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

Alex Hubers
Department of Computer Science
The University of Iowa
Iowa City, Iowa, USA
alexander-hubers@uiowa.edu

*Conference'17, Washington, DC, USA*
2025. ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

## Abstract

Hubers et al. [2024] introduce R$\omega\mu$, a higher-order row calculus, but do not describe any metatheory of its type equivalence relation nor of type reduction. R$\omega\mu$ extends System F$\omega\mu$ with rows, records, variants, row mapping, and a novel *row complement* operator. This paper shows not only that R$\omega\mu$ types enjoy normal forms, but formalizes the normalization-by-evaluation (NbE) of types in the interactive proof assistant Agda. We prove that our normalization algorithm is stable, sound and complete with respect to the type equivalence relation. Consequently, type conversion in R$\omega\mu$ is decidable.

## 1 Introduction

Hubers and Morris [2023] introduce an expressive higher-order row calculus called R$\omega$, which relies on implicit type reductions according to a directed type equivalence relation. Despite this reliance, the authors only provide a proof of *semantic soundness* that well-typed terms inhabit the denotations of well-kinded types. The authors do not characterize the shape of types in normal form, nor prove that the denoted types are sound and complete with respect to the equivalence relation. Hubers et al. [2024] extends the R$\omega$ language to R$\omega\mu$, which is R$\omega$ with recursive types, term-level recursion, and a novel *row complement* operator. The authors similarly extend the proof of semantic soundness, and fail to describe any metatheory of the equivalence relation.

### 1.1 The need for type normalization

R$\omega$ and R$\omega\mu$ each have a type conversion rule. The rule below states that the term $M$ can have its type converted from $\tau$ to $\upsilon$ provided a proof that $\tau$ and $\upsilon$ are equivalent. (For now, let us split environments into kinding environments $\Delta$, evidence environments $\Phi$, and typing environments $\Gamma$.)

$$(\textsc{t-conv})\ \frac{\Delta; \Phi; \Gamma \vdash M : \tau \quad \Delta \vdash \tau = \upsilon : \star}{\Delta; \Phi; \Gamma \vdash M : \upsilon}$$

Conversion rules can complicate metatheory. In particular, Hubers and Morris [2023]; Hubers et al. [2024] do not provide a procedure to decide type equivalence, and so proofs of type conversion must be embedded into the term language. This has a number of consequences:

1. Users of the surface language may be forced to write conversion rules by hand.
2. Decidability of type checking now rests upon the decidability of type conversion.
3. Term-level conversions can block $\beta$-reduction if a conversion is in the head position of an application.
4. Term-level conversions can block proofs of progress. Let `M` have type $\tau$, let `pf` be a proof that $\tau = \upsilon$, and consider the term `conv M pf`; ideally, one would expect this to reduce to `M` (we've changed nothing semantically about the term). But this breaks type preservation, as `conv M pf` (at type $\upsilon$) has stepped to a term at type $\tau$.
5. Inversion of the typing judgment $\Delta; \Phi; \Gamma \vdash M : \tau$—that is, induction over derivations—must consider the possibility that this derivation was constructed via conversion. But conversion from what type? Proofs by induction over derivations often thus get stuck.

All of these complications may be avoided provided a sound and complete normalization algorithm. In such a case, all types are reduced to normal forms, where syntactic comparison is enough to decide equivalence. In effect, the proofs of all conversions have collapsed to just the reflexive case, and so term-level conversions can safely be removed.

### 1.2 Contributions

This paper offers the following as contributions:

1. A normalization procedure for the directed R$\omega$ and R$\omega\mu$ type equivalence relation;
2. the semantics of a novel *row complement* operator;
3. proofs of soundness and completeness of normalization with respect to type equivalence; and
4. a complete mechanization in Agda of R$\omega\mu$ and the claimed metatheoretic results.

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

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

**Figure 1.** Syntax

## 2 The R$\omega\mu$ calculus

Figure 1 describes the syntax of kinds, predicates, and types in R$\omega\mu$.

Labels (i.e., record field and variant constructor names) live at the type level, and are classified by kind $\mathsf{L}$. Rows of kind $\kappa$ are classified by $\mathsf{R}[\kappa]$. When possible, we use $\phi$ for type functions, $\rho$ for row types, and $\xi$ for label types. Singleton types $\#\tau$ are used to cast label-kinded types to types at kind $\star$. $\phi \, \$ \, \rho$ maps the type operator $\phi$ across a row $\rho$. In practice, we often leave the map operator implicit, using kind information to infer the presence of maps. We define a families of $\Pi$ and $\Sigma$ constructors, describing record and variants at various kinds; in practice, we can determine the kind annotation from context. $\mu$ builds isorecursive types. Row literals (or, synonymously, *simple rows*) are sequences of labeled types $\xi_i \triangleright \tau_i$. We write $0 \ldots m$ to denote the set of naturals up to (but not including) $m$. We will frequently use $\varepsilon$ to denote the empty row.

The type $\pi \Rightarrow \tau$ denotes a qualified type. In essence, the predicate $\pi$ restricts the instantiation of the type variables in $\tau$. Our predicates capture relationships among rows: $\rho_1 \lesssim \rho_2$ means that $\rho_1$ is *contained* in $\rho_2$, and $\rho_1 \odot \rho_2 \sim \rho_3$ means that $\rho_1$ and $\rho_2$ can be *combined* to give $\rho_3$.

Finally, R$\omega\mu$ introduces a novel *row complement* operator $\rho_2 \setminus \rho_1$, analogous to a set complement for rows. The complement $\rho_2 \setminus \rho_1$ intuitively means the row obtained by removing any label-type associations in $\rho_1$ from $\rho_2$. In practice, the type $\rho_2 \setminus \rho_1$ is meaningful only when we know that $\rho_1 \lesssim \rho_2$, however constraining the formation of row complements to just this case introduces an unpleasant dependency between predicate evidence and type well-formedness. In practice, it is easy enough to totally define the complement operator on all rows, even without the containment of one by the other.

### 2.1 Type computation in R$\omega\mu$

R$\omega$ and R$\omega\mu$ are quite expressive languages, with succinct and readable types. To some extent, this magic relies on implicit type application, implicit maps, and unresolved type reduction. Let us demonstrate with a few examples.

#### 2.1.1 Reifying variants, reflecting records.

The following R$\omega$ terms witness the duality of records and variants.

```
reify : ∀ z : R[ ★ ], t : ★.
          (Σ z → t) → Π (z → t)
reflect : ∀ z : R[ ★ ], t : ★.
            Π (z → t) → Σ z → t
```

The term `reify` transforms a variant eliminator into a record of individual eliminators; the term `reflect` transforms a record of individual eliminators into a variant eliminator. The syntax above is precise, but arguably so because it hides some latent computation. In particular, what does `z → t` mean? The variable `z` is at kind `R[ ★ ]` and `t` at kind $\star$, so this is an implicit map. Rewriting explicitly yields:

```
  reify : ∀ z : R[ ★ ], t : ★.
            (Σ z → t) → Π ((λs. s → t) $ z)
reflect : ∀ z : R[ ★ ], t : ★.
            Π ((λs. s → t) $ z) → Σ z → t
```

The writing of the former rather than the latter is permitted because the corresponding types are convertible.

#### 2.1.2 Deriving functorality.

We can simulate the deriving of functor typeclass instances: given a record of `fmap` instances at type `Π (Functor z)`, we can give a `Functor` instance for `Σ z`.

```
type Functor : (★ → ★) → ★
type Functor = λf. ∀ a b. (a → b) → f a → f b
fmapS : ∀ z : R[★ → ★].
          Π (Functor z) → Functor (Σ z)
```

When we consider the kind of `Functor z` it becomes apparent that this is another implicit map. Let us write it explicitly and also expand the `Functor` type synonym:

```
fmapS : ∀ z : R[★ → ★].
        Π ((λf. ∀ a b.
            (a → b) → f a → f b) $ z) →
        (λf. ∀ a b. (a → b) → f a → f b) (Σ z)
```

which reduces further to:

```
fmapS : ∀ z : R[★ → ★].
          Π ((λf. ∀ a b.
                (a → b) → f a → f b) $ z) →
          ∀ a b. (a → b) → (Σ z) a → (Σ z) b
```

Intuitively, we suspect that `(Σ z) a` means "the variant of type constructors `z` applied to the type variable `a`". Let us make this intent obvious. First, define a "left-mapping" helper `_??_` with kind `R[ ★ → ★ ] → ★ → R[ ★ ]` as so:

```
r ?? t = (λ f. f t) $ r
```

Now the type of `fmapS` is:

```
fmapS : ∀ z : R[★ → ★].
          Π ((λf. ∀ a b.
                (a → b) → f a → f b) $ z) →
          ∀ a b. (a → b) → Σ (z ?? a) → Σ (z ?? b)
```

And we have what appears to be a normal form. Of course, the type is more interesting when applied to a real value for `z`. Suppose `z` is a functor for naturals, `{'Z ▹ const Unit, 'S ▹ λx. x}`. Then a first pass yields:

```
fmapS {'Z ▹ const Unit, 'S ▹ λx. x} :
        Π ((λf. ∀ a b. (a → b) → f a → f b)
          $ {'Z ▹ const Unit, 'S ▹ λx. x}) →
        ∀ a b. (a → b) →
        Σ ({'Z ▹ const Unit, 'S ▹ λx. x} ?? a) →
        Σ ({'Z ▹ const Unit, 'S ▹ λx. x} ?? b)
```

How do we reduce from here? Regarding the first input, we suspect we would like a record of fmap instances for both the `'Z` and `'S` functors. We further intuit that the subterm `({'Z ▹ const Unit, 'S ▹ λx. x} ?? a)` really ought to mean "the row with `'Z` mapped to `Unit` and `'S` mapped to `a`". Performing the remaining reductions yields:

```
fmapS {'Z ▹ const Unit, 'S ▹ λx. x} :
        Π {'Z ▹ ∀ a b. (a → b) → Unit → Unit,
           'S ▹ ∀ a b. (a → b) → a → b} →
        ∀ a b. (a → b) →
        Σ {'Z ▹ Unit , 'S ▹ a} →
        Σ {'Z ▹ Unit , 'S ▹ b}
```

The point we arrive at is that the precision of some R$\omega$ and R$\omega\mu$ types are supplanted quite effectively by type equivalence. Further, as values are passed to type-operators, the shapes of the types incur forms of reduction beyond simple $\beta$-reduction. In this case, we must map type operators over rows; we next consider the reduction of row complements.

**2.1.3 Desugaring Booleans.** Consider a desugaring of Booleans to Church encodings:

```
type BoolF = { 'T ▹ const Unit ,
               'F ▹ const Unit ,
               'If ▹ λx. Triple x x x}
type LamF  = { 'Lam ▹ Id ,
               'App ▹ λx. Pair x x ,
               'Var ▹ const Nat }
desugar : ∀ y. BoolF ≲ y, LamF ≲ y \ BoolF ⇒
          Π (Functor (y \ BoolF)) →
          μ (Σ y) →
          μ (Σ (y \ BoolF))
```

We will ignore the already stated complications that arise from subexpressions such as `Functor (y \ BoolF)` and skip to the step in which we tell `desugar` what particular row `y` it operates over. Here we know it must have at least the `BoolF` and `LamF` constructors. Let us try something like the following AST, using ++ as pseudonotation for row concatenation to save space.

```
type AST = BoolF ++ LamF ++
           {'Lit ▹ const Int , 'Add ▹ λx. Pair x x }
desugar AST : BoolF ≲ AST, LamF ≲ (AST \ BoolF) ⇒
              Π (Functor (AST \ BoolF)) →
              μ (Σ y) → μ (Σ (AST \ BoolF))
```

When `desugar` is passed `AST` for `z`, the inherent computation in the complement operator is made more obvious. What should `AST \ BoolF` reduce to? Intuitively, we suspect the following to hold:

```
AST \ BoolF = {'Lit ▹ const Int ,
               'Add ▹ λx. Pair x x,
               'Lam ▹ Id ,
               'App ▹ λx. Pair x x ,
               'Var ▹ const Nat }
```

But this computation must be realized, just as (analogously) $\lambda$-redexes are realized by $\beta$-reduction.

## 3 Type Equivalence & Reduction

We define reduction on types $\tau \longrightarrow_{\mathcal{T}} \tau'$ by directing the type equivalence judgment $\Delta \vdash \tau = \tau' : \kappa$ from left to right, defined in Figure 2. We omit conversion and closure rules. We describe each rule when I get around to it.

### 3.1 Normal forms

The syntax of normal types is given in Figure 3.

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

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

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

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

$$(\text{K}_{nf}\text{-}\triangleright)\ \frac{\Delta \vdash_{ne} n : \mathsf{L} \quad \Delta \vdash_{nf} \hat{\tau} : \kappa}{\Delta \vdash_{nf} n \triangleright \hat{\tau} : \mathsf{R}[\kappa]}$$

**Figure 3.** Normal type forms

Normalization reduces applications and maps except when a variable blocks computation, which we represent as a *neutral type*. A neutral type is either a variable or a spine of applications with a variable in head position. We distinguish ground kinds $\gamma$ from functional and row kinds, as neutral types may only be promoted to normal type at ground kind (rule ($\text{K}_{nf}$-NE)): neutral types $n$ at functional kind must $\eta$-expand to have an outer-most $\lambda$-binding (e.g., to $\lambda x.\, n\, x$), and neutral types at row kind are expanded to an inert map by the identity function (e.g., to $(\lambda x.x) \,\$\, n$). Likewise, repeated

$$\boxed{\Delta \vdash \tau = \tau : \kappa} \quad \boxed{\Delta \vdash \pi = \pi}$$

$$(\text{E-}\beta)\ \frac{\Delta \vdash (\lambda\alpha : \kappa.\tau)\ \upsilon : \kappa'}{\Delta \vdash (\lambda\alpha : \kappa.\tau)\ \upsilon = \tau[\upsilon/\alpha] : \kappa'}$$

$$(\text{E-LIFT}_\Xi)\ \frac{\Delta \vdash \rho : \mathsf{R}[\kappa \to \kappa'] \qquad \Delta \vdash \tau : \kappa}{\Delta \vdash (\Xi^{(\kappa \to \kappa')}\rho)\ \tau = \Xi^{(\kappa')}(\rho \mathbin{??} \tau) : \kappa'}\ (\Xi \in \{\Pi, \Sigma\})$$
$$\text{where } \rho \mathbin{??} \tau = (\lambda f. f\ \tau)\ \$\ \rho$$

$$(\text{E-}\backslash)\ \frac{\Delta \vdash \{\xi_i \triangleright \tau_i\}_{i\in 0\ldots n} : \mathsf{R}[\kappa] \qquad \Delta \vdash \{\xi_j \triangleright \tau_j\}_{j\in 0\ldots m} : \mathsf{R}[\kappa]}{\Delta \vdash \{\xi_i \triangleright \tau_i\} \setminus \{\xi_j \triangleright \tau_j\} = \text{subtract}\ \{\xi_i \triangleright \tau_i\}\ \{\xi_j \triangleright \tau_j\} : \mathsf{R}[\kappa]}$$

$$(\text{E-MAP})\ \frac{\Delta \vdash \phi : \kappa_1 \to \kappa_2 \qquad \Delta \vdash \{\xi_i \triangleright \tau_i\}_{i\in 0\ldots n} : \mathsf{R}[\kappa_1]}{\Delta \vdash \phi\ \$\ \{\xi_i \triangleright \tau_i\}_{i\in 0\ldots n} = \{\xi_i \triangleright \phi\ \tau_i\}_{i \in 0\ldots n} : \mathsf{R}[\kappa_2]}$$

$$(\text{E-MAP}_{\text{id}})\ \frac{\Delta \vdash \rho : \mathsf{R}[\kappa]}{\Delta \vdash (\lambda\alpha.\alpha)\ \$\ \rho = \rho : \mathsf{R}[\kappa]}$$

$$(\text{E-MAP}_\circ)\ \frac{\Delta \vdash \phi_1 : \kappa_2 \to \kappa_3 \qquad \Delta \vdash \phi_2 : \kappa_1 \to \kappa_2 \qquad \Delta \vdash \rho : \mathsf{R}[\kappa_1]}{\Delta \vdash \phi_1\ \$\ (\phi_2\ \$\ \rho) = (\phi_1 \circ \phi_2)\ \$\ \rho : \kappa_3}$$
$$\text{where } \phi_1 \circ \phi_2 = \lambda\alpha.\phi_1\ (\phi_2\ \alpha)$$

$$(\text{E-MAP}_\backslash)\ \frac{\Delta \vdash \phi : \kappa_1 \to \kappa_2 \qquad \Delta \vdash \rho_i : \mathsf{R}[\kappa_1]}{\Delta \vdash \phi\ \$\ (\rho_2 \setminus \rho_1) = \phi\ \$\ \rho_2 \setminus \phi\ \$\ \rho_1 : \kappa_2}$$

$$(\text{E-}\Xi)\ \frac{\Delta \vdash \rho : \mathsf{R}[\mathsf{R}[\kappa]]}{\Delta \vdash \Xi^{(\mathsf{R}[\kappa])}\ \rho = \Xi^{(\kappa)}\ \$\ \rho : \mathsf{R}[\kappa]}\ (\Xi \in \{\Pi, \Sigma\})$$

$$(\text{E-}\eta)\ \frac{\Delta \vdash \phi : \kappa_1 \to \kappa_2}{\Delta \vdash \phi = \lambda\alpha : \kappa_1.\phi\ \alpha : \kappa_1 \to \kappa_2}$$

$$\boxed{\text{subtract}\ \rho\ \rho}$$

$$\text{subtract}\ \varepsilon\ \rho = \varepsilon$$
$$\text{subtract}\ \rho\ \varepsilon = \rho$$
$$\text{subtract}\ \{\ell \triangleright \tau, \rho\}\ \{\ell' \triangleright \tau', \rho'\} =$$
$$\begin{cases} \text{subtract}\ \rho\ \rho' & \text{if } \ell = \ell' \text{ and } \tau = \tau' \\ \{\ell \triangleright \tau, \text{subtract}\ \rho\ \{\ell' \triangleright \tau', \rho'\}\} & \text{if } \ell < \ell' \\ \text{subtract}\ \{\ell \triangleright \tau, \rho\}\ \rho' & \text{if } \ell > \ell' \end{cases}$$

**Figure 2.** Type equivalence

maps are necessarily composed according to rule (E-MAP$_\circ$): For example, $\phi_1\ \$\ (\phi_2\ \$\ n)$ normalizes by letting $\phi_1$ and $\phi_2$ compose into $((\phi_1 \circ \phi_2)\ \$\ n)$. By consequence of $\eta$-expansion, records and variants need only be formed at kind $\star$. This means a type such as $\Pi(\ell \triangleright \lambda x.x)$ must reduce to $\lambda x.\Pi(\ell \triangleright x)$, $\eta$-expanding its binder over the $\Pi$. Nested applications of $\Pi$ and $\Sigma$ are also "pushed in" by rule (E-$\Xi$). For example, the type $\Pi\,\Sigma\,(\ell_1 \triangleright (\ell_2 \triangleright \tau))$ has $\Sigma$ mapped over the outer row, reducing to $\Pi(\ell_1 \triangleright \Sigma(\ell_2 \triangleright \tau))$.

The syntax $n \triangleright \hat{\tau}$ separates singleton rows with variable labels from row literals $\hat{\rho}$ with literal labels; rule ($\kappa_{nf}$-$\triangleright$) ensures that $n$ is a well-kinded neutral label. A row is otherwise an inert map $\phi\ \$\ n$ or the complement of two rows $\hat{\tau}_2 \setminus \hat{\tau}_1$. Observe that the complement of two row literals should compute according to rule (E-$\backslash$); we thus require in the kinding of normal row complements ($\kappa_{nf}$-$\backslash$) that one (or both) rows are not literal so that the computation is indeed inert. The remaining normal type syntax does not differ meaningfully from the type syntax; the remaining kinding rules for the judgments $\Delta \vdash_{nf} \hat{\tau} : \kappa$ and $\Delta \vdash_{ne} n : \kappa$ are as expected.

## 3.2 Metatheory

### 3.2.1 Canonicity of normal types.

**3.2.1 Canonicity of normal types.** The normal type syntax is pleasantly partitioned by kind. Due to $\eta$-expansion of functional variables, arrow kinded types are canonically $\lambda$-bound. A normal type at kind $\mathsf{R}[\kappa]$ is either an inert map $\hat{\phi}^\star\ n$, a variable-labeled row $(n \triangleright \hat{\tau})$, the complement of two rows $\hat{\tau}_2 \setminus \hat{\tau}_1$, or a row literal $\hat{\rho}$. The first three cases necessarily have neutral types (recall that at least one of the two rows in a complement is not a row literal). Hence rows in empty contexts are canonically literal. Likewise, the only types with label kind in empty contexts are label literals; recall that we disallowed the formation of $\Pi$ and $\Sigma$ at kind $\mathsf{R}[\mathsf{L}] \to \mathsf{L}$, thereby disallowing non-literal labels such as $\Delta \nvdash \Pi\epsilon : \mathsf{L}$ or $\Delta \nvdash \Pi(\ell_1 \triangleright \ell_2) : \mathsf{L}$.

**Theorem 3.1** (Canonicity). *Let $\hat{\tau} \in \hat{\mathcal{T}}$.*

- *If $\Delta \vdash_{nf} \hat{\tau} : (\kappa_1 \to \kappa_2)$ then $\hat{\tau} = \lambda\alpha : \kappa_1.\hat{\upsilon}$;*
- *if $\epsilon \vdash_{nf} \hat{\tau} : \mathsf{R}[\kappa]$ then $\hat{\tau} = \{\ell_i \triangleright \hat{\tau}_i\}_{i \in 0\ldots m}$.*
- *If $\epsilon \vdash_{nf} \hat{\tau} : \mathsf{L}$, then $\hat{\tau} = \ell$.*

### 3.2.2 Normalization.

**Theorem 3.2** (Normalization). *There exists a normalization function $\Downarrow : \mathcal{T} \to \hat{\mathcal{T}}$ that maps well-kinded types to well-kinded normal forms.*

$\Downarrow$ is realized in Agda intrinsically as a function from derivations of $\Delta \vdash \tau : \kappa$ to derivations of $\Delta \vdash_{nf} \hat{\tau} : \kappa$. Conversely, we witness the inclusion $\hat{\mathcal{T}} \subseteq \mathcal{T}$ as an embedding $\Uparrow : \hat{\mathcal{T}} \to \mathcal{T}$, which casts derivations of $\Delta \vdash_{nf} \hat{\tau} : \kappa$ back to a derivation of $\Delta \vdash \tau : \kappa$; we omit this function and its use in the following claims, as it is effectively the identity function (modulo tags).

The following properties confirm that $\Downarrow$ behaves as a normalization function ought to. The first property, *stability*, asserts that normal forms cannot be further normalized. Stability implies *idempotency* and *surjectivity*.

**Theorem 3.3** (Properties of normalization).

- *(Stability) for all $\hat{\tau} \in \hat{\mathcal{T}}$, $\Downarrow \hat{\tau} = \hat{\tau}$.*
- *(Idempotency) For all $\tau \in \mathcal{T}$, $\Downarrow (\Downarrow \tau) = \Downarrow \tau$.*
- *(Surjectivity) For all $\hat{\tau} \in \hat{\mathcal{T}}$, there exists $\upsilon \in \mathcal{T}$ such that $\hat{\tau} = \Downarrow \upsilon$.*

We now show that $\Downarrow$ indeed reduces faithfully according to the equivalence relation $\Delta \vdash \tau = \tau : \kappa$. Completeness of normalization states that equivalent types normalize to the same form.

**Theorem 3.4** (Completeness). *For well-kinded $\tau, \upsilon \in \mathcal{T}$ at kind $\kappa$, If $\Delta \vdash \tau = \upsilon : \kappa$ then $\Downarrow \tau = \Downarrow \upsilon$.*

Soundness of normalization states that every type is equivalent to its normalization.

**Theorem 3.5** (Soundness). *For well-kinded $\tau \in \mathcal{T}$ at kind $\kappa$, there exists a derivation that $\Delta \vdash \tau = \Downarrow \tau : \kappa$. Equivalently, if $\Downarrow \tau = \Downarrow \upsilon$, then $\Delta \vdash \tau = \upsilon : \kappa$.*

Soundness and completeness together imply, as desired, that $\tau \longrightarrow_{\mathcal{T}} \tau'$ iff $\Downarrow \tau = \Downarrow \tau'$.

#### 3.2.3 Decidability of type conversion.

Equivalence of normal types is syntactically decidable which, in conjunction with soundness and completeness, is sufficient to show that R$\omega\mu$'s equivalence relation is decidable.

**Theorem 3.6** (Decidability). *Given well-kinded $\tau, \upsilon \in \mathcal{T}$ at kind $\kappa$, the judgment $\Delta \vdash \tau = \upsilon : \kappa$ either (i) has a derivation or (ii) has no derivation.*

## 4 Normalization by Evaluation (NbE)

This section and those that follow give a closer examination into how the above metatheory was derived. In particular, we explain the *normalization of types by evaluation* (NbE) involved in deriving a normalization algorithm. We describe the standard components of NbE, but place emphasis on the novelty of normalizing rows and row operators.

Normalization by evaluation comes in a handful of different flavors. In our case, we seek to build a normalization function $\Downarrow : \mathcal{T} \to \hat{\mathcal{T}}$ by interpreting derivations in $\mathcal{T}^{\kappa}_{\Delta}$ (the set of derivations of the judgment $\Delta \vdash \tau : \kappa$) into a semantic domain capable of performing reductions semantically. We then *reify* objects in the semantic domain back to judgments in $\hat{\mathcal{T}}^{\kappa}_{\Delta}$ (the set of derivations of the judgment $\Delta \vdash_{nf} \tau : \kappa$). The mapping of syntax to a semantic domain is typically written as $[\![\cdot]\!]$ and called the *residualizing semantics*. For example, a judgment of the form $\Delta \vdash \phi : \star \to \star$ could be interpreted into a set-theoretic function, allowing applications to be interpreted into set-theoretic applications by that function. In our case, the syntax of the judgments $\Delta \vdash \tau : \kappa$, $\Delta \vdash_{nf} \tau : \kappa$, and $\Delta \vdash_{ne} \tau : \kappa$ are represented as Agda data types (where `Env` is a list of De Bruijn indexed type variables and `Kind` is the type of kinds):

```
data Type : Env → Kind → Set
data NormalType : Env → Kind → Set
data NeutralType : Env → Kind → Set
```

```
SemType : Env → Kind → Set
SemType Δ ★ = NormalType Δ ★
SemType Δ L = NormalType Δ L
SemType Δ₁ (κ₁ → κ₂) = KripkeFunction Δ₁ κ₁ κ₂
SemType Δ R[ κ ] =
  RowType Δ (λ Δ' → SemType Δ' κ) R[ κ ]
```

**Figure 4.** Semantic types

### 4.1 Residualizing semantics

We define our semantic domain in Agda recursively over the syntax of `Kinds` in Figure 4. Care must be taken to not run afoul of Agda's termination and positivity checking.

Types at ground kind $\star$ and `L` are simply interpreted as `NormalTypes`. We interpret arrow-kinded types as *Kripke function spaces*, which permit the application of interpreted function $\phi$ at any environment $\Delta_2$ provided a renaming from $\Delta_1$ into $\Delta_2$.

```
Renaming Δ₁ Δ₂ = TVar Δ₁ κ → TVar Δ₂ κ
KripkeFunction : Env → Kind → Kind → Set
KripkeFunction Δ₁ κ₁ κ₂ = ∀ {Δ₂} →
  Renaming Δ₁ Δ₂ → SemType Δ₂ κ₁ → SemType Δ₂ κ₂
```

The first three equations thus far are standard for this style of Agda mechanization, borrowing from Chapman et al. [2019]. Novel to our development is the interpretation of row-kinded types. First, we define the interpretation of row literals as finitely indexed maps to label-type pairs. (Here the type `Label` is a synonym for `String`, but could be any type with decidable equality and a strict total-order.)

```
Row : Set → Set
Row A = ∃[ n ](Fin n → Label × A)
```

Next, we define a `RowType` inductively as one of four cases: either a row literal constructed by `row`, a neutral-labeled row singleton constructed by `_▹_`, an inert map constructed by `_$_`, or an inert row complement constructed by `_\_` (Figure 5).

Care must be taken to explain some nuances of each constructor. First, the `row` and `_\_` constructors are each constrained by predicates. The `OrderedRow` $\rho$ predicate asserts that $\rho$ has its string labels totally and ascendingly ordered—guaranteeing that labels in the row are unique and that rows are definitionally equal modulo ordering. The `NotRow` $\rho$ predicate asserts simply that $\rho$ was *not* constructed by `row`. In other words, it is not a row literal. This is important, as the complement of two row literals should reduce to a `Row`, so we must disallow the formation of complements in which at least one of the operands is a literal.

The next set of nuances come from dancing around Agda's positivity checker. It would have been preferable for us to have written the `row` and `_$_` constructors as follows:

```
  row      : (ρ : Row (SemType Δ κ)) →
```

```
data RowType (Δ : Env)
             (T : Env → Set) : Kind → Set where
  row      : (ρ : Row (T Δ)) →
              OrderedRow ρ →
              RowType Δ T R[ κ ]
  _▹_     : NeutralType Δ L →
             T Δ →
             RowType Δ T R[ κ ]
  _$_   : (∀ {Δ'} →
                Renaming Δ Δ' →
                NeutralType Δ' κ1 →
                T Δ') →
             NeutralType Δ R[ κ1 ] →
             RowType Δ T R[ κ2 ]
  _\_   : (ρ2 ρ1 : RowType Δ T R[ κ ]) →
             {nor : NotRow ρ2 or notRow ρ1} →
             RowType Δ T R[ κ ]
```

**Figure 5.** Semantic row type

```
               OrderedRow ρ →
               RowType Δ T R[ κ ]
    _$_   : (∀ {Δ'} →
                 Renaming Δ Δ' →
                 SemType Δ' κ1 →
                 SemType Δ' κ2) →
               NeutralType Δ R[ κ1 ] →
               RowType Δ T R[ κ2 ]
```

Such a definition would have necessarily made the types `RowType` and `SemType` mutually inductive-recursive. But this would run afoul of Agda's positivity checker for the following reasons:

1. in the constructor `row`, the input `Row (SemType Δ κ)` makes a recursive call to `SemType Δ κ`, where it's not clear (to Agda) that this is a strictly smaller recursive call. To get around this, we parameterize the `RowType` type by `T : Env → Set` so that we may enforce this recursive call to be structurally smaller—hence the definition of `SemType` at kind `R[ κ ]` passes the argument `(λ Δ' → SemType Δ' κ)`, which varies in environment but is at a strictly smaller kind.
2. The `_$_` constructor takes a `KripkeFunction` as input, in which `SemType Δ' κ1` occurs negatively, which Agda must outright reject. Here we borrow some clever machinery from Allais et al. [2013] and instead make the `KripkeFunction` accept the input `NeutralType Δ' κ1`, which is already defined. The trick is that, as we will show in the next section, every `NeutralType` may be promoted to a `SemType`. In practice this is sufficient for our needs.

```
reflect : NeutralType Δ κ → SemType Δ κ
reify : SemType Δ κ → NormalType Δ κ

reflect {κ = ★} τ = ne τ
reflect {κ = L} τ = ne τ
reflect {κ = κ1 → κ2} =
  λ r v → reflect ((rename r τ) · reify v)
reflect {κ = R[ κ ]} ρ = (λ r n → reflect n) $ ρ
```

**Figure 6.** reflection

### 4.2 Reflection & reification

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 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].)

Reflection is defined in Figure 6. Types at kind ★ and `L` can be promoted straightforwardly with the `ne` constructor. Neutral types at arrow kind must be expanded into Kripke functions. Note that the input `v` has type `SemType Δ κ1` and must be reified; additionally, tau is kinded in environment $\Delta_1$ and so must be renamed to $\Delta_2$, the environment of `v`. The syntax · is used to construct an application of a `neutralType` to a `normalType`. Finally, a neutral row (e.g., a row variable) must be expanded into an inert mapping by `(λ r n → reflect n)`, which is effectively the identity function.

The definition of reification is a little more involved (Figure 7). The first two equations are expected ($\tau$ is already in normal form). Functions are reified effectively by $\eta$-expansion; note that we are using intrinsically-scoped De Bruijn variables, so `Z` constructs the zero'th variable and `S` induces a renaming in which each variable is incremented by one. (Recall that $\phi$ is a Kripke function space and so expects a renaming as argument.) The constructor ` promotes a type variable to a `neutralType`.

The equation of interest is in reifying rows. We pun the `row` constructor to construct row literals at type `NormalType`, which likewise expects a proof that the row is well-ordered. Such a proof is given by the auxiliary lemma `reifyPreservesOrdering`, which proves what it says. Next, we use a helper function `reifyRow` to recursively build a list of `Label-NormalType` pairs (that is, the form of `NormalType` row literals) from a semantic row. The empty case is trivial; the successor case must inspect the head of the list by destructing `P fzero`, i.e., the label-type association of the zero'th finite index. From

```
reify {κ = ★} τ = τ
reify {κ = L} τ = τ
reify {κ = κ₁ → κ₂} ϕ = `λ (reify (ϕ S (` Z)))
reify {κ = R[ κ ]} (row ρ q) =
  row (reifyRow ρ) (reifyPreservesOrdering q)
  where
    reifyRow : Row (SemType Δ κ) →
               List (Label × NormalType Δ κ)
    reifyRow (0 , P) = []
    reifyRow (suc n , P) with P fzero
    ... | (l , τ) =
      (l , reify τ) :: reifyRow (n , P ∘ fsuc)
```

**Figure 7.** reification

there we yield a semantic type $\tau$ which we reify and append to the result of recursing.

Finally, we have asserted that types are reduced modulo $\beta$-reduction and $\eta$-expansion. It follows that a given `NeutralType` should, after reflection and reification, end up in an expanded form. This is precisely how we define the promotion of `NeutralTypes` to `NormalTypes`:

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

This function is necessary: the `NormalType` constructor `ne` stipulates that we may only promote neutral derivations to normal derivations at *ground kind* (rule ($\kappa_{nf}$-NE)). Hence $\eta$-`norm` is the only means by which we may promote neutral types at row or arrow kind.

## 4.3 Helping evaluation

We will build our evaluation function incrementally; we find it clearer to incrementally build helpers for sub-computation (e.g., mapping or the complement) on our way up to full evaluation. We describe these helpers next.

**4.3.1 Semantic application.** We define semantic application straightforwardly as Agda application under the identity renaming.

```
_·'_ : SemType Δ (κ₁ → κ₂) →
       SemType Δ κ₁ →
       SemType Δ κ₂
ϕ ·' v = ϕ id v
```

**4.3.2 Semantic mapping.** Mapping over rows is a form of computation novel to R$\omega\mu$'s equivalence relation. We define the mapping $\phi$ `$` $\rho$ over the four cases a semantic row may take (Figure 8). When $\rho$ is neutral-labeled, we simply apply $\phi$ to its contents. The case where $\rho$ is a row literal is interesting in that our choice of representation for row literals as Agda functions comes to pay off: we may express the mapping of $\phi$ across the row `(n , P)` by pre-composing `P` with $\phi$ (note that we must appropriately fmap $\phi$ over the pair's second component). The mapping of $\phi$ over a complement is distributive, following rule (E-MAP$_\backslash$). Likewise, we follow rule (E-MAP$_\circ$) in grouping the nested map $\phi$ \$ ($\phi_2$ `$ n`) into a composed map.

```
_$'_ : SemType Δ (κ₁ → κ₂) →
       SemType Δ R[ κ₁ ] →
       SemType Δ R[ κ₂ ]
ϕ $' (l ▹ τ) = l ▹ (ϕ ·' τ)
ϕ $' (row (n , P) q) = row (n , fmap (ϕ id) ∘ P)
ϕ $' (ρ₂ \ ρ₁) = (ϕ $ ρ₂) \ (ϕ $ ρ₁)
ϕ₁ $' (ϕ₂ $ n) = (λ r → ϕ₁ r ∘ ϕ₂ r) $ n
```

**Figure 8.** semantic mapping

```
Π' : SemType Δ R[ κ ] → SemType Δ κ
Π' {κ = ★} x = Π (reify x)
Π' {κ = κ₁ → κ₁} ϕ = λ r v → Π' (rename r ϕ ??' v)
Π' {κ = R[ κ ]} x = (λ r v → Π' v) $ x
```

**Figure 9.** Semantic Π

**4.3.3 Semantic complement.**

**4.3.4 Semantic flap.** The rule (E-LIFT$_\Xi$) describes how $\Pi$ and $\Sigma$ reassociate from e.g. $(\Pi\ \rho)$ `a` to $\Pi$ $(\rho$ `?? a`$)$. We define a semantic version of the flap (flipped map) operator as follows:

```
_??'_ : SemType Δ R[ κ₁ → κ₂ ] →
        SemType Δ κ₁ → SemType Δ R[ κ₂ ]
ϕ ??' a = (λ r f → f ·' (rename r a)) $' ϕ
```

**4.3.5 $\Pi$ and $\Sigma$ as operators.** The defining equations for the reduction of $\Pi$ is given in Figure 9. (The logic for $\Sigma$ is identical and omitted.)

We proceed by case splitting on the kind of the input to $\Pi'$. Recall that we may only construct record types in normal form at kind $\star$, and so for the case that $\kappa = \star$ we simply reify the input and construct the record via the $\Pi$ constructor. We exclude the case that $\kappa$ = `L` because it is impossible: in the `Type` syntax, we restrict the formation of the $\Pi$ constructor by the following predicate:

```
NotLabel : Kind → Set
NotLabel ★ = ⊤
NotLabel L = ⊥
NotLabel (κ₁ → κ₂) = NotLabel κ₂
NotLabel R[ κ ] = NotLabel κ
```

This is to say, one may not apply $\Pi$ to an input that is a label, a label-valued function, or a row of labels. Next, when applying `Π'` to a function, we must expand the semantic $\lambda$-binding outwards. Thereafter, we apply rule (E-LIFT$_\Xi$) to explain how `Π'` operates on a single operand. Finally, we implement rule (E-$\Xi$) directly in the last equation: the application of `Π'` to a row-kinded input `x` is simply the mapping of `Π'` over `x`.

### 4.4 Evaluation

Evaluation warrants an environment that maps type variables to semantic types. The identity environment, which fixes the meaning of variables, is given as the composition of reflection and `, the constructor of `NeutralTypes` from `TVars`.

```
SemEnv : Env → Env → Set
SemEnv Δ₁ Δ₂ = TVar Δ₁ → SemType Δ₂ κ
idEnv : SemEnv Δ Δ
idEnv = reflect ∘ `
```

### 4.5 Normalization

Normalization in the NbE approach is simply the composition of reification after evaluation.

```
⇓ : Type Δ κ → NormalType Δ κ
⇓ τ = reify (eval τ idEnv)
```

It will be helpful in the coming metatheory to define an inverse embedding by induction over the `NormalType` structure. The definition is entirely expected and omitted.

```
⇑ : NormalType Δ κ → Type Δ κ
```

## 5 Mechanized metatheory

This section gives a deeper exposition on the metatheory summarized in (§3.2). We forego syntactic tyding of claims and give a deeper explanation of the proof techniques involved.

### 5.1 Stability

Stability follows by simple induction on the input derivation $\Delta \vdash_{nf} \tau : \kappa$. Here it is clearer that we are stating ⇓ is left-inverse to ⇑.

```
stability : ∀ (τ : NormalType Δ κ) → ⇓ (⇑ τ) ≡ τ
```

Stability implies idempotency:

```
idempotency : ∀ (τ : Type Δ κ) →
              (⇑ ∘ ⇓ ∘ ⇑ ∘ ⇓) τ ≡ (⇑ ∘ ⇓) τ
idempotency τ rewrite (⇓ τ) = refl
```

and surjectivity:

```
surjectivity : ∀ (τ : NormalType Δ κ) →
               ∃[ υ ] (⇓ υ ≡ τ)
surjectivity τ = ( ⇑ τ , stability τ )
```

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

```
⇑-inj : ∀ (τ₁ τ₂ : NormalType Δ κ) →
        ⇑ τ₁ ≡ ⇑ τ₂ → τ₁ ≡ τ₂
⇑-inj τ₁ τ₂ eq =
  trans
    (sym (stability τ₁))
    (trans
      (cong ⇓ eq)
      (stability τ₂))
```

### 5.2 A logical relation for completeness

#### 5.2.1 Properties.

#### 5.2.2 Logical environments.

#### 5.2.3 The fundamental theorem and completeness.

### 5.3 A logical relation for soundness

#### 5.3.1 Properties.

#### 5.3.2 Logical environments.

#### 5.3.3 The fundamental theorem and Soundness.

## 6 Most closely related work

## 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.

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.

Alex Hubers, Apoorv Ingle, Andrew Marmaduke, and J. Garrett Morris. Extensible recursive functions, algebraically, 2024. URL https://arxiv.org/abs/2410.11742.