# 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 $\beta$- and $\eta$-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 $\Pi$, $\Sigma$, 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 $\kappa$.

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 $\Delta$ and $\kappa$ 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 $\Delta$ and kind $\kappa$ to specify that $x$ has kind $\kappa$ in kinding environment $\Delta$.

```
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)

  _·_ :
```

```agda
      (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)
```

```
ren_k : Renaming_k Δ_1 Δ_2 → Type Δ_1 κ → Type Δ_2 κ
renPred_k : Renaming_k Δ_1 Δ_2 → Pred Type Δ_1 R[ κ ] → Pred Type Δ_2 R[ κ ]
renRow_k : Renaming_k Δ_1 Δ_2 → SimpleRow Type Δ_1 R[ κ ] → SimpleRow Type Δ_2 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 Δ1 Δ2 → Pred Type Δ1 κ → Pred Type Δ2 κ
subRow_k : Substitution_k Δ1 Δ2 → SimpleRow Type Δ1 R[ κ ] → SimpleRow Type Δ2 R[ κ ]
orderedSubRow_k : (σ : Substitution_k Δ1 Δ2) → (xs : SimpleRow Type Δ1 R[ κ ]) → Ordered xs →
                  Ordered (subRow_k σ xs)
- sub_k σ ϵ = ϵ
sub_k σ (‘ x) = σ x
sub_k σ (‘λ τ) = ‘λ (sub_k (lifts_k σ) τ)
sub_k σ (τ1 · τ2) = (sub_k σ τ1) · (sub_k σ τ2)
sub_k σ (τ1 ‘→ τ2) = (sub_k σ τ1) ‘→ (sub_k σ τ2)
sub_k σ (π ⇒ τ) = subPred_k σ π ⇒ sub_k σ τ
sub_k σ (‘∀ τ) = ‘∀ (sub_k (lifts_k σ) τ)
sub_k σ (μ F) = μ (sub_k σ F)
sub_k σ (Π {notLabel = nl}) = Π {notLabel = nl}
sub_k σ (Σ {notLabel = nl}) = Σ {notLabel = nl}
sub_k σ (lab x) = lab x
sub_k σ ⌊ ℓ ⌋ = ⌊ (sub_k σ ℓ) ⌋
sub_k σ (f <$> a) = sub_k σ f <$> sub_k σ a
sub_k σ (ρ2 \ ρ1) = sub_k σ ρ2 \ sub_k σ ρ1
sub_k σ ((| xs |) oxs) = (| subRow_k σ xs |) (fromWitness (orderedSubRow_k σ xs (toWitness oxs)))
sub_k σ (l ▹ τ) = (sub_k σ l) ▹ (sub_k σ τ)
subRow_k σ [] = []
subRow_k σ ((l , τ) ∷ xs) = (l , sub_k σ τ) ∷ subRow_k σ xs

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

subRow_k-isMap : ∀ (σ : Substitution_k Δ1 Δ2) (xs : SimpleRow Type Δ1 R[ κ ]) →
                 subRow_k σ xs ≡ map (over_r (sub_k σ)) xs

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

subPred_k σ (ρ1 · ρ2 ~ ρ3) = sub_k σ ρ1 · sub_k σ ρ2 ~ sub_k σ ρ3
subPred_k σ (ρ1 ≲ ρ2) = (sub_k σ ρ1) ≲ (sub_k σ ρ2)

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

- Single variable sub_kstitution is a special case of simultaneous sub_kstitution.
_β_k[_] : Type (Δ ,, κ1) κ2 → Type Δ κ1 → Type Δ κ2
B β_k[ A ] = sub_k (extend_k ‘ 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
```

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

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

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

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

Fig. 2. Normal type forms

*2.3.1 Some admissable rules.* We confirm that (i) $\Pi$ and $\Sigma$ are mapped over nested rows, and (ii) $\lambda$-bindings $\eta$-expand over $\Pi$ and $\Sigma$.

```
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
```

# 4 SEMANTIC TYPES

## 4.1 Renaming and substitution

# 5 NORMALIZATION BY EVALUATION

## 5.1 Helping evaluation

## 5.2 Evaluation

# 6 METATHEORY

## 6.1 A logical relation for completeness

### *6.1.1 Properties.*

## 6.2 The fundamental theorem and completeness

## 6.3 A logical relation for soundness

### *6.3.1 Properties.*

## 6.4 The fundamental theorem and soundness

# 7 THE REST OF THE PICTURE

# 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.