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

ALEX HUBERS, The University of Iowa, USA

## ABSTRACT

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

## 1 THE R$\omega\mu$ CALCULUS

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

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

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

Fig. 1. Syntax

### 1.1 Example types and the need for reduction

We will write Rome types in programs in the slightly-altered syntax of *Rosi*, our experimental implementation of R$\omega\mu$. Wand's problem:

```
wand : ∀ l x y z t. x ⊙ y ~ z, {l ▹ t} ≲ z ⇒ #l → Π x → Π y → t
```

Here we can simulate the deriving of functor typeclass instances: given a record of `fmap` instances, I can give you a `Functor` instance for $\Sigma\, z$.

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

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

Let's first elaborate this type by rendering some implicit notation (e.g., maps) explicit.

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

## 2.1 Normal forms

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

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

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

Fig. 2. Normal type forms

## 2.2 Metatheory

*2.2.1 Canonicity of normal types.* The syntax of normal types is defined precisely so as to enjoy canonical forms based on kind.

*2.2.2 Completeness of normalization.*

*2.2.3 Soundness of normalization.*

*2.2.4 Decidability of type conversion.*

# 3 NORMALIZATION BY EVALUATION (NBE)

## 3.1 The semantic domain

## 3.2 reflection & reification

## 3.3 Evaluation

## 3.4 Normalization

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

# 4 MECHANIZING METATHEORY

## 4.1 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) idE
```

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))
```

### 4.2 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

- _≈_ {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)
```

#### 4.2.1 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)
```

### 4.3 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 ρ
```

### 4.4 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})))) × (⟦ ⇑
- ⟦_⟧≈_ {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)
```

#### 4.4.1 *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
```

### 4.4.2 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

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

- 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)
```

## REFERENCES