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$\omega\mu$.

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

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

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. Arguably 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; $(\to)$ as the type of type operators; and $R[\kappa]$ as the type of *rows* containing types at kind κ.

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

```
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. We say that the type variable x is indexed by kinding environment Δ and kind κ to specify that x has kind κ in kinding environment Δ.

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

2.1.1 *Quotienting kinds.*

```
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
```

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

2.2 Type syntax

```
infixr 2 _⇒_
infixl 5 _·_
infixr 5 _≲_
data Pred (Ty : KEnv → Kind → Set) Δ : Kind → Set
data Type Δ : Kind → Set

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

Ordered : SimpleRow Type Δ R[ κ ] → Set
ordered? : ∀ (xs : SimpleRow Type Δ R[ κ ]) → Dec (Ordered xs)

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 ]

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 ★

  ------------------------------
  -- Qualified types

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

  ------------------------------
  -- Rω business

  ⦅_⦆ : (xs : SimpleRow Type Δ R[ κ ]) (ordered : True (ordered? xs)) →
     -----------
     Type Δ R[ κ ]

  -- labels
  lab :

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

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

```
        -----
        Type Δ ★

  -- Row formation
  _▹_ :
          (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.

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

ordered? [] = yes tt
ordered? (x ∷ []) = yes tt
ordered? ((l₁ , _) ∷ (l₂ , _) ∷ xs) with l₁ <? l₂ | ordered? ((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})

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 = r

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

renₖ r (` x) = ` (r x)
renₖ r (`λ τ) = `λ (renₖ (liftₖ r) τ)
renₖ r (τ₁ · τ₂) = (renₖ r τ₁) · (renₖ r τ₂)
renₖ r (τ₁ `→ τ₂) = (renₖ r τ₁) `→ (renₖ r τ₂)
renₖ r (π ⇒ τ) = renPredₖ r π ⇒ renₖ r τ
renₖ r (`∀ τ) = `∀ (renₖ (liftₖ r) τ)
renₖ r (μ F) = μ (renₖ r F)
renₖ r (Π {notLabel = nl}) = Π {notLabel = nl}
renₖ r (Σ {notLabel = nl}) = Σ {notLabel = nl}
renₖ r (lab x) = lab x
renₖ r ⌊ ℓ ⌋ = ⌊ (renₖ r ℓ) ⌋
renₖ r (f <$> m) = renₖ r f <$> renₖ r m
renₖ r (⦅ xs ⦆ oxs) = ⦅ renRowₖ r xs ⦆ (fromWitness (orderedRenRowₖ r xs (toWitness oxs)))
renₖ r (ρ₂ ∖ ρ₁) = renₖ r ρ₂ ∖ renₖ r ρ₁
renₖ r (l ▹ τ) = renₖ r l ▹ renₖ r τ

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

renRowₖ r [] = []
renRowₖ r ((l , τ) ∷ xs) = (l , renₖ r τ) ∷ renRowₖ r xs

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

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

weakenPredₖ : Pred Type Δ R[ κ₂ ] → Pred Type (Δ ,, κ₁) R[ κ₂ ]
weakenPredₖ = renPredₖ S
```

2.2.5 Type substitution. `Substitutionₖ : KEnv → KEnv → Set`

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

-- lifting a substitution over binders.
liftsₖ : Substitutionₖ Δ₁ Δ₂ → Substitutionₖ(Δ₁ ,, κ) (Δ₂ ,, κ)
liftsₖ σ Z = ` Z
liftsₖ σ (S x) = weakenₖ (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.
subₖ : Substitutionₖ Δ₁ Δ₂ → Type Δ₁ κ → Type Δ₂ κ
```

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

$$\overline{(\pi_1 \Rightarrow \tau_1) \equiv\text{t} (\pi_2 \Rightarrow \tau_2)}$$

eq-lab :

$$\frac{\ell_1 \equiv \ell_2 \rightarrow}{\text{lab}\ \{\Delta = \Delta\}\ \ell_1 \equiv\text{t}\ \text{lab}\ \ell_2}$$

eq-row :
$\forall \{\rho_1\ \rho_2 : \text{SimpleRow Type}\ \Delta\ \text{R[}\ \kappa\ \text{]}\} \{o\rho_1 : \text{True (ordered?}\ \rho_1)\}$
$\{o\rho_2 : \text{True (ordered?}\ \rho_2)\} \rightarrow$

$$\frac{\rho_1 \equiv\text{r}\ \rho_2 \rightarrow}{(\!|\ \rho_1\ |\!)\ o\rho_1 \equiv\text{t}\ (\!|\ \rho_2\ |\!)\ o\rho_2}$$

eq-▹ : $\forall \{l_1\ l_2 : \text{Type}\ \Delta\ \text{L}\} \{\tau_1\ \tau_2 : \text{Type}\ \Delta\ \kappa\} \rightarrow$

$$\frac{l_1 \equiv\text{t}\ l_2 \rightarrow \tau_1 \equiv\text{t}\ \tau_2 \rightarrow}{(l_1 \triangleright \tau_1) \equiv\text{t}\ (l_2 \triangleright \tau_2)}$$

eq-\ : $\forall \{\rho_2\ \rho_1\ \upsilon_2\ \upsilon_1 : \text{Type}\ \Delta\ \text{R[}\ \kappa\ \text{]}\} \rightarrow$

$$\frac{\rho_2 \equiv\text{t}\ \upsilon_2 \rightarrow \rho_1 \equiv\text{t}\ \upsilon_1 \rightarrow}{(\rho_2 \setminus \rho_1) \equiv\text{t}\ (\upsilon_2 \setminus \upsilon_1)}$$

- ---------------
- η-laws

eq-η : $\forall \{f : \text{Type}\ \Delta\ (\kappa_1\ \text{‘}\!\rightarrow \kappa_2)\} \rightarrow$

$$\overline{f \equiv\text{t}\ \text{‘}\lambda\ (\text{weaken}_k\ f \cdot (\text{‘ Z}))}$$

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

eq-β : $\forall \{\tau_1 : \text{Type}\ (\Delta\ ,,\ \kappa_1)\ \kappa_2\} \{\tau_2 : \text{Type}\ \Delta\ \kappa_1\} \rightarrow$

$$\overline{((\text{‘}\lambda\ \tau_1) \cdot \tau_2) \equiv\text{t}\ (\tau_1\ \beta_k[\ \tau_2\ ])}$$

eq-labTy :

$$\frac{l \equiv\text{t}\ \text{lab}\ \ell \rightarrow}{}$$

```
  (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
```

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

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

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

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

Fig. 2. Normal type forms

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

```
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$_{\text{id}}$), 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 Δ_1 Δ_2 → NeutralType Δ_1 κ → NeutralType Δ_2 κ
ren_kNF : Renaming_k Δ_1 Δ_2 → NormalType Δ_1 κ → NormalType Δ_2 κ
renRow_kNF : Renaming_k Δ_1 Δ_2 → SimpleRow NormalType Δ_1 R[ κ ] → SimpleRow NormalType Δ_2 R[ κ ]
renPred_kNF : Renaming_k Δ_1 Δ_2 → NormalPred Δ_1 R[ κ ] → NormalPred Δ_2 R[ κ ]
```

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

```
orderedRenRow_kNF : (r : Renaming_k Δ_1 Δ_2) → (xs : SimpleRow NormalType Δ_1 R[ κ ]) → NormalOrdered x
                    NormalOrdered (renRow_kNF r xs)

nsrRen_kNF : ∀ (r : Renaming_k Δ_1 Δ_2) (ρ_1 ρ_2 : NormalType Δ_1 R[ κ ]) → NotSimpleRow ρ_2 or NotSimpleRow
               NotSimpleRow (ren_kNF r ρ_2) or NotSimpleRow (ren_kNF r ρ_1)
nsrRen_kNF' : ∀ (r : Renaming_k Δ_1 Δ_2) (ρ : NormalType Δ_1 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)
⇑ (τ_1 `→ τ_2) = ⇑ τ_1 `→ ⇑ τ_2
⇑ (`∀ 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)))
⇑ (ρ_2 \ ρ_1) = ⇑ ρ_2 \ ⇑ ρ_1
⇑ (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⇑ ((l_1 , _) ∷ (l_2 , _) ∷ ρ) (l_1<l_2 , oρ) = l_1<l_2 , Ordered⇑ ((l_2 , _) ∷ 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 : Renamingₖ Δ₁ Δ₂ → KripkeFunction Δ₁ κ₁ κ₂ → KripkeFunction Δ₂ κ₁ κ₂
renKripke {Δ₁} ρ F {Δ₂} = λ ρ' → F (ρ' ∘ ρ)

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

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

nrRenSem : ∀ (r : Renamingₖ Δ₁ Δ₂) → (ρ : RowType Δ₁ (λ Δ' → SemType Δ' κ) R[ κ ]) →
           NotRow ρ → NotRow (renSem r ρ)
nrRenSem' : ∀ (r : Renamingₖ 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 = renₖNF r τ
renSem {κ = L} r τ = renₖNF r τ
renSem {κ = κ `→ κ₁} r F = renKripke r F
renSem {κ = R[ κ ]} r (ϕ <$> x) = (λ r' → ϕ (r' ∘ r)) <$> (renₖNE r x)
renSem {κ = R[ κ ]} r (l ▹ τ) = (renₖNE r l) ▹ renSem r τ
renSem {κ = R[ κ ]} r (row (n , P) q) = row (n , ( overᵣ (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 , overᵣ (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 : 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-$[\![]\!]\approx$ ($\vdash[\![\ \tau\ ]\!]\approx$)

—————————————————

– If τ_1 normalizes to $\Downarrow\ \tau_2$ then the embedding of τ_1 is equivalent to τ_2

embed-≡t : $\forall\ \{\tau_1 : \text{NormalType}\ \Delta\ \kappa\}\ \{\tau_2 : \text{Type}\ \Delta\ \kappa\} \to \tau_1 \equiv (\Downarrow\ \tau_2) \to\ \Uparrow\ \tau_1\ \equiv\!\text{t}\ \tau_2$

embed-≡t $\{\tau_1 = \tau_1\}\ \{\tau_2\}$ refl = eq-sym (soundness τ_2)

—————————————————

– Soundness implies the converse of completeness, as desired

Completeness^{-1} : $\forall\ \{\Delta\ \kappa\} \to (\tau_1\ \tau_2 : \text{Type}\ \Delta\ \kappa) \to\ \Downarrow\ \tau_1 \equiv\ \Downarrow\ \tau_2 \to \tau_1\ \equiv\!\text{t}\ \tau_2$

Completeness^{-1} $\tau_1\ \tau_2\ eq$ = eq-trans (soundness τ_1) (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.