

# 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  $\beta$ - and  $\eta$ -equivalence—that is, to  $\beta\eta$ -long forms. Because the type system of  $R\omega\mu$  is a strict extension of System  $F\omega\mu$ , type level computation for arrow kinds is isomorphic to reduction of arrow types in the STLC. Novel to this report are the reductions of  $\Pi$ ,  $\Sigma$ , and 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  $\kappa$ . As shorthand, we write  $R^n[\kappa]$  to denote  $n$  repeated applications of  $R$  to the type  $\kappa$ —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  $\Delta$  and  $\kappa$  range over kinding environments and kinds, respectively. Correspondingly, we define *generalized variables* in Agda at these names.

```
private
variable
  Δ Δ1 Δ2 Δ3 : KEnv
  κ κ1 κ2 : Kind
```

The syntax of intrinsically well-scoped De-Bruijn type variables is given below. We say that the kind variable  $x$  is indexed by kinding environment  $\Delta$  and kind  $\kappa$  to specify that  $x$  has kind  $\kappa$  in kinding environment  $\Delta$ .

---

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

```

50 data KVar : KEnv → Kind → Set where
51   Z : KVar (Δ „ κ) κ
52   S : KVar Δ κ1 → KVar (Δ „ κ2) κ1
53
54
55

```

### 3 SYNTAX OF TYPES

$R\omega\mu$  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  $\rho_1$ ,  $\rho_2$ , and  $\rho_3$ . 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  $F\omega\mu$ , qualified type systems, and system  $R\omega$ .

```

61 data Pred (Δ : KEnv) : Kind → Set
62 data Type Δ : Kind → Set
63 data Type Δ where
64
65
66

```

```

65   ‘ :
66     (α : KVar Δ κ) →
67     Type Δ κ
68
69

```

```

69   ‘λ :
70     (τ : Type (Δ „ κ1) κ2) →
71     Type Δ (κ1 ‘→ κ2)
72
73

```

```

73   _·_ :
74     (τ1 : Type Δ (κ1 ‘→ κ2)) →
75     (τ2 : Type Δ κ1) →
76     Type Δ κ2
77
78

```

```

78   _‘→_ :
79     (τ1 : Type Δ ★) →
80     (τ2 : Type Δ ★) →
81     Type Δ ★
82
83

```

```

83   ‘∀ :
84     (τ : Type (Δ „ κ) ★) →
85     Type Δ ★
86
87

```

```

87   μ :
88     (F : Type Δ (★ ‘→ ★)) →
89     Type Δ ★
90
91

```

The first three constructors are analogous to the terms of the STLC. the constructor  $\cdot$  classifies term functions; the constructor  $\forall$  classifies type-in-term quantification; and the constructor  $\mu$  classifies recursive terms. Note that  $\mu$  could be further generalized to kind  $\kappa \rightarrow \star$ ; however, we found that kind  $\star \rightarrow \star$  was sufficient for our needs while simplifying both presentation and mechanization.

The syntax of qualified types is given below.

```

99   $\Rightarrow$  :
100  ( $\pi$  :  $\text{Pred } \Delta R[\kappa_1]$ )  $\rightarrow$  ( $\tau$  :  $\text{Type } \Delta \star$ )  $\rightarrow$ 
101   $\text{Type } \Delta \star$ 
102

```

The type  $\pi \Rightarrow \tau$  states that  $\tau$  is *qualified* by the predicate  $\pi$ —that is, the type variables bound in  $\tau$  are restricted in instantiation to just those that satisfy the predicate  $\pi$ . 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.

```

111 lab :
112  ( $l$  :  $\text{Label}$ )  $\rightarrow$ 
113   $\text{Type } \Delta L$ 
114
115  $\llbracket \_ \rrbracket$  :
116  ( $\tau$  :  $\text{Type } \Delta L$ )  $\rightarrow$ 
117   $\text{Type } \Delta \star$ 
118

```

Label constants in  $R\omega\mu$  are constructed from the type  $\text{Label}$ ; in our mechanization,  $\text{Label}$  is a type synonym for  $\text{String}$ , but one could choose any other candidate with decidable equality. Types at label kind  $L$  may be cast to *label singletons* by the  $\llbracket \_ \rrbracket$  constructor. This makes labels first-class entities: for example, as the type  $\llbracket \text{lab } "1" \rrbracket$  has kind  $\star$ , it can be inhabited by a term.

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

```

125  $\epsilon$  :
126   $\text{Type } \Delta R[\kappa]$ 
127
128  $\triangleright$  :
129  ( $l$  :  $\text{Type } \Delta L$ )  $\rightarrow$  ( $\tau$  :  $\text{Type } \Delta \kappa$ )  $\rightarrow$ 
130   $\text{Type } \Delta R[\kappa]$ 
131
132  $\<\$>$  :
133  ( $f$  :  $\text{Type } \Delta (\kappa_1 \rightarrow \kappa_2)$ )  $\rightarrow$  ( $\tau$  :  $\text{Type } \Delta R[\kappa_1]$ )  $\rightarrow$ 
134   $\text{Type } \Delta R[\kappa_2]$ 
135

```

Rows in  $R\omega\mu$  are either the empty row  $\epsilon$ , a labeled row ( $1 \triangleright \tau$ ), or a row mapping  $f \<\$> \tau$ . The row mapping  $f \<\$> (1 \triangleright \tau)$  describes the lifting of the function  $f$  over row ( $1 \triangleright \tau$ ), which we will define to equal  $(1 \triangleright f \tau)$  in the case where the right hand applicand is a labeled row. We will show that rows in Rome (that is, types at row kind) reduce to either the empty row  $\epsilon$  or a labeled row ( $1 \triangleright \tau$ ) 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  $f \<\$> \tau$  except when  $\tau$  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  $\Pi$  and  $\Sigma$  constructors.

```

148  $\Pi :$ 
149    $\text{Type } \Delta (R[\kappa] \multimap \kappa)$ 
150
151  $\Sigma :$ 
152    $\text{Type } \Delta (R[\kappa] \multimap \kappa)$ 

```

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

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

```

162 data Pred  $\Delta$  where
163    $\_ \cdot \_ \sim \_ :$ 
164      $(\rho_1 \rho_2 \rho_3 : \text{Type } \Delta R[\kappa]) \rightarrow$ 
165      $\text{Pred } \Delta R[\kappa]$ 
166
167    $\_ \lesssim \_ :$ 
168      $(\rho_1 \rho_2 : \text{Type } \Delta R[\kappa]) \rightarrow$ 
169      $\text{Pred } \Delta R[\kappa]$ 

```

Hubers and Morris [2023] implicitly define two type-level row lifting operators, *left mapping*  $\langle \$ \rangle$  and *right mapping*  $\langle ? \rangle$ , but the latter is superfluous. We appeal to the kinds of these operators for their intuition: left mapping  $f \langle \$ \rangle \rho$  lifts a function at arrow kind  $f : \kappa_1 \rightarrow \kappa_2$  into a function at kind  $R[\kappa_1] \rightarrow R[\kappa_2]$  and then applies it to  $\rho : R[\kappa_2]$ . We may define right mapping (named *flap* and written  $\langle ? \rangle$ , after similar Haskell operators) of row function  $f : R[\kappa_1 \rightarrow \kappa_2]$  over type  $\tau : \kappa_1$  using left mapping under the following identity:

$$f \langle ? \rangle \tau = (\lambda g. g \tau) \langle \$ \rangle f$$

which we encode in Agda as follows:

```

182 flap : Type  $\Delta (R[\kappa_1 \multimap \kappa_2] \multimap \kappa_1 \multimap R[\kappa_2])$ 
183 flap = 'λ ('λ (('λ (('Z) · ('(S Z))))  $\langle \$ \rangle$  ('(S Z))))
184
185  $\_ \langle ? \rangle \_ :$  Type  $\Delta (R[\kappa_1 \multimap \kappa_2]) \rightarrow \text{Type } \Delta \kappa_1 \rightarrow \text{Type } \Delta R[\kappa_2]$ 
186  $f \langle ? \rangle a = \text{flap } f \cdot a$ 

```

(We choose to define  $\_ \langle ? \rangle \_$  as the application of *flap* to inputs  $f$  and  $a$  so that we needn't pollute the definition with weakenings of its arguments.)

### 3.1 Type renaming

We closely follow Wadler et al. [2022] and Chapman et al. [2019] 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 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.

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

We will let the metavariable  $\rho$  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.

$\text{lift}_k : \text{Renaming}_k \Delta_1 \Delta_2 \rightarrow \text{Renaming}_k (\Delta_1 \text{ „ } \kappa) (\Delta_2 \text{ „ } \kappa)$   
 $\text{lift}_k \rho \text{ Z} = \text{Z}$   
 $\text{lift}_k \rho (S \ x) = S (\rho \ x)$

We define renaming as a function that translates a kinding derivation in kinding environment  $\Delta_1$  to environment  $\Delta_2$  provided a renaming from  $\Delta_1$  to  $\Delta_2$ . The definition proceeds by induction on the input kinding derivation. In the variable case, we use  $\rho$  to rename variable  $x$ . In the  $\lambda$  and  $\forall$  cases, we must lift the renaming  $\rho$  over the type variable introduced by these binders. The rest of the cases are effectively just congruence over the type structure.

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

$\text{ren}_k \rho (\text{' } x) = \text{' } (\rho \ x)$   
 $\text{ren}_k \rho (\text{' } \lambda \ \tau) = \text{' } \lambda (\text{ren}_k (\text{lift}_k \rho) \ \tau)$   
 $\text{ren}_k \rho (\pi \Rightarrow \tau) = \text{renPred}_k \rho \ \pi \Rightarrow \text{ren}_k \rho \ \tau$   
 $\text{ren}_k \rho (\text{' } \forall \ \tau) = \text{' } \forall (\text{ren}_k (\text{lift}_k \rho) \ \tau)$   
 $\text{ren}_k \rho \ \epsilon = \epsilon$   
 $\text{ren}_k \rho (\tau_1 \cdot \tau_2) = (\text{ren}_k \rho \ \tau_1) \cdot (\text{ren}_k \rho \ \tau_2)$   
 $\text{ren}_k \rho (\tau_1 \text{' } \rightarrow \tau_2) = (\text{ren}_k \rho \ \tau_1) \text{' } \rightarrow (\text{ren}_k \rho \ \tau_2)$   
 $\text{ren}_k \rho (\mu \ F) = \mu (\text{ren}_k \rho \ F)$   
 $\text{ren}_k \rho \ \Pi = \Pi$   
 $\text{ren}_k \rho \ \Sigma = \Sigma$   
 $\text{ren}_k \rho (\text{lab } x) = \text{lab } x$   
 $\text{ren}_k \rho (l \triangleright \tau) = \text{ren}_k \rho \ l \triangleright \text{ren}_k \rho \ \tau$   
 $\text{ren}_k \rho [\ell] = [\text{ren}_k \rho \ \ell]$   
 $\text{ren}_k \rho (f \text{ <$> } m) = \text{ren}_k \rho \ f \text{ <$> } \text{ren}_k \rho \ m$

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

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

Finally, weakening is a special case of renaming.

$\text{weaken}_k : \text{Type} \Delta \kappa_2 \rightarrow \text{Type} (\Delta \text{ „ } \kappa_1) \kappa_2$   
 $\text{weaken}_k = \text{ren}_k \ S$

### 3.2 Type substitution

We wish to give both a declarative and algorithmic treatment of type equivalence. For the latter, we will normalize types to normal forms, meaning types are equivalent iff their normal forms are definitionally equal. For the former, we must define  $\beta$ -substitution syntactically so that we can express  $\beta$ -equivalence of types declaratively. In our development,  $\beta$ -reduction is a special case of substitution.

We define a substitution as a function mapping type variables in context  $\Delta_1$  to types in context  $\Delta_2$ .

$\text{Substitution}_k : \text{KEnv} \rightarrow \text{KEnv} \rightarrow \text{Set}$

$\text{Substitution}_k \Delta_1 \Delta_2 = \forall \{\kappa\} \rightarrow \text{KVar } \Delta_1 \kappa \rightarrow \text{Type } \Delta_2 \kappa$

Substitutions must be lifted over binders, just as is done for renamings.

$\text{lifts}_k : \text{Substitution}_k \Delta_1 \Delta_2 \rightarrow \text{Substitution}_k (\Delta_1 \gg \kappa) (\Delta_2 \gg \kappa)$

$\text{lifts}_k \sigma \text{Z} = \text{'Z}$

$\text{lifts}_k \sigma (\text{S } x) = \text{weaken}_k (\sigma x)$

Substitution is defined inductively over types in a similar fashion to renaming. Note that this is *simultaneous* substitution and renaming—The variable case translates type variable  $x$  to the type  $\sigma x$ , for which the substitution  $\sigma$  also performs a renaming from environment  $\Delta_1$  to  $\Delta_2$ . The rest of the cases (as with renaming) are either congruences over the type structure or congruences plus lifting of the substitution. Again, substitution over predicates is defined mutually recursively.

$\text{sub}_k : \text{Substitution}_k \Delta_1 \Delta_2 \rightarrow \text{Type } \Delta_1 \kappa \rightarrow \text{Type } \Delta_2 \kappa$

$\text{subPred}_k : \text{Substitution}_k \Delta_1 \Delta_2 \rightarrow \text{Pred } \Delta_1 \kappa \rightarrow \text{Pred } \Delta_2 \kappa$

$\text{sub}_k \sigma \epsilon = \epsilon$

$\text{sub}_k \sigma (\text{' } x) = \sigma x$

$\text{sub}_k \sigma (\text{' } \lambda \tau) = \text{' } \lambda (\text{sub}_k (\text{lifts}_k \sigma) \tau)$

$\text{sub}_k \sigma (\tau_1 \cdot \tau_2) = (\text{sub}_k \sigma \tau_1) \cdot (\text{sub}_k \sigma \tau_2)$

$\text{sub}_k \sigma (\tau_1 \text{' } \rightarrow \tau_2) = (\text{sub}_k \sigma \tau_1) \text{' } \rightarrow (\text{sub}_k \sigma \tau_2)$

$\text{sub}_k \sigma (\pi \Rightarrow \tau) = \text{subPred}_k \sigma \pi \Rightarrow \text{sub}_k \sigma \tau$

$\text{sub}_k \sigma (\text{' } \forall \tau) = \text{' } \forall (\text{sub}_k (\text{lifts}_k \sigma) \tau)$

$\text{sub}_k \sigma (\mu F) = \mu (\text{sub}_k \sigma F)$

$\text{sub}_k \sigma (\Pi) = \Pi$

$\text{sub}_k \sigma \Sigma = \Sigma$

$\text{sub}_k \sigma (\text{lab } x) = \text{lab } x$

$\text{sub}_k \sigma (l \triangleright \tau) = \text{sub}_k \sigma l \triangleright \text{sub}_k \sigma \tau$

$\text{sub}_k \sigma [\ell] = [\text{sub}_k \sigma \ell]$

$\text{sub}_k \sigma (f \text{' } \<\$> a) = \text{sub}_k \sigma f \text{' } \<\$> \text{sub}_k \sigma a$

$\text{subPred}_k \sigma (\rho_1 \cdot \rho_2 \sim \rho_3) = \text{sub}_k \sigma \rho_1 \cdot \text{sub}_k \sigma \rho_2 \sim \text{sub}_k \sigma \rho_3$

$\text{subPred}_k \sigma (\rho_1 \lesssim \rho_2) = (\text{sub}_k \sigma \rho_1) \lesssim (\text{sub}_k \sigma \rho_2)$

We define the extension of a substitution  $\sigma$  by the type  $\tau$  functionally. If we had chosen to represent a  $\text{Substitution}_k$  as a list, extension would be done by the cons constructor. In a De-Brujin representation, the most recently appended variable is zero—hence an extension here maps the zero variable to  $\tau$  in the Z case and maps each variable (S  $x$ ) to its value in  $\sigma$  at predecessor  $x$ .

```

295 extendk : Substitutionk Δ1 Δ2 → (τ : Type Δ2 κ) → Substitutionk (Δ1 „ κ) Δ2
296 extendk σ τ Z = τ
297 extendk σ τ (S x) = σ x
298

```

Finally,  $\beta$ -substitution is simply a special case of substitution. Note that the constructor  $\cdot$  has type  $\text{KVar } \Delta \ \kappa \rightarrow \text{Type } \Delta \ \kappa$ , making it a substitution. It is in fact an identity substitution, which fixes the meaning of its type variables, hence it is the substitution we choose to extend when defining  $\beta$ -substitution.

```

303 _βk[_] : Type (Δ „ κ1) κ2 → Type Δ κ1 → Type Δ κ2
304 τ1 βk[ τ2 ] = subk ( extendk ‘ τ2 ) τ1
305

```

#### 4 TYPE EQUIVALENCE

We define type and predicate equivalence mutually recursively. You may think of type equivalence also as a sort of small-step relation on types, as we include rules to equate  $\beta$ -equivalent and  $\eta$ -equivalent types, as well as a number of computational steps a row kinded type may take.

```

311 infix 0 _≡t_
312 infix 0 _≡p_
313 data _≡p_ : Pred Δ R[ κ ] → Pred Δ R[ κ ] → Set
314 data _≡t_ : Type Δ κ → Type Δ κ → Set
315

```

Unless otherwise quantified, let the metavariable  $l$  range over types with label kind, let  $\pi$  range over predicates, and let  $\tau$  and  $v$  range over types:

```

319 private
320   variable
321     l l1 l2 l3 : Type Δ L
322     ρ1 ρ2 ρ3 : Type Δ R[ κ ]
323     π1 π2 : Pred Δ R[ κ ]
324     τ τ1 τ2 τ3 v v1 v2 v3 : Type Δ κ
325

```

The rules for predicate equivalence are uninteresting: two predicates are considered equivalent when their component types are equivalent.

```

328 data _≡p_ where
329   _eq-≤_ :
330     τ1 ≡t v1 → τ2 ≡t v2 →
331     τ1 ≤ τ2 ≡p v1 ≤ v2
332
333   _eq-·~_ :
334     τ1 ≡t v1 → τ2 ≡t v2 → τ3 ≡t v3 →
335     τ1 · τ2 ~ τ3 ≡p v1 · v2 ~ v3
336

```

The first three rules for type equivalence state that it is an equivalence relation.

```

337
338 data _≡t_ where
339   eq-refl :
340     τ ≡t τ
341

```

eq-sym :

$$\tau_1 \equiv \tau_2 \rightarrow$$

$$\tau_2 \equiv \tau_1$$

eq-trans :

$$\tau_1 \equiv \tau_2 \rightarrow \tau_2 \equiv \tau_3 \rightarrow$$

$$\tau_1 \equiv \tau_3$$

Type equivalence is congruent over the total structure of types, including  $\lambda$ -bindings (hence you may view type normalization as being *call-by-value*). We omit the other eight congruence rules.

eq- $\lambda$  :  $\forall \{\tau \ v : \text{Type } (\Delta \text{ „ } \kappa_1) \ \kappa_2\} \rightarrow$

$$\tau \equiv v \rightarrow$$

$$\lambda \tau \equiv \lambda v$$

We have one  $\eta$ -equivalence rule. It is henceforth useful to view the following rules as directed left-to-right, as normal forms are produced on the right-hand side.

eq- $\eta$  :  $\forall \{f : \text{Type } \Delta (\kappa_1 \xrightarrow{\epsilon} \kappa_2)\} \rightarrow$

$$f \equiv \lambda (\text{weaken}_k f \cdot (\lambda))$$

The rules that remain as *computational*—these are precisely the rules we would use to define small-step reduction of types. We begin with the  $\beta$ -equivalence rule, which states that lambda abstractions applied to arguments are equivalent to their beta reduction.

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

$$((\lambda \tau_1) \cdot \tau_2) \equiv (\tau_1 \beta_k [\tau_2])$$

The next two rules specify the computational behavior of mapping over rows. Rule (eq- $\epsilon$ ) states that mapping over the empty row  $\epsilon$  should yield the empty row; rule eq- $\rightarrow$  states that mapping over a labeled row should push the left applicand into the body of the row.

eq- $\epsilon$  :  $\{F : \text{Type } \Delta (\kappa_1 \xrightarrow{\epsilon} \kappa_2)\} \rightarrow$

$$(F \epsilon) \equiv \epsilon$$

eq- $\rightarrow$  :  $\forall \{l\} \{\tau : \text{Type } \Delta \ \kappa_1\} \{F : \text{Type } \Delta (\kappa_1 \xrightarrow{\epsilon} \kappa_2)\} \rightarrow$

$$(F \rightarrow (l \triangleright \tau)) \equiv (l \triangleright (F \cdot \tau))$$

We wish to establish that normal forms of types at row kind are either the empty row  $\epsilon$  or labeled rows. This is, of course, not the case for types in general. For example, the type  $\Pi \cdot (l \triangleright \tau)$  has row kind when  $\tau$  has row kind  $R[\ \kappa ]$ . In this case, rule eq- $\Pi$  pushes the  $\Pi$  over the label so that a canonical form is restored.

eq- $\Pi$  :  $\forall \{l\} \{\tau : \text{Type } \Delta \ R[\ \kappa ]\} \rightarrow$

$$\Pi \cdot (l \triangleright \tau) \equiv (l \triangleright (\Pi \cdot \tau))$$

The application of  $\Pi$  and  $\Sigma$  to a type  $\tau$  at nested-row kind is in fact just the mapping of  $\Pi$  and  $\Sigma$  over  $\tau$ :

eq- $\Pi$  :  $\forall \{\tau : \text{Type } \Delta \ R[\ R[\ \kappa ] ]\} \rightarrow$

$$\Pi \cdot \tau \equiv \Pi \epsilon \tau$$



Likewise to rows, we wish to show that normal forms of types at arrow kind are canonically  $\lambda$ -bound. However, the type  $\Pi \cdot (1 \triangleright \lambda \tau)$  has arrow kind! Rule  $\text{eq-}\Pi\lambda$  pushes the  $\lambda$  outwards in order to restore canonicity and so that application of  $\Pi \cdot (1 \triangleright \lambda \tau)$  to an applicand is simply  $\beta$ -reduction.

```

eq- $\Pi\lambda$  :  $\forall \{l\} \{\tau : \text{Type } (\Delta, \kappa_1) \kappa_2\} \rightarrow$ 
   $\Pi \cdot (l \triangleright \lambda \tau) \equiv \lambda (\Pi \cdot (\text{weaken}_\kappa l \triangleright \tau))$ 

```

Finally, in many cases (such as record concatenation and variant branching) it is necessary to reassociate the application  $(\Pi \rho) \tau$  inward so that  $\Pi$  (or  $\Sigma$ ) are the outermost syntax. We observe the following reassociation identity:

```

eq- $\Pi$ -assoc :  $\forall \{\rho : \text{Type } \Delta (R[\kappa_1 \xrightarrow{\quad} \kappa_2])\} \{\tau : \text{Type } \Delta \kappa_1\} \rightarrow$ 
   $(\Pi \cdot \rho) \cdot \tau \equiv \Pi \cdot (\rho \text{<?>} \tau)$ 

```

The definition of  $\equiv$  concludes by repeating the last four rules, replacing each  $\Pi$  with  $\Sigma$ . As a final aside, it might be thought that we could have rid ourselves of the syntax for mapping by elaborating types at kind  $R[\kappa_1 \rightarrow \kappa_2]$ . For example, the type  $(1 \triangleright \lambda x : \kappa_1. \tau)$  could perhaps have its  $\lambda$  binding pushed outside to yield  $\lambda x : \kappa_1. (1 \triangleright \tau)$ . However, this would not be kind-preserving (the latter has kind  $\kappa_1 \rightarrow R[\kappa_2]$ ), and therefore such a translation would induce a normalization that does not preserve kinds. We believe it would be possible but complicated to consider a kind-changing translation.

## 5 NORMAL TYPES

As is common in other *normalization by evaluation* approaches, we separate *neutral types* from *normal types*. These two definitions are defined mutually inductively with the data type for normal predicates:

```

data NormalType ( $\Delta : \text{KEnv}$ ) : Kind  $\rightarrow$  Set
data NormalPred ( $\Delta : \text{KEnv}$ ) : Kind  $\rightarrow$  Set
data NeutralType  $\Delta$  : Kind  $\rightarrow$  Set

```

A type is neutral if it is (respectively) (i) a variable, (ii) the application of a variable to an argument, or (iii) the mapping of a normal function type over a neutral row type. Intuitively, neutral forms are forms for which computation is "stuck" waiting on a variable to be substituted for a canonical form. Note that this third neutral form (row mapping) is novel to our development, and, in comparison to application, inverts the normal/neutral expectation of its arguments. It captures the stuck nature of a type such as  $(1 \triangleright \lambda x. M) \text{<\$>} \rho$ —that is, we are unable to map a function over a type variable.

```

data NeutralType  $\Delta$  where

```

```

  ' :

```

```

    ( $\alpha : \text{KVar } \Delta \kappa$ )  $\rightarrow$ 
    NeutralType  $\Delta \kappa$ 

```

```

  '·' :

```

```

    ( $f : \text{NeutralType } \Delta (\kappa_1 \xrightarrow{\quad} \kappa)$ )  $\rightarrow$ 
    ( $\tau : \text{NormalType } \Delta \kappa_1$ )  $\rightarrow$ 
    NeutralType  $\Delta \kappa$ 

```

```

  '·<\$>' :

```

```

(F : NormalType Δ (κ1 '→ κ2)) → (τ : NeutralType Δ R[ κ1 ]) →
NeutralType Δ (R[ κ2 ])

```

A predicate is normal if its component types are each normal.

```

data NormalPred Δ where
  _·~_ :
    (ρ1 ρ2 ρ3 : NormalType Δ R[ κ ]) →
    NormalPred Δ R[ κ ]
  _≤_ :
    (ρ1 ρ2 : NormalType Δ R[ κ ]) →
    NormalPred Δ R[ κ ]

```

Because we consider the normalization of types modulo  $\eta$ -equivalence, we wish to restrict our normal types to  $\eta$ -long form. This can be done by restricting the construction of normal-neutral types to just ground kind. This also ensures a canonical form for arrow-kinded normal types, as neutral types at arrow-kind cannot be promoted to normal types. We define a Ground predicate on types that maps all non-arrow kinds to the unit type  $\top$  and maps the arrow kind to  $\perp$ . (In other words, Ground  $\kappa$  is trivially inhabitable so long as  $\kappa \neq \kappa_1 \rightarrow \kappa_2$ .)

```

Ground : Kind → Set
Ground ★ = ⊤
Ground L = ⊤
Ground (κ '→ κ1) = ⊥
Ground R[ κ ] = ⊤

```

It is easy to show that this predicate is decidable.

```

ground? : ∀ κ → Dec (Ground κ)
ground? ★ = yes tt
ground? L = yes tt
ground? (κ '→ κ1) = no (λ ())
ground? R[ κ ] = yes tt

```

Now we may restrict the ne constructor to promoting just neutral types at ground kind by adding the (implicit) requirement that ne only be used when Ground  $\kappa$  is satisfied. To make this evidence easy to populate when  $\kappa$  is known, we employ a well-known proof-by-reflection trick (see [Wadler et al. \[2022\]](#)) and require evidence of the form True (ground?  $\kappa$ ).

```

data NormalType Δ where
  ne :
    (x : NeutralType Δ κ) → {ground : True (ground? κ)} →
    NormalType Δ κ

```

Likewise, to ensure canonical forms of rows, we restrict  $\Pi$  and  $\Sigma$  to formation at kind  $\star$  and L. The constructors for record types are given below.

```

491  $\Pi :$ 
492    $(\rho : \text{NormalType } \Delta \text{ R } [\star]) \rightarrow$ 
493    $\text{NormalType } \Delta \star$ 

```

```

494  $\Pi L :$ 
495    $(\rho : \text{NormalType } \Delta \text{ R } [L]) \rightarrow$ 
496    $\text{NormalType } \Delta L$ 

```

The rest of the `NormalType` syntax is identical to the `Type` syntax with the exception that we remove the ``` constructor for variables and  $\Pi$  and  $\Sigma$  constructors at arbitrary kind. We choose not to omit this syntax, as our proofs of canonicity follow from knowing the totality of `NormalType` constructors.

```

503 -  $F\omega$ 

```

```

504  $\lambda :$ 
505    $(\tau : \text{NormalType } (\Delta ,, \kappa_1) \kappa_2) \rightarrow$ 
506    $\text{NormalType } \Delta (\kappa_1 \rightarrow \kappa_2)$ 

```

```

507  $\_ \rightarrow \_ :$ 
508    $(\tau_1 \tau_2 : \text{NormalType } \Delta \star) \rightarrow$ 
509    $\text{NormalType } \Delta \star$ 

```

```

510  $\forall :$ 
511    $\{\kappa : \text{Kind}\} \rightarrow (\tau : \text{NormalType } (\Delta ,, \kappa) \star) \rightarrow$ 
512    $\text{NormalType } \Delta \star$ 

```

```

513  $\mu :$ 
514    $(F : \text{NormalType } \Delta (\star \rightarrow \star)) \rightarrow$ 
515    $\text{NormalType } \Delta \star$ 

```

```

516 - Qualified types

```

```

517  $\Rightarrow \_ :$ 
518    $(\pi : \text{NormalPred } \Delta \text{ R } [\kappa_1]) \rightarrow (\tau : \text{NormalType } \Delta \star) \rightarrow$ 
519    $\text{NormalType } \Delta \star$ 

```

```

520 -  $R\omega$ 

```

```

521  $\epsilon :$ 
522    $\text{NormalType } \Delta \text{ R } [\kappa]$ 

```

```

523  $\triangleright \_ :$ 
524    $(l : \text{NormalType } \Delta L) \rightarrow$ 
525    $(\tau : \text{NormalType } \Delta \kappa) \rightarrow$ 
526    $\text{NormalType } \Delta \text{ R } [\kappa]$ 

```

```

527  $\text{lab} :$ 
528    $(l : \text{Label}) \rightarrow$ 
529    $\text{NormalType } \Delta L$ 

```

```

530  $\lfloor \_ \rfloor :$ 
531    $(l : \text{NormalType } \Delta L) \rightarrow$ 

```

```

540   NormalType Δ ★
541 Σ :
542   (ρ : NormalType Δ R[ ★ ]) →
543   NormalType Δ ★
544
545 ΣL :
546   (ρ : NormalType Δ R[ L ]) →
547   NormalType Δ L
548

```

## 5.1 Renaming

We define renaming over `NormalTypes` in the same fashion as defined over `Types`. Note that we use the suffix  $_k\text{NF}$  now to denote functions which operate on `NormalType` syntax. Definitions are unsurprising and omitted.

```

554 renkNE      : Renamingk Δ1 Δ2 → NeutralType Δ1 κ → NeutralType Δ2 κ
555 renkNF      : Renamingk Δ1 Δ2 → NormalType Δ1 κ → NormalType Δ2 κ
556 weakenkNF : NormalType Δ κ2 → NormalType (Δ „ κ1) κ2
557 weakenkNE : NeutralType Δ κ2 → NeutralType (Δ „ κ1) κ2
558

```

## 5.2 Properties of normal types

We use Agda to confirm the desired canonicity properties. First, we wish for arrow kinds to be canonically formed by  $\lambda$ -abstractions. This can be shown easily by induction on arrow-kinded  $f$ .

```

563 arrow-canonicity : (f : NormalType Δ (κ1 '→ κ2)) → ∃[ τ ] (f ≡ 'λ τ)
564 arrow-canonicity ('λ f) = f , refl
565

```

Second, we wish for types at row kind to be canonically either (i) a labeled type ( $l \triangleright \tau$ ), (ii) a neutral type, or (iii) the empty row  $\epsilon$ . The `row-canonicity` lemma below states precisely this. Note that we permit row-kinded types to be neutral because we do not  $\eta$ -expand arrow-kinded rows. Recall our discussion above that such an expansion would not be kind-preserving. This means arrow-kinded rows must be permitted to be canonically neutral.

```

571 row-canonicity : (ρ : NormalType Δ R[ κ ]) →
572   ∃[ l ] (Σ[ τ ∈ NormalType Δ κ ] ((ρ ≡ (l ▷ τ)))) or
573   Σ[ τ ∈ NeutralType Δ R[ κ ] ] (ρ ≡ ne τ) or
574   ρ ≡ ε
575
576 row-canonicity (l ▷ τ) = left (l , τ , refl)
577 row-canonicity (ne τ) = right (left (τ , refl))
578 row-canonicity ε = right (right refl)
579

```

## 5.3 Type embeddings

We establish an embedding back from normal types to types below. The embedding is written  $\Uparrow$  because its type is converse to our definition of normalization, written  $\Downarrow$ . We will show in later sections precisely that  $\Uparrow$  is right-inverse to  $\Downarrow$ .

```

585 ↑↑ : NormalType Δ κ → Type Δ κ
586 ↑↑NE : NeutralType Δ κ → Type Δ κ
587 ↑↑Pred : NormalPred Δ R[ κ ] → Pred Δ R[ κ ]
588

```

Much of the embedding is defined by using like-for-like constructors and recursing on the subdata.

```

 $\Uparrow \mathsf{NE} \text{ ( } x \text{ )} = x$ 
 $\Uparrow \mathsf{NE} (\tau_1 \cdot \tau_2) = (\Uparrow \mathsf{NE} \tau_1) \cdot (\Uparrow \tau_2)$ 
 $\Uparrow \mathsf{NE} (F \text{ <\$> } \tau) = (\Uparrow F) \text{ <\$> } (\Uparrow \mathsf{NE} \tau)$ 
 $\Uparrow \mathsf{Pred} (\rho_1 \cdot \rho_2 \sim \rho_3) = (\Uparrow \rho_1) \cdot (\Uparrow \rho_2) \sim (\Uparrow \rho_3)$ 
 $\Uparrow \mathsf{Pred} (\rho_1 \lesssim \rho_2) = (\Uparrow \rho_1) \lesssim (\Uparrow \rho_2)$ 
 $\Uparrow \epsilon = \epsilon$ 
 $\Uparrow (\mathsf{ne} \ x) = \Uparrow \mathsf{NE} \ x$ 
 $\Uparrow (l \triangleright \tau) = (\Uparrow l) \triangleright (\Uparrow \tau)$ 
 $\Uparrow (\lambda \tau) = \lambda (\Uparrow \tau)$ 
 $\Uparrow (\tau_1 \text{ '}\rightarrow \tau_2) = \Uparrow \tau_1 \text{ '}\rightarrow \Uparrow \tau_2$ 
 $\Uparrow (\forall \tau) = \forall (\Uparrow \tau)$ 
 $\Uparrow (\mu \tau) = \mu (\Uparrow \tau)$ 
 $\Uparrow (\mathsf{lab} \ l) = \mathsf{lab} \ l$ 
 $\Uparrow \lfloor \tau \rfloor = \lfloor \Uparrow \tau \rfloor$ 
 $\Uparrow (\pi \Rightarrow \tau) = (\Uparrow \mathsf{Pred} \ \pi) \Rightarrow (\Uparrow \tau)$ 

```

An exception is made for record and variant constructors, which we must reconstruct as applications:

```

 $\Uparrow (\Pi \ x) = \Pi \cdot \Uparrow \ x$ 
 $\Uparrow (\Pi \mathsf{L} \ x) = \Pi \cdot \Uparrow \ x$ 
 $\Uparrow (\Sigma \ x) = \Sigma \cdot \Uparrow \ x$ 
 $\Uparrow (\Sigma \mathsf{L} \ x) = \Sigma \cdot \Uparrow \ x$ 

```

## 6 SEMANTIC TYPES

We next define  $\mathsf{SemType} \ \Delta \ \kappa$ , the semantic interpretation of types.  $\mathsf{SemTypes}$  are defined by induction on the kind  $\kappa$  and mutually-recursively with  $\mathsf{KripkeFunctions}$ , the interpretation of type functions.

```

 $\mathsf{SemType} : \mathsf{KEnv} \rightarrow \mathsf{Kind} \rightarrow \mathsf{Set}$ 
 $\mathsf{KripkeFunction} : \mathsf{KEnv} \rightarrow \mathsf{Kind} \rightarrow \mathsf{Kind} \rightarrow \mathsf{Set}$ 

```

Type functions are interpreted as Kripke function spaces because they must permit arbitrary and intermediate renaming. That is, they are functions at "any world."

```

 $\mathsf{KripkeFunction} \ \Delta_1 \ \kappa_1 \ \kappa_2 = (\forall \ \{\Delta_2\} \rightarrow \mathsf{Renaming}_k \ \Delta_1 \ \Delta_2 \rightarrow \mathsf{SemType} \ \Delta_2 \ \kappa_1 \rightarrow \mathsf{SemType} \ \Delta_2 \ \kappa_2)$ 
 $\mathsf{SemType} \ \Delta_1 \ (\kappa_1 \text{ '}\rightarrow \kappa_2) = \mathsf{KripkeFunction} \ \Delta_1 \ \kappa_1 \ \kappa_2$ 

```

We interpret  $\star$  and  $\mathsf{L}$  kinded types as their normal forms.

```

 $\mathsf{SemType} \ \Delta \ \star = \mathsf{NormalType} \ \Delta \ \star$ 
 $\mathsf{SemType} \ \Delta \ \mathsf{L} = \mathsf{NormalType} \ \Delta \ \mathsf{L}$ 

```

We interpret rows as either nothing (the empty row), just  $(\mathsf{left} \ x)$  for neutral  $x$ , or just  $(\mathsf{right} \ (l \ , \ \tau))$  for normal  $l$  and  $\tau$ . This corresponds directly with rows having these exact normal forms. (In general, we should expect a bijection between the types  $\mathsf{NormalType} \ \Delta \ \kappa$  and  $\mathsf{SemType} \ \Delta \ \kappa$ .)

```

SemType Δ R[ κ ] = Maybe
  ((NeutralType Δ R[ κ ]) or
   (NormalType Δ L × SemType Δ κ))

```

## 6.1 Renaming & substitution

Renaming is defined over semantic types in an obvious fashion. Definitions are again surprising and omitted. Substitution is unnecessary, as type functions are interpreted into an Agda function space.

```

renSem : Renamingk Δ1 Δ2 → SemType Δ1 κ → SemType Δ2 κ
weakenSem : SemType Δ κ1 → SemType (Δ „ κ2) κ1

```

## 6.2 Normalization by evaluation

Our *normalization by evaluation* proceeds in a standard fashion. We will define `reflect`, which maps neutral types to semantic types, and `reify`, which maps semantic types to normal types. We then write an evaluator that takes a Type into the semantic domain. During this process, function applications (and other forms of computation) are reduced. We finally reify the semantic type back to a normal form.

Reflection and reification are defined mutually recursively. We define the type synonym `reifyKripke` the reification of types at arrow kind, for repeated use later.

```

reflect : ∀ {κ} → NeutralType Δ κ → SemType Δ κ
reify : ∀ {κ} → SemType Δ κ → NormalType Δ κ
reifyKripke : KripkeFunction Δ κ1 κ2 → NormalType Δ (κ1 ↦ κ2)

```

Reflection of neutral types at ground kind leaves the type undisturbed.

```

reflect {κ = ★} τ      = ne τ
reflect {κ = L} τ      = ne τ
reflect {κ = R[ κ ]} τ = just (left τ)

```

Reflection of neutral types at arrow kind must be  $\eta$ -expanded into a Kripke function. Note here that is necessary to reify the input  $v$  back to a normal type.

```

reflect {κ = κ1 ↦ κ2} τ = λ ρ v → reflect (renkNE ρ τ · reify v)

```

Reification similarly leaves ground types undisturbed. Semantic types at  $\star$  and label kind are already in normal form; semantic types at row kind must be translated from their semantic constructors to their `NormalType` constructors. This process simply exhibits (half of) the bijection between the canonical forms of row kind normal types and the semantic domain at row kind.

```

reify {κ = ★} τ = τ
reify {κ = L} τ = τ
reify {κ = R[ κ ]} (just (left x)) = ne x
reify {κ = R[ κ ]} (just (right (l , τ))) = l ▷ (reify τ)
reify {κ = R[ κ ]} nothing = ε

```

Semantic functions must be reified from Agda functions back into `NormalType` syntax. This is done by reifying the application of semantic function `F` to the reflection of the  $\eta$ -expanded variable `Z`.

```
reify {κ = κ1 '→ κ2} F = reifyKripke F
reifyKripke {κ1 = κ1} F = 'λ (reify (F S (reflect {κ = κ1} (' Z))))
```

Observe that neutral types can be forced into  $\eta$ -long form simply by composing reification and reflection. This will prove helpful later, as the neutral type former `ne` has the same type except restricted to ground kind, but we will need to be able to promote from neutral to normal type at *all* kinds.

```
η-norm : NeutralType Δ κ → NormalType Δ κ
η-norm = reify ∘ reflect
```

Towards writing an evaluator, we define a semantic environment as function mapping kinding variables to semantic types.

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

Environment extension and lifting can be written in a straightforward manner.

```
extende : (η : Env Δ1 Δ2) → (V : SemType Δ2 κ) → Env (Δ1 „ κ) Δ2
lifte : Env Δ1 Δ2 → Env (Δ1 „ κ) (Δ2 „ κ)
```

The identity environment now maps type variables to semantic types. Unlike in [Chapman et al. \[2019\]](#), this environment can no longer be truly said to be an identity: type variables are de facto put into  $\eta$ -long form during reflection. However this change is mandatory for normalization, so we cannot define an environment that does not.

```
idEnv : Env Δ Δ
idEnv = reflect ∘ ‘
```

### 6.3 Helping evaluation

In aid of writing an evaluator, we found it helpful to develop *semantic* notions of the syntax introduced by  $R\omega\mu$ . We can define a row-kinded constructors easily:

```
_▷V_ : SemType Δ L → SemType Δ κ → SemType Δ R[ κ ]
_▷V_ {κ = κ} ℓ τ = just (right (ℓ , τ))

ne-R : NeutralType Δ R[ κ ] → SemType Δ R[ κ ]
ne-R = just ∘ left

εV : SemType Δ R[ κ ]
εV = nothing
```

Application of semantic types is simply Agda application within the identity renaming.

```
_·V_ : SemType Δ (κ1 '→ κ2) → SemType Δ κ1 → SemType Δ κ2
F ·V V = F id V
```

The definition of semantic row mapping varies by the shape of the row  $V$  over which we are lifting. If  $V$  is neutral, so too must the mapping of  $F$  over  $!V!$  be neutral. Hence we reify  $F$  to normal form and leave its mapping in neutral form. If  $V$  is a labeled row ( $l \triangleright \tau$ ), we push the application of  $F$  over  $\tau$ . Finally, if  $V$  is the empty row, its mapping is empty.

```

_<$>V_ : SemType  $\Delta (\kappa_1 \xrightarrow{\quad} \kappa_2) \rightarrow$  SemType  $\Delta R[\kappa_1] \rightarrow$  SemType  $\Delta R[\kappa_2]$ 
_<$>V_ { $\kappa_1 = \kappa_1$ } { $\kappa_2$ }  $F$  (just (left  $x$ )) = just (left (reifyKripke  $F$  <$>  $x$ ))
_<$>V_ { $\kappa_1 = \kappa_1$ } { $\kappa_2$ }  $F$  (just (right ( $l, \tau$ ))) = just (right ( $l, F \cdot V \tau$ ))
_<$>V_ { $\kappa_1 = \kappa_1$ } { $\kappa_2$ }  $F$  nothing = nothing

```

Although the flap operator  $_{<?>}$  is expressible as a special case of row mapping, we nevertheless find it a useful abstraction. It is defined below in terms of semantic row mapping; we find it likewise helpful to give a type synonym `apply` to the left hand side of this equation.

```

apply : SemType  $\Delta \kappa_1 \rightarrow$  SemType  $\Delta ((\kappa_1 \xrightarrow{\quad} \kappa_2) \xrightarrow{\quad} \kappa_2)$ 
apply  $a = \lambda \rho F \rightarrow F \cdot V$  (renSem  $\rho a$ )

infixr 0 _<?>V_
_<?>V_ : SemType  $\Delta R[\kappa_1 \xrightarrow{\quad} \kappa_2] \rightarrow$  SemType  $\Delta \kappa_1 \rightarrow$  SemType  $\Delta R[\kappa_2]$ 
 $f$  <?>V  $a =$  apply  $a$  <$>V  $f$ 

```

Much of the latent computation in  $R\omega\mu$  occurs under outermost  $\Pi$  and  $\Sigma$  syntax. To this end, we chose to represent  $\Pi$  and  $\Sigma$  as arrow-kinded type-constants—meaning they will evaluate into Agda functions. This provides an opportunity to concisely abstract their reduction logic. We define a semantic combinator for the  $\Pi$  type constant below. The first two equations state that record types at  $\star$  and label kind may be formed provided normal bodies; The third equation pushes the  $\lambda$ -binding of  $F$  outside of the record type; the fourth equation states that application *is* mapping at nested row kind.

```

 $\Pi V$  : SemType  $\Delta R[\kappa] \rightarrow$  SemType  $\Delta \kappa$ 
 $\Pi V$  { $\kappa = \star$ }  $x = \Pi$  (reify  $x$ )
 $\Pi V$  { $\kappa = L$ }  $x = \Pi L$  (reify  $x$ )
 $\Pi V$  { $\kappa = \kappa_1 \xrightarrow{\quad} \kappa_2$ }  $F = \lambda \rho v \rightarrow \Pi V$  (renSem  $\rho F$  <?>V  $v$ )
 $\Pi V$  { $\kappa = R[\kappa]$ }  $x = (\lambda \rho v \rightarrow \Pi V v)$  <$>V  $x$ 

```

We omit the definition of  $\Sigma V$ , as it is identical logic modulo appropriate renaming.



## 6.4 Evaluation

## 7 COMPLETENESS

### 7.1 Type Equivalence

### 7.2 A logical relation

### 7.3 The fundamental theorem & completeness

## 8 SOUNDNESS

### 8.1 A logical relation

### 8.2 The fundamental theorem & soundness

## 9 STABILITY

## 10 REMARK

### 10.1 Comparison to Chapman et al. [2019]

Our mechanization has closely resembled that of Chapman et al. [2019]. Our definition of semantic types, however, has differed, as our normalization is with respect to both  $\beta$ - and  $\eta$ -equivalence, whereas Chapman et al's is simply  $\beta$ -equivalence. Changing this definition simplifies some things and complicates others. The definition of semantic types is simpler: whereas Chapman et al permit function types to be interpreted as `NeutralTypes`, ours must be interpreted into solely Kripke function spaces. This complicates the definitions of `reify` and `reflect`, which must become mutually recursive, as we are unable to reflect neutral types at arrow kind to neutral types. We will show later that some of Chapman et al's metatheory relies on neutral forms to not be disturbed by normalization. This complicates the definition of term-level, normality-preserving substitution.

## REFERENCES

- 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/>.