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 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.

1 The R$\omega\mu$ calculus

For reference, Figure 1 describes the syntax of kinds, predicates, and types in R$\omega\mu$.

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

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

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 Contact Information: Alex Hubers, Department of Computer Science, The University of Iowa, 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 *Quotienting kinds.*

We will find it necessary to quotient kinds by two partitions for reasons which we discuss later. 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 labelled 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 now lay the groundwork to describe the type system of R$\omega\mu$. We represent the judgment $\Gamma \vdash \tau : \kappa$ intrinsically as the data type `Type`; The data type `Pred` represents well-kinded predicates. 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` datatype is indexed abstractly by type `Ty`.

```
infixr 2 _⇒_
infixl 5 _·_
infixr 5 _≲_
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. Again, a row literal of `Types` and of types in normal form will not differ in shape, and so `SimpleRow` abstracts over its content type `Ty`.

```
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 ascendingly according to some total order $<$. We will restrict the formation of rows to just those that are ordered, which has three key consequences: first, it guarantees a normal form (later) for simple rows; second, it only permits variable labels in singleton rows; and third, it enforces that labels be unique in each row. It is easy to show that the `Ordered` predicate is decidable (definition omitted).

```
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 first 6 cases are standard for System F$\omega\mu$.

```
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 ★

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

  μ :
    (ϕ : Type Δ (★ ‘→ ★)) →
    Type Δ ★
```

The constructor _⇒_ forms a qualified type given a well-kinded predicate.

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

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

```
lab :
  (l : Label) →
  Type Δ L

- label constant formation
⌊_⌋ :
  (τ : Type Δ L) →
  Type Δ ★
```

We finally describe row formation.

```
⦅_⦆ :
  (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 ]

- Record formation
Π :
  {notLabel : True (notLabel? κ)} →
  Type Δ (R[ κ ] ‘→ κ)

- Variant formation
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 ordering 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
```

```
map-overᵣ : ∀ (ρ : SimpleRow Type Δ₁ R[ κ₁ ]) (f : Type Δ₁ κ₁ → Type Δ₁ κ₂) →
              Ordered ρ → Ordered (map (overᵣ f) ρ)
map-overᵣ [] f oρ = tt
map-overᵣ (x :: []) f oρ = tt
map-overᵣ ((l₁ , _) :: (l₂ , _) :: ρ) f (l₁<l₂ , oρ) = l₁<l₂ , (map-overᵣ ((l₂ , _) :: ρ) f oρ)
```

2.2.2 *Flipped map operator.*

```
- Flapping.
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.*

```
infix 0 _∈L_

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)
l ∈L? [] = no (λ { () })
l ∈L? ((l’ , _) :: xs) with l ≟ l’
... | yes refl = yes Here
... | no     p with l ∈L? xs
... | yes p = yes (There p)
... | no q = no λ { Here → p refl ; (There x) → q x }

_\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.*

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

- lifting over binders.
liftₖ : Renamingₖ Δ₁ Δ₂ → Renamingₖ (Δ₁ ,, κ) (Δ₂ ,, κ)
liftₖ ρ Z = Z
liftₖ ρ (S x) = S (ρ x)

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

```
orderedRenRow_k : (r : Renaming_k Δ_1 Δ_2) → (xs : SimpleRow Type Δ_1 R[ κ ]) → Ordered xs →
                  Ordered (renRow_k r xs)

ren_k r (‘ x) = ‘ (r x)
ren_k r (‘λ τ) = ‘λ (ren_k (lift_k r) τ)
ren_k r (τ_1 · τ_2) = (ren_k r τ_1) · (ren_k r τ_2)
ren_k r (τ_1 ‘→ τ_2) = (ren_k r τ_1) ‘→ (ren_k r τ_2)
ren_k r (π ⇒ τ) = renPred_k r π ⇒ ren_k r τ
ren_k r (‘∀ τ) = ‘∀ (ren_k (lift_k r) τ)
ren_k r (μ F) = μ (ren_k r F)
ren_k r (Π {notLabel = nl}) = Π {notLabel = nl}
ren_k r (Σ {notLabel = nl}) = Σ {notLabel = nl}
ren_k r (lab x) = lab x
ren_k r ⌊ ℓ ⌋ = ⌊ (ren_k r ℓ) ⌋
ren_k r (f <$> m) = ren_k r f <$> ren_k r m
ren_k r (⦅ xs ⦆ oxs) = ⦅ renRow_k r xs ⦆ (fromWitness (orderedRenRow_k r xs (toWitness oxs)))
ren_k r (ρ_2 ∖ ρ_1) = ren_k r ρ_2 ∖ ren_k r ρ_1
ren_k r (l ▹ τ) = ren_k r l ▹ ren_k r τ

renPred_k ρ (ρ_1 · ρ_2 ~ ρ_3) = ren_k ρ ρ_1 · ren_k ρ ρ_2 ~ ren_k ρ ρ_3
renPred_k ρ (ρ_1 ≲ ρ_2) = (ren_k ρ ρ_1) ≲ (ren_k ρ ρ_2)

renRow_k r [] = []
renRow_k r ((l , τ) ∷ xs) = (l , ren_k r τ) ∷ renRow_k r xs

orderedRenRow_k r [] oxs = tt
orderedRenRow_k r ((l , τ) ∷ []) oxs = tt
orderedRenRow_k r ((l_1 , τ) ∷ (l_2 , υ) ∷ xs) (l_1<l_2 , oxs) = l_1<l_2 , orderedRenRow_k r ((l_2 , υ) ∷ xs) oxs

weaken_k : Type Δ κ_2 → Type (Δ ,, κ_1) κ_2
weaken_k = ren_k S

weakenPred_k : Pred Type Δ R[ κ_2 ] → Pred Type (Δ ,, κ_1) R[ κ_2 ]
weakenPred_k = renPred_k S
```

2.2.5 Type substitution.

```
Substitution_k : KEnv → KEnv → Set
Substitution_k Δ_1 Δ_2 = ∀ {κ} → TVar Δ_1 κ → Type Δ_2 κ

-- lifting a substitution over binders.
lifts_k : Substitution_k Δ_1 Δ_2 → Substitution_k(Δ_1 ,, κ) (Δ_2 ,, κ)
lifts_k σ Z = ‘ Z
lifts_k σ (S x) = weaken_k (σ x)

-- This is simultaneous substitution: Given subst σ and type τ, we replace *all*
-- variables in τ with the types mapped to by σ.
sub_k : Substitution_k Δ_1 Δ_2 → Type Δ_1 κ → Type Δ_2 κ
subPred_k : Substitution_k Δ_1 Δ_2 → Pred Type Δ_1 κ → Pred Type Δ_2 κ
subRow_k : Substitution_k Δ_1 Δ_2 → SimpleRow Type Δ_1 R[ κ ] → SimpleRow Type Δ_2 R[ κ ]
orderedSubRow_k : (σ : Substitution_k Δ_1 Δ_2) → (xs : SimpleRow Type Δ_1 R[ κ ]) → Ordered xs →
```

```
                         Ordered (subRowₖ σ xs)
-- 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 {notLabel = nl}) = Π {notLabel = nl}
subₖ σ (Σ {notLabel = nl}) = Σ {notLabel = nl}
subₖ σ (lab x) = lab x
subₖ σ ⌊ ℓ ⌋ = ⌊ (subₖ σ ℓ) ⌋
subₖ σ (f <$> a) = subₖ σ f <$> subₖ σ a
subₖ σ (ρ₂ ∖ ρ₁) = subₖ σ ρ₂ ∖ subₖ σ ρ₁
subₖ σ (⦅ xs ⦆ oxs) = ⦅ subRowₖ σ xs ⦆ (fromWitness (orderedSubRowₖ σ xs (toWitness oxs)))
subₖ σ (l ▹ τ) = (subₖ σ l) ▹ (subₖ σ τ)
subRowₖ σ [] = []
subRowₖ σ ((l , τ) ∷ xs) = (l , subₖ σ τ) ∷ subRowₖ σ xs

orderedSubRowₖ r [] oxs = tt
orderedSubRowₖ r ((l , τ) ∷ []) oxs = tt
orderedSubRowₖ r ((l₁ , τ) ∷ (l₂ , υ) ∷ xs) (l₁<l₂ , oxs) = l₁<l₂ , orderedSubRowₖ r ((l₂ , υ) ∷ xs) oxs

subRowₖ-isMap : ∀ (σ : Substitutionₖ Δ₁ Δ₂) (xs : SimpleRow Type Δ₁ R[ κ ]) →
                          subRowₖ σ xs ≡ map (overᵣ (subₖ σ)) xs

subRowₖ-isMap σ [] = refl
subRowₖ-isMap σ (x ∷ xs) = cong₂ _∷_ refl (subRowₖ-isMap σ xs)

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

-- Extension of a substitution by A
extendₖ : Substitutionₖ Δ₁ Δ₂ → (A : Type Δ₂ κ) → Substitutionₖ(Δ₁ ,, κ) Δ₂
extendₖ σ A Z = A
extendₖ σ A (S x) = σ x

-- Single variable subₖstitution is a special case of simultaneous subₖstitution.
_βₖ[_] : Type (Δ ,, κ₁) κ₂ → Type Δ κ₁ → Type Δ κ₂
B βₖ[ A ] = subₖ (extendₖ ` A) B
```

2.3 Type equivalence

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

```
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
data _≡r_ : SimpleRow Type Δ R[ κ ] → SimpleRow Type Δ R[ κ ] → Set 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
data _≡t_ where
  - ----------------
  - Eq. relation

    eq-refl :
      --
      τ ≡t τ
    eq-sym :

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

```
    τ₁ ≡t τ₂ → τ₂ ≡t τ₃ →
    ─────────
    τ₁ ≡t τ₃

-- ──────────────--
-- Congruence rules
  eq-→ :
    τ₁ ≡t τ₂ → υ₁ ≡t υ₂ →
    ──────────--
    τ₁ `→ υ₁ ≡t τ₂ `→ υ₂
  eq-∀ :
    τ ≡t υ →
    ──────--
    `∀ τ ≡t `∀ υ
  eq-μ :
    τ ≡t υ →
    ──────--
    μ τ ≡t μ υ
  eq-λ : ∀ {τ υ : Type (Δ ,, κ₁) κ₂} →
    τ ≡t υ →
    ──────--
    `λ τ ≡t `λ υ
  eq-· :
    τ₁ ≡t υ₁ → τ₂ ≡t υ₂ →
    ─────────
    τ₁ · τ₂ ≡t υ₁ · υ₂
  eq-<$> : ∀ {τ₁ υ₁ : Type Δ (κ₁ `→ κ₂)} {τ₂ υ₂ : Type Δ R[ κ₁ ]} →
    τ₁ ≡t υ₁ → τ₂ ≡t υ₂ →
    ─────────
    τ₁ <$> τ₂ ≡t υ₁ <$> υ₂
  eq-⌊⌋ :
    τ ≡t υ →
    ─────--
    ⌊ τ ⌋ ≡t ⌊ υ ⌋
  eq-⇒ :
         π₁ ≡p π₂ → τ₁ ≡t τ₂ →
    ──────────
    (π₁ ⇒ τ₁) ≡t (π₂ ⇒ τ₂)
```

```
  eq-lab :

        ℓ₁ ≡ ℓ₂ →
        -------
        lab {Δ = Δ} ℓ₁ ≡t lab ℓ₂

  eq-row :
    ∀ {ρ₁ ρ₂ : SimpleRow Type Δ R[ κ ]} {oρ₁ : True (ordered? ρ₁)}
       {oρ₂ : True (ordered? ρ₂)} →

    ρ₁ ≡r ρ₂ →
    -------------------------
    ⦅ ρ₁ ⦆ oρ₁ ≡t ⦅ ρ₂ ⦆ oρ₂

  eq-▹ : ∀ {l₁ l₂ : Type Δ L} {τ₁ τ₂ : Type Δ κ} →

        l₁ ≡t l₂   → τ₁ ≡t τ₂ →
        -------------------------
        (l₁ ▹ τ₁) ≡t (l₂ ▹ τ₂)

  eq-\ : ∀ {ρ₂ ρ₁ υ₂ υ₁ : Type Δ R[ κ ]} →

        ρ₂ ≡t υ₂ → ρ₁ ≡t υ₁ →
        -------------------------
        (ρ₂ \ ρ₁) ≡t (υ₂ \ υ₁)

-- -----------------------
-- η-laws

  eq-η : ∀ {f : Type Δ (κ₁ `→ κ₂)} →

    ------------------
    f ≡t `λ (weakenₖ f · (` Z))

-- -----------------------
-- Computational laws

  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

  eq-▹$ : ∀ {l} {τ : Type Δ κ₁} {F : Type Δ (κ₁ `→ κ₂)} →
```

```
    ---------------
    (F <$> (l ▹ τ)) ≡t (l ▹ (F · τ))

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

    -------------------
    F <$> (ρ₂ \ ρ₁) ≡t (F <$> ρ₂) \ (F <$> ρ₁)

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

  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

  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)
  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
```

Finally, it is helpful to reflect instances of propositional equality in Agda to proofs of type-equivalence.

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

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

$$\boxed{\Delta \vdash_{nf} \hat{\tau} : \kappa} \boxed{\Delta \vdash_{ne} n : \kappa}$$

$$(\kappa_{nf}\text{-NE}) \frac{\Delta \vdash_{ne} n : \gamma}{\Delta \vdash_{nf} n : \gamma} \qquad (\kappa_{nf}\text{-}\backslash) \frac{\Delta \vdash_{nf} \hat{\tau}_i : \mathsf{R}^{\kappa} \qquad \hat{\tau}_1 \notin \hat{\mathcal{P}} \text{ or } \hat{\tau}_2 \notin \hat{\mathcal{P}}}{\Delta \vdash_{nf} \hat{\tau}_2 \setminus \hat{\tau}_1 : \mathsf{R}^{\kappa}} \qquad (\kappa_{nf}\text{-}\triangleright) \frac{\Delta \vdash_{ne} n : \mathsf{L} \qquad \Delta \vdash_{nf} \hat{\tau} : \kappa}{\Delta \vdash_{nf} n \triangleright \hat{\tau} : \mathsf{R}^{\kappa}}$$

Fig. 2. Normal type forms

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

2.3.1 *Some admissable rules.*

We confirm that (i) Π and Σ are mapped over nested rows, and (ii) λ-bindings η-expand over Π and 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

We define reduction on types $\tau \longrightarrow_{\mathcal{T}} \tau'$ by directing the type equivalence judgment $\varepsilon \vdash \tau = \tau' : \kappa$ from left to right (with the exception of rule (E-MAP_{id}), which reduces right-to-left).

3.1 Mechanized syntax

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

NormalPred : KEnv → Kind → Set
NormalPred = Pred NormalType

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

IsNeutral IsNormal : NormalType Δ κ → Set
isNeutral? : ∀ (τ : NormalType Δ κ) → Dec (IsNeutral τ)
isNormal? : ∀ (τ : NormalType Δ κ) → Dec (IsNormal τ)

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

```
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

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

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

  `λ :
        (τ : NormalType (Δ ,, κ₁) κ₂) →
        ─────────────
        NormalType Δ (κ₁ `→ κ₂)

  _`→_ :
        (τ₁ τ₂ : NormalType Δ ★) →
        ────────
        NormalType Δ ★

  `∀    :

        (τ : NormalType (Δ ,, κ) ★) →
        ─────────────────
        NormalType Δ ★

  μ     :

        (φ : NormalType Δ (★ `→ ★)) →
        ─────────────
        NormalType Δ ★

  ────────────────────────────
  - Qualified types
```

```
  _⇒_ :

          (π : NormalPred Δ R[ κ₁ ]) → (τ : NormalType Δ ★) →
          ----------
          NormalType Δ ★

  ------------------------------

  -- Rω business

  ⦅_⦆ : (ρ : SimpleRow NormalType Δ R[ κ ]) → (oρ : True (normalOrdered? ρ)) →
        -----------
        NormalType Δ R[ κ ]
--    -- labels
  lab :

        (l : Label) →
        ----
        NormalType Δ L

  -- label constant formation
  ⌊_⌋ :
        (l : NormalType Δ L) →
        --------
        NormalType Δ ★

  Π :

        (ρ : NormalType Δ R[ ★ ]) →
        --------
        NormalType Δ ★

  Σ :

        (ρ : NormalType Δ R[ ★ ]) →
        -------
        NormalType Δ ★

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

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

   ------------------------------------ -- Ordered predicate

NormalOrdered [] = ⊤
NormalOrdered ((l , _) ∷ []) = ⊤
NormalOrdered ((l₁ , _) ∷ (l₂ , τ) ∷ xs) = l₁ < l₂ × NormalOrdered ((l₂ , τ) ∷ xs)
```

```
normalOrdered? [] = yes tt
normalOrdered? ((l , τ) :: []) = yes tt
normalOrdered? ((l1 , _) :: (l2 , _) :: xs) with l1 <? l2 | normalOrdered? ((l2 , _) :: xs)
... | yes p | yes q = yes (p , q)
... | yes p | no q = no (λ { (_ , oxs) → q oxs })
... | no p | yes q  = no (λ { (x , _) → p x})
... | no p | no q   = no (λ { (x , _) → p x})
```

```
NotSimpleRow (ne x) = ⊤
NotSimpleRow ((ϕ <$> τ)) = ⊤
NotSimpleRow ((| ρ |) oρ) = ⊥
NotSimpleRow (τ \ τ1) = ⊤
NotSimpleRow (x ▹n τ) = ⊤
```

3.2 Properties 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:

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

row-canonicity-∅ : (ρ : NormalType ∅ R[ κ ]) →
                     ∃[ xs ] Σ[ oxs ∈ True (normalOrdered? xs) ]
                     (ρ ≡ (| xs |) oxs)
row-canonicity-∅ (ne x) = ⊥-elim (noNeutrals x)
row-canonicity-∅ ((| ρ |) oρ) = ρ , oρ , refl
row-canonicity-∅ ((ρ \ ρ1) {nsr}) = ⊥-elim (noComplements nsr refl)
row-canonicity-∅ (l ▹n ρ) = ⊥-elim (noNeutrals l)
row-canonicity-∅ ((ϕ <$> ρ)) = ⊥-elim (noNeutrals ρ)

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.

```
ren_kNE : Renaming_k Δ1 Δ2 → NeutralType Δ1 κ → NeutralType Δ2 κ
ren_kNF : Renaming_k Δ1 Δ2 → NormalType Δ1 κ → NormalType Δ2 κ
renRow_kNF : Renaming_k Δ1 Δ2 → SimpleRow NormalType Δ1 R[ κ ] → SimpleRow NormalType Δ2 R[ κ ]
renPred_kNF : Renaming_k Δ1 Δ2 → NormalPred Δ1 R[ κ ] → NormalPred Δ2 R[ κ ]
```

Care must be given to ensure that the `NormalOrdered` and `NotSimpleRow` predicates are preserved.

```
orderedRenRow_kNF : (r : Renaming_k Δ1 Δ2) → (xs : SimpleRow NormalType Δ1 R[ κ ]) → NormalOrdered x
                    NormalOrdered (renRow_kNF r xs)

nsrRen_kNF : ∀ (r : Renaming_k Δ1 Δ2) (ρ1 ρ2 : NormalType Δ1 R[ κ ]) → NotSimpleRow ρ2 or NotSimpleRow
               NotSimpleRow (ren_kNF r ρ2) or NotSimpleRow (ren_kNF r ρ1)
nsrRen_kNF' : ∀ (r : Renaming_k Δ1 Δ2) (ρ : NormalType Δ1 R[ κ ]) → NotSimpleRow ρ →
               NotSimpleRow (ren_kNF r ρ)
```

3.4 Embedding

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

⇑ (ne x) = ⇑NE x
⇑ (‘λ τ) = ‘λ (⇑ τ)
⇑ (τ1 ‘→ τ2) = ⇑ τ1 ‘→ ⇑ τ2
⇑ (‘∀ τ) = ‘∀ (⇑ τ)
⇑ (μ τ) = μ (⇑ τ)
⇑ (lab l) = lab l
⇑ ⌊ τ ⌋ = ⌊ ⇑ τ ⌋
⇑ (Π x) = Π · ⇑ x
⇑ (Σ x) = Σ · ⇑ x
⇑ (π ⇒ τ) = (⇑Pred π) ⇒ (⇑ τ)
⇑ (⦅ ρ ⦆ oρ) = ⦅ ⇑Row ρ ⦆ (fromWitness (Ordered⇑ ρ (toWitness oρ)))
⇑ (ρ2 \ ρ1) = ⇑ ρ2 \ ⇑ ρ1
⇑ (l ▹_n τ) = (⇑NE l) ▹ (⇑ τ)
⇑ ((F <$> τ)) = (⇑ F) <$> (⇑NE τ)

⇑Row [] = []
⇑Row ((l , τ) :: ρ) = ((l , ⇑ τ) :: ⇑Row ρ)

Ordered⇑ [] oρ = tt
Ordered⇑ (x :: []) oρ = tt
Ordered⇑ ((l1 , _) :: (l2 , _) :: ρ) (l1<l2 , oρ) = l1<l2 , Ordered⇑ ((l2 , _) :: ρ) oρ
```

```
⇑Row-isMap : ∀ (xs : SimpleRow NormalType Δ₁ R[ κ ]) →
                   ⇑Row xs ≡ map (λ { (l , τ) → l , ⇑ τ }) xs
⇑Row-isMap [] = refl
⇑Row-isMap (x :: xs) = cong₂ _::_ refl (⇑Row-isMap xs)

⇑NE (‘ x) = ‘ x
⇑NE (τ₁ · τ₂) = (⇑NE τ₁) · (⇑ τ₂)

⇑Pred (ρ₁ · ρ₂ ~ ρ₃) = (⇑ ρ₁) · (⇑ ρ₂) ~ (⇑ ρ₃)
⇑Pred (ρ₁ ≲ ρ₂) = (⇑ ρ₁) ≲ (⇑ ρ₂)
```

4 Semantic types

```
--------------------------------------------------------------------------------
-- Semantic types (definition)

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

--------------------------------------------------------------------------------
-- Ordered predicate on semantic rows

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

--------------------------------------------------------------------------------
-- Defining SemType Δ R[ κ ]

data RowType (Δ : KEnv) (𝒯 : KEnv → Set) : Kind → Set
NotRow : ∀ {Δ : KEnv} {𝒯 : KEnv → Set} → RowType Δ 𝒯 R[ κ ] → Set
notRows? : ∀ {Δ : KEnv} {𝒯 : KEnv → Set} → (ρ₂ ρ₁ : RowType Δ 𝒯 R[ κ ]) → Dec (NotRow ρ₂ or NotRow ρ

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

  _▹_ : NeutralType Δ L → 𝒯 Δ → RowType Δ 𝒯 R[ κ ]

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

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

NotRow (x ▹ x₁) = ⊤
NotRow (row ρ x) = ⊥
```

```
NotRow (ρ \ ρ₁) = ⊤
NotRow (ϕ <$> ρ) = ⊤

notRows? (x ▹ x₁) ρ₁ = yes (left tt)
notRows? (ρ₂ \ ρ₃) ρ₁ = yes (left tt)
notRows? (ϕ <$> ρ) ρ₁ = yes (left tt)
notRows? (row ρ x) (x₁ ▹ x₂) = yes (right tt)
notRows? (row ρ x) (row ρ₁ x₁) = no (λ { (left ()) ; (right ()) })
notRows? (row ρ x) (ρ₁ \ ρ₂) = yes (right tt)
notRows? (row ρ x) (ϕ <$> τ) = yes (right tt)

--------------------------------------------------
- Defining Semantic types

SemType : KEnv → Kind → Set
SemType Δ ★ = NormalType Δ ★
SemType Δ L = NormalType Δ L
SemType Δ₁ (κ₁ `→ κ₂) = (∀ {Δ₂} → (r : Renamingₖ Δ₁ Δ₂) (v : SemType Δ₂ κ₁) → SemType Δ₂ κ₂)
SemType Δ R[ κ ] = RowType Δ (λ Δ' → SemType Δ' κ) R[ κ ]

--------------------------------------------------
- aliases

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)

--------------------------------------------------
- Truncating a row preserves ordering

ordered-cut : ∀ {n : ℕ} → {P : Fin (suc n) → Label × SemType Δ κ} →
              OrderedRow (suc n , P) → OrderedRow (n , P ∘ fsuc)
ordered-cut {n = zero} oρ = tt
ordered-cut {n = suc n} oρ = oρ .snd

--------------------------------------------------
- Ordering is preserved by mapping

orderedOverᵣ : ∀ {n} {P : Fin n → Label × SemType Δ κ₁} →
               (f : SemType Δ κ₁ → SemType Δ κ₂) →
               OrderedRow (n , P) → OrderedRow (n , overᵣ f ∘ P)
orderedOverᵣ {n = zero} {P} f oρ = tt
orderedOverᵣ {n = suc zero} {P} f oρ = tt
orderedOverᵣ {n = suc (suc n)} {P} f oρ = (oρ .fst) , (orderedOverᵣ f (oρ .snd))

--------------------------------------------------
- Semantic row operators

_::_ :   Label × SemType Δ κ → Row (SemType Δ κ) → Row (SemType Δ κ)
```

```
τ :: (n , P) = suc n , λ { fzero → τ
                         ; (fsuc x) → P x }
-- the empty row
εV : Row (SemType Δ κ)
εV = 0 , λ ()
```

4.1 Renaming and substitution

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

renSem : Renaming_k Δ1 Δ2 → SemType Δ1 κ → SemType Δ2 κ
renRow : Renaming_k Δ1 Δ2 →
         Row (SemType Δ1 κ) →
         Row (SemType Δ2 κ)

orderedRenRow : ∀ {n} {P : Fin n → Label × SemType Δ1 κ} → (r : Renaming_k Δ1 Δ2) →
                OrderedRow' n P → OrderedRow' n (λ i → (P i .fst) , renSem r (P i .snd))

nrRenSem : ∀ (r : Renaming_k Δ1 Δ2) → (ρ : RowType Δ1 (λ Δ' → SemType Δ' κ) R[ κ ]) →
           NotRow ρ → NotRow (renSem r ρ)
nrRenSem' : ∀ (r : Renaming_k Δ1 Δ2) → (ρ2 ρ1 : RowType Δ1 (λ Δ' → SemType Δ' κ) R[ κ ]) →
           NotRow ρ2 or NotRow ρ1 → NotRow (renSem r ρ2) or NotRow (renSem r ρ1)

renSem {κ = ★} r τ = ren_kNF r τ
renSem {κ = L} r τ = ren_kNF r τ
renSem {κ = κ `→ κ1} r F = renKripke r F
renSem {κ = R[ κ ]} r (φ <$> x) = (λ r' → φ (r' ∘ r)) <$> (ren_kNE r x)
renSem {κ = R[ κ ]} r (l ▹ τ) = (ren_kNE r l) ▹ renSem r τ
renSem {κ = R[ κ ]} r (row (n , P) q) = row (n , ( over_r (renSem r) ∘ P)) (orderedRenRow r q)
renSem {κ = R[ κ ]} r ((ρ2 \ ρ1) {nr}) = (renSem r ρ2 \ renSem r ρ1) {nr = nrRenSem' r ρ2 ρ1 nr}

nrRenSem' r ρ2 ρ1 (left x) = left (nrRenSem r ρ2 x)
nrRenSem' r ρ2 ρ1 (right y) = right (nrRenSem r ρ1 y)

nrRenSem r (x ▹ x1) nr = tt
nrRenSem r (ρ \ ρ1) nr = tt
nrRenSem r (φ <$> ρ) nr = tt

orderedRenRow {n = zero} {P} r o = tt
orderedRenRow {n = suc zero} {P} r o = tt
orderedRenRow {n = suc (suc n)} {P} r (l1<l2 , o) = l1<l2 , (orderedRenRow {n = suc n} {P ∘ fsuc} r o)

renRow φ (n , P) = n , over_r (renSem φ) ∘ P

weakenSem : SemType Δ κ1 → SemType (Δ ,, κ2) κ1
weakenSem {Δ} {κ1} τ = renSem {Δ1 = Δ} {κ = κ1} S τ
```

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

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 : Renaming_k Δ₁ Δ₂) → (τ : NormalType Δ₁ R[ ★ ]) → ren_kNF ρ (Ξ★ τ) ≡ Ξ★ (ren_kNF 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 : Renaming_k Δ1 Δ2) {V1 V2 : SemType Δ2 κ1} →
  V1 ≈ V2 → F ρ V1 ≈ G ρ V2

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

Uniform {Δ1} {κ1} {κ2} F =
  ∀ {Δ2 Δ3} (ρ1 : Renaming_k Δ1 Δ2) (ρ2 : Renaming_k 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 : Renaming_k Δ1 Δ2) (ρ2 : Renaming_k Δ2 Δ3) (V : NeutralType Δ2 κ1) →
  (renSem ρ2 (F ρ1 V)) ≈ F (ρ2 ∘ ρ1) (ren_kNE 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.