

# 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, Figure 1 describes the syntax of kinds, predicates, and types in  $R\omega\mu$ . We forego further description to the next section.

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

Kinds  $\kappa ::= \star \mid L \mid R^K \mid \kappa \rightarrow \kappa$   
Predicates  $\pi, \psi ::= \rho \lesssim \rho \mid \rho \odot \rho \sim \rho$   
Types  $\mathcal{T} \ni \phi, \tau, v, \rho, \xi ::= \alpha \mid \pi \Rightarrow \tau \mid \forall \alpha : \kappa. \tau \mid \lambda \alpha : \kappa. \tau \mid \tau \tau$   
 $\mid \{\xi_i \triangleright \tau_i\}_{i \in 0 \dots m} \mid \ell \mid \# \tau \mid \phi \$ \rho \mid \rho \setminus \rho$   
 $\mid \tau \rightarrow \tau \mid \Pi \mid \Sigma \mid \mu \phi$

Fig. 1. Syntax

### 1.1 Example types

Wand's problem and a record modifier:

```
wand :  $\forall l \ x \ y \ z \ t. \ x \odot y \sim z, \{l \triangleright t\} \lesssim z \Rightarrow \#l \rightarrow \Pi x \rightarrow \Pi y \rightarrow t$   
modify :  $\forall l \ t \ u \ y \ z1 \ z2. \{l \triangleright t\} \odot y \sim z1, \{l \triangleright u\} \odot y \sim z2 \Rightarrow$   
 $\#l \rightarrow (t \rightarrow u) \rightarrow \Pi z1 \rightarrow \Pi z2$ 
```

"Deriving" functor typeclass instances:

```
type Functor :  $(\star \rightarrow \star) \rightarrow \star$   
type Functor =  $\lambda f. \forall a \ b. (a \rightarrow b) \rightarrow f \ a \rightarrow f \ b$ 
```

```
fmapS :  $\forall z : R[\star \rightarrow \star]. \Pi (Functor \ z) \rightarrow Functor (\Sigma \ z)$   
fmapP :  $\forall z : R[\star \rightarrow \star]. \Pi (Functor \ z) \rightarrow Functor (\Pi \ z)$ 
```

And a desugaring of booleans to Church encodings:

```
desugar :  $\forall y. BoolF \lesssim y, LamF \lesssim y \setminus BoolF \Rightarrow$   
 $\Pi (Functor (y \setminus BoolF)) \rightarrow \mu (\Sigma \ y) \rightarrow \mu (\Sigma (y \setminus BoolF))$ 
```

## 2 MECHANIZED SYNTAX

### 2.1 Kind syntax

Our formalization of  $R\omega\mu$  types is *intrinsic*, meaning we define the syntax of *typing* and *kinding judgments*, foregoing any formalization of or indexing-by untyped syntax. The only "untyped" syntax is that of kinds, which are well-formed grammatically. We give the syntax of kinds and kinding environments below.

```
data Kind : Set where
```

```
  ★   : Kind
```

```
  L   : Kind
```

```
  _'→_ : Kind → Kind → Kind
```

```
  R[ ] : Kind → Kind
```

```
data KEnv : Set where
```

```
  ∅ : KEnv
```

```
  _„_ : KEnv → Kind → KEnv
```

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  $\kappa$ . Kinding environments are isomorphic to lists of kinds.

The syntax of intrinsically well-scoped De-Brujin type variables is given below. Type variables indexed in this way are analogous to the  $\_ \in \_$  relation for Agda lists—that is, each type variable is itself a proof of its location within the kinding environment. Let the metavariables  $\Delta$  and  $\kappa$  range over kinding environments and kinds, respectively.

```
data TVar : KEnv → Kind → Set where
```

```
  Z : TVar ( $\Delta$  „  $\kappa$ )  $\kappa$ 
```

```
  S : TVar  $\Delta$   $\kappa_1 \rightarrow$  TVar ( $\Delta$  „  $\kappa_2$ )  $\kappa_1$ 
```

**2.1.1 Partitioning kinds.** It will be necessary to partition kinds by two predicates. The predicate  $\text{NotLabel } \kappa$  is satisfied if  $\kappa$  is neither of label kind, a row of label kind, nor a type operator that returns a labeled kind. It is trivial to show that this predicate is decidable.

```
NotLabel : Kind → Set
```

```
NotLabel ★ = ⊤
```

```
NotLabel L = ⊥
```

```
NotLabel ( $\kappa_1$  '→  $\kappa_2$ ) = NotLabel  $\kappa_2$ 
```

```
NotLabel R[  $\kappa$  ] = NotLabel  $\kappa$ 
```

```
notLabel? : ∀  $\kappa \rightarrow$  Dec (NotLabel  $\kappa$ )
```

```
notLabel? ★ = yes tt
```

```
notLabel? L = no λ ()
```

```
notLabel? ( $\kappa$  '→  $\kappa_1$ ) = notLabel?  $\kappa_1$ 
```

```
notLabel? R[  $\kappa$  ] = notLabel?  $\kappa$ 
```

The predicate  $\text{Ground } \kappa$  is satisfied when  $\kappa$  is the kind of types or labels, and is necessary to reserve the promotion of neutral types to just those at these kinds. It is again trivial to show that this predicate is decidable, and so a definition of  $\text{ground?}$  is omitted.

```
Ground : Kind → Set
```

```
ground? : ∀  $\kappa \rightarrow$  Dec (Ground  $\kappa$ )
```

```
Ground ★ = ⊤
```

```
Ground L = ⊤
```

```
Ground ( $\kappa$  '→  $\kappa_1$ ) = ⊥
```

```
Ground R[  $\kappa$  ] = ⊥
```

## 2.2 Type syntax

We represent the judgment  $\Gamma \vdash \tau : \kappa$  intrinsically as the data type  $\text{Type } \Delta \kappa$ . The data type  $\text{Pred } \text{Type } \Delta \text{R}[\kappa]$  represents well-kinded predicates indexed by  $\text{Type } \Delta \kappa$ . The two are necessarily mutually inductive. Note that the syntax of predicates will be the same for both types and normalized types, and so the  $\text{Pred}$  data type is indexed abstractly by type  $\text{Ty}$ .

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

We must also define syntax for *simple rows*, that is, row literals. For uniformity of kind indexing, we define a  $\text{SimpleRow}$  by pattern matching on the syntax of kinds. Like with  $\text{Pred}$ , simple rows are indexed by abstract type  $\text{Ty}$  so that we may reuse the same pattern for normalized types.

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

A simple row is *ordered* if it is of length  $\leq 1$  or its corresponding labels are ordered according to some total order  $<$ . We will restrict the formation of row literals to just those that are ordered, which has two key consequences: first, it guarantees a normal form (later) for simple rows, and second, it enforces that labels be unique in each row. It is easy to show that the  $\text{Ordered}$  predicate is decidable.

```
Ordered : SimpleRow Type Δ R[κ] → Set
ordered? : ∀ (xs : SimpleRow Type Δ R[κ]) → Dec (Ordered xs)
Ordered [] = ⊤
Ordered (x :: []) = ⊤
Ordered ((l1, _) :: (l2, τ) :: xs) = l1 < l2 × Ordered ((l2, τ) :: xs)
```

The syntax of well-kinded predicates is exactly as expected.

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

The syntax of kinding judgments is given below. The formation rules for  $\lambda$ -abstractions, applications, arrow types, and  $\forall$  and  $\mu$  types are standard and omitted. The constructor  $\_ \Rightarrow \_$  forms a qualified type given a well-kinded predicate  $\pi$  and a  $\star$ -kinded body  $\tau$ . Labels are formed from label literals and cast to kind  $\star$  via the  $\_ \_$  constructor. The remaining constructors describe row formation: The constructor  $\_ \_$  forms a row literal from a well-ordered simple row. We additionally allow the syntax  $\_ \triangleright \_$  for constructing row singletons of (perhaps) variable label; this role can be performed by  $\_ \_$  when the label is a literal. The  $\_ \_ \_$  constructor describes the map of a type operator over a row.  $\Pi$  and  $\Sigma$  form records and variants from rows for which the  $\text{NotLabel}$  predicate is satisfied. Finally, the  $\_ \setminus \_$  constructor forms the relative complement of two rows. The novelty in this report will come from showing how types of these forms reduce.

```
data Type Δ where
  ' : (α : TVar Δ κ) → Type Δ κ
  _⇒_ : (π : Pred Type Δ R[κ1]) → (τ : Type Δ ★) → Type Δ ★
  lab : (l : Label) → Type Δ L
```

```

148   [ ] : (τ : Type Δ L) → Type Δ ★
149   (|_) : (xs : SimpleRow Type Δ R[ κ ]) (ordered : True (ordered? xs)) → Type Δ R[ κ ]
150   _>_ : (l : Type Δ L) → (τ : Type Δ κ) → Type Δ R[ κ ]
151   _<$>_ : (φ : Type Δ (κ1 '→ κ2)) → (τ : Type Δ R[ κ1 ]) → Type Δ R[ κ2 ]
152   Π : {notLabel : True (notLabel? κ)} → Type Δ (R[ κ ] '→ κ)
153   Σ : {notLabel : True (notLabel? κ)} → Type Δ (R[ κ ] '→ κ)
154   _\ _ : Type Δ R[ κ ] → Type Δ R[ κ ] → Type Δ R[ κ ]
155

```

2.2.1 *The ordered predicate.* We impose on the  $(|_)$  constructor a witness of the form `True (ordered? xs)`, although it may seem more intuitive to have instead simply required a witness that `Ordered xs`. The reason for this is that the `True` predicate quotients each proof down to a single inhabitant `tt`, which grants us proof irrelevance when comparing rows. This is desirable and yields congruence rules that would otherwise be blocked by two differing proofs of well-orderedness. The congruence rule below asserts that two simple rows are equivalent even with differing proofs. (This pattern is replicable for any decidable predicate.)

```

164   cong-SimpleRow : {sr1 sr2 : SimpleRow Type Δ R[ κ ]}
165                   {wf1 : True (ordered? sr1)} {wf2 : True (ordered? sr2)} →
166                   sr1 ≡ sr2 → (| sr1 |) wf1 ≡ (| sr2 |) wf2
167   cong-SimpleRow {sr1 = sr1} { } {wf1} {wf2} refl
168   rewrite Dec→Irrelevant (Ordered sr1) (ordered? sr1) wf1 wf2 = refl
169

```

In the same fashion, we impose on  $\Pi$  and  $\Sigma$  a similar restriction that their kinds satisfy the `NotLabel` predicate, although our reason for this restriction is instead metatheoretic: without it, nonsensical labels could be formed such as  $\Pi \text{ (lab "a" } \triangleright \text{ lab "b")}$  or  $\Pi \epsilon$ . Each of these types have kind `L`, which violates a label canonicity theorem we later show that all label-kinded types in normal form are label literals or neutral.

### 2.2.2 Flipped map operator.

Hubers and Morris [2023] had a left- and right-mapping operator, but only one is necessary. The flipped application (`flap`) operator is defined below. Its type reveals its purpose.

```

180   flap : Type Δ (R[ κ1 '→ κ2 ] '→ κ1 '→ R[ κ2 ])
181   flap = 'λ ('λ (('λ (('Z) · ('(S Z)))) <$> ('(S Z))))
182
183   _??_ : Type Δ (R[ κ1 '→ κ2 ]) → Type Δ κ1 → Type Δ R[ κ2 ]
184   f ?? a = flap · f · a
185

```

### 2.2.3 The (syntactic) complement operator.

It is necessary to give a syntactic account of the computation incurred by the complement of two row literals so that we can state this computation later in the type equivalence relation. First, define a relation  $\ell \in_L \rho$  that is inhabited when the label literal  $\ell$  occurs in the row  $\rho$ . This relation is decidable ( $\_ \in_L ?\_$ , definition omitted).

```

192   data _∈L_ : (l : Label) → SimpleRow Type Δ R[ κ ] → Set where
193     Here : ∀ {τ : Type Δ κ} {xs : SimpleRow Type Δ R[ κ ]} {l : Label} →
194           l ∈L (l, τ) :: xs
195     There : ∀ {τ : Type Δ κ} {xs : SimpleRow Type Δ R[ κ ]} {l l' : Label} →
196

```

$l \in L \text{ } xs \rightarrow l \in L (l', \tau) :: xs$   
 $\_ \in L? \_ : \forall (l : \text{Label}) (xs : \text{SimpleRow Type } \Delta \text{ } R[\kappa]) \rightarrow \text{Dec } (l \in L \text{ } xs)$

We now define the syntactic *row complement* effectively as a filter: when a label on the left is found in the row on the right, we exclude that labeled entry from the resulting row.

$\_ \backslash s \_ : \forall (xs \text{ } ys : \text{SimpleRow Type } \Delta \text{ } R[\kappa]) \rightarrow \text{SimpleRow Type } \Delta \text{ } R[\kappa]$   
 $[] \backslash s \text{ } ys = []$   
 $((l, \tau) :: xs) \backslash s \text{ } ys \text{ with } l \in L? \text{ } ys$   
 $\dots \mid \text{yes } \_ = xs \backslash s \text{ } ys$   
 $\dots \mid \text{no } \_ = (l, \tau) :: (xs \backslash s \text{ } ys)$

#### 2.2.4 Type renaming and substitution.

A type variable renaming is a map from type variables in environment  $\Delta_1$  to type variables in environment  $\Delta_2$ .

$\text{Renaming}_k : \text{KEnv} \rightarrow \text{KEnv} \rightarrow \text{Set}$   
 $\text{Renaming}_k \Delta_1 \Delta_2 = \forall \{\kappa\} \rightarrow \text{TVar } \Delta_1 \kappa \rightarrow \text{TVar } \Delta_2 \kappa$

This definition and approach is standard for the intrinsic style (cf. Chapman et al. [2019]; Wadler et al. [2022]) and so definitions are omitted. The only deviation of interest is that we have an obligation to show that renaming preserves the well-orderedness of simple rows. Note that we use the suffix  $\_k$  for common operations over the Type and Pred syntax; we will use the suffix  $\_k\text{NF}$  for equivalent operations over the normal type syntax.

$\text{orderedRenRow}_k : (r : \text{Renaming}_k \Delta_1 \Delta_2) \rightarrow (xs : \text{SimpleRow Type } \Delta_1 \text{ } R[\kappa]) \rightarrow \text{Ordered } xs \rightarrow \text{Ordered } (\text{renRow}_k \text{ } r \text{ } xs)$

A substitution is a map from type variables to types.

$\text{Substitution}_k : \text{KEnv} \rightarrow \text{KEnv} \rightarrow \text{Set}$   
 $\text{Substitution}_k \Delta_1 \Delta_2 = \forall \{\kappa\} \rightarrow \text{TVar } \Delta_1 \kappa \rightarrow \text{Type } \Delta_2 \kappa$

Parallel renaming and substitution is likewise standard for this approach, and so definitions are omitted. As will become a theme, we must show that substitution preserves row well-orderedness.

$\text{orderedSubRow}_k : (\sigma : \text{Substitution}_k \Delta_1 \Delta_2) \rightarrow (xs : \text{SimpleRow Type } \Delta_1 \text{ } R[\kappa]) \rightarrow \text{Ordered } xs \rightarrow \text{Ordered } (\text{subRow}_k \text{ } \sigma \text{ } xs)$

Two operations of note: extension of a substitution  $\sigma$  appends a new type  $A$  as the zero'th De Bruijn index.  $\beta$ -substitution is a special case of substitution in which we only substitute the most recently freed variable.

$\text{extend}_k : \text{Substitution}_k \Delta_1 \Delta_2 \rightarrow (A : \text{Type } \Delta_2 \kappa) \rightarrow \text{Substitution}_k (\Delta_1 \text{ } \text{, } \kappa) \Delta_2$   
 $\text{extend}_k \sigma \text{ } A \text{ } Z = A$   
 $\text{extend}_k \sigma \text{ } A \text{ } (S \text{ } x) = \sigma \text{ } x$

$\_ \beta_k \_ : \text{Type } (\Delta \text{ } \text{, } \kappa_1) \kappa_2 \rightarrow \text{Type } \Delta \kappa_1 \rightarrow \text{Type } \Delta \kappa_2$   
 $B \beta_k [A] = \text{sub}_k (\text{extend}_k \text{ } A) B$

### 2.3 Type equivalence

We define reduction on types  $\tau \longrightarrow_{\mathcal{T}} \tau'$  by directing the following type equivalence judgment  $\Delta \vdash \tau = \tau' : \kappa$  from left to right. We equate types under the relation  $\_ \equiv t \_$ , predicates under the relation  $\_ \equiv p \_$ , and row literals under the relation  $\_ \equiv r \_$ .

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

Row literals and predicates are equated in an obvious fashion.

```
data _≡r_ 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 v1 → τ2 ≡t v2 → τ1 ≤ τ2 ≡p v1 ≤ v2
  _eq-··~_ : τ1 ≡t v1 → τ2 ≡t v2 → τ3 ≡t v3 →
    τ1 · τ2 ~ τ3 ≡p v1 · v2 ~ v3
```

The first three type equivalence rules enforce that  $\_ \equiv t \_$  forms an equivalence relation.

```
data _≡t_ where
  eq-refl : τ ≡t τ
  eq-sym : τ1 ≡t τ2 → τ2 ≡t τ1
  eq-trans : τ1 ≡t τ2 → τ2 ≡t τ3 → τ1 ≡t τ3
```

We next have a number of congruence rules. As this is type-level normalization, we equate under binders such as  $\lambda$  and  $\forall$ . The rule for congruence under  $\lambda$  bindings is below; the remaining congruence rules are omitted.

```
eq-λ : ∀ {τ v : Type (Δ „ κ1) κ2} → τ ≡t v → ‘λ τ ≡t ‘λ v
```

We have two "expansion" rules and one composition rule. Firstly, arrow-kinded types are  $\eta$ -expanded to have an outermost lambda binding. This later ensures canonicity of arrow-kinded types.

```
eq-η : ∀ {f : Type Δ (κ1 ‘→ κ2)} → f ≡t ‘λ (weakenk f · (‘Z))
```

Analogously, row-kinded variables left alone are expanded to a map by the identity function. Additionally, nested maps are composed together into one map. These rules together ensure canonical forms for row-kinded normal types. (Observe that these two rules are effectively functorial laws.)

```
eq-map-id : ∀ {κ} {τ : Type Δ R[ κ ]} → τ ≡t (‘λ {κ1 = κ} (‘Z)) <$> τ
eq-map-◦ : ∀ {κ3} {f : Type Δ (κ2 ‘→ κ3)} {g : Type Δ (κ1 ‘→ κ2)} {τ : Type Δ R[ κ1 ]} →
  (f <$> (g <$> τ)) ≡t (‘λ (weakenk f · (weakenk g · (‘Z)))) <$> τ
```

We now describe the computational rules that incur type reduction. Rule  $\text{eq-}\beta$  is the usual  $\beta$ -reduction rule. Rule  $\text{eq-labTy}$  asserts that the constructor  $\_ \triangleright \_$  is indeed superfluous when describing singleton rows with a label literal; singleton rows of the form  $(\ell \triangleright \tau)$  are normalized into row literals.

$\text{eq-}\beta : \forall \{\tau_1 : \text{Type } (\Delta \triangleright \kappa_1) \kappa_2\} \{\tau_2 : \text{Type } \Delta \kappa_1\} \rightarrow ((\lambda \tau_1) \cdot \tau_2) \equiv (\tau_1 \beta_k [\tau_2])$   
 $\text{eq-labTy} : l \equiv \text{lab } \ell \rightarrow (l \triangleright \tau) \equiv ([(\ell, \tau)])$

The rule  $\text{eq-}\$$  describes that mapping  $F$  over a singleton row is simply application of  $F$  over the row's contents. Rule  $\text{eq-map}$  asserts exactly the same except for row literals; the function  $\text{map}_2$  is simply  $\text{fmap}$  over a pair's right component. Rule  $\text{eq-}\langle \$ \rangle \setminus$  asserts that mapping  $F$  over a row complement is distributive.

$\text{eq-}\$ : \forall \{l : \text{Type } \Delta \kappa_1\} \{F : \text{Type } \Delta (\kappa_1 \xrightarrow{\cdot} \kappa_2)\} \rightarrow$   
 $(F \langle \$ \rangle (l \triangleright \tau)) \equiv (l \triangleright (F \cdot \tau))$   
 $\text{eq-map} : \forall \{F : \text{Type } \Delta (\kappa_1 \xrightarrow{\cdot} \kappa_2)\} \{\rho : \text{SimpleRow Type } \Delta R[\kappa_1]\} \{\text{op} : \text{True } (\text{ordered? } \rho)\} \rightarrow$   
 $\{\text{op}' : \text{True } (\text{ordered? } (\text{map } (\text{map}_2 (F \cdot \_)) \rho))\} \rightarrow$   
 $F \langle \$ \rangle ([\rho] \text{ op}) \equiv [\text{map } (\text{map}_2 (F \cdot \_)) \rho] \text{ op}'$   
 $\text{eq-}\langle \$ \rangle \setminus : \forall \{F : \text{Type } \Delta (\kappa_1 \xrightarrow{\cdot} \kappa_2)\} \{\rho_2 \rho_1 : \text{Type } \Delta R[\kappa_1]\} \rightarrow$   
 $F \langle \$ \rangle (\rho_2 \setminus \rho_1) \equiv (F \langle \$ \rangle \rho_2) \setminus (F \langle \$ \rangle \rho_1)$

The rules  $\text{eq-}\Pi$  and  $\text{eq-}\Sigma$  give the defining equations of  $\Pi$  and  $\Sigma$  at nested row kind. This is to say, application of  $\Pi$  to a nested row is equivalent to mapping  $\Pi$  over the row.

$\text{eq-}\Pi : \forall \{\rho : \text{Type } \Delta R[R[\kappa]]\} \{nl : \text{True } (\text{notLabel? } \kappa)\} \rightarrow$   
 $\Pi \{notLabel = nl\} \cdot \rho \equiv \Pi \{notLabel = nl\} \langle \$ \rangle \rho$   
 $\text{eq-}\Sigma : \forall \{\rho : \text{Type } \Delta R[R[\kappa]]\} \{nl : \text{True } (\text{notLabel? } \kappa)\} \rightarrow$   
 $\Sigma \{notLabel = nl\} \cdot \rho \equiv \Sigma \{notLabel = nl\} \langle \$ \rangle \rho$

The next two rules assert that  $\Pi$  and  $\Sigma$  can reassociate from left-to-right except with the new right-applicand "flapped".

$\text{eq-}\Pi\text{-assoc} : \forall \{\rho : \text{Type } \Delta (R[\kappa_1 \xrightarrow{\cdot} \kappa_2])\} \{\tau : \text{Type } \Delta \kappa_1\} \{nl : \text{True } (\text{notLabel? } \kappa_2)\} \rightarrow$   
 $(\Pi \{notLabel = nl\} \cdot \rho) \cdot \tau \equiv \Pi \{notLabel = nl\} \cdot (\rho ?? \tau)$   
 $\text{eq-}\Sigma\text{-assoc} : \forall \{\rho : \text{Type } \Delta (R[\kappa_1 \xrightarrow{\cdot} \kappa_2])\} \{\tau : \text{Type } \Delta \kappa_1\} \{nl : \text{True } (\text{notLabel? } \kappa_2)\} \rightarrow$   
 $(\Sigma \{notLabel = nl\} \cdot \rho) \cdot \tau \equiv \Sigma \{notLabel = nl\} \cdot (\rho ?? \tau)$

Finally, the rule  $\text{eq-comp1}$  gives computational content to the relative row complement operator applied to row literals. (We defined the syntactic complement  $\_ \setminus s \_$  precisely for this reason.)

$\text{eq-comp1} : \forall \{xs \ ys : \text{SimpleRow Type } \Delta R[\kappa]\}$   
 $\{\text{oxs} : \text{True } (\text{ordered? } xs)\} \{\text{oys} : \text{True } (\text{ordered? } ys)\} \{\text{ozs} : \text{True } (\text{ordered? } (xs \setminus s \ ys))\} \rightarrow$   
 $([\text{xs}] \text{ oxs}) \setminus ([\text{ys}] \text{ oys}) \equiv ([\text{xs } \setminus s \ \text{ys}]) \text{ ozs}$

Before concluding, we share an auxiliary definition that reflects instances of propositional equality in Agda to proofs of type-equivalence. The same role could be performed via Agda's `subst` but without the convenience.

$\text{inst} : \forall \{\tau_1 \ \tau_2 : \text{Type } \Delta \kappa\} \rightarrow \tau_1 \equiv \tau_2 \rightarrow \tau_1 \equiv \tau_2$   
 $\text{inst refl} = \text{eq-refl}$

2.3.1 *Some admissible rules.* Early versions of this equivalence relation imposed the following two rules directly; they intuit how we think  $\Pi$  and  $\Sigma$  ought to reduce as applicands. However, we can confirm their admissibility. The first rule states that  $\Pi$  is mapped over nested rows, and the second (definition omitted) states that  $\lambda$ -bindings  $\eta$ -expand over  $\Pi$ . (These results hold identically for  $\Sigma$ .)

$$\begin{aligned} \text{eq-}\Pi\triangleright &: \forall \{l\} \{ \tau : \text{Type } \Delta \text{ R}[\kappa] \} \{ nl : \text{True } (\text{notLabel? } \kappa) \} \rightarrow \\ & (\Pi \{ \text{notLabel} = nl \} \cdot (l \triangleright \tau)) \equiv t (l \triangleright (\Pi \{ \text{notLabel} = nl \} \cdot \tau)) \\ \text{eq-}\Pi\triangleright &= \text{eq-trans eq-}\Pi \text{ eq-}\triangleright\$ \\ \text{eq-}\Pi\lambda &: \forall \{l\} \{ \tau : \text{Type } (\Delta \text{ ,, } \kappa_1) \kappa_2 \} \{ nl : \text{True } (\text{notLabel? } \kappa_2) \} \rightarrow \\ & \Pi \{ \text{notLabel} = nl \} \cdot (l \triangleright ' \lambda \tau) \equiv ' \lambda (\Pi \{ \text{notLabel} = nl \} \cdot (\text{weaken}_\kappa l \triangleright \tau)) \end{aligned}$$

### 3 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. Figure 2 specifies the syntax and typing of normal types, given as reference. We describe the syntax in more depth by describing its intrinsic mechanization.

	Type variables $\alpha \in \mathcal{A}$	Labels $\ell \in \mathcal{L}$
Ground Kinds	$\gamma ::= \star \mid \mathsf{L}$	
Kinds	$\kappa ::= \gamma \mid \kappa \rightarrow \kappa \mid \mathsf{R}^\kappa$	
Row Literals	$\hat{\mathcal{P}} \ni \hat{\rho} ::= \{ \ell_i \triangleright \hat{\tau}_i \}_{i \in 0 \dots m}$	
Neutral Types	$n ::= \alpha \mid n \hat{\tau}$	
Normal Types	$\hat{\mathcal{T}} \ni \hat{\tau}, \hat{\phi} ::= n \mid \hat{\phi} \$ n \mid \hat{\rho} \mid \hat{\pi} \Rightarrow \hat{\tau} \mid \forall \alpha : \kappa. \hat{\tau} \mid \lambda \alpha : \kappa. \hat{\tau}$ $\mid n \triangleright \hat{\tau} \mid \ell \mid \# \hat{\tau} \mid \hat{\tau} \setminus \hat{\tau} \mid \Pi \hat{\tau} \mid \Sigma \hat{\tau}$	

Fig. 2. Normal type forms

#### 3.1 Mechanized syntax

We define `NormalTypes` and `NormalPreds` analogously to `Types` and `Preds`. Recall that `Pred` and `SimpleRow` are indexed by the type of their contents, so we can reuse some code.

```
data NormalType (Δ : KEnv) : Kind → Set
NormalPred : KEnv → Kind → Set
NormalPred = Pred NormalType
```

We must declare an analogous orderedness predicate, this time for normal types. Its definition is nearly identical.

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



Further, we define the predicate `NotSimpleRow`  $\rho$  to be true precisely when  $\rho$  is not a simple row. This is necessary because the row complement  $\rho_2 \setminus \rho_1$  should reduce when each  $\rho_i$  is a row literal. So it is necessary when forming normal row-complements to specify that at least one of the complement operands is a non-literal. The predicate `True` (`notSimpleRows?`  $\rho_1$   $\rho_2$ ) is satisfied precisely in this case.

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

Neutral types are type variables and applications with type variables in head position.

```
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
```

We define the normal type syntax firstly by restricting the promotion of neutral types to normal forms at only *ground* kind. As discussed above, we restrict the formation of inert row complements to just those in which at least one operand is non-literal. We define inert maps as part of the `NormalType` syntax rather than the `NeutralType` syntax. Observe that a consequence of this decision (as opposed to letting the form `_<$>_` be neutral) is that all inert maps must have the mapped function composed into just one applicand. For example, the type  $\phi_2 \langle \$ \rangle (\phi_1 \text{ n})$  must recompose into  $(\lambda \alpha. (\phi_2 (\phi_1 \alpha))) \langle \$ \rangle \text{ n}$  to be in normal form. Finally, we need only permit the formation of records and variants at kind  $\star$ , and we restrict the formation of neutral-labeled rows to just the singleton constructor `_▷n_`. The remaining cases are identical to the regular `Type` syntax and omitted.

```
data NormalType Δ where
  ne : (x : NeutralType Δ κ) → {ground : True (ground? κ)} → NormalType Δ κ
  _\_ : (ρ2 ρ1 : NormalType Δ R[ κ ]) → {nsr : True (notSimpleRows? ρ2 ρ1)} →
    NormalType Δ R[ κ ]
  _<$>_ : (φ : NormalType Δ (κ1 '→ κ2)) → NeutralType Δ R[ κ1 ] → NormalType Δ R[ κ2 ]
  Π : (ρ : NormalType Δ R[ ★ ]) → NormalType Δ ★
  Σ : (ρ : NormalType Δ R[ ★ ]) → NormalType Δ ★
  _▷n_ : (l : NeutralType Δ L) (τ : NormalType Δ κ) → NormalType Δ R[ κ ]
```

### 3.2 Canonicity 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.

<pre>noNeutrals : NeutralType ∅ κ → ⊥ noNeutrals (n · τ) = noNeutrals n</pre>	<pre>noComplements : ∀   {ρ<sub>1</sub> ρ<sub>2</sub> ρ<sub>3</sub> : NormalType ∅ R[ κ ]}   (nsr : True (notSimpleRows? ρ<sub>3</sub> ρ<sub>2</sub>)) →   ρ<sub>1</sub> ≡ (ρ<sub>3</sub> \ ρ<sub>2</sub>) {nsr} →   ⊥</pre>
---	--

Now, in any context an arrow-kinded type is canonically  $\lambda$ -bound:

$\text{arrow-canonicity} : (f : \text{NormalType } \Delta (\kappa_1 \xrightarrow{\text{'}} \kappa_2)) \rightarrow \exists [\tau] (f \equiv \text{' } \lambda \tau)$   
 $\text{arrow-canonicity } (\text{' } \lambda f) = f, \text{ refl}$

A row in an empty context is necessarily a row literal:

$\text{row-canonicity-}\emptyset : (\rho : \text{NormalType } \emptyset \text{ R } [\kappa]) \rightarrow$   
 $\quad \exists [xs] \Sigma [oxs \in \text{True } (\text{normalOrdered? } xxs)]$   
 $\quad (\rho \equiv \llbracket xs \rrbracket oxs)$   
 $\text{row-canonicity-}\emptyset (\llbracket \rho \rrbracket op) = \rho, op, \text{ refl}$

And a label-kinded type is necessarily a label literal:

$\text{label-canonicity-}\emptyset : \forall (l : \text{NormalType } \emptyset \text{ L}) \rightarrow \exists [s] (l \equiv \text{lab } s)$   
 $\text{label-canonicity-}\emptyset (\text{ne } x) = \perp\text{-elim } (\text{noNeutrals } x)$   
 $\text{label-canonicity-}\emptyset (\text{lab } s) = s, \text{ refl}$

### 3.3 Renaming

Renaming over normal types is defined in an entirely straightforward manner. Types and definitions are omitted.

### 3.4 Embedding

The goal is to normalize a given  $\tau : \text{Type } \Delta \kappa$  to a normal form at type  $\text{NormalType } \Delta \kappa$ . It is of course much easier to first describe the inverse embedding, which recasts a normal form back to its original type. Definitions are expected and omitted.

$\Uparrow : \text{NormalType } \Delta \kappa \rightarrow \text{Type } \Delta \kappa$   
 $\Uparrow \text{Row} : \text{SimpleRow NormalType } \Delta \text{ R } [\kappa] \rightarrow \text{SimpleRow Type } \Delta \text{ R } [\kappa]$   
 $\Uparrow \text{NE} : \text{NeutralType } \Delta \kappa \rightarrow \text{Type } \Delta \kappa$   
 $\Uparrow \text{Pred} : \text{NormalPred } \Delta \text{ R } [\kappa] \rightarrow \text{Pred Type } \Delta \text{ R } [\kappa]$

Note that it is precisely in "embedding" the `NormalOrdered` predicate that we establish half of the requisite isomorphism between a normal row being normal-ordered and its embedding being ordered. We will have to show the other half (that is, that ordered rows have normal-ordered evaluations) during normalization.

$\text{Ordered}\Uparrow : \forall (\rho : \text{SimpleRow NormalType } \Delta \text{ R } [\kappa]) \rightarrow \text{NormalOrdered } \rho \rightarrow$   
 $\quad \text{Ordered } (\Uparrow \text{Row } \rho)$

## 4 SEMANTIC TYPES

We have finally set the stage to discuss the process of normalizing types by evaluation. We first must define a semantic image of Types into which we will evaluate. Crucially, neutral types must *reflect* into this domain, and elements of this domain must *reify* to normal forms.

Let us first define the image of row literals to be `Fin`-indexed maps.

$\text{Row} : \text{Set} \rightarrow \text{Set}$   
 $\text{Row } A = \exists [n] (\text{Fin } n \rightarrow \text{Label } \times A)$

Naturally, we required a predicate on such rows to indicate that they are well-ordered.

```

491 OrderedRow' :  $\forall \{A : \text{Set}\} \rightarrow (n : \mathbb{N}) \rightarrow (\text{Fin } n \rightarrow \text{Label} \times A) \rightarrow \text{Set}$ 
492 OrderedRow' zero P =  $\top$ 
493 OrderedRow' (suc zero) P =  $\top$ 
494 OrderedRow' (suc (suc n)) P = (P fzero .fst < P (fsuc fzero) .fst)  $\times$  OrderedRow' (suc n) (P  $\circ$  fsuc)
495
496 OrderedRow :  $\forall \{A\} \rightarrow \text{Row } A \rightarrow \text{Set}$ 
497 OrderedRow (n , P) = OrderedRow' n P
498

```

We may now define the totality of forms a row-kinded type might take in the semantic domain (the `RowType` data type). We evaluate row literals into `Rows` via the row constructor; note that the argument  $\mathcal{T}$  maps kinding environments to types. In practice, this is how we specify that a row contains types in environment  $\Delta$ .

```

504 data RowType ( $\Delta : \text{KEnv}$ ) ( $\mathcal{T} : \text{KEnv} \rightarrow \text{Set}$ ) : Kind  $\rightarrow$  Set
505 NotRow :  $\forall \{\Delta : \text{KEnv}\} \{\mathcal{T} : \text{KEnv} \rightarrow \text{Set}\} \rightarrow \text{RowType } \Delta \mathcal{T} \text{R}[\kappa] \rightarrow \text{Set}$ 
506
507 data RowType  $\Delta \mathcal{T}$  where
508   row : ( $\rho : \text{Row } (\mathcal{T} \Delta)$ )  $\rightarrow$  RowType  $\Delta \mathcal{T} \text{R}[\kappa]$ 
509   _>_ : NeutralType  $\Delta \text{L} \rightarrow \mathcal{T} \Delta \rightarrow$  RowType  $\Delta \mathcal{T} \text{R}[\kappa]$ 
510   _\_ : ( $\rho_2 \rho_1 : \text{RowType } \Delta \mathcal{T} \text{R}[\kappa]$ )  $\rightarrow \{nr : \text{NotRow } \rho_2 \text{ or NotRow } \rho_1\} \rightarrow$ 
511       RowType  $\Delta \mathcal{T} \text{R}[\kappa]$ 
512   _<$>_ : ( $\phi : \forall \{\Delta'\} \rightarrow \text{Renaming}_k \Delta \Delta' \rightarrow \text{NeutralType } \Delta' \kappa_1 \rightarrow \mathcal{T} \Delta'$ )  $\rightarrow$ 
513       NeutralType  $\Delta \text{R}[\kappa_1]$   $\rightarrow$ 
514       RowType  $\Delta \mathcal{T} \text{R}[\kappa_2]$ 
515

```

Neutral-labeled singleton rows are evaluated into the `_>_` constructor; inert complements are evaluated into the `_\_` constructor. Just as `OrderedRow` is the semantic version of row well-orderedness, the predicate `NotRow` asserts that a given `RowType` is not a row literal (constructed by `row`). This ensures that complements constructed by `_\_` are indeed inert. Regarding the inert map constructor, we would like to compose nested maps. Borrowing from Allais et al. [2013], we thus interpret the left applicand of a map as a Kripke function space mapping neutral types in environment  $\Delta'$  to the type  $\mathcal{T} \Delta'$ , which we will later specify to be that of semantic types in environment  $\Delta'$  at kind  $\kappa$ . To avoid running afoul of Agda's positivity checker, we let the domain type of this Kripke function be *neutral types*, which may always be reflected into semantic types. We define semantic types (`SemType`) below, but replacing `NeutralType  $\Delta' \kappa_1$`  with `SemType  $\Delta' \kappa_1$`  would not be strictly positive.

We finally define the semantic domain by induction on the kind  $\kappa$ . Types with  $\star$  and label kind are simply `NormalTypes`. We interpret functions into *Kripke function spaces*—that is, functions that operate over `SemType` inputs at any possible environment  $\Delta_2$ , provided a renaming into  $\Delta_2$ . We interpret row-kinded types into the `RowType` type, defined above. Note some more trickery which we have borrowed from Allais et al. [2013]: we cannot pass `SemType` itself as an argument to `RowType` (which would violate termination checking), but we can instead pass to `RowType` the function  $(\lambda \Delta' \rightarrow \text{SemType } \Delta' \kappa)$ , which enforces a strictly smaller recursive call on the kind  $\kappa$ . Observe too that abstraction over the kinding environment  $\Delta'$  is necessary because our representation of inert maps `_<$>_` interprets the mapped applicand as a Kripke function space over neutral type

```

540 SemType : KEnv → Kind → Set
541 SemType Δ ★ = NormalType Δ ★
542 SemType Δ L = NormalType Δ L
543 SemType Δ1 (κ1 '→ κ2) = (∀ {Δ2} → (r : Renamingk Δ1 Δ2)
544                               (v : SemType Δ2 κ1) → SemType Δ2 κ2)
545 SemType Δ R[ κ ] = RowType Δ (λ Δ' → SemType Δ' κ) R[ κ ]
546

```

For abbreviation later, we alias our two types of Kripke function spaces as so:

```

550 KripkeFunction : KEnv → Kind → Kind → Set   KripkeFunctionNE : KEnv → Kind → Kind → Set
551 KripkeFunction Δ1 κ1 κ2 =                     KripkeFunctionNE Δ1 κ1 κ2 =
552   (∀ {Δ2} → Renamingk Δ1 Δ2 →              (∀ {Δ2} → Renamingk Δ1 Δ2 →
553     SemType Δ2 κ1 → SemType Δ2 κ2)           NeutralType Δ2 κ1 → SemType Δ2 κ2)
554

```

#### 4.1 Renaming

Renaming a Kripke function is nothing more than providing the appropriate renaming to the function.

```

558 renSem : Renamingk Δ1 Δ2 → SemType Δ1 κ → SemType Δ2 κ
559 renKripke : Renamingk Δ1 Δ2 → KripkeFunction Δ1 κ1 κ2 → KripkeFunction Δ2 κ1 κ2
560 renKripke {Δ1} ρ F {Δ2} = λ ρ' → F (ρ' ∘ ρ)
561

```

Renaming a row is simply pre-composition of the renaming  $r$  over the row's map  $P$ . The helper  $\text{map}_2$  lifts  $\text{renSem } r$  over the tuple, applying  $\text{renSem } r$  to the second component.

```

565 renRow : Renamingk Δ1 Δ2 → Row (SemType Δ1 κ) → Row (SemType Δ2 κ)
566 renRow r (n , P) = n , map2 (renSem r) ∘ P
567

```

Renaming over semantic types is otherwise defined in a straightforward manner. At kinds  $\star$  and  $L$ , we defer to the renaming of normal types. The other cases are described above or simply compositional. Some care must be given to ensure that the  $\text{NotRow}$  and well-ordered predicates are preserved. (We omit the auxiliary lemmas  $\text{orderedRenRow}$  and  $\text{nrRenSem}'$ .)

```

573 renSem {κ = ★} r τ = renkNF r τ
574 renSem {κ = L} r τ = renkNF r τ
575 renSem {κ = κ' '→ κ1} r F = renKripke r F
576 renSem {κ = R[ κ ]} r (φ <$> x) = (λ r' → φ (r' ∘ r)) <$> (renkNE r x)
577 renSem {κ = R[ κ ]} r (row (n , P) q) = row (renRow r (n , P)) (orderedRenRow r q)
578 renSem {κ = R[ κ ]} r (l ▷ τ) = (renkNE r l) ▷ renSem r τ
579 renSem {κ = R[ κ ]} r ((ρ2 \ ρ1) {nr}) = (renSem r ρ2 \ renSem r ρ1) {nr = nrRenSem' r ρ2 ρ1 nr}
580

```

## 5 NORMALIZATION BY EVALUATION (NBE)

We have now declared three domains: the syntax of types, the syntax of normal and neutral types, and the embedded domain of semantic types. Normalization by evaluation (NbE), as we follows it, involves producing a *reflection* from neutral types to semantic types, a *reification* from semantic types to normal types, and an *evaluation* from types to semantic types. It follows thereafter that normalization is the reification of evaluation. Because we reason about types modulo  $\eta$ -expansion,

reflection and reification are necessarily mutually recursive. (This is not the case however with e.g. Chapman et al. [2019].)

We describe the reflection logic before reification. Types at kind  $\star$  and  $\text{L}$  can be promoted straightforwardly with the `ne` constructor. A neutral row (e.g., a row variable) must be expanded into an inert mapping by  $(\lambda r\ n \rightarrow \text{reflect}\ n)$ , which is effectively the identity function. Finally, neutral types at arrow kind must be expanded into Kripke functions. Note that the input  $v$  has type  $\text{SemType}\ \Delta\ \kappa_1$  and must be reified.

`reflect` :  $\forall \{\kappa\} \rightarrow \text{NeutralType}\ \Delta\ \kappa \rightarrow \text{SemType}\ \Delta\ \kappa$

`reify` :  $\forall \{\kappa\} \rightarrow \text{SemType}\ \Delta\ \kappa \rightarrow \text{NormalType}\ \Delta\ \kappa$

`reflect`  $\{\kappa = \star\} \tau = \text{ne } \tau$

`reflect`  $\{\kappa = \text{L}\} \tau = \text{ne } \tau$

`reflect`  $\{\kappa = \text{R}[\ \kappa\ ]\} \rho = (\lambda r\ n \rightarrow \text{reflect}\ n) \langle \$ \rangle \rho$

`reflect`  $\{\kappa = \kappa_1 \xrightarrow{\text{'}} \kappa_2\} \tau = \lambda \rho\ v \rightarrow \text{reflect}\ (\text{ren}_\kappa \text{NE } \rho\ \tau \cdot \text{reify}\ v)$

Stopping here.

`reifyKripke` :  $\text{KripkeFunction}\ \Delta\ \kappa_1\ \kappa_2 \rightarrow \text{NormalType}\ \Delta\ (\kappa_1 \xrightarrow{\text{'}} \kappa_2)$

`reifyKripkeNE` :  $\text{KripkeFunctionNE}\ \Delta\ \kappa_1\ \kappa_2 \rightarrow \text{NormalType}\ \Delta\ (\kappa_1 \xrightarrow{\text{'}} \kappa_2)$

`reifyKripke`  $\{\kappa_1 = \kappa_1\} F = \lambda (\text{reify}\ (F\ S\ (\text{reflect}\ \{\kappa = \kappa_1\} ((\text{' } Z))))$

`reifyKripkeNE`  $F = \lambda (\text{reify}\ (F\ S\ (\text{' } Z)))$

`reifyRow'` :  $(n : \mathbb{N}) \rightarrow (\text{Fin } n \rightarrow \text{Label} \times \text{SemType}\ \Delta\ \kappa) \rightarrow \text{SimpleRow}\ \text{NormalType}\ \Delta\ \text{R}[\ \kappa\ ]$

`reifyRow'` `zero`  $P = []$

`reifyRow'`  $(\text{succ } n)\ P \text{ with } P\ \text{fzero}$

$\dots \mid (l, \tau) = (l, \text{reify } \tau) :: \text{reifyRow}'\ n\ (P \circ \text{fsucc})$

`reifyRow` :  $\text{Row}\ (\text{SemType}\ \Delta\ \kappa) \rightarrow \text{SimpleRow}\ \text{NormalType}\ \Delta\ \text{R}[\ \kappa\ ]$

`reifyRow`  $(n, P) = \text{reifyRow}'\ n\ P$

`reifyRowOrdered` :  $\forall (\rho : \text{Row}\ (\text{SemType}\ \Delta\ \kappa)) \rightarrow \text{OrderedRow}\ \rho \rightarrow \text{NormalOrdered}\ (\text{reifyRow}\ \rho)$

`reifyRowOrdered'` :  $\forall (n : \mathbb{N}) \rightarrow (P : \text{Fin } n \rightarrow \text{Label} \times \text{SemType}\ \Delta\ \kappa) \rightarrow$

$\text{OrderedRow}\ (n, P) \rightarrow \text{NormalOrdered}\ (\text{reifyRow}\ (n, P))$

`reifyRowOrdered'` `zero`  $P\ op = \text{tt}$

`reifyRowOrdered'`  $(\text{succ } \text{zero})\ P\ op = \text{tt}$

`reifyRowOrdered'`  $(\text{succ } (\text{succ } n))\ P\ (l_1 < l_2, ih) = l_1 < l_2, (\text{reifyRowOrdered}'\ (\text{succ } n)\ (P \circ \text{fsucc})\ ih)$

`reifyRowOrdered`  $(n, P)\ op = \text{reifyRowOrdered}'\ n\ P\ op$

`reifyPreservesNR` :  $\forall (\rho_1\ \rho_2 : \text{RowType}\ \Delta\ (\lambda\ \Delta' \rightarrow \text{SemType}\ \Delta'\ \kappa)\ \text{R}[\ \kappa\ ]) \rightarrow$

$(nr : \text{NotRow}\ \rho_1 \text{ or } \text{NotRow}\ \rho_2) \rightarrow \text{NotSimpleRow}\ (\text{reify}\ \rho_1) \text{ or } \text{NotSimpleRow}\ (\text{reify}\ \rho_2)$

`reifyPreservesNR'` :  $\forall (\rho_1\ \rho_2 : \text{RowType}\ \Delta\ (\lambda\ \Delta' \rightarrow \text{SemType}\ \Delta'\ \kappa)\ \text{R}[\ \kappa\ ]) \rightarrow$

$(nr : \text{NotRow}\ \rho_1 \text{ or } \text{NotRow}\ \rho_2) \rightarrow \text{NotSimpleRow}\ (\text{reify}\ ((\rho_1 \setminus \rho_2)\ \{nr\}))$

`reify`  $\{\kappa = \star\} \tau = \tau$

`reify`  $\{\kappa = \text{L}\} \tau = \tau$

`reify`  $\{\kappa = \kappa_1 \xrightarrow{\text{'}} \kappa_2\} F = \text{reifyKripke}\ F$

`reify`  $\{\kappa = \text{R}[\ \kappa\ ]\} (l \triangleright \tau) = (l \triangleright_n (\text{reify } \tau))$

```

638 reify {κ = R[ κ ]} (row ρ q) = (reifyRow ρ) (fromWitness (reifyRowOrdered ρ q))
639 reify {κ = R[ κ ]} ((φ <$> τ)) = (reifyKripkeNE φ <$> τ)
640 reify {κ = R[ κ ]} ((φ <$> τ) \ ρ₂) = (reify (φ <$> τ) \ reify ρ₂) {nsr = tt}
641 reify {κ = R[ κ ]} ((l ▷ τ) \ ρ) = (reify (l ▷ τ) \ (reify ρ)) {nsr = tt}
642 reify {κ = R[ κ ]} (row ρ x \ ρ'@(x₁ ▷ x₂)) = (reify (row ρ x) \ reify ρ') {nsr = tt}
643 reify {κ = R[ κ ]} ((row ρ x \ row ρ₁ x₁) {left ()})
644 reify {κ = R[ κ ]} ((row ρ x \ row ρ₁ x₁) {right ()})
645 reify {κ = R[ κ ]} (row ρ x \ (φ <$> τ)) = (reify (row ρ x) \ reify (φ <$> τ)) {nsr = tt}
646 reify {κ = R[ κ ]} ((row ρ x \ ρ'@((ρ₁ \ ρ₂) {nr'})) {nr}) = ((reify (row ρ x) \ (reify ((ρ₁ \ ρ₂) {nr'}))) {nsr = from
647 reify {κ = R[ κ ]} (((ρ₂ \ ρ₁) {nr'}) \ ρ) {nr}) = ((reify ((ρ₂ \ ρ₁) {nr'})) \ reify ρ) {fromWitness (reifyPreservesN
648
649
650 reifyPreservesNR (x₁ ▷ x₂) ρ₂ (left x) = left tt
651 reifyPreservesNR ((ρ₁ \ ρ₃) {nr}) ρ₂ (left x) = left (reifyPreservesNR' ρ₁ ρ₃ nr)
652 reifyPreservesNR (φ <$> ρ) ρ₂ (left x) = left tt
653 reifyPreservesNR ρ₁ (x ▷ x₁) (right y) = right tt
654 reifyPreservesNR ρ₁ ((ρ₂ \ ρ₃) {nr}) (right y) = right (reifyPreservesNR' ρ₂ ρ₃ nr)
655 reifyPreservesNR ρ₁ ((φ <$> ρ₂)) (right y) = right tt
656
657 reifyPreservesNR' (x₁ ▷ x₂) ρ₂ (left x) = tt
658 reifyPreservesNR' (ρ₁ \ ρ₃) ρ₂ (left x) = tt
659 reifyPreservesNR' (φ <$> n) ρ₂ (left x) = tt
660 reifyPreservesNR' (φ <$> n) ρ₂ (right y) = tt
661 reifyPreservesNR' (x ▷ x₁) ρ₂ (right y) = tt
662 reifyPreservesNR' (row ρ x) (x₁ ▷ x₂) (right y) = tt
663 reifyPreservesNR' (row ρ x) (ρ₂ \ ρ₃) (right y) = tt
664 reifyPreservesNR' (row ρ x) (φ <$> n) (right y) = tt
665 reifyPreservesNR' (ρ₁ \ ρ₃) ρ₂ (right y) = tt
666
667
668 - η normalization of neutral types
669
670 η-norm : NeutralType Δ κ → NormalType Δ κ
671 η-norm = reify ∘ reflect
672
673 - - Semantic environments
674
675 Env : KEnv → KEnv → Set
676 Env Δ₁ Δ₂ = ∀ {κ} → TVar Δ₁ κ → SemType Δ₂ κ
677
678 idEnv : Env Δ Δ
679 idEnv = reflect ∘ '
680
681 extende : (η : Env Δ₁ Δ₂) → (V : SemType Δ₂ κ) → Env (Δ₁ ,, κ) Δ₂
682 extende η V Z = V
683 extende η V (S x) = η x
684
685 lifte : Env Δ₁ Δ₂ → Env (Δ₁ ,, κ) (Δ₂ ,, κ)
686 lifte {Δ₁} {Δ₂} {κ} η = extende (weakenSem ∘ η) (idEnv Z)

```

## 5.1 Helping evaluation

### - Semantic application

$\_ \cdot \_ : \text{SemType } \Delta (\kappa_1 \xrightarrow{\epsilon} \kappa_2) \rightarrow \text{SemType } \Delta \kappa_1 \rightarrow \text{SemType } \Delta \kappa_2$   
 $F \cdot \_ V = F \text{ id } V$

### - Semantic complement

$\_ \in \text{Row} \_ : \forall \{m\} \rightarrow (l : \text{Label}) \rightarrow$   
 $(Q : \text{Fin } m \rightarrow \text{Label} \times \text{SemType } \Delta \kappa) \rightarrow$   
 $\text{Set}$   
 $\_ \in \text{Row} \_ \{m = m\} l Q = \Sigma [i \in \text{Fin } m] (l \equiv Q i .fst)$   
 $\_ \in \text{Row}? \_ : \forall \{m\} \rightarrow (l : \text{Label}) \rightarrow$   
 $(Q : \text{Fin } m \rightarrow \text{Label} \times \text{SemType } \Delta \kappa) \rightarrow$   
 $\text{Dec } (l \in \text{Row } Q)$   
 $\_ \in \text{Row}? \_ \{m = \text{zero}\} l Q = \text{no } \lambda \{ () \}$   
 $\_ \in \text{Row}? \_ \{m = \text{suc } m\} l Q \text{ with } l \stackrel{?}{=} Q \text{ fzero} .fst$   
 $\dots \mid \text{yes } p = \text{yes } (\text{fzero} , p)$   
 $\dots \mid \text{no } p \text{ with } l \in \text{Row}? (Q \circ \text{fsuc})$   
 $\dots \mid \text{yes } (n , q) = \text{yes } ((\text{fsuc } n) , q)$   
 $\dots \mid \text{no } q = \text{no } \lambda \{ (\text{fzero} , q') \rightarrow p \text{ } q' ; (\text{fsuc } n , q') \rightarrow q (n , q') \}$

$\text{compl} : \forall \{n\} \rightarrow$   
 $(P : \text{Fin } n \rightarrow \text{Label} \times \text{SemType } \Delta \kappa)$   
 $(Q : \text{Fin } m \rightarrow \text{Label} \times \text{SemType } \Delta \kappa) \rightarrow$   
 $\text{Row } (\text{SemType } \Delta \kappa)$   
 $\text{compl } \{n = \text{zero}\} \{m\} P Q = \epsilon V$   
 $\text{compl } \{n = \text{suc } n\} \{m\} P Q \text{ with } P \text{ fzero} .fst \in \text{Row}? Q$   
 $\dots \mid \text{yes } \_ = \text{compl } (P \circ \text{fsuc}) Q$   
 $\dots \mid \text{no } \_ = (P \text{ fzero}) :: (\text{compl } (P \circ \text{fsuc}) Q)$

### - - Semantic complement preserves well-ordering

$\text{lemma} : \forall \{n\} \{m\} \{q\} \rightarrow$   
 $(P : \text{Fin } (\text{suc } n) \rightarrow \text{Label} \times \text{SemType } \Delta \kappa)$   
 $(Q : \text{Fin } m \rightarrow \text{Label} \times \text{SemType } \Delta \kappa) \rightarrow$   
 $(R : \text{Fin } (\text{suc } q) \rightarrow \text{Label} \times \text{SemType } \Delta \kappa) \rightarrow$   
 $\text{OrderedRow } (\text{suc } n , P) \rightarrow$   
 $\text{compl } (P \circ \text{fsuc}) Q \equiv (\text{suc } q , R) \rightarrow$   
 $P \text{ fzero} .fst < R \text{ fzero} .fst$

$\text{lemma } \{n = \text{suc } n\} \{q = q\} P Q R \text{ oP eq}_1 \text{ with } P (\text{fsuc } \text{fzero}) .fst \in \text{Row}? Q$

$\text{lemma } \{\kappa = \_ \} \{\text{suc } n\} \{q = q\} P Q R \text{ oP refl} \mid \text{no } \_ = \text{oP} .fst$

$\dots \mid \text{yes } \_ = \text{<-trans } \{i = P \text{ fzero} .fst\} \{j = P (\text{fsuc } \text{fzero}) .fst\} \{k = R \text{ fzero} .fst\} (\text{oP} .fst) (\text{lemma } \{n = n\} (P \circ \text{fsuc}) Q)$

$\text{ordered-}:: : \forall \{n\} \{m\} \rightarrow$

```

736      (P : Fin (suc n) → Label × SemType Δ κ)
737      (Q : Fin m → Label × SemType Δ κ) →
738      OrderedRow (suc n , P) →
739      OrderedRow (compl (P ∘ fsuc) Q) → OrderedRow (P fzero :: compl (P ∘ fsuc) Q)
740 ordered-:: {n = n} P Q oP oC with compl (P ∘ fsuc) Q | inspect (compl (P ∘ fsuc)) Q
741 ... | zero , R | _ = tt
742 ... | suc n , R | [[ eq ]] = lemma P Q R oP eq , oC
743
744 ordered-compl : ∀ {n m} →
745   (P : Fin n → Label × SemType Δ κ)
746   (Q : Fin m → Label × SemType Δ κ) →
747   OrderedRow (n , P) → OrderedRow (m , Q) → OrderedRow (compl P Q)
748 ordered-compl {n = zero} P Q op1 op2 = tt
749 ordered-compl {n = suc n} P Q op1 op2 with P fzero .fst ∈Row? Q
750 ... | yes _ = ordered-compl (P ∘ fsuc) Q (ordered-cut op1) op2
751 ... | no _ = ordered-:: P Q op1 (ordered-compl (P ∘ fsuc) Q (ordered-cut op1) op2)
752
753 -----
754 - Semantic complement on Rows
755
756 _\v_ : Row (SemType Δ κ) → Row (SemType Δ κ) → Row (SemType Δ κ)
757 (n , P) \v (m , Q) = compl P Q
758
759 ordered\v : ∀ (ρ2 ρ1 : Row (SemType Δ κ)) → OrderedRow ρ2 → OrderedRow ρ1 → OrderedRow (ρ2 \v ρ1)
760 ordered\v (n , P) (m , Q) op2 op1 = ordered-compl P Q op2 op1
761
762 -----
763 - - - - Semantic lifting
764
765 _<$>V_ : SemType Δ (κ1 '→ κ2) → SemType Δ R[ κ1 ] → SemType Δ R[ κ2 ]
766 NotRow<$> : ∀ {F : SemType Δ (κ1 '→ κ2)} {ρ2 ρ1 : RowType Δ (λ Δ' → SemType Δ' κ1) R[ κ1 ]} →
767   NotRow ρ2 or NotRow ρ1 → NotRow (F <$>V ρ2) or NotRow (F <$>V ρ1)
768
769 F <$>V (l ▷ τ) = l ▷ (F ·V τ)
770 F <$>V row (n , P) q = row (n , map2 (F id) ∘ P) (orderedMap2 (F id) q)
771 F <$>V ((ρ2 \ ρ1) {nr}) = ((F <$>V ρ2) \ (F <$>V ρ1)) {NotRow<$> nr}
772 F <$>V (G <$> n) = (λ {Δ'} r → F r ∘ G r) <$> n
773
774 NotRow<$> {F = F} {x1 ▷ x2} {ρ1} (left x) = left tt
775 NotRow<$> {F = F} {ρ2 \ ρ3} {ρ1} (left x) = left tt
776 NotRow<$> {F = F} {φ <$> n} {ρ1} (left x) = left tt
777
778 NotRow<$> {F = F} {ρ2} {x ▷ x1} (right y) = right tt
779 NotRow<$> {F = F} {ρ2} {ρ1 \ ρ3} (right y) = right tt
780 NotRow<$> {F = F} {ρ2} {φ <$> n} (right y) = right tt
781
782 -----
783 - - - - Semantic complement on SemTypes
784

```



```

785  $\_ \backslash V\_ : \text{SemType } \Delta \text{ R}[\kappa] \rightarrow \text{SemType } \Delta \text{ R}[\kappa] \rightarrow \text{SemType } \Delta \text{ R}[\kappa]$ 
786  $\text{row } \rho_2 \text{ } \rho_2 \backslash V \text{ row } \rho_1 \text{ } \rho_1 = \text{row } (\rho_2 \backslash v \rho_1) (\text{ordered} \backslash v \rho_2 \rho_1 \text{ } \rho_2 \rho_1 \text{ } \rho_2 \rho_1)$ 
787  $\rho_2 @ (x \triangleright x_1) \backslash V \rho_1 = (\rho_2 \backslash \rho_1) \{nr = \text{left tt}\}$ 
788  $\rho_2 @ (\text{row } \rho \text{ } x) \backslash V \rho_1 @ (x_1 \triangleright x_2) = (\rho_2 \backslash \rho_1) \{nr = \text{right tt}\}$ 
789  $\rho_2 @ (\text{row } \rho \text{ } x) \backslash V \rho_1 @ (\_ \backslash \_) = (\rho_2 \backslash \rho_1) \{nr = \text{right tt}\}$ 
790  $\rho_2 @ (\text{row } \rho \text{ } x) \backslash V \rho_1 @ (\_ <\$> \_) = (\rho_2 \backslash \rho_1) \{nr = \text{right tt}\}$ 
791  $\rho @ (\rho_2 \backslash \rho_3) \backslash V \rho' = (\rho \backslash \rho') \{nr = \text{left tt}\}$ 
792  $\rho @ (\phi <\$> n) \backslash V \rho' = (\rho \backslash \rho') \{nr = \text{left tt}\}$ 
793
794  $\text{-- Semantic flap}$ 
795
796  $\text{apply} : \text{SemType } \Delta \kappa_1 \rightarrow \text{SemType } \Delta ((\kappa_1 \xrightarrow{\text{'}} \kappa_2) \xrightarrow{\text{'}} \kappa_2)$ 
797  $\text{apply } a = \lambda \rho \text{ } F \rightarrow F \cdot V (\text{renSem } \rho \text{ } a)$ 
798
799  $\text{infixr } 0 \text{ } \_<?>V\_$ 
800  $\_<?>V\_ : \text{SemType } \Delta \text{ R}[\kappa_1 \xrightarrow{\text{'}} \kappa_2] \rightarrow \text{SemType } \Delta \kappa_1 \rightarrow \text{SemType } \Delta \text{ R}[\kappa_2]$ 
801  $f <?>V a = \text{apply } a <\$>V f$ 
802

```

## 5.2 $\Pi$ and $\Sigma$ as operators

```

803
804  $\text{record Xi : Set where}$ 
805    $\text{field}$ 
806      $\Xi\star : \forall \{\Delta\} \rightarrow \text{NormalType } \Delta \text{ R}[\star] \rightarrow \text{NormalType } \Delta \star$ 
807      $\text{ren-}\star : \forall (\rho : \text{Renaming}_k \Delta_1 \Delta_2) \rightarrow (\tau : \text{NormalType } \Delta_1 \text{ R}[\star]) \rightarrow \text{ren}_k \text{NF } \rho (\Xi\star \tau) \equiv \Xi\star (\text{ren}_k \text{NF } \rho \tau)$ 
808
809  $\text{open Xi}$ 
810  $\xi : \forall \{\Delta\} \rightarrow \text{Xi} \rightarrow \text{SemType } \Delta \text{ R}[\kappa] \rightarrow \text{SemType } \Delta \kappa$ 
811  $\xi \{ \kappa = \star \} \Xi x = \Xi . \Xi\star (\text{reify } x)$ 
812  $\xi \{ \kappa = L \} \Xi x = \text{lab "impossible"}$ 
813  $\xi \{ \kappa = \kappa_1 \xrightarrow{\text{'}} \kappa_2 \} \Xi F = \lambda \rho \text{ } v \rightarrow \xi \Xi (\text{renSem } \rho \text{ } F <?>V v)$ 
814  $\xi \{ \kappa = \text{R}[\kappa] \} \Xi x = (\lambda \rho \text{ } v \rightarrow \xi \Xi v) <\$>V x$ 
815
816  $\Pi\text{-rec } \Sigma\text{-rec} : \text{Xi}$ 
817  $\Pi\text{-rec} = \text{record}$ 
818    $\{ \Xi\star = \Pi ; \text{ren-}\star = \lambda \rho \text{ } \tau \rightarrow \text{refl} \}$ 
819  $\Sigma\text{-rec} =$ 
820    $\text{record}$ 
821    $\{ \Xi\star = \Sigma ; \text{ren-}\star = \lambda \rho \text{ } \tau \rightarrow \text{refl} \}$ 
822
823  $\Pi V \Sigma V : \forall \{\Delta\} \rightarrow \text{SemType } \Delta \text{ R}[\kappa] \rightarrow \text{SemType } \Delta \kappa$ 
824  $\Pi V = \xi \Pi\text{-rec}$ 
825  $\Sigma V = \xi \Sigma\text{-rec}$ 
826
827  $\xi\text{-Kripke} : \text{Xi} \rightarrow \text{KripkeFunction } \Delta \text{ R}[\kappa] \kappa$ 
828  $\xi\text{-Kripke } \Xi \rho \text{ } v = \xi \Xi v$ 
829
830  $\Pi\text{-Kripke } \Sigma\text{-Kripke} : \text{KripkeFunction } \Delta \text{ R}[\kappa] \kappa$ 
831  $\Pi\text{-Kripke} = \xi\text{-Kripke } \Pi\text{-rec}$ 
832  $\Sigma\text{-Kripke} = \xi\text{-Kripke } \Sigma\text{-rec}$ 
833

```

### 5.3 Evaluation

```

834 eval : Type  $\Delta_1 \kappa \rightarrow \text{Env } \Delta_1 \Delta_2 \rightarrow \text{SemType } \Delta_2 \kappa$ 
835 evalPred : Pred Type  $\Delta_1 \text{R}[\kappa] \rightarrow \text{Env } \Delta_1 \Delta_2 \rightarrow \text{NormalPred } \Delta_2 \text{R}[\kappa]$ 
836
837 evalRow :  $(\rho : \text{SimpleRow Type } \Delta_1 \text{R}[\kappa]) \rightarrow \text{Env } \Delta_1 \Delta_2 \rightarrow \text{Row } (\text{SemType } \Delta_2 \kappa)$ 
838 evalRowOrdered :  $(\rho : \text{SimpleRow Type } \Delta_1 \text{R}[\kappa]) \rightarrow (\eta : \text{Env } \Delta_1 \Delta_2) \rightarrow \text{Ordered } \rho \rightarrow \text{OrderedRow } (\text{evalRow } \rho \eta)$ 
839
840 evalRow []  $\eta = \epsilon V$ 
841 evalRow  $((l, \tau) :: \rho) \eta = (l, (\text{eval } \tau \eta)) :: \text{evalRow } \rho \eta$ 
842
843  $\Downarrow \text{Row-isMap} : \forall (\eta : \text{Env } \Delta_1 \Delta_2) \rightarrow (xs : \text{SimpleRow Type } \Delta_1 \text{R}[\kappa]) \rightarrow$ 
844  $\text{reifyRow } (\text{evalRow } xs \eta) \equiv \text{map } (\lambda \{ (l, \tau) \rightarrow l, (\text{reify } (\text{eval } \tau \eta)) \}) xs$ 
845
846  $\Downarrow \text{Row-isMap } \eta [] = \text{refl}$ 
847  $\Downarrow \text{Row-isMap } \eta (x :: xs) = \text{cong}_2 \_::\_ \text{refl } (\Downarrow \text{Row-isMap } \eta xs)$ 
848
849 evalPred  $(\rho_1 \cdot \rho_2 \sim \rho_3) \eta = \text{reify } (\text{eval } \rho_1 \eta) \cdot \text{reify } (\text{eval } \rho_2 \eta) \sim \text{reify } (\text{eval } \rho_3 \eta)$ 
850 evalPred  $(\rho_1 \lesssim \rho_2) \eta = \text{reify } (\text{eval } \rho_1 \eta) \lesssim \text{reify } (\text{eval } \rho_2 \eta)$ 
851
852 eval  $\{\kappa = \kappa\} (\text{' } x) \eta = \eta x$ 
853 eval  $\{\kappa = \kappa\} (\tau_1 \cdot \tau_2) \eta = (\text{eval } \tau_1 \eta) \cdot V (\text{eval } \tau_2 \eta)$ 
854 eval  $\{\kappa = \kappa\} (\tau_1 \text{' } \rightarrow \tau_2) \eta = (\text{eval } \tau_1 \eta) \text{' } \rightarrow (\text{eval } \tau_2 \eta)$ 
855
856 eval  $\{\kappa = \star\} (\pi \Rightarrow \tau) \eta = \text{evalPred } \pi \eta \Rightarrow \text{eval } \tau \eta$ 
857 eval  $\{\Delta_1\} \{\kappa = \star\} (\forall \tau) \eta = \forall (\text{eval } \tau (\text{lifte } \eta))$ 
858 eval  $\{\kappa = \star\} (\mu \tau) \eta = \mu (\text{reify } (\text{eval } \tau \eta))$ 
859 eval  $\{\kappa = \star\} \lfloor \tau \rfloor \eta = \lfloor \text{reify } (\text{eval } \tau \eta) \rfloor$ 
860 eval  $(\rho_2 \setminus \rho_1) \eta = \text{eval } \rho_2 \eta \setminus V \text{eval } \rho_1 \eta$ 
861 eval  $\{\kappa = L\} (\text{lab } l) \eta = \text{lab } l$ 
862 eval  $\{\kappa = \kappa_1 \text{' } \rightarrow \kappa_2\} (\lambda \tau) \eta = \lambda \rho v \rightarrow \text{eval } \tau (\text{extende } (\lambda \{\kappa\} v' \rightarrow \text{renSem } \{\kappa = \kappa\} \rho (\eta v')) v)$ 
863 eval  $\{\kappa = \text{R}[\kappa] \text{' } \rightarrow \kappa\} \Pi \eta = \Pi\text{-Kripke}$ 
864 eval  $\{\kappa = \text{R}[\kappa] \text{' } \rightarrow \kappa\} \Sigma \eta = \Sigma\text{-Kripke}$ 
865 eval  $\{\kappa = \text{R}[\kappa]\} (f <\$> a) \eta = (\text{eval } f \eta) <\$> V (\text{eval } a \eta)$ 
866 eval  $(\ll \rho \gg \rho \rho) \eta = \text{row } (\text{evalRow } \rho \eta) (\text{evalRowOrdered } \rho \eta (\text{toWitness } \rho \rho))$ 
867 eval  $(l \triangleright \tau) \eta \text{ with eval } l \eta$ 
868 ... | ne  $x = (x \triangleright \text{eval } \tau \eta)$ 
869 ... | lab  $l_1 = \text{row } (1, \lambda \{ \text{fzero} \rightarrow (l_1, \text{eval } \tau \eta) \}) \text{tt}$ 
870 evalRowOrdered []  $\eta \rho \rho = \text{tt}$ 
871 evalRowOrdered  $(x_1 :: []) \eta \rho \rho = \text{tt}$ 
872 evalRowOrdered  $((l_1, \tau_1) :: (l_2, \tau_2) :: \rho) \eta (l_1 < l_2, \rho \rho) \text{ with}$ 
873  $\text{evalRow } \rho \eta \mid \text{evalRowOrdered } ((l_2, \tau_2) :: \rho) \eta \rho \rho$ 
874 ... | zero, P |  $ih = l_1 < l_2, \text{tt}$ 
875 ... | suc  $n, P \mid ih_1, ih_2 = l_1 < l_2, ih_1, ih_2$ 

```

### 5.4 Normalization

```

876
877  $\Downarrow : \forall \{\Delta\} \rightarrow \text{Type } \Delta \kappa \rightarrow \text{NormalType } \Delta \kappa$ 
878  $\Downarrow \tau = \text{reify } (\text{eval } \tau \text{idEnv})$ 
879
880  $\Downarrow \text{Pred} : \forall \{\Delta\} \rightarrow \text{Pred Type } \Delta \text{R}[\kappa] \rightarrow \text{Pred NormalType } \Delta \text{R}[\kappa]$ 

```

```

883  $\Downarrow \text{Pred } \pi = \text{evalPred } \pi \text{ idEnv}$ 
884  $\Downarrow \text{Row} : \forall \{ \Delta \} \rightarrow \text{SimpleRow Type } \Delta \text{ R} [ \kappa ] \rightarrow \text{SimpleRow NormalType } \Delta \text{ R} [ \kappa ]$ 
885  $\Downarrow \text{Row } \rho = \text{reifyRow } (\text{evalRow } \rho \text{ idEnv})$ 
886
887  $\Downarrow \text{NE} : \forall \{ \Delta \} \rightarrow \text{NeutralType } \Delta \kappa \rightarrow \text{NormalType } \Delta \kappa$ 
888  $\Downarrow \text{NE } \tau = \text{reify } (\text{eval } (\Downarrow \text{NE } \tau) \text{ idEnv})$ 
889

```

## 6 METATHEORY

### 6.1 Stability

```

893  $\text{stability} : \forall (\tau : \text{NormalType } \Delta \kappa) \rightarrow \Downarrow (\Downarrow \tau) \equiv \tau$ 
894  $\text{stabilityNE} : \forall (\tau : \text{NeutralType } \Delta \kappa) \rightarrow \text{eval } (\Downarrow \text{NE } \tau) (\text{idEnv } \{ \Delta \}) \equiv \text{reflect } \tau$ 
895  $\text{stabilityPred} : \forall (\pi : \text{NormalPred } \Delta \text{ R} [ \kappa ] ) \rightarrow \text{evalPred } (\Downarrow \text{Pred } \pi) \text{ idEnv } \equiv \pi$ 
896  $\text{stabilityRow} : \forall (\rho : \text{SimpleRow NormalType } \Delta \text{ R} [ \kappa ] ) \rightarrow \text{reifyRow } (\text{evalRow } (\Downarrow \text{Row } \rho) \text{ idEnv}) \equiv \rho$ 
897

```

Stability implies surjectivity and idempotency.

```

899  $\text{idempotency} : \forall (\tau : \text{Type } \Delta \kappa) \rightarrow (\Downarrow \circ \Downarrow \circ \Downarrow \circ \Downarrow) \tau \equiv (\Downarrow \circ \Downarrow) \tau$ 
900  $\text{idempotency } \tau \text{ rewrite stability } (\Downarrow \tau) = \text{refl}$ 
901
902  $\text{surjectivity} : \forall (\tau : \text{NormalType } \Delta \kappa) \rightarrow \exists [ v ] (\Downarrow v \equiv \tau)$ 
903  $\text{surjectivity } \tau = (\Downarrow \tau, \text{stability } \tau)$ 
904

```

Dual to surjectivity, stability also implies that embedding is injective.

```

906  $\Downarrow \text{-inj} : \forall (\tau_1 \tau_2 : \text{NormalType } \Delta \kappa) \rightarrow \Downarrow \tau_1 \equiv \Downarrow \tau_2 \rightarrow \tau_1 \equiv \tau_2$ 
907  $\Downarrow \text{-inj } \tau_1 \tau_2 \text{ eq} = \text{trans } (\text{sym } (\text{stability } \tau_1)) (\text{trans } (\text{cong } \Downarrow \text{ eq}) (\text{stability } \tau_2))$ 
908

```

### 6.2 A logical relation for completeness

```

910  $\text{subst-Row} : \forall \{ A : \text{Set} \} \{ n m : \mathbb{N} \} \rightarrow (n \equiv m) \rightarrow (f : \text{Fin } n \rightarrow A) \rightarrow \text{Fin } m \rightarrow A$ 
911  $\text{subst-Row refl } f = f$ 
912

```

- Completeness relation on semantic types

```

914  $\_ \approx \_ : \text{SemType } \Delta \kappa \rightarrow \text{SemType } \Delta \kappa \rightarrow \text{Set}$ 
915  $\_ \approx_2 \_ : \forall \{ A \} \rightarrow (x y : A \times \text{SemType } \Delta \kappa) \rightarrow \text{Set}$ 
916  $(l_1, \tau_1) \approx_2 (l_2, \tau_2) = l_1 \equiv l_2 \times \tau_1 \approx \tau_2$ 
917  $\_ \approx \text{R} \_ : (\rho_1 \rho_2 : \text{Row } (\text{SemType } \Delta \kappa)) \rightarrow \text{Set}$ 
918  $(n, P) \approx \text{R } (m, Q) = \Sigma [ pf \in (n \equiv m) ] (\forall (i : \text{Fin } m) \rightarrow (\text{subst-Row } pf P) i \approx_2 Q i)$ 
919

```

```

920  $\text{PointEqual} \approx : \forall \{ \Delta_1 \} \{ \kappa_1 \} \{ \kappa_2 \} (F G : \text{KripkeFunction } \Delta_1 \kappa_1 \kappa_2) \rightarrow \text{Set}$ 
921  $\text{PointEqualNE} \approx : \forall \{ \Delta_1 \} \{ \kappa_1 \} \{ \kappa_2 \} (F G : \text{KripkeFunctionNE } \Delta_1 \kappa_1 \kappa_2) \rightarrow \text{Set}$ 
922  $\text{Uniform} : \forall \{ \Delta \} \{ \kappa_1 \} \{ \kappa_2 \} \rightarrow \text{KripkeFunction } \Delta \kappa_1 \kappa_2 \rightarrow \text{Set}$ 
923  $\text{UniformNE} : \forall \{ \Delta \} \{ \kappa_1 \} \{ \kappa_2 \} \rightarrow \text{KripkeFunctionNE } \Delta \kappa_1 \kappa_2 \rightarrow \text{Set}$ 
924

```

```

925  $\text{convNE} : \kappa_1 \equiv \kappa_2 \rightarrow \text{NeutralType } \Delta \text{ R} [ \kappa_1 ] \rightarrow \text{NeutralType } \Delta \text{ R} [ \kappa_2 ]$ 
926  $\text{convNE refl } n = n$ 
927

```

```

927  $\text{convKripkeNE}_1 : \forall \{ \kappa_1' \} \rightarrow \kappa_1 \equiv \kappa_1' \rightarrow \text{KripkeFunctionNE } \Delta \kappa_1 \kappa_2 \rightarrow \text{KripkeFunctionNE } \Delta \kappa_1' \kappa_2$ 
928  $\text{convKripkeNE}_1 \text{ refl } f = f$ 
929

```

```

930  $\_ \approx \_ \{ \kappa = \star \} \tau_1 \tau_2 = \tau_1 \equiv \tau_2$ 
931

```

$\approx_{-} \{ \kappa = \mathbf{L} \} \tau_1 \tau_2 = \tau_1 \equiv \tau_2$   
 $\approx_{-} \{ \Delta_1 \} \{ \kappa = \kappa_1 \xrightarrow{\quad} \kappa_2 \} F G =$   
 $\text{Uniform } F \times \text{Uniform } G \times \text{PointEqual} \approx_{-} \{ \Delta_1 \} F G$   
 $\approx_{-} \{ \Delta_1 \} \{ \mathbf{R} [ \kappa_2 ] \} ( \_ \text{< \$ > } \_ \{ \kappa_1 \} \phi_1 n_1 ) ( \_ \text{< \$ > } \_ \{ \kappa_1 \} \phi_2 n_2 ) =$   
 $\Sigma [ pf \in ( \kappa_1 \equiv \kappa_1' ) ]$   
 $\text{UniformNE } \phi_1$   
 $\times \text{UniformNE } \phi_2$   
 $\times ( \text{PointEqualNE} \approx_{-} ( \text{convKripkeNE}_1 pf \phi_1 ) \phi_2$   
 $\times \text{convNE } pf n_1 \equiv n_2 )$   
 $\approx_{-} \{ \Delta_1 \} \{ \mathbf{R} [ \kappa_2 ] \} ( \phi_1 \text{< \$ >} n_1 ) \_ = \perp$   
 $\approx_{-} \{ \Delta_1 \} \{ \mathbf{R} [ \kappa_2 ] \} \_ ( \phi_1 \text{< \$ >} n_1 ) = \perp$   
 $\approx_{-} \{ \Delta_1 \} \{ \mathbf{R} [ \kappa ] \} ( l_1 \triangleright \tau_1 ) ( l_2 \triangleright \tau_2 ) = l_1 \equiv l_2 \times \tau_1 \approx \tau_2$   
 $\approx_{-} \{ \Delta_1 \} \{ \mathbf{R} [ \kappa ] \} ( x_1 \triangleright x_2 ) ( \text{row } \rho x_3 ) = \perp$   
 $\approx_{-} \{ \Delta_1 \} \{ \mathbf{R} [ \kappa ] \} ( x_1 \triangleright x_2 ) ( \rho_2 \setminus \rho_3 ) = \perp$   
 $\approx_{-} \{ \Delta_1 \} \{ \mathbf{R} [ \kappa ] \} ( \text{row } \rho x_1 ) ( x_2 \triangleright x_3 ) = \perp$   
 $\approx_{-} \{ \Delta_1 \} \{ \mathbf{R} [ \kappa ] \} ( \text{row } ( n , P ) x_1 ) ( \text{row } ( m , Q ) x_2 ) = ( n , P ) \approx \mathbf{R} ( m , Q )$   
 $\approx_{-} \{ \Delta_1 \} \{ \mathbf{R} [ \kappa ] \} ( \text{row } \rho x_1 ) ( \rho_2 \setminus \rho_3 ) = \perp$   
 $\approx_{-} \{ \Delta_1 \} \{ \mathbf{R} [ \kappa ] \} ( \rho_1 \setminus \rho_2 ) ( x_1 \triangleright x_2 ) = \perp$   
 $\approx_{-} \{ \Delta_1 \} \{ \mathbf{R} [ \kappa ] \} ( \rho_1 \setminus \rho_2 ) ( \text{row } \rho x_1 ) = \perp$   
 $\approx_{-} \{ \Delta_1 \} \{ \mathbf{R} [ \kappa ] \} ( \rho_1 \setminus \rho_2 ) ( \rho_3 \setminus \rho_4 ) = \rho_1 \approx \rho_3 \times \rho_2 \approx \rho_4$   
 $\text{PointEqual} \approx_{-} \{ \Delta_1 \} \{ \kappa_1 \} \{ \kappa_2 \} F G =$   
 $\forall \{ \Delta_2 \} ( \rho : \text{Renaming}_k \Delta_1 \Delta_2 ) \{ V_1 V_2 : \text{SemType } \Delta_2 \kappa_1 \} \rightarrow$   
 $V_1 \approx V_2 \rightarrow F \rho V_1 \approx G \rho V_2$   
 $\text{PointEqualNE} \approx_{-} \{ \Delta_1 \} \{ \kappa_1 \} \{ \kappa_2 \} F G =$   
 $\forall \{ \Delta_2 \} ( \rho : \text{Renaming}_k \Delta_1 \Delta_2 ) ( V : \text{NeutralType } \Delta_2 \kappa_1 ) \rightarrow$   
 $F \rho V \approx G \rho V$   
 $\text{Uniform } \{ \Delta_1 \} \{ \kappa_1 \} \{ \kappa_2 \} F =$   
 $\forall \{ \Delta_2 \Delta_3 \} ( \rho_1 : \text{Renaming}_k \Delta_1 \Delta_2 ) ( \rho_2 : \text{Renaming}_k \Delta_2 \Delta_3 ) ( V_1 V_2 : \text{SemType } \Delta_2 \kappa_1 ) \rightarrow$   
 $V_1 \approx V_2 \rightarrow ( \text{renSem } \rho_2 ( F \rho_1 V_1 ) ) \approx ( \text{renKripke } \rho_1 F \rho_2 ( \text{renSem } \rho_2 V_2 ) )$   
 $\text{UniformNE } \{ \Delta_1 \} \{ \kappa_1 \} \{ \kappa_2 \} F =$   
 $\forall \{ \Delta_2 \Delta_3 \} ( \rho_1 : \text{Renaming}_k \Delta_1 \Delta_2 ) ( \rho_2 : \text{Renaming}_k \Delta_2 \Delta_3 ) ( V : \text{NeutralType } \Delta_2 \kappa_1 ) \rightarrow$   
 $( \text{renSem } \rho_2 ( F \rho_1 V ) ) \approx F ( \rho_2 \circ \rho_1 ) ( \text{ren}_k \text{NE } \rho_2 V )$   
 $\text{Env} \approx_{-} : ( \eta_1 \eta_2 : \text{Env } \Delta_1 \Delta_2 ) \rightarrow \text{Set}$   
 $\text{Env} \approx_{-} \eta_1 \eta_2 = \forall \{ \kappa \} ( x : \text{TVar } \_ \kappa ) \rightarrow ( \eta_1 x ) \approx ( \eta_2 x )$   
 $- \text{ extension}$   
 $\text{extend} \approx_{-} : \forall \{ \eta_1 \eta_2 : \text{Env } \Delta_1 \Delta_2 \} \rightarrow \text{Env} \approx_{-} \eta_1 \eta_2 \rightarrow$   
 $\{ V_1 V_2 : \text{SemType } \Delta_2 \kappa \} \rightarrow$   
 $V_1 \approx V_2 \rightarrow$   
 $\text{Env} \approx_{-} ( \text{extende } \eta_1 V_1 ) ( \text{extende } \eta_2 V_2 )$   
 $\text{extend} \approx_{-} p q \mathbf{Z} = q$   
 $\text{extend} \approx_{-} p q ( \mathbf{S} v ) = p v$

### 6.2.1 Properties.

$\text{reflect-}\approx : \forall \{\tau_1 \tau_2 : \text{NeutralType } \Delta \kappa\} \rightarrow \tau_1 \equiv \tau_2 \rightarrow \text{reflect } \tau_1 \approx \text{reflect } \tau_2$   
 $\text{reify-}\approx : \forall \{V_1 V_2 : \text{SemType } \Delta \kappa\} \rightarrow V_1 \approx V_2 \rightarrow \text{reify } V_1 \equiv \text{reify } V_2$   
 $\text{reifyRow-}\approx : \forall \{n\} (P Q : \text{Fin } n \rightarrow \text{Label} \times \text{SemType } \Delta \kappa) \rightarrow$   
 $(\forall (i : \text{Fin } n) \rightarrow P i \approx_2 Q i) \rightarrow$   
 $\text{reifyRow } (n, P) \equiv \text{reifyRow } (n, Q)$

### 6.3 The fundamental theorem and completeness

$\text{fundC} : \forall \{\tau_1 \tau_2 : \text{Type } \Delta_1 \kappa\} \{\eta_1 \eta_2 : \text{Env } \Delta_1 \Delta_2\} \rightarrow$   
 $\text{Env-}\approx \eta_1 \eta_2 \rightarrow \tau_1 \equiv \tau_2 \rightarrow \text{eval } \tau_1 \eta_1 \approx \text{eval } \tau_2 \eta_2$   
 $\text{fundC-pred} : \forall \{\pi_1 \pi_2 : \text{Pred Type } \Delta_1 \text{R}[\kappa]\} \{\eta_1 \eta_2 : \text{Env } \Delta_1 \Delta_2\} \rightarrow$   
 $\text{Env-}\approx \eta_1 \eta_2 \rightarrow \pi_1 \equiv \pi_2 \rightarrow \text{evalPred } \pi_1 \eta_1 \equiv \text{evalPred } \pi_2 \eta_2$   
 $\text{fundC-Row} : \forall \{\rho_1 \rho_2 : \text{SimpleRow Type } \Delta_1 \text{R}[\kappa]\} \{\eta_1 \eta_2 : \text{Env } \Delta_1 \Delta_2\} \rightarrow$   
 $\text{Env-}\approx \eta_1 \eta_2 \rightarrow \rho_1 \equiv \rho_2 \rightarrow \text{evalRow } \rho_1 \eta_1 \approx \text{evalRow } \rho_2 \eta_2$   
 $\text{idEnv-}\approx : \forall \{\Delta\} \rightarrow \text{Env-}\approx (\text{idEnv } \{\Delta\}) (\text{idEnv } \{\Delta\})$   
 $\text{idEnv-}\approx x = \text{reflect-}\approx \text{refl}$   
 $\text{completeness} : \forall \{\tau_1 \tau_2 : \text{Type } \Delta \kappa\} \rightarrow \tau_1 \equiv \tau_2 \rightarrow \Downarrow \tau_1 \equiv \Downarrow \tau_2$   
 $\text{completeness } eq = \text{reify-}\approx (\text{fundC idEnv-}\approx eq)$   
 $\text{completeness-row} : \forall \{\rho_1 \rho_2 : \text{SimpleRow Type } \Delta \text{R}[\kappa]\} \rightarrow \rho_1 \equiv \rho_2 \rightarrow \Downarrow \text{Row } \rho_1 \equiv \Downarrow \text{Row } \rho_2$

### 6.4 A logical relation for soundness

$\text{infix } 0 \llbracket \_ \rrbracket \approx \_$   
 $\llbracket \_ \rrbracket \approx \_ : \forall \{\kappa\} \rightarrow \text{Type } \Delta \kappa \rightarrow \text{SemType } \Delta \kappa \rightarrow \text{Set}$   
 $\llbracket \_ \rrbracket \approx \text{ne\_} : \forall \{\kappa\} \rightarrow \text{Type } \Delta \kappa \rightarrow \text{NeutralType } \Delta \kappa \rightarrow \text{Set}$   
 $\llbracket \_ \rrbracket \text{r}\approx \_ : \forall \{\kappa\} \rightarrow \text{SimpleRow Type } \Delta \text{R}[\kappa] \rightarrow \text{Row } (\text{SemType } \Delta \kappa) \rightarrow \text{Set}$   
 $\llbracket \_ \rrbracket \approx_2 \_ : \forall \{\kappa\} \rightarrow \text{Label} \times \text{Type } \Delta \kappa \rightarrow \text{Label} \times \text{SemType } \Delta \kappa \rightarrow \text{Set}$   
 $\llbracket (l_1, \tau) \rrbracket \approx_2 (l_2, V) = (l_1 \equiv l_2) \times (\llbracket \tau \rrbracket \approx V)$   
 $\text{SoundKripke} : \text{Type } \Delta_1 (\kappa_1 \xrightarrow{\text{'}} \kappa_2) \rightarrow \text{KripkeFunction } \Delta_1 \kappa_1 \kappa_2 \rightarrow \text{Set}$   
 $\text{SoundKripkeNE} : \text{Type } \Delta_1 (\kappa_1 \xrightarrow{\text{'}} \kappa_2) \rightarrow \text{KripkeFunctionNE } \Delta_1 \kappa_1 \kappa_2 \rightarrow \text{Set}$   

- $\tau$  is equivalent to neutral ‘n’ if it’s equivalent
- to the  $\eta$  and map-id expansion of n

 $\llbracket \_ \rrbracket \approx \text{ne\_} \tau n = \tau \equiv \uparrow (\eta\text{-norm } n)$   
 $\llbracket \_ \rrbracket \approx \_ \{\kappa = \star\} \tau_1 \tau_2 = \tau_1 \equiv \uparrow \tau_2$   
 $\llbracket \_ \rrbracket \approx \_ \{\kappa = \text{L}\} \tau_1 \tau_2 = \tau_1 \equiv \uparrow \tau_2$   
 $\llbracket \_ \rrbracket \approx \_ \{\Delta_1\} \{\kappa = \kappa_1 \xrightarrow{\text{'}} \kappa_2\} f F = \text{SoundKripke } f F$   
 $\llbracket \_ \rrbracket \approx \_ \{\Delta\} \{\kappa = \text{R}[\kappa]\} \tau (\text{row } (n, P) \text{ op}) =$   
 $\text{let } xs = \uparrow \text{Row } (\text{reifyRow } (n, P)) \text{ in}$   
 $(\tau \equiv \llbracket xs \rrbracket (\text{fromWitness } (\text{Ordered} \uparrow (\text{reifyRow } (n, P)) (\text{reifyRowOrdered } n P \text{ op})))) \times$   
 $(\llbracket xs \rrbracket \text{r}\approx (n, P))$

1030  $\llbracket \_ \rrbracket \approx \_ \{ \Delta \} \{ \kappa = R[ \_ ] \} \tau (l \triangleright V) = (\tau \equiv (\uparrow \text{NE } l \triangleright \uparrow (\text{reify } V))) \times (\llbracket \uparrow (\text{reify } V) \rrbracket \approx V)$   
 1031  $\llbracket \_ \rrbracket \approx \_ \{ \Delta \} \{ \kappa = R[ \_ ] \} \tau ((\rho_2 \setminus \rho_1) \{nr\}) = (\tau \equiv (\uparrow (\text{reify } ((\rho_2 \setminus \rho_1) \{nr\})))) \times (\llbracket \uparrow (\text{reify } \rho_2) \rrbracket \approx \rho_2) \times (\llbracket \uparrow (\text{reify } \rho_1) \rrbracket \approx \rho_1)$   
 1032  $\llbracket \_ \rrbracket \approx \_ \{ \Delta \} \{ \kappa = R[ \_ ] \} \tau (\phi \text{ <\$> } n) =$   
 1033  $\exists [f] ((\tau \equiv (f \text{ <\$> } \uparrow \text{NE } n)) \times (\text{SoundKripkeNE } f \phi))$   
 1034  $\llbracket \_ \rrbracket r \approx (\text{zero}, P) = \top$   
 1035  $\llbracket \_ \rrbracket r \approx (\text{succ } n, P) = \perp$   
 1036  $\llbracket x :: \rho \rrbracket r \approx (\text{zero}, P) = \perp$   
 1037  $\llbracket x :: \rho \rrbracket r \approx (\text{succ } n, P) = (\llbracket x \rrbracket \approx_2 (P \text{ fzero})) \times (\llbracket \rho \rrbracket r \approx (n, P \circ \text{fsucc}))$   
 1038  
 1039 **SoundKripke**  $\{ \Delta_1 = \Delta_1 \} \{ \kappa_1 = \kappa_1 \} \{ \kappa_2 = \kappa_2 \} f F =$   
 1040  $\forall \{ \Delta_2 \} (\rho : \text{Renaming}_k \Delta_1 \Delta_2) \{ v V \} \rightarrow$   
 1041  $\llbracket v \rrbracket \approx V \rightarrow$   
 1042  $\llbracket (\text{ren}_k \rho f \cdot v) \rrbracket \approx (\text{renKripke } \rho F \cdot V V)$   
 1043  
 1044 **SoundKripkeNE**  $\{ \Delta_1 = \Delta_1 \} \{ \kappa_1 = \kappa_1 \} \{ \kappa_2 = \kappa_2 \} f F =$   
 1045  $\forall \{ \Delta_2 \} (r : \text{Renaming}_k \Delta_1 \Delta_2) \{ v V \} \rightarrow$   
 1046  $\llbracket v \rrbracket \approx_{\text{ne}} V \rightarrow$   
 1047  $\llbracket (\text{ren}_k r f \cdot v) \rrbracket \approx (F r V)$   
 1048

#### 6.4.1 Properties.

1050 **reflect**- $\llbracket \_ \rrbracket \approx \_ : \forall \{ \tau : \text{Type } \Delta \kappa \} \{ v : \text{NeutralType } \Delta \kappa \} \rightarrow$   
 1051  $\tau \equiv \uparrow \text{NE } v \rightarrow \llbracket \tau \rrbracket \approx (\text{reflect } v)$   
 1052 **reify**- $\llbracket \_ \rrbracket \approx \_ : \forall \{ \tau : \text{Type } \Delta \kappa \} \{ V : \text{SemType } \Delta \kappa \} \rightarrow$   
 1053  $\llbracket \tau \rrbracket \approx V \rightarrow \tau \equiv \uparrow (\text{reify } V)$   
 1054  $\eta\text{-norm}\text{-}\equiv : \forall (\tau : \text{NeutralType } \Delta \kappa) \rightarrow \uparrow (\eta\text{-norm } \tau) \equiv \uparrow \text{NE } \tau$   
 1055 **subst**- $\llbracket \_ \rrbracket \approx \_ : \forall \{ \tau_1 \tau_2 : \text{Type } \Delta \kappa \} \rightarrow$   
 1056  $\tau_1 \equiv \tau_2 \rightarrow \{ V : \text{SemType } \Delta \kappa \} \rightarrow \llbracket \tau_1 \rrbracket \approx V \rightarrow \llbracket \tau_2 \rrbracket \approx V$   
 1057  
 1058

#### 6.4.2 Logical environments.

1060  $\llbracket \_ \rrbracket \approx_{\text{e}} \_ : \forall \{ \Delta_1 \Delta_2 \} \rightarrow \text{Substitution}_k \Delta_1 \Delta_2 \rightarrow \text{Env } \Delta_1 \Delta_2 \rightarrow \text{Set}$   
 1061  $\llbracket \_ \rrbracket \approx_{\text{e}} \_ \{ \Delta_1 \} \sigma \eta = \forall \{ \kappa \} (\alpha : \text{TVar } \Delta_1 \kappa) \rightarrow \llbracket (\sigma \alpha) \rrbracket \approx (\eta \alpha)$   
 1062  
 1063 **– Identity relation**  
 1064 **idSR**  $: \forall \{ \Delta_1 \} \rightarrow \llbracket ' \rrbracket \approx_{\text{e}} (\text{idEnv } \{ \Delta_1 \})$   
 1065 **idSR**  $\alpha = \text{reflect}\text{-}\llbracket \_ \rrbracket \approx \text{eq}\text{-refl}$   
 1066

### 6.5 The fundamental theorem and soundness

1068 **fundS**  $: \forall \{ \Delta_1 \Delta_2 \kappa \} (\tau : \text{Type } \Delta_1 \kappa) \{ \sigma : \text{Substitution}_k \Delta_1 \Delta_2 \} \{ \eta : \text{Env } \Delta_1 \Delta_2 \} \rightarrow$   
 1069  $\llbracket \sigma \rrbracket \approx_{\text{e}} \eta \rightarrow \llbracket \text{sub}_k \sigma \tau \rrbracket \approx (\text{eval } \tau \eta)$   
 1070 **fundSRow**  $: \forall \{ \Delta_1 \Delta_2 \kappa \} (xs : \text{SimpleRow Type } \Delta_1 R[ \_ ] \kappa) \{ \sigma : \text{Substitution}_k \Delta_1 \Delta_2 \} \{ \eta : \text{Env } \Delta_1 \Delta_2 \} \rightarrow$   
 1071  $\llbracket \sigma \rrbracket \approx_{\text{e}} \eta \rightarrow \llbracket \text{subRow}_k \sigma xs \rrbracket r \approx (\text{evalRow } xs \eta)$   
 1072 **fundSPred**  $: \forall \{ \Delta_1 \kappa \} (\pi : \text{Pred Type } \Delta_1 R[ \_ ] \kappa) \{ \sigma : \text{Substitution}_k \Delta_1 \Delta_2 \} \{ \eta : \text{Env } \Delta_1 \Delta_2 \} \rightarrow$   
 1073  $\llbracket \sigma \rrbracket \approx_{\text{e}} \eta \rightarrow (\text{subPred}_k \sigma \pi) \equiv \uparrow \text{Pred } (\text{evalPred } \pi \eta)$   
 1074  
 1075

– Fundamental theorem when substitution is the identity

$\text{sub}_k\text{-id} : \forall (\tau : \text{Type } \Delta \kappa) \rightarrow \text{sub}_k \tau \equiv \tau$

$\vdash \llbracket \_ \rrbracket \approx : \forall (\tau : \text{Type } \Delta \kappa) \rightarrow \llbracket \tau \rrbracket \approx \text{eval } \tau \text{ idEnv}$

$\vdash \llbracket \tau \rrbracket \approx = \text{subst-}\llbracket \_ \rrbracket \approx (\text{inst } (\text{sub}_k\text{-id } \tau)) (\text{fundS } \tau \text{ idSR})$

---

– Soundness claim

$\text{soundness} : \forall \{\Delta_1 \kappa\} \rightarrow (\tau : \text{Type } \Delta_1 \kappa) \rightarrow \tau \equiv \text{t } \Downarrow (\Downarrow \tau)$

$\text{soundness } \tau = \text{reify-}\llbracket \_ \rrbracket \approx (\vdash \llbracket \tau \rrbracket \approx)$

---

– If  $\tau_1$  normalizes to  $\Downarrow \tau_2$  then the embedding of  $\tau_1$  is equivalent to  $\tau_2$

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

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

---

– Soundness implies the converse of completeness, as desired

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

$\text{Completeness}^{-1} \tau_1 \tau_2 \text{ eq} = \text{eq-trans } (\text{soundness } \tau_1) (\text{embed-}\equiv \text{t } \text{eq})$

## 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](https://doi.org/10.1007/978-3-030-33636-3_10).
- Alex Hubers and J. Garrett Morris. Generic programming with extensible data types: Or, making ad hoc extensible data types less ad hoc. *Proc. ACM Program. Lang.*, 7(ICFP):356–384, 2023. doi: 10.1145/3607843. URL <https://doi.org/10.1145/3607843>.
- Philip Wadler, Wen Kokke, and Jeremy G. Siek. *Programming Language Foundations in Agda*. August 2022. URL <https://plfa.inf.ed.ac.uk/20.08/>.