈_ {κ = ★} τ V = τ ≡t ⇑ V
⟦_⟧≈_ {κ = L} τ V = τ ≡t ⇑ V
```

In the row case, we assert (resp.) that (i) if τ relates to `nothing` then it must be equivalent to the empty row; (ii) if τ relates to a neutral row then it must be equivalent to a neutral row; and (iii) if τ relates to a labeled rows then it must be equivalent to a labeled row and that labeled row's component type must relate to itself.

```
⟦_⟧≈_ {κ = R[ κ ]} τ nothing = τ ≡t ϵ
⟦_⟧≈_ {κ = R[ κ ]} τ (just (left n)) = τ ≡t (⇑NE n)
⟦_⟧≈_ {κ = R[ κ ]} τ (just (right (l , υ))) = (τ ≡t ⇑ (l ▹ reify υ)) × (⟦ ⇑ (reify υ) ⟧≈ υ)
```

In the functional case, we assert that logically related functions map related inputs to related outputs.

```
⟦_⟧≈_ {Δ₁} {κ = κ₁ ‘→ κ₂} f F = SoundKripke f F
SoundKripke {Δ₁ = Δ₁} {κ₁ = κ₁} {κ₂ = κ₂} f F =
  (∀ {Δ₂} (ρ : Renamingₖ Δ₁ Δ₂) {v V} →
    ⟦ v ⟧≈ V →
    ⟦ (renₖ ρ f · v) ⟧≈ (renKripke ρ F ·V V))
```

5.4 Properties of the soundness relation

We reflect type equivalence to the relation and reify the relation to type equivalence as so.

```
reflect-⟦⟧≈ : ∀ {τ : Type Δ κ} {υ : NeutralType Δ κ} →
              τ ≡t ⇑NE υ → ⟦ τ ⟧≈ (reflect υ)
reify-⟦⟧≈ : ∀ {τ : Type Δ κ} {V : SemType Δ κ} →
              ⟦ τ ⟧≈ V → τ ≡t ⇑ (reify V)
```

5.4.1 The fundamental theorem & soundness. Towards defining the fundamental theorem, we first define a relation between syntactic environments (substitutions) and semantic environments. Intuitively, the substitution σ is related to the environment η if each type mapped to by σ point-wise relates to the semantic type mapped to by η.

```
⟦_⟧≈e_ : ∀ {Δ₁ Δ₂} → Substitutionₖ Δ₁ Δ₂ → Env Δ₁ Δ₂ → Set
⟦_⟧≈e_ {Δ₁} σ η = ∀ {κ} (α : TVar Δ₁ κ) → ⟦ (σ α) ⟧≈ (η α)
```

The fundamental theorem for soundness states that the substitution of τ by σ is related to the evaluation of τ by η. Intuitively, substitution may be thought of as a syntactic notion of evaluation, and hence we are stating that syntactic and semantic evaluations relate.

```
fundS : ∀ {Δ₁ Δ₂ κ}(τ : Type Δ₁ κ){σ : Substitutionₖ Δ₁ Δ₂}{η : Env Δ₁ Δ₂} →
          ⟦ σ ⟧≈e η → ⟦ subₖ σ τ ⟧≈ (eval τ η)
```

We show that the identity substitution ` is related to the identity environment:

```
idSR : ∀ {Δ₁} → ⟦ ‘ ⟧≈e (idEnv {Δ₁})
idSR α = reflect-⟦⟧≈ eq-refl
```

and also show that ` is indeed an identity substitution: it fixes the meaning of the type over which it is substituted.

```
subₖ-id : ∀ (τ : Type Δ κ) → subₖ ‘ τ ≡ τ
```

Soundness follows as a special case of the fundamental theorem.

```
Soundness : ∀ {Δ₁ κ} → (τ : Type Δ₁ κ) → τ ≡t ⇑ (⇓ τ)
Soundness τ = subst (_≡t ⇑ (⇓ τ)) (subₖ-id τ) ((reify-⟦⟧≈ (fundS τ idSR)))
```

6 REMARK

6.1 Comparison to Chapman et al. [2019]

Our mechanization has closely resembled that of Chapman et al. [2019]. Our definition of semantic types, however, has differed, as our normalization is with respect to both β- and η-equivalence, whereas Chapman et al's is simply β-equivalence. Changing this definition simplifies some things and complicates others. The definition of semantic types is simpler: whereas Chapman et al permit function types to be interpreted as `NeutralTypes`, ours must be interpreted into solely Kripke function spaces. This complicates the definitions of `reify` and `reflect`, which must become mutually recursive, as we are unable to reflect neutral types at arrow kind to neutral types. We will show later that some of Chapman et al's metatheory relies on neutral forms to not be disturbed by normalization. This complicates the definition of term-level, normality-preserving substitution.

REFERENCES

Guillaume Allais, Pierre Boutillier, and Conor McBride. New equations for neutral terms: A sound and complete decision procedure, formalized, 2013. URL https://arxiv.org/abs/1304.0809.

James Chapman, Roman Kireev, Chad Nester, and Philip Wadler. System F in agda, for fun and profit. In Graham Hutton, editor, *Mathematics of Program Construction - 13th International Conference, MPC 2019, Porto, Portugal, October 7-9, 2019, Proceedings*, volume 11825 of *Lecture Notes in Computer Science*, pages 255–297. Springer, 2019. ISBN 978-3-030-33635-6. doi: 10.1007/978-3-030-33636-3_10. URL https://doi.org/10.1007/978-3-030-33636-3_10.

Alex Hubers and J. Garrett Morris. Generic programming with extensible data types: Or, making ad hoc extensible data types less ad hoc. *Proc. ACM Program. Lang.*, 7(ICFP):356–384, 2023. doi: 10.1145/3607843. URL https://doi.org/10.1145/3607843.

Philip Wadler, Wen Kokke, and Jeremy G. Siek. *Programming Language Foundations in Agda.* August 2022. URL https://plfa.inf.ed.ac.uk/20.08/.