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

ALEX HUBERS, The University of Iowa, USA

Abstract

We describe the normalization-by-evaluation (NbE) of types in R$\omega\mu$, a row calculus with recursive types, qualified types, and a novel *row complement* operator. Types are normalized to $\beta\eta$-long forms modulo a type equivalence relation. Because the type system of R$\omega\mu$ is a strict extension of System F$\omega\mu$, much of the type reduction is isomorphic to reduction of terms in the STLC. Novel to this report are the reductions of row, record, and variant types.

1 The R$\omega\mu$ calculus

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

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

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

Fig. 1. Syntax

1.1 Example types and the need for reduction

We will write Rome types in the slightly-altered syntax of *Rosi*, our experimental implementation of R$\omega\mu$. Wand's problem. Let us consider the role of type computation in a handful of R$\omega\mu$ types.

1.1.1 Wand's problem. R$\omega\mu$, Rω, Rose, and thus Rosi stem from the desire to express Wand's problem (and its dual `dnaw`):

```
wand : ∀ l x y z t. x ⊙ y ~ z, {l ▷ t} ≲ z ⇒ #l → Π x → Π y → t
dnaw : ∀ l x y z t. x ⊙ y ~ z, {l ▷ t} ≲ z ⇒ #l →
       (Σ x → t) → (Σ y → t) → Σ z → t
```

The novelty here is that, in other languages with extensible records and variants (e.g., OCaml), one has to specify precisely in which of the inputs `l` occurs. Here, the predicates `x ⊙ y ~ z` and `{l ▷ t} ≲ z` are sufficient to express that the input row `z` has an `{l ▷ t}` field without specifying if it is in `x` or `y`. This type demonstrates that Rose is cool. However, Rose is not higher-order, and so there is no type-level reduction to perform. Things become more complicated in the higher order scenario.

Author's Contact Information: Alex Hubers, Department of Computer Science, The University of Iowa, Iowa City, Iowa, USA, alexander-hubers@uiowa.edu.

1.1.2 Deriving functorality. Our family of languages is quite expressive, and the types even look quite readable! To some extent, this magic relies on implicit type application, mapping, and type reduction. Let us demonstrate. Here we can simulate the deriving of functor typeclass instances: given a record of `fmap` instances at type `Π (Functor z)`, I can give you 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)
```

Pay close attention: what is the type of `Functor z`? This is implicitly a map. Let us write it as such 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's 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 something resembling a normal form. Of course, the type is more interesting when applied to a real value for z. Suppose z is `{'l ▹ λx. x}`. Then a first pass yields:

```
fmapS {'l ▹ λx.x} :
        Π ((λf. ∀ a b. (a → b) → f a → f b) $ {'l ▹ λx.x}) →
        ∀ a b. (a → b) → Σ ({'l ▹ λx.x} ?? a) → Σ ({'l ▹ λx.x} ?? b)
```

How do we reduce from here? Regarding the first input, we suspect we would like a record with `'l` mapped to `λx.x` applied to the `Functor` type. We further intuit that the subterm `({'l ▹ λx.x} ?? a)` really ought to mean "the row with `'l` mapped to a". At this point I will perform multiple steps of computation simultaneously so as to retain the reader's attention.

```
fmapS {'l ▹ λx.x} :
        Π ({'l ▹ ∀ a b. (a → b) → a → b}) →
        ∀ a b. (a → b) → Σ ({'l ▹ a}) → Σ ({'l ▹ b})
```

The point we arrive at is that the elegance of some Rω and R$\omega\mu$ types are masked quite effectively by foregoing type reduction. Further, as intermediate values are passed to type-operators, the shapes of the types can begin to vary drastically, given that the computational rules of type reduction in R$\omega\mu$ are often not obvious or trivial.

1.1.3 Desugaring Booleans. Lastly, we emphasize the rule of reduction in computing complements. 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's try something like the following AST, using ⧺ as pseudonotation for row concatenation.

```
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.

1.2 The need for type normalization

Metatheory is difficult, particularly in the presence of conversion rules, of which both Rω and R$\omega\mu$ have. 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 equvivalent:

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

Conversion rules complicate metatheory. To list a few reasons:

(1) decidability of type checking now rests upon the decidability of type conversion.
(2) Conversion rules block proofs of progress. Let `M` have type `t`, let `pf` be a proof that `t = u`, 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 `u`) has stepped to a term at type `t`.
(3) 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.

2 Type Equivalence & Reduction

We define reduction on types $\tau \longrightarrow_{\mathcal{T}} \tau'$ by directing the type equivalence judgment $\varepsilon \vdash \tau = \tau' : \kappa$ from left to right, defined in Figure 2. Note that in some rules we will annotate Π and Σ with the kind of their contents, e.g., $\Sigma^{(\star)}$ has kind $\mathsf{R}^\star \to \star$.

2.1 Normal forms

The syntax of normal types is given in Figure 3. We carefully define the normal type syntax so that no type $\hat{\tau} \in \hat{\mathcal{T}}$ could reasonably reduce further to some other $\tau' \in \hat{\mathcal{T}}$. Hence we write $\tau \not\longrightarrow_{\mathcal{T}}$ synonymously with $\tau \in \hat{\mathcal{T}}$ to indicate that τ is well-kinded and has no further reductions. We define a normalization function in Agda to materialize this sentiment later.

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

$$(\text{E-}\beta)\ \frac{\Delta \vdash (\lambda\alpha : \kappa.\tau)\, \upsilon : \kappa'}{\Delta \vdash (\lambda\alpha : \kappa.\tau)\, \upsilon = \tau[\upsilon/\alpha] : \kappa'} \qquad (\text{E-LIFT}_\Xi)\ \frac{\Delta \vdash \rho : \mathrm{R}^{\kappa \to \kappa'} \quad \Delta \vdash \tau : \kappa}{\Delta \vdash (\Xi^{(\kappa \to \kappa')}\, \rho)\, \tau = \Xi^{(\kappa')}\, (\rho^{\$}\, \tau) : \kappa'}\ (\Xi \in \{\Pi, \Sigma\})$$
$$\text{where } \rho^{\$}\, \tau = (\lambda f.f\, \tau)\, \$\, \rho$$

$$(\text{E-}\setminus)\ \frac{\Delta \vdash \rho_i : \mathrm{R}^{\kappa}}{\Delta \vdash \rho_2 \setminus \rho_1 = \text{subtract}\, \rho_2\, \rho_1 : \mathrm{R}^{\kappa}} \qquad (\text{E-MAP})\ \frac{\Delta \vdash \phi : \kappa_1 \to \kappa_2 \quad \Delta \vdash \{\xi_i \triangleright \tau_i\}_{i \in 0 \ldots n} : \mathrm{R}^{\kappa_1}}{\Delta \vdash \phi\, \$\, \{\xi_i \triangleright \tau_i\}_{i \in 0 \ldots n} = \{\xi_i \triangleright \phi\, \tau_i\}_{i \in 0 \ldots n} : \mathrm{R}^{\kappa_2}}$$

$$(\text{E-MAP}_{\text{id}})\ \frac{\Delta \vdash \rho : \mathrm{R}^{\kappa}}{\Delta \vdash (\lambda\alpha.\alpha)\, \$\, \rho = \rho : \mathrm{R}^{\kappa}} \qquad (\text{E-MAP}_{\circ})\ \frac{\Delta \vdash \phi_1 : \kappa_2 \to \kappa_3 \quad \Delta \vdash \phi_2 : \kappa_1 \to \kappa_2 \quad \Delta \vdash \rho : \mathrm{R}^{\kappa_1}}{\Delta \vdash \phi_1\, \$\, (\phi_2\, \$\, \rho) = (\phi_1 \circ \phi_2)\, \$\, \rho : \kappa_3}$$
$$\text{where } \phi_1 \circ \phi_2 = \lambda\alpha.\phi_1\, (\phi_2\, \alpha)$$

$$(\text{E-MAP}_{\setminus})\ \frac{\Delta \vdash \phi : \kappa_1 \to \kappa_2 \quad \Delta \vdash \rho_i : \mathrm{R}^{\kappa_1}}{\Delta \vdash \phi\, \$\, (\rho_2 \setminus \rho_1) = \phi\, \$\, \rho_2 \setminus \phi\, \$\, \rho_1 : \kappa_2} \qquad (\text{E-}\Xi)\ \frac{\Delta \vdash \rho : \mathrm{R}^{\kappa}}{\Delta \vdash \Xi^{(\mathrm{R}^{\kappa})}\, \rho = \Xi^{(\kappa)}\, \$\, \rho : \mathrm{R}^{\kappa}}\ (\Xi \in \{\Pi, \Sigma\})$$

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

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

Fig. 2. Type equivalence

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

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

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

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

Fig. 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 (κ_{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$_\circ$): 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-$\setminus$); we thus require in the kinding of normal row complements (κ_{nf}-$\setminus$) 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.

2.2 Metatheory

2.2.1 Canonicity of normal types. The normal type syntax is pleasantly partitioned by kind. Due to η-expansion of functional variables, arrow kinded types are canonically λ-bound. A normal type at kind R^κ 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 Π and Σ 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 2.1 (Canonicity). *Let $\hat{\tau} \not\longrightarrow_{\mathcal{T}}$.*

- *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 $\epsilon \vdash \hat{\tau} : \mathsf{L}$, then $\hat{\tau} = \ell$.*

2.2.2 Normalization.

Theorem 2.2 (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 2.3 (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 2.4 (Completeness). *For well-kinded* $\tau, \upsilon \in \mathcal{T}$ *at kind* κ, *If* $\Delta \vdash \tau = \upsilon : \kappa$ *then* $\Downarrow \tau = \Downarrow \upsilon$.

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

Theorem 2.5 (Soundness). *For well-kinded* $\tau \in \mathcal{T}$ *at kind* κ, *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 $\Downarrow \tau = \Downarrow \tau'$.

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. Consequently, the user has no obligation to provide proofs of equivalence in type and predicate conversion (rules (T-CONV) and (N-CONV)).

2.2.3 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. This has a number of desirable consequences (see (§1.2)).

Theorem 2.6 (Decidability). *Given well-kinded* $\tau, \upsilon \in \mathcal{T}$ *at kind* κ, *the judgment* $\Delta \vdash \tau = \upsilon : \kappa$ *either (i) has a derivation or (ii) has no derivation.*

3 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 and emphasize where our approach has differed. Emphasis is placed on the novelty of normalizing rows and row operators.

3.1 The semantic domain

3.2 Reflection & reification

3.3 Evaluation

3.4 Normalization

4 Metatheory again, or: logical relations

This section gives a deeper exposition on the metatheory summarized (§2.2). We forego syntactic tyding of claims and give a deeper explanation of the proof techniques involved.

4.1 Stability

Stability follows by simple induction on typing derivations.

Theorem 4.1 (stability).

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