Type Normalization in R$\omega\mu$

ALEX HUBERS, The University of Iowa, USA

1 INTRODUCTION

We describe the normalization-by-evaluation (NBE) of types in R$\omega\mu$. 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 label bound terms.

2 SYNTAX OF KINDS

Our formalization of R$\omega\mu$ types is *intrinsic*, meaning we define the syntax of *typing* and *kinding judgments*, foregoing any description of untyped syntax. The syntax of types is indexed by kinding environments and kinds, defined 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. As shorthand, we write $R^n[\kappa]$ to denote n repeated applications of R to the type κ–e.g., $R^3[\kappa]$ is shorthand for $R[R[R[\kappa]]]$.

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

```
data KEnv : Set where
  ϵ : 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 kind variable x is indexed by kinding environment Δ and kind κ to specify that x has kind κ in kinding environment Δ.

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

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

3 SYNTAX OF TYPES

Rωμ is a qualified type system with predicates of the form $\rho_1 \lesssim \rho_2$ and $\rho_1 \cdot \rho_2 \sim \rho_3$ for row-kinded types ρ_1, ρ_2, and ρ_3. Because predicates occur in types and types occur in predicates, the syntax of well-kinded types and well-kinded predicates are mutually recursive. The syntax for each is given below. we describe (in this order) the syntactic components belonging to System Fωμ, qualified type systems, and system Rω.

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

  ‘ :
      (α : KVar 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 ★

  ‘∀ :

          (τ : Type (Δ „ κ) ★) →
          Type Δ ★

  μ :

          (F : Type Δ (★ ‘→ ★)) →
          Type Δ ★
```

The first three constructors are analogous to the terms of the STLC. the constructor `→ classifies term functions; the constructor `∀ classifies type-in-term quantification; and the constructor μ classifies recursive terms. Note that μ could be further generalized to kind κ `→ ★; however, we

found that kind $\star$ `⟶ $\star$ was sufficient for our needs while simplifying both presentation and mechanization.

The syntax of qualified types is given below.

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

The type π `⇒` τ states that τ is *qualified* by the predicate π—that is, the type variables bound in τ are restricted in instantiation to just those that satisfy the predicate π. This is completely analogous to identical syntax used in Haskell to introduce typeclass qualification. Predicates are defined below (after the presentation of type syntax).

We now describe the syntax exclusive to R$\omega\mu$, beginning with label kind introduction and elimination. Labels are first-class entities in R$\omega\mu$, and may be represented by both constants and variables.

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

Label constants in R$\omega\mu$ are constructed from the type `Label`; in our mechanization, `Label` is a type synonym for `String`, but one could choose any other candidate with decidable equality. Types at label kind `L` may be cast to *label singletons* by the $\lfloor\_\rfloor$ constructor. This makes labels first-class entities: for example, as the type `⌊ lab "l" ⌋` has kind $\star$, it can be inhabited by a term.

Types at row kind are constructed by one of the following three constructors.

```
ϵ :
  Type Δ R[ κ ]
_▷_ :
      (l : Type Δ L) → (τ : Type Δ κ) →
      Type Δ R[ κ ]
_<$>_ :
  (f : Type Δ (κ₁ `⟶ κ₂)) → (τ : Type Δ R[ κ₁ ]) →
  Type Δ R[ κ₂ ]
```

Rows in R$\omega\mu$ are either the empty row ϵ, a labeled row `(l ▷ τ)`, or a row mapping `f <$> τ`. We will show that rows in Rome (that is, types at row kind) reduce to either the empty row ϵ or a labeled row `(l ▷ τ)` after normalization. There are two important consequences of this canonicity: firstly, we treat row mapping _<$>_ as having latent computation to perform (there are no normal types with form `f <$> τ` except when τ is a neutral variable). The second consequence is that we

do not permit the formation of rows with more than one label-type association. Such rows are instead formed as type variables with predicates specifying the shape of the row.

Rows in R$\omega\mu$ are eliminated by the Π and Σ constructors.

```
Π :
  Type Δ (R[ κ ] ‘→ κ)

Σ :
  Type Δ (R[ κ ] ‘→ κ)
```

Given a type ρ at row kind, $\Pi\rho$ constructs a record with label-type associations from ρ and $\Sigma\rho$ constructs a variant that has label and type from ρ. We choose to represent Π and Σ as type constants at kind `(R[ κ ] `→ κ)`; we will show that many applications of Π and Σ induce type reductions, and hence it is convenient to group such reductions with type application.

Finally, the syntax of predicates is given below. The predicate $\rho_1 \lesssim \rho_2$ states that label-to-type mappings in ρ_1 are a subset of those in ρ_2; the predicate $\rho_1 \cdot \rho_2 \sim \rho_3$ states that the combination of mappings in ρ_1 and ρ_2 equals ρ_3.[1]

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

  _≲_ :
        (ρ₁ ρ₂ : Type Δ R[ κ ]) →
        Pred Δ R[ κ ]
```

3.1 Type renaming

We closely follow PLFA and SFFP (citations needed) in defining a *type renaming* as a function from type variables in one kinding environment to type variables in another. This is the *parallel renaming and substitution* approach (citation needed) for which weakening and single variable substitution are special cases. The code we establish now will be mimicked again for both normal types and for terms; many names are reused, and so we find it helpful to index duplicate names by a suffix. The suffix $_k$ specifies that this definition describes the Type syntax.

```
Renaming_k : KEnv → KEnv → Set
Renaming_k Δ₁ Δ₂ = ∀ {κ} → KVar Δ₁ κ → KVar Δ₂ κ
```

We will let the metavariable ρ range over both renamings and types at row kind.

Lifting can be thought of as the weakening of a renaming, and permits renamings to be pushed under binders.

```
lift_k : Renaming_k Δ₁ Δ₂ → Renaming_k (Δ₁ „ κ) (Δ₂ „ κ)
lift_k ρ Z = Z
lift_k ρ (S x) = S (ρ x)
```

[1] I do not know if I should generalize this description to be row-theory agnostic or if I should specialize it to a specific row theory.

We define renaming as a function that translates a kinding derivation in kinding environment Δ_1 to environment Δ_2 provided a renaming from Δ_1 to Δ_2. The definition proceeds by induction on the input kinding derivation; we describe only the interesting cases, omitting the cases which are effectively just congruence over the type structure. In the variable case, we use ρ to rename variable x. In the `$\lambda$ and `$\forall$ cases, we must lift the renaming ρ over the type variable introduced by these binders.

$$\mathsf{ren}_k : \mathsf{Renaming}_k\ \Delta_1\ \Delta_2 \to \mathsf{Type}\ \Delta_1\ \kappa \to \mathsf{Type}\ \Delta_2\ \kappa$$
$$\mathsf{renPred}_k : \mathsf{Renaming}_k\ \Delta_1\ \Delta_2 \to \mathsf{Pred}\ \Delta_1\ \mathsf{R}[\ \kappa\ ] \to \mathsf{Pred}\ \Delta_2\ \mathsf{R}[\ \kappa\ ]$$

$$\mathsf{ren}_k\ \rho\ (\text{`}\ x) = \text{`}\ (\rho\ x)$$
$$\mathsf{ren}_k\ \rho\ (\text{`}\lambda\ \tau) = \text{`}\lambda\ (\mathsf{ren}_k\ (\mathsf{lift}_k\ \rho)\ \tau)$$
$$\mathsf{ren}_k\ \rho\ (\pi \Rightarrow \tau) = \mathsf{renPred}_k\ \rho\ \pi \Rightarrow \mathsf{ren}_k\ \rho\ \tau$$
$$\mathsf{ren}_k\ \rho\ (\text{`}\forall\ \tau) = \text{`}\forall\ (\mathsf{ren}_k\ (\mathsf{lift}_k\ \rho)\ \tau)$$

As `Type` and `Pred` are mutually inductive, we must define $\mathtt{renPred}_k$ as mutually recursive to $\mathtt{ren}_k$. Its definition is completely unsuprising.

$$\mathsf{renPred}_k\ \rho\ (\rho_1 \cdot \rho_2 \sim \rho_3) = \mathsf{ren}_k\ \rho\ \rho_1 \cdot \mathsf{ren}_k\ \rho\ \rho_2 \sim \mathsf{ren}_k\ \rho\ \rho_3$$
$$\mathsf{renPred}_k\ \rho\ (\rho_1 \lesssim \rho_2) = (\mathsf{ren}_k\ \rho\ \rho_1) \lesssim (\mathsf{ren}_k\ \rho\ \rho_2)$$