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 Rω, 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 Rω language to R$\omega\mu$, which is Rω 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

Rω and R$\omega\mu$ each have a type conversion rule. The rule below states that the term M can have its type converted from τ to υ provided a proof that τ and υ are equivalent. (For now, let us split environments into kinding environments Δ, evidence environments Φ, and typing environments Γ.)

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

Conversion rules can complicate metatheory in an intrinsic setting. Hubers and Morris [2023]; Hubers et al. [2024] each provide an intrinsic semantics and do not provide a procedure to decide type checking or type equivalence. Proofs of type conversion are thus necessarily embedded into the term language. This has a number of consequences:

1. Users of the surface language are 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 β-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 τ, 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 υ) has stepped to a term at type τ.
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$\omega\mu$ type equivalence relation;
2. a semantics of the type-level *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.

$$
\begin{array}{llcl}
\multicolumn{4}{c}{\text{Type variables } \alpha \in \mathcal{A} \qquad \text{Labels } \ell \in \mathcal{L}} \\[1ex]
\text{Kinds} & \kappa & ::= & \star \mid \mathsf{L} \mid \mathsf{R}[\kappa] \mid \kappa \to \kappa \\
\text{Predicates} & \pi, \psi & ::= & \rho \lesssim \rho \mid \rho \odot \rho \sim \rho \\
\text{Types} & \mathcal{T} \ni \phi, \tau, \rho, \xi & ::= & \alpha \mid T \mid \tau \to \tau \mid \pi \Rightarrow \tau \\
& & \mid & \forall \alpha : \kappa . \tau \mid \lambda \alpha : \kappa . \tau \mid \tau\, \tau \\
& & \mid & \{\xi_i \triangleright \tau_i\}_{i \in 0 \ldots m} \mid \ell \mid \# \tau \\
& & \mid & \phi \,\$\, \rho \mid \rho \setminus \rho \\
\text{Type constants} & T & ::= & \Pi^{(\kappa)} \mid \Sigma^{(\kappa)} \mid \mu
\end{array}
$$

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 L. Rows of kind κ are classified by R$[\kappa]$. When possible, we use ϕ for type functions, ρ for row types, and ξ for label types. Singleton types $\#\tau$ are used to cast label-kinded types to types at kind $\star$. $\phi \,\$\, \rho$ maps the type operator ϕ across a row ρ. In practice, we often leave the map operator implicit, using kind information to infer the presence of maps. We define a families of Π and Σ constructors, describing record and variants at various kinds; in practice, we can determine the kind annotation from context. μ 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 ε to denote the empty row.

The type $\pi \Rightarrow \tau$ denotes a qualified type. In essence, the predicate π restricts the instantiation of the type variables in τ. Our predicates capture relationships among rows: $\rho_1 \lesssim \rho_2$ means that ρ_1 is *contained* in ρ_2, and $\rho_1 \odot \rho_2 \sim \rho_3$ means that ρ_1 and ρ_2 can be *combined* to give ρ_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 ρ_1 from ρ_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$\omega\mu$

Rω 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 Rω 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 Rω 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 β-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) λ-redexes are realized by β-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.

3.1 Normal forms

The syntax of normal types is given in Figure 3.

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

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

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

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

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

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 γ from functional and row kinds, as neutral types may only be promoted to normal type at ground kind (rule ($\textsc{K}_{nf}$-NE)): neutral types n at functional kind must η-expand to have an outer-most λ-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 maps are necessarily composed according to rule (E-MAP$_\circ$):

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

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

$$(\text{E-LIFT}_\Xi)\ \frac{\Delta \vdash \rho : \mathsf{R}[\kappa_1 \to \kappa_2] \quad \Delta \vdash \tau : \kappa}{\Delta \vdash (\Xi^{(\kappa_1 \to \kappa_2)}\,\rho)\,\tau = \Xi^{(\kappa_2)}(\rho\,??\,\tau) : \kappa_2}\ (\Xi \in \{\Pi, \Sigma\})$$
$$\text{where } \rho\,??\,\tau = (\lambda f.f\,\tau)\,\$\,\rho$$

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

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

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

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

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

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

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

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

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

Figure 2. Type equivalence

For example, $\phi_1\,\$\,(\phi_2\,\$\,n)$ normalizes by letting ϕ_1 and ϕ_2 compose into $((\phi_1 \circ \phi_2)\,\$\,n)$. By consequence of η-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 $\lambda x.\Pi(\ell \triangleright x)$, η-expanding its binder over the Π. Nested applications of Π and Σ are also "pushed in" by rule (E-Ξ). For example, the type $\Pi\,\Sigma\,(\ell_1 \triangleright (\ell_2 \triangleright \tau))$ has Σ 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 (κ_{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 (κ_{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.

4 Normalization by Evaluation (NbE)

This section describes our methodology, which is largely inspired by the *normalization by evaluation* algorithm and metatheory of Chapman et al. [2019], although we have made significant extensions to their approach in order to capture the computation of rows. Our work also differs in some design choices (see (§6)). Our full development is available as part of the anonymous supplementary materials. The code we present here is summarized and tidied for display in print and easier digestion, but otherwise remains faithful to the development in behavior and intent. The claims of this section are annotated with the corresponding points in our full artifact.

Normalization by evaluation comes in a handful of different flavors. In our intrinsic case, we seek to build a normalization function $\Downarrow : \mathcal{T}^\kappa_\Delta \to \hat{\mathcal{T}}^\kappa_\Delta$ 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
```

We will interpret the `Type` and `NeutralType` types into Agda function spaces in order to leverage Agda's meta-level computation, then reify these semantic objects back to `NormalType` syntax.

4.1 Residualizing semantics

We define our semantic domain in Agda recursively over the syntax of `Kind`s in Figure 4.

```
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

```
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

Types at ground kind ★ and L are simply interpreted as NormalTypes. We interpret arrow-kinded types as *Kripke function spaces*, which permit the application of interpreted function ϕ at any environment Δ_2 provided a renaming from Δ_1 into Δ_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 ρ predicate asserts that ρ 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 ρ predicate asserts simply that ρ 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 and termination checking. It would have been preferable for us to have written the row and _$_ constructors as follows:

```
  row      : (ρ : Row (SemType Δ κ)) →
              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 termination and positivity checkers 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.

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

```
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

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 η-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 $\star$ 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 κ_1 and must be reified; additionally, τ is kinded in environment Δ_1 and so must be renamed to Δ_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 (τ is already in normal form). Functions are reified effectively by η-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 ϕ is a Kripke function space and so expects a renaming as argument.) The constructor ` promotes a type variable to a `neutralType`, which is reflected so that it may be passed to $\phi$ `S`. The remaining equations describe the reification of the four row cases. When the input is a neutral-labeled row singleton, we need only create a `NeutralType`-labeled singleton with the body $\tau$ reified. The case of an inert complement $\rho_2 \setminus \rho_1$ remains an inert complement at type `NormalType`. Finally, we reify the inert map $\phi$ `$` $\tau$ by reifying $\phi$ analogously to the $\kappa_1$ `→ κ_2 case and mapping it over the reification of τ.

The equation of interest is in reifying row literals. 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`. 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

```
reify {κ = ★} τ = τ
reify {κ = L} τ = τ
reify {κ = κ₁ '→ κ₂} ϕ =
  `λ (reify (ϕ S (reflect (` Z))))
reify {κ = R[ κ ]} (l ▹ τ) = l ▹ (reify τ)
reify {κ = R[ κ ]} (ρ₂ \ ρ₁) = reify ρ₂ \ reify ρ₁
reify {κ = R[ κ ]} (ϕ $ τ) =
  `λ (reify (ϕ S (` Z))) $ (reify τ)
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

the head of the list by destructing `P fzero`, i.e., the label-type association of the zero'th finite index. From there we yield a semantic type τ which we reify and append to the result of recursing.

Finally, we have asserted that types are reduced modulo β-reduction and η-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 `NeutralType`s to `NormalType`s:

```
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 (κ_{nf}-NE)). Hence η-`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 ϕ `$` ρ over the four cases a semantic row may

```
_$'_ : 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

```
_In?_ : Label → Row (SemType Δ κ) → Bool

_\'_ : Row (SemType Δ κ) → Row (SemType Δ κ) →
            Row (SemType Δ κ)
(zero , P) \' (m , Q) = 0 , λ ()
(suc n , P) \' (m , Q) with P fzero .fst In? Q
... | true = (P ∘ fsuc) \' Q
... | false = suc n , λ { fzero → P fzero ,
                             fsuc _ → (P ∘ fsuc) \' Q }
```

Figure 9. Semantic complement

take (Figure 8). When ρ is neutral-labeled, we simply apply ϕ to its contents. The case where ρ 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 ϕ across the row `(n , P)` by pre-composing `P` with ϕ (note that we must appropriately fmap ϕ over the pair's second component). The mapping of ϕ over a complement is distributive, following rule ($\text{E-MAP}_{\backslash}$). Likewise, we follow rule ($\text{E-MAP}_{\circ}$) in grouping the nested map ϕ `$` (ϕ_2 `$ n`) into a composed map.

4.3.3 Semantic complement.

The complement of two row-kinded semantic types is always inert when one (or both) are not row literals, and thus constructed simply by the `_\_` constructor. The interesting case is when we must reduce two row literals to another row literal (Figure 9). Here our implementation differs slightly to the syntactic presentation in Figure 2. We proceed by induction on the length of the left-hand row: The resulting row is the empty row `0 , λ ()` when the left-hand row is empty. (That is to say, an empty row minus any other row is empty.) Otherwise, we check if the label of the head entry in `P`, `P fzero .fst`, is in the right-hand row. If so, we omit it and proceed with recursion. If not, we retain it.

4.3.4 Semantic flap.

The rule (E-LIFT_{Ξ}) describes how Π and Σ reassociate from e.g. (Π ρ) `a` to Π (ρ `??` `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
```

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

Figure 10. Semantic Π

4.3.5 Semantic Π and Σ.

The defining equations for the reduction of Π is given in Figure 10. (The logic for Σ is identical and omitted.)

The input row to `Π'` has kind `R[ κ ]`; we proceed by destructing κ. 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 `NormalType` constructor Π. We exclude the case that κ = `L` because it is impossible: in the `Type` syntax, we restrict the formation of the Π 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 Π to an input that is a row of labels, a label-valued function, or a nested row of labels. Next, when applying `Π'` to a function, we must expand the semantic λ-binding outwards. Thereafter, we apply rule (E-LIFT_{Ξ}) to explain how `Π'` operates on a single operand. Finally, we implement rule (E-Ξ) 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 κ → SemType Δ₂ κ
idEnv : SemEnv Δ Δ
idEnv = reflect ∘ `
```

We describe only the interesting cases of evaluation (Figure 11); the rest are purely compositional.

The first equation states that variables evaluate to their meaning in environment η. The equations for application `_·_`, row complement `_\_`, record and variant operators Π and Σ, and mapping `_$_` defer to the semantic helpers defined in (§4.3). The evaluation of a function `` `λ τ `` is simply the evaluation of the body in the environment η expanded with semantic object `v`, being careful to rename appropriately as this is a Kripke function. Evaluation of labeled singletons must check if the label is a neutral variable `n` or label literal ℓ; in the former case, we evaluate to an inert singleton using

```
eval : Type Δ1 → SemEnv Δ1 Δ2 → SemType Δ2 κ
eval (` x) η = η x
eval (τ1 · τ2) η = (eval τ1 η) ·' (eval τ2 η)
eval (ρ2 \ ρ1) η = eval ρ2 η \' eval ρ1 η
eval (`λ τ) η = λ r v →
  eval τ (extend (rename r ∘ η) v)
eval Π η = λ r v → Π' v
eval Σ η = λ r v → Σ' v
eval (ϕ $ τ) η = eval ϕ η $' eval n τ
eval (l ▹ τ) η with eval l η
... | ne n = (n ▹ eval τ η)
... | lab ℓ = row (1 , λ { fzero →
                               (ℓ , eval τ η )}) tt
eval (row ρ q) η = row
  (evalRow ρ η)
  (evalPreservesOrdering q)
where
  evalRow : List (Label × (Type Δ1 κ)) →
            SemEnv Δ1 Δ2 →
            Row (SemType Δ2 κ)
  evalRow [] η = 0 , λ ()
  evalRow ((l , τ) :: ρ) η = λ { fzero → eval τ η ,
                             fsuc _ → evalRow ρ η }
```

Figure 11. Evaluation

the RowType constructor `_▹_`; in the latter, we evaluate to a row literal in which `fzero` points to (`ℓ , eval τ η`). The term `tt : Unit` is the evidence that this row literal is trivially ordered. Finally, we evaluate row literals by recursion: the empty case evaluates to the empty Row, `0 , λ ()`; the cons case evaluates to a row in which `fzero` maps to the evaluation of τ, while `fsuc` otherwise proceeds recursively. Again, we have an obligation to prove that evaluation preserves the ordering evidence `q`, which is performed by the auxiliary lemma `evalPreservesOrdering`.

4.5 Normalization

Theorem 4.1 (Normalization). *There exists a normalization function $\Downarrow : \mathcal{T}_\Delta^\kappa \to \hat{\mathcal{T}}_\Delta^\kappa$, that maps well-kinded types to well-kinded normal forms.*

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 necessary in the coming metatheory to define an inverse embedding by induction over the `NormalType` structure. The definitions are entirely expected and omitted.

```
⇑ : NormalType Δ κ → Type Δ κ
⇑NE : NeutralType Δ κ → Type Δ κ
```

5 Mechanized metatheory

This section describes the Agda formalization of our metatheory, including proofs and proof outlines where space permits.

5.1 Canonicity of normal types

Theorem 5.1 (Canonicity). *Let $\hat{\tau} \in \hat{\mathcal{T}}_\Delta^\kappa$.*

- *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 $\hat{\tau} = \ell$.*

Normal forms are partitioned by kind, which can easily be shown by case splitting on `NormalType` inputs. We first demonstrate that neutrals cannot exist in an empty environment:

```
noNeutrals : NeutralType [] κ → ⊥
noNeutrals (n · τ) = noNeutrals n
```

Now, in any context an arrow-kinded type is canonically λ-bound:

```
arrow-canonicity : (ϕ : NormalType Δ (κ1 '→ κ2)) →
                         ∃[ τ ](ϕ ≡ `λ τ)
arrow-canonicity (`λ τ) = τ , refl
```

A row in an empty context is necessarily a row literal (all omitted cases are eliminated by `⊥-elim`):

```
row-canonicity : (ρ : NormalType [] R[ κ]) →
                    ∃[ (xs , oxs) ]
                    ( ρ ≡ row xs oxs)
row-canonicity (row ρ q) = ρ , q , refl
```

And a label-kinded type is necessarily a label literal (where `lab` constructs a label literal):

```
label-canonicity : (l : NormalType [] L) →
                      ∃[ ℓ ] (l ≡ lab ℓ)
label-canonicity (ne x) = ⊥-elim (noNeutrals x)
label-canonicity (lab s) = s , refl
```

5.2 Stability

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 5.2 (Properties of normalization).

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

Stability follows by simple induction on the input derivation $\Delta \vdash_{nf} \tau : \kappa$. We are in essence just stating that normallization ($\Downarrow$) is left-inverse to embedding ($\Uparrow$).

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

Stability implies idempotency:

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

5.3 A logical relation for completeness

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 5.3 (Completeness)**.** *For well-kinded $\tau, \upsilon \in \mathcal{T}^{\kappa}_{\Delta}$, If $\Delta \vdash \tau = \upsilon : \kappa$ then $\Downarrow \tau = \Downarrow \upsilon$.*

We define the equivalence relation of Figure 2 as an inductive, intrinsically typed relation in Agda.

```
data _≡t_ : Type Δ κ → Type Δ κ → set
```

and hence completeness may be stated as follows:

```
completeness : ∀ τ1 τ2. τ1 ≡t τ2 → ⇓ τ1 ≡ ⇓ τ2
```

We prove completeness via a logical relation `_≈_` on semantic types that specifies when two semantic objects are equivalent modulo uniformity ([Allais et al. 2013; Chapman et al. 2019]) and pointwise functional extensionality. We define `_≈_` recursively over the kinds of the inputs τ_1 and τ_2 (Figure 12).

The completeness logical relation is defined compositionally in the cases where κ = ★, κ = L, or κ = R[κ] and the equated rows are neutral-labeled or inert complements. In the case that $\kappa = \kappa_1$ `→ κ_2, we assert that the Kripke functions ϕ_1 and ϕ_2 are *uniform* and *extensionally equivalent* to one another (Figure 13). Uniformity and states effectively that passing a renaming r_2 to a Kripke function applied to its argument is equivalent to renaming it algorithmically. The uniformity property is attributable to Allais et al. [2013] but simplified drastically by Chapman et al. [2019]. The `PointEqual` predicate circumvents any need to postulate functional extensionality; rather, we assert that ϕ_1 and ϕ_2 map equivalent inputs to equivalent outputs (a property to be expected of a logical relation).

```
_≈_ {κ = ★} τ1 τ2 = τ1 ≡ τ2
_≈_ {κ = L} τ1 τ2 = τ1 ≡ τ2
_≈_ {κ = κ1 `→ κ2} φ1 φ2 =
  Uniform φ1 × Uniform φ2 × PointEqual φ1 φ2
_≈_ {κ = R[ κ ]} (ℓ1 ▹ τ1) (ℓ2 ▹ τ2) = ℓ1 ≡ ℓ2 × τ1 ≈ τ2
_≈_ {κ = R[ κ2 ]} (_$_ {κ1} φ1 n1) (_$_ {κ1'} φ2 n2) =
  ∃[ pf : κ1 ≡ κ1' ]
    UniformNE φ1 ×
    UniformNE φ2 ×
    PointEqualNE (convKripkeNE pf φ1) φ2 ×
    convNE pf n1 ≡ n2
_≈_ {κ = R[ κ ]} (ρ2 \ ρ1) (ρ4 \ ρ3) = ρ2 ≈ ρ4 × ρ1 ≈ ρ3
_≈_ {κ = R[ κ ]} (row ρ1 q) (row ρ2 g) = ρ1 ≈R ρ2
  where
    (ℓ1 , τ1) ≈2 (ℓ2 , τ2) = ℓ1 ≡ ℓ2 × τ1 ≈ τ2
    (n , P) ≈R (m , Q) = ∃[ pf : n ≡ m]
                          (∀ (i : fin m) →
                          (subst-Row pf P) i ≈2 Q i)
```

Figure 12. Completeness relation

```
Uniform : KripkeFunction Δ κ1 κ2 → Set
Uniform φ = ∀ (r1 : Renaming Δ1 Δ2)
              (r2 : Renaming Δ2 Δ3)
              (v1 v2 : SemType Δ2 κ1) →
              v1 ≈ v2 →
              rename r2 (φ r1 v1) ≈
              φ (r2 ∘ r1) (rename r2 v2)
PointEqual : ( φ1 φ2 : KripkeFunction Δ κ1 κ2) → Set
PointEqual φ1 φ2 = ∀ (r : Renaming Δ1 Δ2)
                     {v1 v2 : SemType Δ2κ1} →
                     v1 ≈ v2 →
                     φ1 r v1 ≈ φ2 r v2
```

Figure 13. Uniformity and point equality

The predicates `UniformNE` and `PointEqualNE` are entirely analogous to `Uniform` and `PointEqual` except that they describe Kripke functions in which the domain is a `NeutralType` rather than `SemType`. In the case that we are equating two inert maps, we must additionally assert that the domains of ϕ_1 and ϕ_2 (that is, κ_1 and κ_1 ', resp.) are equivalent. The helpers `convKripkeNE` and `convNE` convert ϕ_1 and n_1 appropriately so to be indexed by kind κ_1 '.

Finally, we equate row literals under the ≈R relation, which states that (i) the two rows' lengths are equal, and (ii) the two rows have pointwise related contents. Note that we must use an auxiliary helper `subst-Row` to convert the length `n` indexing `P` to be `m`.

5.3.1 Properties.

Propositionally equal neutral types reflect to equivalent semantic objects:

```
reflect-≈ : ∀ {τ1 τ2 : NeutralType Δ κ} →
            τ1 ≡ τ2 → reflect τ1 ≈ reflect τ2
```

Dually, equivalent semantic objects reify to propositionally equal types.

```
reify-≈ : λ∀ {v₁ v₂ : SemType Δ κ} →
                v₁ ≈ v₂ →
                reify v₁ ≡ reify v₂
```

5.3.2 Logical environments.

We lift the relation _≈_ to a relation on semantic environments η_1 and η_2 by asserting that the two are pointwise related.

```
_≈e_ : (η₁ η₂ : SemEnv Δ₁ Δ₂) → Set
η₁ ≈e η₂ = ∀ (α : TVar Δ₁ κ) → (η₁ α) ≈ (η₂ α)
```

The identity semantic environment relates to itself under the reflection of propositional equality, as witnessed by `idEnv-≈`.

```
idEnv-≈ : idEnv ≈e idEnv
idEnv-≈ α = reflect-≈ refl
```

5.3.3 The fundamental theorem and completeness.

The fundamental theorem for the completeness relation states that equivalent types evaluate to related semantic objects. The proof of the fundamental theorem is by induction over the type equivalence witness $\tau_1 \equiv t\ \tau_2$.

```
fundC : η₁ ≈e η₂ → τ₁ ≡t τ₂ →
            eval τ₁ η₁ ≈ eval τ₂ η₂
```

Completeness follows from the fundamental theorem in the identity semantic environment.

```
completeness : ∀ τ₁ τ₂. τ₁ ≡t τ₂ → ⇓ τ₁ ≡ ⇓ τ₂
completeness τ₁ τ₂ eq = reify-≈ (fundC idEnv-≈ eq)
```

5.4 A logical relation for soundness

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

Theorem 5.4 (Soundness). *For well-kinded $\tau \in \mathcal{T}^{\kappa}_{\Delta}$, there exists a derivation that $\Delta \vdash \tau =\Downarrow \tau : \kappa$. Equivalently, if $\Downarrow \tau = \Downarrow \upsilon$, then $\Delta \vdash \tau = \upsilon : \kappa$.*

In Agda, soundness states specifically that each type is equivalent to its embedded normalization:

```
soundness : ∀ (τ : Type Δ κ) → τ ≡t ⇑ (⇓ τ)
```

This is enough to imply the converse of completeness. We use an auxiliary transfer lemma, `embed-≡t`, to state that if the type τ_1 is equal to a normalization of τ_2, then the embedding of τ_1 is equivalent to τ_2.

```
embed-≡t : ∀ (τ₁ : NormalType Δ κ) (τ₂ : Type Δ κ) →
                τ₁ ≡ (⇓ τ₂) → ⇑ τ₁ ≡t τ₂
embed-≡t refl = sym (soundness τ₂)

conv-completeness : (τ₁ τ₂ : Type Δ κ) →
                        ⇓ τ₁ ≡ ⇓ τ₂ → τ₁ ≡t τ₂
conv-completeness τ₁ τ₂ eq =
  trans (soundness τ₁) (embed-≡t eq)
```

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

```
1  ⟦_⟧≈_ : Type Δ κ → SemType Δ κ → Set
2  ⟦_⟧≈_ {κ = ★} τ   v = τ ≡t ⇑ v
3  ⟦_⟧≈_ {κ = L} τ   v = τ ≡t ⇑ v
4  ⟦_⟧≈_ {κ = κ₁ `→ κ₂} ϕ F = SoundKripke ϕ F
5  ⟦_⟧≈_ {κ = R[ κ ]} τ (row (n , P) q) =
6    ∃[ xs ]
7    ∃[ oxs ]
8    (τ ≡t row xs oxs) ×
9    ⟦ xs ⟧R≈ (n , P)
10 ⟦_⟧≈_ {κ = R[ κ ]} τ (n ▹ v) =
11   ∃[ υ ]
12   (τ ≡t (⇑NE n ▹ υ)) ×
13   ⟦ υ ⟧≈ v
14 ⟦_⟧≈_ {κ = R[ κ ]} τ (ρ₂ \ ρ₁) =
15   (τ ≡t ⇑ (reify (ρ₂ \ ρ₁))) ×
16   ⟦ ⇑ (reify ρ₂) ⟧≈ ρ₂ ×
17   ⟦ ⇑ (reify ρ₁) ⟧≈ ρ₁
18 ⟦_⟧≈_ {κ = R[ κ ]} τ (F $ n) =
19   ∃[ ϕ ]
20   (τ ≡t (ϕ $ ⇑NE n)) ×
21   (SoundKripkeNE ϕ F)
```

Figure 14. Soundness relation

In Figure 14, we define a logical relation for soundness by relating un-normalized types to semantic objects. The first two cases (lines 2-3) state that τ relates to v at ground kind when τ is equivalent to the embedding of v. On line 4, we relate arrow-kinded types; Figure 16 describes when syntactic type operators relate to Kripke functions. The definitions are largely the same for semantic and neutral Kripke functions except that, in the neutral case, we require instead that τ is equivalent to the η-expansion of n. The definitions otherwise assert that related inputs map to related outputs.

The cases that follow relate syntactic rows with semantic rows. On line 5 we relate row literals *existentially*:[1] we assert that there exists a syntactic row literal `xs : List (Label × Type Δ κ)` and a well-orderedness predicate `oxs` such that τ is equivalent to `row xs oxs` and, further, `xs` is pointwise equivalent to `(n , P)`. The pointwise relation `.⟦_⟧R≈_` of syntactic and semantic row literals is defined in Figure 15, and is more or less to be expected: two row literals are related if they are of the same length and have related contents.

On lines 10-13 we relate row singletons in a straightforward fashion; lines 14-17 relate row complements compositionally. Finally, we relate inert row maps by asserting the existence of a ϕ that is sound with respect to the `KripkeFunctionNE F` (Figure 16).

[1] In practice, we actually know precisely that τ should be equivalent to the embedding of the reification of `row (n , P) q`; either style will work.

```
⟦ [] ⟧R≈_ (zero , P) = ⊤
⟦ [] ⟧R≈_ (suc n , P) = ⊥
⟦ x :: ρ ⟧R≈_ (zero , P) = ⊥
⟦ x :: ρ ⟧R≈_ (suc n , P) =
  ⟦ x ⟧≈2 (P fzero) ×
  ⟦ ρ ⟧R≈ (n , P ∘ fsuc)
  where
    ⟦ (ℓ₁ , τ) ⟧≈2 (ℓ₂ , v) =
      (ℓ₁ ≡ ℓ₂) × ⟦ τ ⟧≈ v
```

Figure 15. Soundness relation (row literals)

```
SoundKripke : Type Δ₁ (κ₁ `→ κ₂) →
              KripkeFunction Δ₁ κ₁ κ₂ → Set
SoundKripke ϕ F =
  (r : Renaming Δ₁ Δ₂)
  {τ : Type Δ₂ κ} {v : SemType Δ₂ κ}
  ⟦ τ ⟧≈ v →
  ⟦ rename r ϕ · τ ⟧≈ F r v

SoundKripkeNE : Type Δ₁ (κ₁ `→ κ₂) →
                KripkeFunctionNE Δ₁ κ₁ κ₂ → Set
SoundKripkeNE ϕ F =
  (r : Renaming Δ₁Δ₂)
  {τ : Type Δ₁ κ} {n : NeutralType Δ₁ κ}
  τ ≡t ⇑ (η-norm n) →
  ⟦ rename r ϕ · n ⟧≈ F r n
```

Figure 16. Soundness of Kripke functions

5.4.1 Properties.

Analogous to the completeness relation, equivalence of neutral types can be reflected into the soundness relation.

```
reflect-⟦⟧≈ : ∀ {τ : Type Δ κ}
                {n : NeutralType Δ κ} →
                τ ≡t ⇑NE n  → ⟦ τ ⟧≈ (reflect n)
```

And the relation of `Type` τ to semantic type `v` can be reified to type equivalence:

```
reify-⟦⟧≈ : ∀ {τ : Type Δ κ}
              {v : SemType Δ κ} →
              ⟦ τ ⟧≈ v →
              τ ≡t ⇑ (reify v)
```

5.4.2 Logical environments.

A syntactic substitution is related to a semantic environment when the substitution relates at each point in the environment.

```
⟦_⟧e≈ : Substitution Δ₁ Δ₂ → SemEnv Δ₁ Δ₂ → Set
⟦ σ ⟧e≈ η = (α : TVar Δ₁ κ) → ⟦ σ α ⟧≈ (η α)
```

Here a syntactic substitution is a map from type variables to Types:

```
Substitution : Env → Env → Set
Substitution Δ₁ Δ₂ = TVar Δ₁ κ → Type Δ₂ κ
```

5.4.3 The fundamental theorem and Soundness.

Finally, the fundamental theorem of soundness states that the syntactic substitution of a type τ by σ is related to its semantic evaluation in η, provided σ and η are related. The definition is by induction over τ.

```
fundS : (τ : Type Δ₁ κ) →
        ⟦ σ ⟧e≈ η →
        ⟦ sub σ τ ⟧≈ (eval τ η)
```

Soundness follows from a trivial case of the fundamental theorem where the environments related are the identity substitution ` and the identity semantic environment `idEnv`:

```
⊢⟦_⟧≈ : (τ : Type Δ κ)  → ⟦ τ ⟧≈ eval τ idEnv
soundness : ∀ (τ : Type Δ κ) → τ ≡t ⇑ (⇓ τ)
soundness τ = reify-⟦⟧≈ (⊢⟦ τ ⟧≈)
```

5.5 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 5.5 (Decidability). *Given well-kinded $\tau, \upsilon \in \mathcal{T}^{\kappa}_{\Delta}$, the judgment $\Delta \vdash \tau = \upsilon : \kappa$ either (i) has a derivation or (ii) has no derivation.*

It is easy enough (but tedious) to implement a syntactic decidability check on the shapes of normal forms; simply proceed by case analysis on both τ_1 and τ_2.

```
_≡?_ : ∀ (τ₁ τ₂ : NormalType Δ κ) → Dec (τ₁ ≡ τ₂)
```

Paired with soundness and completeness, we get an effective decision procedure to decide the relation τ_1 ≡t τ_2 as follows:

```
_≡t?_ : ∀ (τ₁ τ₂ : Type Δ κ) → Dec (τ₁ ≡t τ₂)
τ₁ ≡t? τ₂  with (⇓ τ₁) ≡? (⇓ τ₂)
... | yes p = yes
    (trans
        (soundness τ₁)
        (embed-≡t p))
... | no  p = no (λ x → p (completeness x))
```

6 Most closely related work

We conclude with a discussion of closely related work.

6.1 Intrinsic mechanization of System F$\omega\mu$.

Our technical development owes a huge debt to two papers in particular. We closely follow the formalization patterns and proof techniques of Chapman et al. [2019]; indeed, this paper is in some sense an extension of their work from System F$\omega\mu$ to system R$\omega\mu$. In turn, Chapman et al. [2019] themselves follow closely Allais et al. [2013], from whom we looked to in finding the correct semantic image of rows (borrowing from their semantic image of lists). Our paper differs in a few key ways. Firstly, we introduce label and row kinds to the syntax, incurring an additional burden to reduce row maps, the row operators Π and Σ, and row complements. We also reason about types modulo η-equivalence of functions and also expansion of rows to inert maps, which made many proof definitions harder to define. (For example, Chapman et al. [2019] do not need a mutually recursive reify and reflect, which in turn made many auxiliary lemmas harder to define.)

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.