Normalization By Evaluation of Types in R$\omega\mu$

ALEX HUBERS, The University of Iowa, USA

ABSTRACT

We describe the normalization-by-evaluation (NbE) of types in R$\omega\mu$, a row calculus with recursive types, qualified types, and a novel *row complement* operator. Types are normalized to $\beta\eta$-long forms modulo a type equivalence relation. Because the type system of R$\omega\mu$ is a strict extension of System F$\omega\mu$, much of the type reduction is isomorphic to reduction of terms in the STLC. Novel to this report are the reductions of row, record, and variant types.

1 THE R$\omega\mu$ CALCULUS

For reference, Figure 1 describes the syntax of kinds, predicates, and types in R$\omega\mu$. We forego further description to the next section.

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

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

Fig. 1. Syntax

1.1 Example types

Wand's problem and a record modifier:

```
wand : ∀ l x y z t. x ⊙ y ~ z, {l ▹ t} ≲ z ⇒ #l → Π x → Π y → t
modify : ∀ l t u y z1 z2. {l ▹ t} ⊙ y ~ z1, {l ▹ u} ⊙ y ~ z2 ⇒
         #l → (t → u) → Π z1 → Π z2
```

"Deriving" functor typeclass instances:

```
type Functor : (★ → ★) → ★
type Functor = λf. ∀ a b. (a → b) → f a → f b

fmapS : ∀ z : R[★ → ★]. Π (Functor z) → Functor (Σ z)
fmapP : ∀ z : R[★ → ★]. Π (Functor z) → Functor (Π z)
```

And a desugaring of booleans to Church encodings:

```
desugar : ∀ y. BoolF ≲ y, LamF ≲ y \ BoolF ⇒
          Π (Functor (y \ BoolF)) → μ (Σ y) → μ (Σ (y \ BoolF))
```

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

2 MECHANIZED 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 or indexing-by untyped syntax. The only "untyped" syntax is that of kinds, which are well-formed grammatically. We give the syntax of kinds and kinding environments 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; $(\rightarrow)$ 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.

```
data KEnv : Set where
  ∅ : 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. Type variables indexed in this way are analogous to the `_∈_` relation for Agda lists—that is, each type variable is itself a proof of its location within the kinding environment.

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

2.1.1 Partitioning kinds. It will be necessary to partition kinds by two predicates. The predicate `NotLabel κ` is satisfied if κ is neither of label kind, a row of label kind, nor a type operator that returns a labeled kind. It is trivial to show that this predicate is decidable.

```
NotLabel : Kind → Set
NotLabel ★ = ⊤
NotLabel L = ⊥
NotLabel (κ₁ `→ κ₂) = NotLabel κ₂
NotLabel R[ κ ] = NotLabel κ
```

```
notLabel? : ∀ κ → Dec (NotLabel κ)
notLabel? ★ = yes tt
notLabel? L = no λ ()
notLabel? (κ `→ κ₁) = notLabel? κ₁
notLabel? R[ κ ] = notLabel? κ
```

The predicate `Ground κ` is satisfied when κ is the kind of types or labels, and is necessary to reserve the promotion of neutral types to just those at these kinds. It is again trivial to show that this predicate is decidable, and so a definition of `ground?` is omitted.

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

2.2 Type syntax

We represent the judgment $\Gamma \vdash \tau : \kappa$ intrinsically as the data type `Type Δ κ`. The data type `Pred Type Δ R[ κ ]` represents well-kinded predicates indexed by `Type Δ κ`. The two are necessarily mutually inductive. Note that the syntax of predicates will be the same for both types and normalized types, and so the `Pred` data type is indexed abstractly by type `Ty`.

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

We must also define syntax for *simple rows*, that is, row literals. For uniformity of kind indexing, we define a `SimpleRow` by pattern matching on the syntax of kinds. Like with `Pred`, simple rows are indexed by abstract type `Ty` so that we may reuse the same pattern for normalized types.

```
SimpleRow : (Ty : KEnv → Kind → Set) → KEnv → Kind → Set
SimpleRow Ty Δ R[ κ ] = List (Label × Ty Δ κ)
SimpleRow _ _ _ = ⊥
```

A simple row is *ordered* if it is of length ≤ 1 or its corresponding labels are ordered according to some total order $<$. We will restrict the formation of row literals to just those that are ordered, which has two key consequences: first, it guarantees a normal form (later) for simple rows, and second, it enforces that labels be unique in each row. It is easy to show that the `Ordered` predicate is decidable.

```
Ordered : SimpleRow Type Δ R[ κ ] → Set
ordered? : ∀ (xs : SimpleRow Type Δ R[ κ ]) → Dec (Ordered xs)
Ordered [] = ⊤
Ordered (x ∷ []) = ⊤
Ordered ((l₁ , _) ∷ (l₂ , τ) ∷ xs) = l₁ < l₂ × Ordered ((l₂ , τ) ∷ xs)
```

The syntax of well-kinded predicates is exactly as expected.

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

The syntax of kinding judgments is given below. The formation rules for λ-abstractions, applications, arrow types, and $\forall$ and μ types are standard and omitted.

```
data Type Δ where
  ‘ : (α : TVar Δ κ) → Type Δ κ
```

The constructor _⇒_ forms a qualified type given a well-kinded predicate π and a $\star$-kinded body τ.

```
_⇒_ : (π : Pred Type Δ R[ κ₁ ]) → (τ : Type Δ ★) → Type Δ ★
```

Labels are formed from label literals and cast to kind $\star$ via the ⌊_⌋ constructor.

```
lab : (l : Label) → Type Δ L
⌊_⌋ : (τ : Type Δ L) → Type Δ ★
```

We finally describe row formation. The constructor ⦅_⦆ forms a row literal from a well-ordered simple row. We additionally allow the syntax _▹_ for constructing row singletons of (perhaps) variable label; this role can be performed by ⦅_⦆ when the label is a literal. The _<$>_ constructor describes the map of a type operator over a row. Π and Σ form records and variants from rows for which the `NotLabel` predicate is satisfied. Finally, the __ constructor forms the relative complement of two rows. The novelty in this report will come from showing how types of these forms reduce.

```
⦅_⦆ : (xs : SimpleRow Type Δ R[ κ ]) (ordered : True (ordered? xs)) → Type Δ R[ κ ]
_▹_ : (l : Type Δ L) → (τ : Type Δ κ) → Type Δ R[ κ ]
_<$>_ : (ϕ : Type Δ (κ₁ ‘→ κ₂)) → (τ : Type Δ R[ κ₁ ]) → Type Δ R[ κ₂ ]
Π : {notLabel : True (notLabel? κ)} → Type Δ (R[ κ ] ‘→ κ)
Σ : {notLabel : True (notLabel? κ)} → Type Δ (R[ κ ] ‘→ κ)
_\_ : Type Δ R[ κ ] → Type Δ R[ κ ] → Type Δ R[ κ ]
```

2.2.1 The ordered predicate. We impose on the ⦅_⦆ constructor a witness of the form `True (ordered? xs)`, although it may seem more intuitive to have instead simply required a witness that `Ordered xs`. The reason for this is that the `True` predicate quotients each proof down to a single inhabitant `tt`, which grants us proof irrelevance when comparing rows. This is desirable and yields congruence rules that would otherwise be blocked by two differing proofs of well-orderedness. The congruence rule below asserts that two simple rows are equivalent even with differing proofs. (This pattern is replicable for any decidable predicate.)

```
cong-SimpleRow : {sr₁ sr₂ : SimpleRow Type Δ R[ κ ]}
                 {wf₁ : True (ordered? sr₁)} {wf₂ : True (ordered? sr₂)} →
                 sr₁ ≡ sr₂ → ⦅ sr₁ ⦆ wf₁ ≡ ⦅ sr₂ ⦆ wf₂
cong-SimpleRow {sr₁ = sr₁} {_} {wf₁} {wf₂} refl
  rewrite Dec→Irrelevant (Ordered sr₁) (ordered? sr₁) wf₁ wf₂ = refl
```

In the same fashion, we impose on Π and Σ a similar restriction that their kinds satisfy the `NotLabel` predicate, although our reason for this restriction is instead metatheoretic: without it, nonsensical labels could be formed such as Π `(lab "a" ▹ lab "b")` or Π ϵ. Each of these types have kind L, which violates a label canonicity theorem we later show that all label-kinded types in normal form are label literals or neutral.

2.2.2 Flipped map operator.

Hubers and Morris [2023] had a left- and right-mapping operator, but only one is necessary. The flipped application (flap) operator is defined below. Its type reveals its purpose.

```
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
```

2.2.3 The (syntactic) complement operator.

It is necessary to give a syntactic account of the computation incurred by the complement of two row literals so that we can state this computation later in the type equivalence relation. First, define a relation $\ell \in_L \rho$ that is inhabited when the label literal ℓ occurs in the row ρ. This relation is decidable (_∈L?_, definition omitted).

```
data _∈L_ : (l : Label) → SimpleRow Type Δ R[ κ ] → Set where
  Here : ∀ {τ : Type Δ κ} {xs : SimpleRow Type Δ R[ κ ]} {l : Label} →
         l ∈L (l , τ) :: xs
  There : ∀ {τ : Type Δ κ} {xs : SimpleRow Type Δ R[ κ ]} {l l’ : Label} →
          l ∈L xs → l ∈L (l’ , τ) :: xs
_∈L?_ : ∀ (l : Label) (xs : SimpleRow Type Δ R[ κ ]) → Dec (l ∈L xs)
```

We now define the syntactic *row complement* effectively as a filter: when a label on the left is found in the row on the right, we exclude that labeled entry from the resulting row.

```
_\s_ : ∀ (xs ys : SimpleRow Type Δ R[ κ ]) → SimpleRow Type Δ R[ κ ]
[] \s ys = []
((l , τ) :: xs) \s ys with l ∈L? ys
... | yes _ = xs \s ys
... | no _ = (l , τ) :: (xs \s ys)
```

2.2.4 Type renaming and substitution.

A type variable renaming is a map from type variables in environment Δ_1 to type variables in environment Δ_2.

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

This definition and approach is standard for the intrinsic style (*cf.* Chapman et al. [2019]; Wadler et al. [2022]) and so definitions are omitted. The only deviation of interest is that we have an obligation to show that renaming preserves the well-orderedness of simple rows. Note that we use the suffix $_k$ for common operations over the Type and Pred syntax; we will use the suffix $_k$NF for equivalent operations over the normal type syntax.

```
orderedRenRowₖ : (r : Renamingₖ Δ₁ Δ₂) → (xs : SimpleRow Type Δ₁ R[ κ ]) → Ordered xs →
                 Ordered (renRowₖ r xs)
```

A substitution is a map from type variables to types.

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

Parallel renaming and substitution is likewise standard for this approach, and so definitions are omitted. As will become a theme, we must show that substitution preserves row well-orderedness.

```
orderedSubRowₖ : (σ : Substitutionₖ Δ₁ Δ₂) → (xs : SimpleRow Type Δ₁ R[ κ ]) → Ordered xs →
                   Ordered (subRowₖ σ xs)
```

Two operations of note: extension of a substitution σ appends a new type A as the zero'th De Bruijn index. β-substitution is a special case of substitution in which we only substitute the most recently freed variable.

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

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

2.3 Type equivalence

We define reduction on types $\tau \longrightarrow_{\mathcal{T}} \tau'$ by directing the following type equivalence judgment $\Delta \vdash \tau = \tau' : \kappa$ from left to right. We equate types under the relation _≡t_, predicates under the relation _≡p_, and row literals under the relation _≡r_.

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

Declare the following as generalized metavariables to reduce clutter. (N.b., generalized variables in Agda are not dependent upon eachother, e.g., it is not true that ρ_1 and ρ_2 must have equal kinds when ρ_1 and ρ_2 appear in the same type signature.)

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

Row literals and predicates are equated in an obvious fashion.

```
data _≡r_ where
  eq-[] : _≡r_ {Δ = Δ} {κ = κ} [] []
  eq-cons : {xs ys : SimpleRow Type Δ R[ κ ]} →
    ℓ₁ ≡ ℓ₂ → τ₁ ≡t τ₂ → xs ≡r ys →
    ((ℓ₁ , τ₁) ∷ xs) ≡r ((ℓ₂ , τ₂) ∷ ys)
```

```
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 type equivalence rules enforce that `_≡t_` forms 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
```

We next have a number of congruence rules. As this is type-level normalization, we equate under binders such as λ and $\forall$. The rule for congruence under λ bindings is below; the remaining congruence rules are omitted.

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

We have two "expansion" rules and one composition rule. Firstly, arrow-kinded types are η-expanded to have an outermost lambda binding. This later ensures canonicity of arrow-kinded types.

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

Analogously, row-kinded variables left alone are expanded to a map by the identity function. Additionally, nested maps are composed together into one map. These rules together ensure canonical forms for row-kinded normal types. Observe that the last two rules are effectively functorial laws.

```
eq-map-id : ∀ {κ} {τ : Type Δ R[ κ ]} → τ ≡t (‘λ {κ₁ = κ} (‘ Z)) <$> τ
eq-map-∘ : ∀ {κ₃} {f : Type Δ (κ₂ ‘→ κ₃)} {g : Type Δ (κ₁ ‘→ κ₂)} {τ : Type Δ R[ κ₁ ]} →
  (f <$> (g <$> τ)) ≡t (‘λ (weakenₖ f · (weakenₖ g · (‘ Z)))) <$> τ
```

We now describe the computational rules that incur type reduction. Rule `eq-β` is the usual β-reduction rule. Rule `eq-labTy` asserts that the constructor `_▹_` is indeed superfluous when describing singleton rows with a label literal; singleton rows of the form `(ℓ ▹ τ)` are normalized into row literals.

```
eq-β : ∀ {τ₁ : Type (Δ ,, κ₁) κ₂} {τ₂ : Type Δ κ₁} →
  ((‘λ τ₁) · τ₂) ≡t (τ₁ βₖ[ τ₂ ])
eq-labTy : l ≡t lab ℓ → (l ▹ τ) ≡t ⦅ [ (ℓ , τ) ] ⦆ tt
```

The rule `eq-▹$` describes that mapping `F` over a singleton row is simply application of `F` over the row's contents. Rule `eq-map` asserts exactly the same except for row literals; the function $\texttt{over}_r$ (definition omitted) is simply fmap over a pair's right component. Rule `eq-<$>-\` asserts that mapping `F` over a row complement is distributive.

```
eq-▹$ : ∀ {l} {τ : Type Δ κ₁} {F : Type Δ (κ₁ ‘→ κ₂)} →
  (F <$> (l ▹ τ)) ≡t (l ▹ (F · τ))
eq-map : ∀ {F : Type Δ (κ₁ ‘→ κ₂)} {ρ : SimpleRow Type Δ R[ κ₁ ]} {oρ : True (ordered? ρ)} →
  F <$> (⦅ ρ ⦆ oρ) ≡t ⦅ map (overᵣ (F ·_)) ρ ⦆ (fromWitness (map-overᵣ ρ (F ·_) (toWitness oρ)))
```

```
eq-<$>-\ : ∀ {F : Type Δ (κ₁ ‘→ κ₂)} {ρ₂ ρ₁ : Type Δ R[ κ₁ ]} →
  F <$> (ρ₂ \ ρ₁) ≡t (F <$> ρ₂) \ (F <$> ρ₁)
```

The rules `eq-Π` and `eq-Σ` give the defining equations of Π and Σ at nested row kind. This is to say, application of Π to a nested row is equivalent to mapping Π over the row.

```
eq-Π : ∀ {ρ : Type Δ R[ R[ κ ] ]} {nl : True (notLabel? κ)} →
     Π {notLabel = nl} · ρ ≡t Π {notLabel = nl} <$> ρ
eq-Σ : ∀ {ρ : Type Δ R[ R[ κ ] ]} {nl : True (notLabel? κ)} →
     Σ {notLabel = nl} · ρ ≡t Σ {notLabel = nl} <$> ρ
```

The next two rules assert that Π and Σ can reassociate from left-to-right except with the new right-applicand "flapped".

```
eq-Π-assoc : ∀ {ρ : Type Δ (R[ κ₁ ‘→ κ₂ ])} {τ : Type Δ κ₁} {nl : True (notLabel? κ₂)} →
  (Π {notLabel = nl} · ρ) · τ ≡t Π {notLabel = nl} · (ρ ?? τ)
eq-Σ-assoc : ∀ {ρ : Type Δ (R[ κ₁ ‘→ κ₂ ])} {τ : Type Δ κ₁} {nl : True (notLabel? κ₂)} →
  (Σ {notLabel = nl} · ρ) · τ ≡t Σ {notLabel = nl} · (ρ ?? τ)
```

Finally, the rule `eq-compl` gives computational content to the relative row complement operator applied to row literals.

```
eq-compl : ∀ {xs ys : SimpleRow Type Δ R[ κ ]}
  {oxs : True (ordered? xs)} {oys : True (ordered? ys)} {ozs : True (ordered? (xs \s ys))} →
  ((⦅ xs ⦆ oxs) \ (⦅ ys ⦆ oys) ≡t ⦅ (xs \s ys) ⦆ ozs
```

Before concluding, we share an auxiliary definition that reflects instances of propositional equality in Agda to proofs of type-equivalence. The same role could be performed via Agda's `subst` but without the convenience.

```
inst : ∀ {τ₁ τ₂ : Type Δ κ} → τ₁ ≡ τ₂ → τ₁ ≡t τ₂
inst refl = eq-refl
```

2.3.1 *Some admissable rules.*

In early versions of this equivalence relation, we thought it would be necessary to impose the following two rules directly. However, we can confirm their admissability. The first rule states that Π is mapped over nested rows, and the second (definition omitted) states that λ-bindings η-expand over Π. (These results hold identically for Σ.)

```
eq-Π▹ : ∀ {l} {τ : Type Δ R[ κ ]}{nl : True (notLabel? κ)} →
       (Π {notLabel = nl} · (l ▹ τ)) ≡t (l ▹ (Π {notLabel = nl} · τ))
eq-Π▹ = eq-trans eq-Π eq-▹$

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

3 NORMAL FORMS

By directing the type equivalence relation we define computation on types. This serves as a sort of specification on the shape normal forms of types ought to have. Our grammar for normal types must be carefully crafted so as to be neither too "large" nor too "small". In particular, we wish our normalization algorithm to be *stable*, which implies surjectivity. Hence if the normal syntax is too large—i.e., it produces junk types—then these junk types will have pre-images in the domain of normalization. Inversely, if the normal syntax is too small, then there will be types whose normal forms cannot be expressed. Figure 2 specifies the syntax and typing of normal types, given as reference. We describe the syntax in more depth by describing its intrinsic mechanization.

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

$$\begin{array}{llcl}
\text{Ground Kinds} & \gamma & ::= & \star \mid \mathsf{L} \\
\text{Kinds} & \kappa & ::= & \gamma \mid \kappa \to \kappa \mid \mathsf{R}^{\kappa} \\
\text{Row Literals} & \hat{\mathcal{P}} \ni \hat{\rho} & ::= & \{\ell_i \triangleright \hat{\tau}_i\}_{i \in 0 \ldots m} \\
\text{Neutral Types} & n & ::= & \alpha \mid n\,\hat{\tau} \\
\text{Normal Types} & \hat{\mathcal{T}} \ni \hat{\tau}, \hat{\phi} & ::= & n \mid \hat{\phi}\,\$\,n \mid \hat{\rho} \mid \hat{\pi} \Rightarrow \hat{\tau} \mid \forall \alpha : \kappa.\hat{\tau} \mid \lambda \alpha : \kappa.\hat{\tau} \\
& & \mid & n \triangleright \hat{\tau} \mid \ell \mid \#\hat{\tau} \mid \hat{\tau} \setminus \hat{\tau} \mid \Pi\,\hat{\tau} \mid \Sigma\,\hat{\tau}
\end{array}$$

Fig. 2. Normal type forms

3.1 Mechanized syntax

We define `NormalTypes` and `NormalPreds` analogously to `Types` and `Preds`. Recall that `Pred` and `SimpleRow` are indexed by the type of their contents, so we can reuse some code.

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

We must declare an analogous orderedness predicate, this time for normal types. Its definition is nearly identical.

```
NormalOrdered : SimpleRow NormalType Δ R[ κ ] → Set
normalOrdered? : ∀ (xs : SimpleRow NormalType Δ R[ κ ]) → Dec (NormalOrdered xs)
```

Further, we define the predicate `NotSimpleRow` ρ to be true precisely when ρ is not a simple row. This is necessary because the row complement $\rho_2 \setminus \rho_1$ should reduce when each ρ_i is a row literal. So it is necessary when forming normal row-complements to specify that at least one of the complement operands is a non-literal. The predicate `True (notSimpleRows?` ρ_1 ρ_2`)` is satisfied precisely in this case.

```
NotSimpleRow : NormalType Δ R[ κ ] → Set
notSimpleRows? : ∀ (τ₁ τ₂ : NormalType Δ R[ κ ]) →
                  Dec (NotSimpleRow τ₁ or NotSimpleRow τ₂)
```

Neutral types are type variables and applications with type variables in head position.

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

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

We define the normal type syntax firstly by restricting the promotion of neutral types to normal forms at only *ground* kind.

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

As discussed above, we restrict the formation of inert row complements to just those in which at least one operand is non-literal.

```
_\_ : (ρ₂ ρ₁ : NormalType Δ R[ κ ]) → {nsr : True (notSimpleRows? ρ₂ ρ₁)} →
      NormalType Δ R[ κ ]
```

We define inert maps as part of the `NormalType` syntax rather than the `NeutralType` syntax. Observe that a consequence of this decision (as opposed to letting the form _<$>_ be neutral) is that all inert maps must have the mapped function composed into just one applicand. For example, the type ϕ_2 `<$>` (ϕ_1 `n`) must recompose into (`` `λ α. (ϕ₂ (ϕ₁ α)) <$> n ``) to be in normal form.

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

we need only permit the formation of records and variants at kind ★, and we restrict the formation of neutral-labeled rows to just the singleton constructor $\_\triangleright_n\_$.

```
Π : (ρ : NormalType Δ R[ ★ ]) → NormalType Δ ★
Σ : (ρ : NormalType Δ R[ ★ ]) → NormalType Δ ★
_▹ₙ_ : (l : NeutralType Δ L) (τ : NormalType Δ κ) → NormalType Δ R[ κ ]
```

The remaining cases are identical to the regular `Type` syntax and omitted.

3.2 Canonicity of normal types

The syntax of normal types is defined precisely so as to enjoy canonical forms based on kind. We first demonstrate that neutral types and inert complements cannot occur in empty contexts.

```
noNeutrals : NeutralType ∅ κ → ⊥

noNeutrals (n · τ) = noNeutrals n

noComplements : ∀ {ρ₁ ρ₂ ρ₃ : NormalType ∅ R[ κ ]}
                (nsr : True (notSimpleRows? ρ₃ ρ₂)) →
                ρ₁ ≡ (ρ₃ \ ρ₂) {nsr} →
                ⊥
```

Now, in any context an arrow-kinded type is canonically λ-bound:

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

A row in an empty context is necessarily a row literal:

```
row-canonicity-∅ : (ρ : NormalType ∅ R[ κ ]) →
                    ∃[ xs ] Σ[ oxs ∈ True (normalOrdered? xs) ]
```

```
                    (ρ ≡ ⦅ xs ⦆ oxs)
row-canonicity-∅ (⦅ ρ ⦆ oρ) = ρ , oρ , refl
row-canonicity-∅ (ne x) = ⊥-elim (noNeutrals x)
row-canonicity-∅ ((ρ \ ρ₁) {nsr}) = ⊥-elim (noComplements nsr refl)
row-canonicity-∅ (l ▹ₙ ρ) = ⊥-elim (noNeutrals l)
row-canonicity-∅ ((ϕ <$> ρ)) = ⊥-elim (noNeutrals ρ)
```

And a label-kinded type is necessarily a label literal:

```
label-canonicity-∅ : ∀ (l : NormalType ∅ L) → ∃[ s ] (l ≡ lab s)
label-canonicity-∅ (ne x) = ⊥-elim (noNeutrals x)
label-canonicity-∅ (lab s) = s , refl
```

3.3 Renaming

Renaming over normal types is defined in an entirely straightforward manner. Types and definitions are omitted.

3.4 Embedding

The goal is to normalize a given τ : `Type` Δ κ to a normal form at type `NormalType` Δ κ. It is of course much easier to first describe the inverse embedding, which recasts a normal form back to its original type. Definitions are expected and omitted.

```
⇑ : NormalType Δ κ → Type Δ κ
⇑Row : SimpleRow NormalType Δ R[ κ ] → SimpleRow Type Δ R[ κ ]
⇑NE : NeutralType Δ κ → Type Δ κ
⇑Pred : NormalPred Δ R[ κ ] → Pred Type Δ R[ κ ]
```

Note that it is precisely in "embedding" the `NormalOrdered` predicate that we establish half of the requisite isomorphism between a normal row being normal-ordered and its embedding being ordered. We will have to show the other half (that is, that ordered rows have normal-ordered evaluations) during normalization.

```
Ordered⇑ : ∀ (ρ : SimpleRow NormalType Δ R[ κ ]) → NormalOrdered ρ →
             Ordered (⇑Row ρ)
```

4 SEMANTIC TYPES

We have finally set the stage to discuss the process of normalizing types by evaluation. We first must define a semantic image of `Type`s into which we will evaluate. Crucially, neutral types must *reflect* into this domain, and elements of this domain must *reify* to normal forms.

Let us first define the image of row literals to be `Fin`-indexed maps.

```
Row : Set → Set
Row A = ∃[ n ](Fin n → Label × A)
```

Naturally, we required a predicate on such rows to indicate that they are well-ordered.

```
OrderedRow' : ∀ {A : Set} → (n : ℕ) → (Fin n → Label × A) → Set
OrderedRow' zero P = ⊤
OrderedRow' (suc zero) P = ⊤
```

```
OrderedRow' (suc (suc n)) P = (P fzero .fst < P (fsuc fzero) .fst) × OrderedRow' (suc n) (P ∘ fsuc)

OrderedRow : ∀ {A} → Row A → Set
OrderedRow (n , P) = OrderedRow' n P
```

We may now define the totality of forms a row-kinded type might take in the semantic domain (the `RowType` data type). We evaluate row literals into `Rows` via the `row` constructor; note that the argument $\mathcal{T}$ maps kinding environments to types. In practice, this is how we specify that a `row` contains types in environment Δ.

```
data RowType (Δ : KEnv) (𝒯 : KEnv → Set) : Kind → Set
NotRow : ∀ {Δ : KEnv} {𝒯 : KEnv → Set} → RowType Δ 𝒯 R[ κ ] → Set

data RowType Δ 𝒯 where
  row : (ρ : Row (𝒯 Δ)) → OrderedRow ρ → RowType Δ 𝒯 R[ κ ]
```

Neutral-labeled singleton rows are evaluated into the `_▹_` constructor; inert complements are evaluated into the `_\_` constructor. Just as `OrderedRow` is the semantic version of row well-orderedness, the predicate `NotRow` asserts that a given `RowType` is not a row literal (constructed by `row`). This ensures that complements constructed by `_\_` are indeed inert.

```
_▹_ : NeutralType Δ L → 𝒯 Δ → RowType Δ 𝒯 R[ κ ]
_\_ : (ρ₂ ρ₁ : RowType Δ 𝒯 R[ κ ]) → {nr : NotRow ρ₂ or NotRow ρ₁} →
      RowType Δ 𝒯 R[ κ ]
```

We would like to compose nested maps. Borrowing from Allais et al. [2013], we thus interpret the left applicand of a map as a Kripke function space mapping neutral types in environment `Δ'` to the type `𝒯 Δ'`, which we will later specify to be that of semantic types in environment `Δ'` at kind κ. To avoid running afoul of Agda's positivity checker, we let the domain type of this Kripke function be *neutral types*, which may always be reflected into semantic types. We define semantic types (`SemType`) below, but replacing `NeutralType Δ' κ₁` with `SemType Δ' κ₁` would not be strictly positive.

```
_<$>_ : (ϕ : ∀ {Δ'} → Renaming_k Δ Δ' → NeutralType Δ' κ₁ → 𝒯 Δ') →
        NeutralType Δ R[ κ₁ ] →
        RowType Δ 𝒯 R[ κ₂ ]
```

We finally define the semantic domain by induction on the kind κ. Types with ★ and label kind are simply `NormalTypes`.

```
SemType : KEnv → Kind → Set
SemType Δ ★ = NormalType Δ ★
SemType Δ L = NormalType Δ L
```

We interpret functions into *Kripke function spaces*—that is, functions that operate over `SemType` inputs at any possible environment Δ_2, provided a renaming into Δ_2.

```
SemType Δ₁ (κ₁ `→ κ₂) = (∀ {Δ₂} → (r : Renaming_k Δ₁ Δ₂)
                          (v : SemType Δ₂ κ₁) → SemType Δ₂ κ₂)
```

We interpret row-kinded types into the `RowType` type, defined above. Note some more trickery which we have borrowed from Allais et al. [2013]: we cannot pass `SemType` itself as an argument to `RowType` (which would violate termination checking), but we can instead pass to `RowType` the function `(λ Δ' → SemType Δ' κ)`, which enforces a strictly smaller recursive call on the kind κ. Observe too that abstraction over the kinding environment `Δ'` is necessary because our representation of inert maps `_<$>_` interprets the mapped applicand as a Kripke function space over neutral type domain.

```
SemType Δ R[ κ ] = RowType Δ (λ Δ' → SemType Δ' κ) R[ κ ]
```

For abbreviation later, we alias our two types of Kripke function spaces as so:

```
KripkeFunction : KEnv → Kind → Kind → Set
KripkeFunctionNE : KEnv → Kind → Kind → Set
KripkeFunction Δ₁ κ₁ κ₂ = (∀ {Δ₂} → Renamingₖ Δ₁ Δ₂ →
  SemType Δ₂ κ₁ → SemType Δ₂ κ₂)
KripkeFunctionNE Δ₁ κ₁ κ₂ = (∀ {Δ₂} → Renamingₖ Δ₁ Δ₂ →
  NeutralType Δ₂ κ₁ → SemType Δ₂ κ₂)
```

4.1 Renaming

Renaming over normal types is defined in a straightforward manner. Observe that renaming a Kripke function is nothing more than providing the appropriate renaming to the function.

```
renKripke : Renamingₖ Δ₁ Δ₂ → KripkeFunction Δ₁ κ₁ κ₂ → KripkeFunction Δ₂ κ₁ κ₂
renKripke {Δ₁} ρ F {Δ₂} = λ ρ' → F (ρ' ∘ ρ)
```

We will make some reference to semantic renaming, so we give it the name `renSem` here. Its definition is expected.

```
renSem : Renamingₖ Δ₁ Δ₂ → SemType Δ₁ κ → SemType Δ₂ κ
```

5 NORMALIZATION BY EVALUATION

```
reflect : ∀ {κ} → NeutralType Δ κ → SemType Δ κ
reify : ∀ {κ} → SemType Δ κ → NormalType Δ κ

reflect {κ = ★} τ       = ne τ
reflect {κ = L} τ       = ne τ
reflect {κ = R[ κ ]} ρ = (λ r n → reflect n) <$> ρ
reflect {κ = κ₁ `→ κ₂} τ = λ ρ v → reflect (renₖNE ρ τ · reify v)

reifyKripke : KripkeFunction Δ κ₁ κ₂ → NormalType Δ (κ₁ `→ κ₂)
reifyKripkeNE : KripkeFunctionNE Δ κ₁ κ₂ → NormalType Δ (κ₁ `→ κ₂)
reifyKripke {κ₁ = κ₁} F = `λ (reify (F S (reflect {κ = κ₁} ((` Z)))))
reifyKripkeNE F = `λ (reify (F S (` Z)))

reifyRow' : (n : ℕ) → (Fin n → Label × SemType Δ κ) → SimpleRow NormalType Δ R[ κ ]
reifyRow' zero P = []
reifyRow' (suc n) P with P fzero
... | (l , τ) = (l , reify τ) :: reifyRow' n (P ∘ fsuc)

reifyRow : Row (SemType Δ κ) → SimpleRow NormalType Δ R[ κ ]
reifyRow (n , P) = reifyRow' n P
```

```agda
reifyRowOrdered : ∀ (ρ : Row (SemType Δ κ)) → OrderedRow ρ → NormalOrdered (reifyRow ρ)
reifyRowOrdered' : ∀ (n : ℕ) → (P : Fin n → Label × SemType Δ κ) →
                   OrderedRow (n , P) → NormalOrdered (reifyRow (n , P))

reifyRowOrdered' zero P oρ = tt
reifyRowOrdered' (suc zero) P oρ = tt
reifyRowOrdered' (suc (suc n)) P (l₁<l₂ , ih) = l₁<l₂ , (reifyRowOrdered' (suc n) (P ∘ fsuc) ih)

reifyRowOrdered (n , P) oρ = reifyRowOrdered' n P oρ

reifyPreservesNR : ∀ (ρ₁ ρ₂ : RowType Δ (λ Δ' → SemType Δ' κ) R[ κ ]) →
                   (nr : NotRow ρ₁ or NotRow ρ₂) → NotSimpleRow (reify ρ₁) or NotSimpleRow (reify ρ

reifyPreservesNR' : ∀ (ρ₁ ρ₂ : RowType Δ (λ Δ' → SemType Δ' κ) R[ κ ]) →
                   (nr : NotRow ρ₁ or NotRow ρ₂) → NotSimpleRow (reify ((ρ₁ \ ρ₂) {nr}))

reify {κ = ★} τ = τ
reify {κ = L} τ = τ
reify {κ = κ₁ `→ κ₂} F = reifyKripke F
reify {κ = R[ κ ]} (l ▹ τ) = (l ▹ₙ (reify τ))
reify {κ = R[ κ ]} (row ρ q) = ⦅ reifyRow ρ ⦆ (fromWitness (reifyRowOrdered ρ q))
reify {κ = R[ κ ]} ((φ <$> τ)) = (reifyKripkeNE φ <$> τ)
reify {κ = R[ κ ]} ((φ <$> τ) \ ρ₂) = (reify (φ <$> τ) \ reify ρ₂) {nsr = tt}
reify {κ = R[ κ ]} ((l ▹ τ) \ ρ) = (reify (l ▹ τ) \ (reify ρ)) {nsr = tt}
reify {κ = R[ κ ]} (row ρ x \ ρ'@(x₁ ▹ x₂)) = (reify (row ρ x) \ reify ρ') {nsr = tt}
reify {κ = R[ κ ]} ((row ρ x \ row ρ₁ x₁) {left ()})
reify {κ = R[ κ ]} ((row ρ x \ row ρ₁ x₁) {right ()})
reify {κ = R[ κ ]} (row ρ x \ (φ <$> τ)) = (reify (row ρ x) \ reify (φ <$> τ)) {nsr = tt}
reify {κ = R[ κ ]} ((row ρ x \ ρ'@((ρ₁ \ ρ₂) {nr'})) {nr}) = ((reify (row ρ x)) \ (reify ((ρ₁ \ ρ₂) {nr'}))) {nsr = from
reify {κ = R[ κ ]} ((((ρ₂ \ ρ₁) {nr'}) \ ρ) {nr}) = ((reify ((ρ₂ \ ρ₁) {nr'})) \ reify ρ) {fromWitness (reifyPreservesN

reifyPreservesNR (x₁ ▹ x₂) ρ₂ (left x) = left tt
reifyPreservesNR ((ρ₁ \ ρ₃) {nr}) ρ₂ (left x) = left (reifyPreservesNR' ρ₁ ρ₃ nr)
reifyPreservesNR (φ <$> ρ) ρ₂ (left x) = left tt
reifyPreservesNR ρ₁ (x ▹ x₁) (right y) = right tt
reifyPreservesNR ρ₁ ((ρ₂ \ ρ₃) {nr}) (right y) = right (reifyPreservesNR' ρ₂ ρ₃ nr)
reifyPreservesNR ρ₁ ((φ <$> ρ₂)) (right y) = right tt

reifyPreservesNR' (x₁ ▹ x₂) ρ₂ (left x) = tt
reifyPreservesNR' (ρ₁ \ ρ₃) ρ₂ (left x) = tt
reifyPreservesNR' (φ <$> n) ρ₂ (left x) = tt
reifyPreservesNR' (φ <$> n) ρ₂ (right y) = tt
reifyPreservesNR' (x ▹ x₁) ρ₂ (right y) = tt
reifyPreservesNR' (row ρ x) (x₁ ▹ x₂) (right y) = tt
reifyPreservesNR' (row ρ x) (ρ₂ \ ρ₃) (right y) = tt
reifyPreservesNR' (row ρ x) (φ <$> n) (right y) = tt
reifyPreservesNR' (ρ₁ \ ρ₃) ρ₂ (right y) = tt
```

```
--------------------------------
-- η normalization of neutral types

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

-- --------------------------------
-- - Semantic environments

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

idEnv : Env Δ Δ
idEnv = reflect ∘ `

extende : (η : Env Δ₁ Δ₂) → (V : SemType Δ₂ κ) → Env (Δ₁ ,, κ) Δ₂
extende η V Z = V
extende η V (S x) = η x

lifte : Env Δ₁ Δ₂ → Env (Δ₁ ,, κ) (Δ₂ ,, κ)
lifte {Δ₁} {Δ₂} {κ} η = extende (weakenSem ∘ η) (idEnv Z)
```

5.1 Helping evaluation

```
----------
-- Semantic application

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

----------
-- Semantic complement

_∈Row_ : ∀ {m} → (l : Label) →
             (Q : Fin m → Label × SemType Δ κ) →
             Set
_∈Row_ {m = m} l Q = Σ[ i ∈ Fin m ] (l ≡ Q i .fst)

_∈Row?_ : ∀ {m} → (l : Label) →
             (Q : Fin m → Label × SemType Δ κ) →
             Dec (l ∈Row Q)
_∈Row?_ {m = zero} l Q = no λ { () }
_∈Row?_ {m = suc m} l Q with l ≟ Q fzero .fst
... | yes p = yes (fzero , p)
... | no    p with l ∈Row? (Q ∘ fsuc)
... | yes (n , q) = yes ((fsuc n) , q)
... | no          q = no λ { (fzero , q') → p q' ; (fsuc n , q') → q (n , q') }

compl : ∀ {n m} →
        (P : Fin n → Label × SemType Δ κ)
        (Q : Fin m → Label × SemType Δ κ) →
```

```
          Row (SemType Δ κ)
compl {n = zero} {m} P Q = εV
compl {n = suc n} {m} P Q with P fzero .fst ∈Row? Q
... | yes _ = compl (P ∘ fsuc) Q
... | no _ = (P fzero) :: (compl (P ∘ fsuc) Q)

- ——————————————————————————————
- - Semantic complement preserves well-ordering
lemma : ∀ {n m q} →
              (P : Fin (suc n) → Label × SemType Δ κ)
              (Q : Fin m → Label × SemType Δ κ) →
              (R : Fin (suc q) → Label × SemType Δ κ) →
                 OrderedRow (suc n , P) →
                 compl (P ∘ fsuc) Q ≡ (suc q , R) →
              P fzero .fst < R fzero .fst
lemma {n = suc n} {q = q} P Q R oP eq₁ with P (fsuc fzero) .fst ∈Row? Q
lemma {κ = _} {suc n} {q = q} P Q R oP refl | no _ = oP .fst
... | yes _ = <-trans {i = P fzero .fst} {j = P (fsuc fzero) .fst} {k = R fzero .fst} (oP .fst) (lemma {n = n} (P ∘ fsuc) Q

ordered-:: : ∀ {n m} →
                    (P : Fin (suc n) → Label × SemType Δ κ)
                    (Q : Fin m → Label × SemType Δ κ) →
                    OrderedRow (suc n , P) →
                    OrderedRow (compl (P ∘ fsuc) Q) → OrderedRow (P fzero :: compl (P ∘ fsuc) Q)
ordered-:: {n = n} P Q oP oC with compl (P ∘ fsuc) Q | inspect (compl (P ∘ fsuc)) Q
... | zero , R | _ = tt
... | suc n , R | [[ eq ]] = lemma P Q R oP eq , oC

ordered-compl : ∀ {n m} →
                    (P : Fin n → Label × SemType Δ κ)
                    (Q : Fin m → Label × SemType Δ κ) →
                    OrderedRow (n , P) → OrderedRow (m , Q) → OrderedRow (compl P Q)
ordered-compl {n = zero} P Q oρ₁ oρ₂ = tt
ordered-compl {n = suc n} P Q oρ₁ oρ₂ with P fzero .fst ∈Row? Q
... | yes _ = ordered-compl (P ∘ fsuc) Q (ordered-cut oρ₁) oρ₂
... | no _ = ordered-:: P Q oρ₁ (ordered-compl (P ∘ fsuc) Q (ordered-cut oρ₁) oρ₂)

——————————————————————————————
- Semantic complement on Rows

_\v_ : Row (SemType Δ κ) → Row (SemType Δ κ) → Row (SemType Δ κ)
(n , P) \v (m , Q) = compl P Q

ordered\v : ∀ (ρ₂ ρ₁ : Row (SemType Δ κ)) → OrderedRow ρ₂ → OrderedRow ρ₁ → OrderedRow (ρ₂ \v ρ₁)
ordered\v (n , P) (m , Q) oρ₂ oρ₁ = ordered-compl P Q oρ₂ oρ₁

- - - ——————————————————————————————
```

```
---- Semantic lifting

_<$>V_ : SemType Δ (κ₁ `→ κ₂) → SemType Δ R[ κ₁ ] → SemType Δ R[ κ₂ ]
NotRow<$> : ∀ {F : SemType Δ (κ₁ `→ κ₂)} {ρ₂ ρ₁ : RowType Δ (λ Δ' → SemType Δ' κ₁) R[ κ₁ ]} →
            NotRow ρ₂ or NotRow ρ₁ → NotRow (F <$>V ρ₂) or NotRow (F <$>V ρ₁)

F <$>V (l ▹ τ) = l ▹ (F ·V τ)
F <$>V row (n , P) q = row (n , overᵣ (F id) ∘ P) (orderedOverᵣ (F id) q)
F <$>V ((ρ₂ \ ρ₁) {nr}) = ((F <$>V ρ₂) \ (F <$>V ρ₁)) {NotRow<$> nr}
F <$>V (G <$> n) = (λ {Δ'} r → F r ∘ G r) <$> n

NotRow<$> {F = F} {x₁ ▹ x₂} {ρ₁} (left x) = left tt
NotRow<$> {F = F} {ρ₂ \ ρ₃} {ρ₁} (left x) = left tt
NotRow<$> {F = F} {φ <$> n} {ρ₁} (left x) = left tt

NotRow<$> {F = F} {ρ₂} {x ▹ x₁} (right y) = right tt
NotRow<$> {F = F} {ρ₂} {ρ₁ \ ρ₃} (right y) = right tt
NotRow<$> {F = F} {ρ₂} {φ <$> n} (right y) = right tt

--- ———————————————————————————
---- Semantic complement on SemTypes

_\V_ : SemType Δ R[ κ ] → SemType Δ R[ κ ] → SemType Δ R[ κ ]
row ρ₂ oρ₂ \V row ρ₁ oρ₁ = row (ρ₂ \v ρ₁) (ordered\v ρ₂ ρ₁ oρ₂ oρ₁)
ρ₂@(x ▹ x₁) \V ρ₁ = (ρ₂ \ ρ₁) {nr = left tt}
ρ₂@(row ρ x) \V ρ₁@(x₁ ▹ x₂) = (ρ₂ \ ρ₁) {nr = right tt}
ρ₂@(row ρ x) \V ρ₁@(_ \ _) = (ρ₂ \ ρ₁) {nr = right tt}
ρ₂@(row ρ x) \V ρ₁@(_ <$> _) = (ρ₂ \ ρ₁) {nr = right tt}
ρ@(ρ₂ \ ρ₃) \V ρ' = (ρ \ ρ') {nr = left tt}
ρ@(φ <$> n) \V ρ' = (ρ \ ρ') {nr = left tt}

- ———————————————————————————
-- Semantic flap

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
```

5.2 Π and Σ as operators

```
record Xi : Set where
  field
    Ξ★ : ∀ {Δ} → NormalType Δ R[ ★ ] → NormalType Δ ★
    ren-★ : ∀ (ρ : Renamingₖ Δ₁ Δ₂) → (τ : NormalType Δ₁ R[ ★ ]) → renₖNF ρ (Ξ★ τ) ≡ Ξ★ (renₖNF ρ τ)

open Xi
ξ : ∀ {Δ} → Xi → SemType Δ R[ κ ] → SemType Δ κ
ξ {κ = ★} Ξ x = Ξ .Ξ★ (reify x)
```

```
ξ {κ = L} Ξ x = lab "impossible"
ξ {κ = κ₁ `→ κ₂} Ξ F = λ ρ v → ξ Ξ (renSem ρ F <?>V v)
ξ {κ = R[ κ ]} Ξ x = (λ ρ v → ξ Ξ v) <$>V x

Π-rec Σ-rec : Xi
Π-rec = record
  { Ξ★ = Π ; ren-★ = λ ρ τ → refl }
Σ-rec =
  record
  { Ξ★ = Σ ; ren-★ = λ ρ τ → refl }

ΠV ΣV : ∀ {Δ} → SemType Δ R[ κ ] → SemType Δ κ
ΠV = ξ Π-rec
ΣV = ξ Σ-rec

ξ-Kripke : Xi → KripkeFunction Δ R[ κ ] κ
ξ-Kripke Ξ ρ v = ξ Ξ v

Π-Kripke Σ-Kripke : KripkeFunction Δ R[ κ ] κ
Π-Kripke = ξ-Kripke Π-rec
Σ-Kripke = ξ-Kripke Σ-rec
```

5.3 Evaluation

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

evalRow : (ρ : SimpleRow Type Δ₁ R[ κ ]) → Env Δ₁ Δ₂ → Row (SemType Δ₂ κ)
evalRowOrdered : (ρ : SimpleRow Type Δ₁ R[ κ ]) → (η : Env Δ₁ Δ₂) → Ordered ρ → OrderedRow (evalRow

evalRow [] η = εV
evalRow ((l , τ) ∷ ρ) η = (l , (eval τ η)) ∷ evalRow ρ η

⇓Row-isMap : ∀ (η : Env Δ₁ Δ₂) → (xs : SimpleRow Type Δ₁ R[ κ ]) →
                reifyRow (evalRow xs η) ≡ map (λ { (l , τ) → l , (reify (eval τ η)) }) xs
⇓Row-isMap η [] = refl
⇓Row-isMap η (x ∷ xs) = cong₂ _∷_ refl (⇓Row-isMap η xs)

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)

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)

eval {κ = ★} (π ⇒ τ) η = evalPred π η ⇒ eval τ η
eval {Δ₁} {κ = ★} (`∀ τ) η = `∀ (eval τ (lifte η))
eval {κ = ★} (μ τ) η = μ (reify (eval τ η))
eval {κ = ★} ⌊ τ ⌋ η = ⌊ reify (eval τ η) ⌋
eval (ρ₂ ∖ ρ₁) η = eval ρ₂ η ∖V eval ρ₁ η
eval {κ = L} (lab l) η = lab l
```

```
eval {κ = κ₁ `→ κ₂} (`λ τ) η = λ ρ v → eval τ (extende (λ {κ} v' → renSem {κ = κ} ρ (η v')) v)
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 (⦅ ρ ⦆ oρ) η = row (evalRow ρ η) (evalRowOrdered ρ η (toWitness oρ))
eval (l ▹ τ) η with eval l η
... | ne x = (x ▹ eval τ η)
... | lab l₁ = row (1 , λ { fzero → (l₁ , eval τ η) }) tt
evalRowOrdered [] η oρ = tt
evalRowOrdered (x₁ ∷ []) η oρ = tt
evalRowOrdered ((l₁ , τ₁) ∷ (l₂ , τ₂) ∷ ρ) η (l₁<l₂ , oρ) with
  evalRow ρ η | evalRowOrdered ((l₂ , τ₂) ∷ ρ) η oρ
... | zero , P | ih = l₁<l₂ , tt
... | suc n , P | ih₁ , ih₂ = l₁<l₂ , ih₁ , ih₂
```

5.4 Normalization

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

⇓Pred : ∀ {Δ} → Pred Type Δ R[ κ ] → Pred NormalType Δ R[ κ ]
⇓Pred π = evalPred π idEnv

⇓Row : ∀ {Δ} → SimpleRow Type Δ R[ κ ] → SimpleRow NormalType Δ R[ κ ]
⇓Row ρ = reifyRow (evalRow ρ idEnv)

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

6 METATHEORY

6.1 Stability

```
stability : ∀ (τ : NormalType Δ κ) → ⇓ (⇑ τ) ≡ τ
stabilityNE : ∀ (τ : NeutralType Δ κ) → eval (⇑NE τ) (idEnv {Δ}) ≡ reflect τ
stabilityPred : ∀ (π : NormalPred Δ R[ κ ]) → evalPred (⇑Pred π) idEnv ≡ π
stabilityRow : ∀ (ρ : SimpleRow NormalType Δ R[ κ ]) → reifyRow (evalRow (⇑Row ρ) idEnv) ≡ ρ
```

Stability implies surjectivity and idempotency.

```
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 )
```

Dual to surjectivity, stability also implies that embedding is injective.

```
⇑-inj : ∀ (τ₁ τ₂ : NormalType Δ κ) → ⇑ τ₁ ≡ ⇑ τ₂ → τ₁ ≡ τ₂
⇑-inj τ₁ τ₂ eq = trans (sym (stability τ₁)) (trans (cong ⇓ eq) (stability τ₂))
```

6.2 A logical relation for completeness

```
subst-Row : ∀ {A : Set} {n m : ℕ} → (n ≡ m) → (f : Fin n → A) → Fin m → A
subst-Row refl f = f

-- Completeness relation on semantic types
_≈_ : SemType Δ κ → SemType Δ κ → Set
_≈₂_ : ∀ {A} → (x y : A × SemType Δ κ) → Set
(l₁ , τ₁) ≈₂ (l₂ , τ₂) = l₁ ≡ l₂ × τ₁ ≈ τ₂
_≈R_ : (ρ₁ ρ₂ : Row (SemType Δ κ)) → Set
(n , P) ≈R (m , Q) = Σ[ pf ∈ (n ≡ m) ] (∀ (i : Fin m) → (subst-Row pf P) i ≈₂ Q i)

PointEqual-≈ : ∀ {Δ₁} {κ₁} {κ₂} (F G : KripkeFunction Δ₁ κ₁ κ₂) → Set
PointEqualNE-≈ : ∀ {Δ₁} {κ₁} {κ₂} (F G : KripkeFunctionNE Δ₁ κ₁ κ₂) → Set
Uniform : ∀ {Δ} {κ₁} {κ₂} → KripkeFunction Δ κ₁ κ₂ → Set
UniformNE : ∀ {Δ} {κ₁} {κ₂} → KripkeFunctionNE Δ κ₁ κ₂ → Set

convNE : κ₁ ≡ κ₂ → NeutralType Δ R[ κ₁ ] → NeutralType Δ R[ κ₂ ]
convNE refl n = n

convKripkeNE₁ : ∀ {κ₁'} → κ₁ ≡ κ₁' → KripkeFunctionNE Δ κ₁ κ₂ → KripkeFunctionNE Δ κ₁' κ₂
convKripkeNE₁ refl f = f

_≈_ {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
_≈_ {Δ₁} {R[ κ₂ ]} (_<$>_ {κ₁} φ₁ n₁) (_<$>_ {κ₁'} φ₂ n₂) =
  Σ[ pf ∈ (κ₁ ≡ κ₁') ]
    UniformNE φ₁
  × UniformNE φ₂
  × (PointEqualNE-≈ (convKripkeNE₁ pf φ₁) φ₂
  × convNE pf n₁ ≡ n₂)
_≈_ {Δ₁} {R[ κ₂ ]} (φ₁ <$> n₁) _ = ⊥
_≈_ {Δ₁} {R[ κ₂ ]} _ (φ₁ <$> n₁) = ⊥
_≈_ {Δ₁} {R[ κ ]} (l₁ ▹ τ₁) (l₂ ▹ τ₂) = l₁ ≡ l₂ × τ₁ ≈ τ₂
_≈_ {Δ₁} {R[ κ ]} (x₁ ▹ x₂) (row ρ x₃) = ⊥
_≈_ {Δ₁} {R[ κ ]} (x₁ ▹ x₂) (ρ₂ \ ρ₃) = ⊥
_≈_ {Δ₁} {R[ κ ]} (row ρ x₁) (x₂ ▹ x₃) = ⊥
_≈_ {Δ₁} {R[ κ ]} (row (n , P) x₁) (row (m , Q) x₂) = (n , P) ≈R (m , Q)
_≈_ {Δ₁} {R[ κ ]} (row ρ x₁) (ρ₂ \ ρ₃) = ⊥
_≈_ {Δ₁} {R[ κ ]} (ρ₁ \ ρ₂) (x₁ ▹ x₂) = ⊥
_≈_ {Δ₁} {R[ κ ]} (ρ₁ \ ρ₂) (row ρ x₁) = ⊥
_≈_ {Δ₁} {R[ κ ]} (ρ₁ \ ρ₂) (ρ₃ \ ρ₄) = ρ₁ ≈ ρ₃ × ρ₂ ≈ ρ₄

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

```
PointEqualNE-≈ {Δ1} {κ1} {κ2} F G =
  ∀ {Δ2} (ρ : Renamingk Δ1 Δ2) (V : NeutralType Δ2 κ1) →
  F ρ V ≈ G ρ V

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))

UniformNE {Δ1} {κ1} {κ2} F =
  ∀ {Δ2 Δ3} (ρ1 : Renamingk Δ1 Δ2) (ρ2 : Renamingk Δ2 Δ3) (V : NeutralType Δ2 κ1) →
  (renSem ρ2 (F ρ1 V)) ≈ F (ρ2 ∘ ρ1) (renkNE ρ2 V)

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

-- extension
extend-≈ : ∀ {η1 η2 : Env Δ1 Δ2} → Env-≈ η1 η2 →
            {V1 V2 : SemType Δ2 κ} →
            V1 ≈ V2 →
            Env-≈ (extende η1 V1) (extende η2 V2)
extend-≈ p q Z = q
extend-≈ p q (S v) = p v
```

6.2.1 Properties.

```
reflect-≈ : ∀ {τ1 τ2 : NeutralType Δ κ} → τ1 ≡ τ2 → reflect τ1 ≈ reflect τ2
reify-≈   : ∀ {V1 V2 : SemType Δ κ} → V1 ≈ V2 → reify V1 ≡ reify V2
reifyRow-≈ : ∀ {n} (P Q : Fin n → Label × SemType Δ κ) →
               (∀ (i : Fin n) → P i ≈2 Q i) →
               reifyRow (n , P) ≡ reifyRow (n , Q)
```

6.3 The fundamental theorem and completeness

```
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 Type Δ1 R[ κ ]} {η1 η2 : Env Δ1 Δ2} →
               Env-≈ η1 η2 → π1 ≡p π2 → evalPred π1 η1 ≡ evalPred π2 η2
fundC-Row : ∀ {ρ1 ρ2 : SimpleRow Type Δ1 R[ κ ]} {η1 η2 : Env Δ1 Δ2} →
               Env-≈ η1 η2 → ρ1 ≡r ρ2 → evalRow ρ1 η1 ≈R evalRow ρ2 η2

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

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

completeness-row : ∀ {ρ1 ρ2 : SimpleRow Type Δ R[ κ ]} → ρ1 ≡r ρ2 → ⇓Row ρ1 ≡ ⇓Row ρ2
```

6.4 A logical relation for soundness

```
infix 0 ⟦_⟧≈_
⟦_⟧≈_ : ∀ {κ} → Type Δ κ → SemType Δ κ → Set
⟦_⟧≈ne_ : ∀ {κ} → Type Δ κ → NeutralType Δ κ → Set

⟦_⟧r≈_ : ∀ {κ} → SimpleRow Type Δ R[ κ ] → Row (SemType Δ κ) → Set
⟦_⟧≈₂_ : ∀ {κ} → Label × Type Δ κ → Label × SemType Δ κ → Set
⟦ (l₁ , τ) ⟧≈₂ (l₂ , V) = (l₁ ≡ l₂) × (⟦ τ ⟧≈ V)

SoundKripke : Type Δ₁ (κ₁ ‘→ κ₂) → KripkeFunction Δ₁ κ₁ κ₂ → Set
SoundKripkeNE : Type Δ₁ (κ₁ ‘→ κ₂) → KripkeFunctionNE Δ₁ κ₁ κ₂ → Set

-- τ is equivalent to neutral ‘n‘ if it's equivalent
-- to the η and map-id expansion of n
⟦_⟧≈ne_ τ n = τ ≡t ⇑ (η-norm n)

⟦_⟧≈_ {κ = ★} τ₁ τ₂ = τ₁ ≡t ⇑ τ₂
⟦_⟧≈_ {κ = L} τ₁ τ₂ = τ₁ ≡t ⇑ τ₂
⟦_⟧≈_ {Δ₁} {κ = κ₁ ‘→ κ₂} f F = SoundKripke f F
⟦_⟧≈_ {Δ} {κ = R[ κ ]} τ (row (n , P) oρ) =
  let xs = ⇑Row (reifyRow (n , P)) in
  (τ ≡t ⦅ xs ⦆ (fromWitness (Ordered⇑ (reifyRow (n , P)) (reifyRowOrdered’ n P oρ)))) ×
  (⟦ xs ⟧r≈ (n , P))
⟦_⟧≈_ {Δ} {κ = R[ κ ]} τ (l ▹ V) = (τ ≡t (⇑NE l ▹ ⇑ (reify V))) × (⟦ ⇑ (reify V) ⟧≈ V)
⟦_⟧≈_ {Δ} {κ = R[ κ ]} τ ((ρ₂ ∖ ρ₁) {nr}) = (τ ≡t (⇑ (reify ((ρ₂ ∖ ρ₁) {nr})))) × (⟦ ⇑ (reify ρ₂) ⟧≈ ρ₂) × (⟦ ⇑ (reify ρ
⟦_⟧≈_ {Δ} {κ = R[ κ ]} τ (φ <$> n) =
  ∃[ f ] ((τ ≡t (f <$> ⇑NE n)) × (SoundKripkeNE f φ))
⟦ [] ⟧r≈ (zero , P) = ⊤
⟦ [] ⟧r≈ (suc n , P) = ⊥
⟦ x ∷ ρ ⟧r≈ (zero , P) = ⊥
⟦ x ∷ ρ ⟧r≈ (suc n , P) = (⟦ x ⟧≈₂ (P fzero)) × ⟦ ρ ⟧r≈ (n , P ∘ fsuc)

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)

SoundKripkeNE {Δ₁ = Δ₁} {κ₁ = κ₁} {κ₂ = κ₂} f F =
  ∀ {Δ₂} (r : Renamingₖ Δ₁ Δ₂) {v V} →
    ⟦ v ⟧≈ne V →
    ⟦ (renₖ r f · v) ⟧≈ (F r V)
```

6.4.1 *Properties.*

```
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)
η-norm-≡t : ∀ (τ : NeutralType Δ κ) → ⇑ (η-norm τ) ≡t ⇑NE τ
```

```
subst-⟦⟧≈ : ∀ {τ₁ τ₂ : Type Δ κ} →
  τ₁ ≡t τ₂ → {V : SemType Δ κ} → ⟦ τ₁ ⟧≈ V → ⟦ τ₂ ⟧≈ V
```

6.4.2 *Logical environments.*

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

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

6.5 The fundamental theorem and soundness

```
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)
fundSRow : ∀ {Δ₁ Δ₂ κ}(xs : SimpleRow Type Δ₁ R[ κ ]){σ : Substitutionₖ Δ₁ Δ₂}{η : Env Δ₁ Δ₂} →
          ⟦ σ ⟧≈e η → ⟦ subRowₖ σ xs ⟧r≈ (evalRow xs η)
fundSPred : ∀ {Δ₁ κ}(π : Pred Type Δ₁ R[ κ ]){σ : Substitutionₖ Δ₁ Δ₂}{η : Env Δ₁ Δ₂} →
          ⟦ σ ⟧≈e η → (subPredₖ σ π) ≡p ⇑Pred (evalPred π η)

--------------------------------------------------------------------------------
-- Fundamental theorem when substitution is the identity
subₖ-id : ∀ (τ : Type Δ κ) → subₖ ‘ τ ≡ τ

⊢⟦_⟧≈ : ∀ (τ : Type Δ κ) → ⟦ τ ⟧≈ eval τ idEnv
⊢⟦ τ ⟧≈ = subst-⟦⟧≈ (inst (subₖ-id τ)) (fundS τ idSR)

--------------------------------------------------------------------------------
-- Soundness claim

soundness : ∀ {Δ₁ κ} → (τ : Type Δ₁ κ) → τ ≡t ⇑ (⇓ τ)
soundness τ = reify-⟦⟧≈ (⊢⟦ τ ⟧≈)

 --------------------------------------------------------------------------------
-- If τ₁ normalizes to ⇓ τ₂ then the embedding of τ₁ is equivalent to τ₂

embed-≡t : ∀ {τ₁ : NormalType Δ κ} {τ₂ : Type Δ κ} → τ₁ ≡ (⇓ τ₂) → ⇑ τ₁ ≡t τ₂
embed-≡t {τ₁ = τ₁} {τ₂} refl = eq-sym (soundness τ₂)

--------------------------------------------------------------------------------
-- Soundness implies the converse of completeness, as desired

Completeness⁻¹ : ∀ {Δ κ} → (τ₁ τ₂ : Type Δ κ) → ⇓ τ₁ ≡ ⇓ τ₂ → τ₁ ≡t τ₂
Completeness⁻¹ τ₁ τ₂ eq = eq-trans (soundness τ₁) (embed-≡t eq)
```

7 THE REST OF THE PICTURE

In the remainder of the development, we intrinsically represent terms as typing judgments indexed by normal types. We then give a typed reduction relation on terms and show progress.

8 MOST CLOSELY RELATED WORK

8.0.1 *Chapman et al. [2019].*

8.0.2 *Allais et al. [2013].*

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/.