Contents

1 Normalisation by Evaluation

Normalisation by Evaluation (NbE) [1, 2] is a normalisation algorithm for lambda calculus terms, which operates by first evaluating terms into a semantic domain (specifically, the "presheaf model"), and then inverting the evaluation function to "quote" back normal forms. It can be motivated from multiple directions:

- **No reliance on small-step reductions:** NbE is structurally recursive, and is therefore not reliant on a separate strong normalisation result to justify termination. This can be especially useful in settings where a strongly normalising set of small-step reductions is difficult to identify (e.g. dealing with η-expansion).
- **Applicability to quotiented syntax:** Following on from the first point, unlike term-rewriting-based approaches to normalisation, NbE does not rely on distinguishing $\beta\eta$-convertible terms (the algorithm can be structured in such a way as to simply map families of convertible terms to values [3]). This enables working with more "semantic" [4] definitions of type theory (e.g. Categories with Families, or CwFs) where terms are quotiented by conversion, providing soundness "for free".
- **Efficiency:** NbE avoids repeated expensive single-substitutions (which need to traverse the whole syntax tree each time to possibly replace variables with the substitutee) [7]. Instead, the mappings between variables and semantic values are tracked in a persistent map (the "environment"), such that variables can be looked up exactly when they are evaluated.

For the application on NbE in this project, only the last of these points is truly relevant. Specifically, we do not plan to directly prove normalisation of type theory with local equational assumptions via NbE, primarily because I am unaware of a good way to justify rewriting-to-completion without going down to the level of an ordering on terms.

Instead, following [8], we shall employ NbE as the algorithm to decide conversion in our prototype Haskell typechecker. On top of the efficiency benefits, NbE is also relatively simple to implement, and as we shall see, is quite compatible with **smart case** in the sense that the extensions necessary to support local equations are minimal.

To introduce NbE, we will begin by deriving the algorithm for the Simply-Typed Lambda Calculus (STLC), staying within our Agda-flavoured ITT meta-theory. We will then move on to discuss how the algorithm extends to dependently-typed object languages, and the optimisations that become available when implementing inside a non-total metalanguage. Finally, we will cover the extensions to NbE necessary to support **smart case**.

Quotienting by conversion is especially attractive in the setting of dependent types, where intrinsically-typed syntax must otherwise be defined mutually with conversion to account for definitional equality [5, 6].

[1]: Berger et al. (1991), *An inverse of the evaluation functional for typed lambda-calculus*
[2]: Altenkirch et al. (1995), *Categorical reconstruction of a reduction free normalization proof*
[3]: Altenkirch et al. (2017), *Normalisation by evaluation for type theory, in type theory*
[4]: Kaposi et al. (2025), *Type Theory in Type Theory Using a Strictified Syntax*
[5]: Danielsson (2006), *A formalisation of a dependently typed language as an inductive-recursive family*
[6]: Kovács (2023), *TTinTTasSetoid.agda*
[7]: Kovács (2023), *smalltt*
[8]: Coquand (1996), *An algorithm for type-checking dependent types*

1.1 Syntax

> There is no such thing as a free variable.
> There are only variables bound in the context.
>
> *Conor McBride [9]*

To implement a normalisation algorithm, we need a syntax of terms to normalise. We will start with an *intrinsically-typed* syntax of STLC. That is, instead of first defining a grammar of terms and then separate typing relations, we will define our syntax as an indexed family such that only well-typed terms can be constructed.

Under the intrinsically-typed paradigm, simple types and contexts are still defined as usual. We include functions $A \hat{\to} B$, pairs $A \hat{\times} B$, unit $\hat{\mathbb{1}}$ and the empty type $\hat{\mathbb{0}}$, and define contexts as backwards lists of types.

We use hats "ˆ" to distinguish object-level STLC types from the analagous meta-level **Type**s.

```
data Ty : Type where
  _→̂_ : Ty → Ty → Ty
  _×̂_ : Ty → Ty → Ty
  𝟙̂   : Ty
  𝟘̂   : Ty
data Ctx : Type where
  ε   : Ctx
  _,_ : Ctx → Ty → Ctx
```

Variables are then merely proofs that a particular type occurs in the context. After erasing the indexing, we are effectively left with de Bruijn variables [10], natural numbers counting the number of binders between the use of a variable and the location it was bound.

[10]: De Bruijn (1972), *Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem*

```
data Var : Ctx → Ty → Type where
  vz : Var (Γ , A) A
  vs : Var Γ B → Var (Γ , A) B
```

Terms embed variables, and then also include the standard introduction and elimination rules for $\_\hat{\to}\_$, $\_\hat{\times}\_$, $\hat{\mathbb{1}}$.

To distinguish applications and abstractions of the meta-theory with those of the object language, we annotate λs with a hat and use the binary operator $\_\cdot\_$ instead of plain juxtaposition.

```
data Tm : Ctx → Ty → Type where
  `_  : Var Γ A → Tm Γ A
  λ̂_  : Tm (Γ , A) B → Tm Γ (A →̂ B)
  _·_ : Tm Γ (A →̂ B) → Tm Γ A → Tm Γ B
  _,_ : Tm Γ A → Tm Γ B → Tm Γ (A ×̂ B)
  π₁  : Tm Γ (A ×̂ B) → Tm Γ A
  π₂  : Tm Γ (A ×̂ B) → Tm Γ B
  ⟨⟩  : Tm Γ 𝟙̂
```

Note that while our syntax is instrinsically-typed and to some extent CwF-inspired, we have not gone so far as to actually quotient by conversion (we won't even define a conversion relation explicitly). This is merely for practical convenience - i.e. to avoid getting bogged down in the details, we will implement NbE, and in-doing-so prove termination and type-preservation, but for constraints of time, leave the full proof that NbE decides conversion to cited work (e.g. [11]).

[11]: Kovács (2017), *A machine-checked correctness proof of normalization by evaluation for simply typed lambda calculus*

1.1.1 Substitution and Renaming

While NbE itself does not require substitutions, our explorations on the path towards NbE will. NbE also does need at the very least some way to weaken terms (that is, embed terms into extended contexts), so the subset of parallel substitutions where variables may only be substituted for other variables, known as *renamings*, will be directly useful.

We define parallel renaming and substitution operations by recursion on our syntax. Following [12], we avoid duplication between renaming and substitution by factoring via a boolean algebra of Sorts, valued either V or T with V < T. We will skip over most of the details of how to encode this in Agda but explicitly define Sort-parameterised terms:

[12]: Altenkirch et al. (2025), *Substitution without copy and paste*

```
Tm[_] : Sort → Ctx → Ty → Type
Tm[ V ] ≔ Var
Tm[ T ] ≔ Tm
```

and lists of terms (parameterised by the sort of the terms, the context they exist in, and the list of types of each of the terms themselves).

```
data Tms[_] : Sort → Ctx → Ctx → Type where
  ε   : Tms[ q ] Δ ε
  _,_ : Tms[ q ] Δ Γ → Tm[ q ] Δ A → Tms[ q ] Δ (Γ , A)
```

We can simultaneously interpret lists of variables as renamings, Ren ≔ Tms[V] and lists of terms as full substitutions Sub ≔ Tms[T], with the following recursively defined substitution operation:

```
_[_] : Tm[ q ] Γ A → Tms[ r ] Δ Γ → Tm[ q ⊔ r ] Δ A
```

We also define a number of recursively-defined operations to build and manipulate renamings/substitutions, including id : Ren Γ Γ to build identity renamings (a backwards list of increasing variables), single weakenings wk : Ren (Γ , A) Γ, single substitutions <_> : Tm[q] Γ A → Tms[q] Γ (Γ , A), and composition _;_ : Tms[q] Δ Γ → Tms[r] Θ Δ → Tms[q ⊔ r] Θ Γ.

We refer to [12] for the details of how to define these operations.

1.2 Naive Normalisation

As a warm-up to NbE, we will start by implementing "naive" normalisation, i.e. recursing on a term, contracting β-redexes where possible by applying single-substitutions. As we are implementing the algorithm in the total language of Agda, we will detail how termination can be justified in terms of strong normalisation.

We first define our goal: β-normal forms, Nf Γ A, inductively (mutually recursively with stuck, neutral terms, Ne Γ A) along with the obvious injections back into ordinary terms, ⌜_⌝, ⌜_⌝ne.

Note that neutrals are comprised of spines of elimination forms while introduction rules are restricted to Nf, to rule-out β-redexes syntactically.

```
data Ne : Ctx → Ty → Type
data Nf : Ctx → Ty → Type

data Ne where
  `_  : Var Γ A → Ne Γ A
  _·_ : Ne Γ (A ↣ B) → Nf Γ A → Ne Γ B
  π₁  : Ne Γ (A ⨯ B) → Ne Γ A
  π₂  : Ne Γ (A ⨯ B) → Ne Γ B
```

```
data Nf where
  ne    : Ne Γ A → Nf Γ A
  λ̂_    : Nf (Γ , A) B → Nf Γ (A →̂ B)
  _,_   : Nf Γ A → Nf Γ B → Nf Γ (A ×̂ B)
  ⟨⟩    : Nf Γ 𝟙̂
⌜_⌝     : Nf Γ A → Tm Γ A
⌜_⌝ne   : Ne Γ A → Tm Γ A
```

To enforce η-normality for $\_\hat{\to}\_$, $\_\hat{\times}\_$ and $\hat{\mathbb{1}}$, we could restrict embedded neutrals in Nf to only those of empty type, $\hat{\mathbb{0}}$. $\beta\eta$-normal forms accounting for $\mathbb{0}$-η are more complicated [13].

[13]: Scherer (2017), *Deciding equivalence with sums and the empty type*

We can then attempt to define normalisation by recursion on terms, relying on substitution to contract β-redexes (for now focusing only on the cases for abstraction and application):

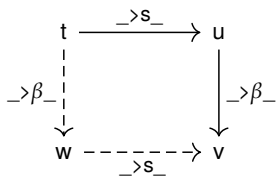
```
norm : Tm Γ A → Nf Γ A
nf-app : Nf Γ (A →̂ B) → Nf Γ A → Nf Γ B

norm (λ̂ t)    ≡ λ̂ (norm t)
norm (t · u)  ≡ nf-app (norm t) (norm u)

nf-app (ne t) u ≡ ne (t · u)
nf-app (λ̂ t)  u ≡ norm (⌜ t ⌝ [ < ⌜ u ⌝ > ])
```

In a partial language, when applied to normalising terms, this definition is sufficient! The single substitutions are less efficient on terms with multiple β-redexes than the NbE approach of tracking all variable mappings in a single environment, but it can be optimised with memoisation and annotating subterms with the sets of variables which are actually used (i.e. to avoid unnecessary traversals during substitution).

Note that normal forms are not stable under substitution (i.e. substitution can create new β-redexes), so we cannot easily define substitution on normal forms to resolve this. It is perhaps worth mentioning though, that if one is more careful with the representation of neutral spines (among other things), pushing in this direction can lead to another structurally recursive normalisation algorithm known as *hereditary substitution* [14]. Unfortunately, it is currently unknown whether this technique extends to dependent types.

In a total setting, unfortunately, naive normalisation is clearly not well-founded by structural recursion. `⌜ t ⌝ [ < ⌜ u ⌝ > ]` is not necessarily structurally smaller than t or u.

Making naive normalisation total relies on a strong normalisation result: we need to know that β-reduction, $\_>\beta\_$, is well-founded. Actually, we will make use of the accessibility of typed terms w.r.t. interleaved structural ordering, `_>s_`, and β-reduction, but luckily obtaining this from traditional strong normalisation is not too difficult [15]. Note that $\_>\beta\_$ commutes with `_>s_` in the sense that `t >s u → u >β v → ∃[ w ] t >β w × w >s v`, or as a diagram:



We therefore skip ahead to defining a single `_>_` relation on terms encompassing both structural and reduction orderings, and assume we have a proof that this combined order is well-founded.

Classically, strong normalisation can be defined as there existing no infinite chains of reductions. To justify induction w.r.t. reduction order constructively, we must instead use accessibility predicates. `Acc R x` can be thought of as the type of finite-depth trees starting at `x`, with branches corresponding to single steps along `_>_` and minimal elements w.r.t. relation `R` at the leaves.

[14]: Keller et al. (2010), *Hereditary substitutions for simple types, formalized*

[15]: Guillaume et al. (2024), *Combining Accessibility Proofs and Structural Ordering*

```
data _>β_ : Tm Γ A → Tm Γ A → Type where
  -- Congruence
  l·   : t₁ >β t₂ → t₁ · u >β t₂ · u
  ·r   : u₁ >β u₂ → t · u₁ >β t · u₂
  λ̂_   : t₁ >β t₂ → λ̂ t₁ >β λ̂ t₂
  -- etc...
  -- Reductions
  β    : (λ̂ t) · u >β t [ < u > ]
  π₁β  : π₁ (t , u) >β t
  π₂β  : π₂ (t , u) >β u
data _>s_ : Tm Γ A → Tm Δ B → Type where
  -- Structural ordering
  l·>  : t · u >s t
  ·r>  : t · u >s u
  λ̂>   : λ̂ t >s t

data _>βs_ : BTm → BTm → Type where
  β>  : t >β u → ⟪ t ⟫ >βs ⟪ u ⟫
  s>  : t >s u → ⟪ t ⟫ >βs ⟪ u ⟫
-- All terms are strongly normalisable w.r.t. _>βs_
wf : Π (t : Tm Γ A) → SN _>βs_ ⟪ t ⟫
```

Normalisation can then be made total by consistently returning evidence that there exists a (possibly empty) chain of reductions $\_>\beta^*\_$ to go from the input term to the resulting normal form.

We denote the transitive closure and reflexive-transitive closures of orders with $\_+$ and $\_\hat{*}$ respectively.

```
Nf> : Π Γ A → Tm Γ A → Type
Nf> Γ A t ≔ Σ (Nf Γ A) (λ tᴺᶠ → t >β* ⌜ tᴺᶠ ⌝)
```

Actually using our accessibility predicate in naive normalisation gets quite cluttered, but the main idea is to ensure that we are always making progress with respect to $\_>\beta s\_$.

We again skip the cases for pairs and the unit type here as they are routine.

```
norm   : Π (t : Tm Γ A) → SN _>βs+_ ⟪ t ⟫ → Nf> Γ A t
nf-app : Π (tᴺᶠ : Nf Γ (A ↣ B)) (uᴺᶠ : Nf Γ A)
       → SN _>βs+_ ⟪ t · u ⟫ → t · u >β* ⌜ tᴺᶠ ⌝ · ⌜ uᴺᶠ ⌝
       → Nf> Γ B (t · u)
norm (` i)    a ≔ ne (` i) , ε
norm (λ̂ t)    (acc a)
  using tᴺᶠ   , t>tᴺᶠ  ≡ norm t (a ⟪ s> λ̂> ⟫)
  ≔ (λ̂ tᴺᶠ) , λ̂* t>tᴺᶠ
norm (t · u)  (acc a)
  using tᴺᶠ   , t>tᴺᶠ   ≡ norm t (a ⟪ s> l·> ⟫)
     | uᴺᶠ , u>uᴺᶠ ≡ norm u (a ⟪ s> ·r> ⟫)
  ≔ nf-app tᴺᶠ uᴺᶠ (acc a) (t>tᴺᶠ ·* u>uᴺᶠ)
nf-app (ne t) u _ tu>tuᴺᶠ
  ≔ ne (t · u) , tu>tuᴺᶠ
nf-app (λ̂ t)  u (acc a) ε
  using tuᴺᶠ , tu>tuᴺᶠ ≡ norm (⌜ t ⌝ [ < ⌜ u ⌝ > ]) (a ⟪ β> β ⟫)
  ≔ tuᴺᶠ , β :: tu>tuᴺᶠ
nf-app (λ̂ t)  u (acc a) (p :: q)
  using tuᴺᶠ , tu>tuᴺᶠ ≡ norm (⌜ t ⌝ [ < ⌜ u ⌝ > ])
                              (a (β> p ::+ (map* _ β> q ∘* ⟪ β> β ⟫*)))
  ≔ tuᴺᶠ , (p :: q ∘* ⟪ β ⟫* ∘* tu>tuᴺᶠ)
```

1.3 The Standard Model

If our aim is to derive an algorithm which reduces terms while staying structurally recursive, our focus should be the case for application. i.e. when aiming to produce $\mathsf{Nf}\ \Gamma\ \mathsf{As}$ directly by recursion on our syntax, we failed to derive a structurally recursive algorithm because there is no analogue of $\_\cdot\_ : \mathsf{Tm}\ \Gamma\ (\mathsf{A} \hat{\to} \mathsf{B}) \to \mathsf{Tm}\ \Gamma\ \mathsf{A} \to \mathsf{Tm}\ \Gamma\ \mathsf{B}$ for normal forms.

As a step towards deriving an improved normalisation algorithm that gets around this problem, we will look at the similar but slightly easier problem of merely evaluating closed STLC terms. The key idea here will be to interpret STLC types into their metatheoretic (i.e. in **Type**) counterparts. This way, function-typed terms will be evaluated into meta-level functions, which can be directly applied to their evaluated arguments, while still satisfying expected β-equalities. This idea is known as the standard model of type theory (also the "meta-circular interpretation").

As values will not contain free variables, we will also interpret contexts as left-nested pair types (environments).

$$
\begin{aligned}
&\text{-- } \textit{Closed values}\\
&[\![\_]\!]^{\mathsf{Ty}} : \mathsf{Ty} \to \mathbf{Type}\\
&[\![\ \hat{0}\ ]\!]^{\mathsf{Ty}} \coloneqq \mathbb{0}\\
&[\![\ \mathsf{A} \hat{\to} \mathsf{B}\ ]\!]^{\mathsf{Ty}} \coloneqq [\![\ \mathsf{A}\ ]\!]^{\mathsf{Ty}} \to [\![\ \mathsf{B}\ ]\!]^{\mathsf{Ty}}\\
&[\![\ \mathsf{A} \hat{\times} \mathsf{B}\ ]\!]^{\mathsf{Ty}} \coloneqq [\![\ \mathsf{A}\ ]\!]^{\mathsf{Ty}} \times [\![\ \mathsf{B}\ ]\!]^{\mathsf{Ty}}\\
&[\![\ \hat{1}\ ]\!]^{\mathsf{Ty}} \coloneqq \mathbb{1}\\
&\text{-- } \textit{Environments}\\
&[\![\_]\!]^{\mathsf{Ctx}} : \mathsf{Ctx} \to \mathbf{Type}\\
&[\![\ \varepsilon\ ]\!]^{\mathsf{Ctx}} \coloneqq \mathbb{0}\\
&[\![\ \Gamma\ ,\ \mathsf{A}\ ]\!]^{\mathsf{Ctx}} \coloneqq [\![\ \Gamma\ ]\!]^{\mathsf{Ctx}} \times [\![\ \mathsf{A}\ ]\!]^{\mathsf{Ty}}
\end{aligned}
$$

Terms are then interpreted as functions from environments to closed values, so in non-empty contexts, variables can pick out their associated values. In other words, we can *evaluate* a term of type A in context Γ into a closed value of type A, $[\![\ \mathsf{A}\ ]\!]^{\mathsf{Ty}}$, given an environment $\rho : [\![\ \Gamma\ ]\!]^{\mathsf{Ctx}}$. Application directly applies values using application of the meta-theory and abstraction extends environments with new values, using abstraction of the meta. Given we are working inside of a constructive type theory, meta-functions are computable-by-construction and so termination is ensured merely by structural recursion on our syntax.

$$
\begin{aligned}
&\mathsf{lookup} : \mathsf{Var}\ \Gamma\ \mathsf{A} \to [\![\ \Gamma\ ]\!]^{\mathsf{Ctx}} \to [\![\ \mathsf{A}\ ]\!]^{\mathsf{Ty}}\\
&\mathsf{lookup}\ \mathsf{vz}\ \ (\rho\ ,\ \mathsf{x}) \coloneqq \mathsf{x}\\
&\mathsf{lookup}\ (\mathsf{vs}\ \mathsf{i})\ (\rho\ ,\ \mathsf{x}) \coloneqq \mathsf{lookup}\ \mathsf{i}\ \rho\\
&[\![\_]\!]^{\mathsf{Tm}} : \mathsf{Tm}\ \Gamma\ \mathsf{A} \to [\![\ \Gamma\ ]\!]^{\mathsf{Ctx}} \to [\![\ \mathsf{A}\ ]\!]^{\mathsf{Ty}}\\
&[\![\ \hat{\lambda}\ \mathsf{t}\ ]\!]^{\mathsf{Tm}}\ \rho \coloneqq \lambda\ \mathsf{x} \to [\![\ \mathsf{t}\ ]\!]^{\mathsf{Tm}}\ (\rho\ ,\ \mathsf{x})\\
&[\![\ \mathsf{t} \cdot \mathsf{u}\ ]\!]^{\mathsf{Tm}}\ \rho \coloneqq ([\![\ \mathsf{t}\ ]\!]^{\mathsf{Tm}}\ \rho)\ ([\![\ \mathsf{u}\ ]\!]^{\mathsf{Tm}}\ \rho)\\
&[\![\ \mathsf{t}\ ,\ \mathsf{u}\ ]\!]^{\mathsf{Tm}}\ \rho \coloneqq ([\![\ \mathsf{t}\ ]\!]^{\mathsf{Tm}}\ \rho)\ ,\ ([\![\ \mathsf{u}\ ]\!]^{\mathsf{Tm}}\ \rho)\\
&[\![\ \pi_1\ \mathsf{t}\ ]\!]^{\mathsf{Tm}}\ \rho \coloneqq \mathsf{proj}_1\ ([\![\ \mathsf{t}\ ]\!]^{\mathsf{Tm}}\ \rho)\\
&[\![\ \pi_2\ \mathsf{t}\ ]\!]^{\mathsf{Tm}}\ \rho \coloneqq \mathsf{proj}_2\ ([\![\ \mathsf{t}\ ]\!]^{\mathsf{Tm}}\ \rho)\\
&[\![\ \langle\rangle\ ]\!]^{\mathsf{Tm}}\ \rho \coloneqq \langle\rangle\\
&[\![\ \grave{}\ \mathsf{i}\ ]\!]^{\mathsf{Tm}}\ \rho \coloneqq \mathsf{lookup}\ \mathsf{i}\ \rho
\end{aligned}
$$

Of course, this algorithm is not sufficient for normalisation. Without an environment of closed values to evaluate with respect to, we cannot hope

to inspect the structure of evaluated terms (i.e. at the meta-level, functions like $[\![\ \Gamma\ ]\!]^{\mathsf{Ctx}} \to [\![\ \mathsf{A}\ ]\!]^{\mathsf{Ty}}$ are opaque). Similarly, even with an environment, we cannot inspect the structure of higher order ($\hat{\to}$-typed) values beyond testing their behaviour on particular inputs given these are again opaque meta-language functions. The "problem" we are encountering is that our values have no first-order representation of variables.

It turns out, by carefully defining a similar model, based on presheaves, we can embed stuck, first-order variables into values[1] , implement evaluation in open contexts and even *invert* evaluation, "quoting" back into normalised first-order terms (i.e. our normal forms). This *evaluation* followed by *quoting* is exactly normalisation by evaluation.

1: In fact, we are forced to include general, stuck neutral terms to support application where the LHS is a variable.

1.4 The Presheaf Model

Central to the presheaf model (perhaps unsurprisingly) is the concept of a presheaves: contravariant functors into **Type**. We will specifically focus on presheaves on the category of renamings. Being able to weaken values (i.e. introduce new fresh variables) via renamings will be critical when defining quotation into normal forms.

The objects in the category of renamings are contexts Ctx and the morphisms are renamings $\mathsf{Ren}\ \Delta\ \Gamma$.

```
-- Presheaves on the category of renamings
record PshRen (F : Ctx → Type) : Type where
  field
    ren     : Ren Δ Γ → F Γ → F Δ
    ren-id  : Π {x} → ren (id {Γ ≔ Γ}) x  =  x
    ren-;   : Π {x} → ren δ (ren σ x)  =  ren (δ ; σ) x
```

Note that renamings are not the only option here. "Thinnings" are a subset of renamings where variables of the target can only be retained or dropped (not permuted or duplicated) and yet still form a category and encompass weakenings.

The standard model can be seen as interpreting object-level types into the corresponding objects in the category **Type** (i.e. where **Type**s are objects and functions are morphisms). In the presheaf model, we instead interpret into corresponding objects in the category of presheaves (a category where objects are presheaves, and morphisms are natural transformations).

For example, the unit presheaf (i.e. the terminal object in the category of presheaves) is simply $\mathbb{1}^{Psh} \coloneqq \lambda\ \Gamma \to \mathbb{1}$. Similarly, the products in the category of presheaves can be constructed as $\mathsf{F} \times^{Psh} \mathsf{G} \coloneqq \lambda\ \Gamma \to \mathsf{F}\ \Gamma \times \mathsf{G}\ \Gamma$.

The exponential object in the category of presheaves is a bit more subtle. We might try to follow the pattern and define $\mathsf{F} \to^{Psh} \mathsf{G} \coloneqq \lambda\ \Gamma \to \mathsf{F}\ \Gamma \to^{Psh} \mathsf{G}\ \Gamma$ but this doesn't quite work. When trying to implement $\mathsf{ren} : \mathsf{Ren}\ \Delta\ \Gamma \to (\mathsf{F} \to^{Psh} \mathsf{G})\ \Gamma \to (\mathsf{F} \to^{Psh} \mathsf{G})\ \Delta$ we only have access to an $\mathsf{F}\ \Delta$ and a function which accepts $\mathsf{F}\ \Gamma$s[2] . The solution is to quantify over renamings, i.e. $\mathsf{F} \to^{Psh} \mathsf{G} \coloneqq \lambda\ \Gamma \to \Pi\ \{\Delta\} \to \mathsf{Ren}\ \Delta\ \Gamma \to \mathsf{F}\ \Delta \to \mathsf{G}\ \Delta$ [16].

2: Note the $\mathsf{Ren}\ \Delta\ \Gamma$ renaming can only transform $\mathsf{F}\ \Gamma$s into $\mathsf{F}\ \Delta$s, not the other way around.

[16]: 1Lab Development Team (2025), *Exponential objects in presheaf categories*

These are (almost) all the ingredients we need to define NbE values. Types in a context Γ are merely interpreted as the corresponding constructs in the category of presheaves.

$$
\begin{aligned}
&[\![\_]\!]^{\mathsf{Psh}} : \mathsf{Ty} \to \mathsf{Ctx} \to \mathbf{Type} \\
&[\![\ \mathsf{A} \hat{\to} \mathsf{B}\ ]\!]^{\mathsf{Psh}}\ \Gamma \coloneqq \Pi\ \{\Delta\} \to \mathsf{Ren}\ \Delta\ \Gamma \to [\![\ \mathsf{A}\ ]\!]^{\mathsf{Psh}}\ \Delta \to [\![\ \mathsf{B}\ ]\!]^{\mathsf{Psh}}\ \Delta \\
&[\![\ \mathsf{A} \hat{\times} \mathsf{B}\ ]\!]^{\mathsf{Psh}}\ \Gamma \coloneqq [\![\ \mathsf{A}\ ]\!]^{\mathsf{Psh}}\ \Gamma \times [\![\ \mathsf{B}\ ]\!]^{\mathsf{Psh}}\ \Gamma \\
&[\![\ \hat{\mathbb{1}}\ ]\!]^{\mathsf{Psh}}\ \Gamma \coloneqq \mathbb{1} \\
&[\![\ \hat{\mathbb{0}}\ ]\!]^{\mathsf{Psh}}\ \Gamma \coloneqq \mathbb{0}
\end{aligned}
$$

A final subtlety arises with the empty type $\hat{\mathbb{0}}$. While $\lambda\ \Gamma \to \mathbb{0}$ does satisfy all the necessary laws of an initial object, and terms of type $\hat{\mathbb{0}}$ can only occur inside empty contexts (i.e. contexts containing $\hat{\mathbb{0}}$), when evaluating a variable of type $\hat{\mathbb{0}}$, we cannot hope to produce a proof of $\mathbb{0}$ (i.e. the context containing the empty type does not mean evaluation can give up - normalisation requires evaluating in all contexts).

To solve this, we must embed neutrals into the model. E.g. we could interpret $\hat{\mathbb{0}}$ as $\lambda\ \Gamma \to \mathsf{Ne}\ \Gamma\ \hat{\mathbb{0}}$. $\lambda\ \Gamma \to \mathsf{Ne}\ \Gamma\ \hat{\mathbb{0}}$ is obviously not an initial object in the category of presheaves, so by doing this we have slightly broken our model, but it turns out that only the η laws for $\hat{\mathbb{0}}$ are actually lost (which lines up exactly with the consequences of embedding neutrals to Nf). We are aiming only to β-normalise terms, and will actually take a more extreme option, embedding neutrals of all types as to line up more closely with our β-normal forms.

This issue is not unique to the empty type. All "positive" types, e.g. booleans, coproducts, natural numbers etc... experience a similar problem. [17] explores using a model based on sheaves (instead of presheaves) to fix this more elegantly in the case of coproducts, but in general (e.g. for natural numbers) deciding "categorical" ($\beta\eta$) equivalence is undecidable.

[17]: Altenkirch et al. (2001), *Normalization by evaluation for typed lambda calculus with coproducts*

```
Val    : Ctx → Ty → Type
PshVal : Ctx → Ty → Type

Val Γ A ≔ PshVal Γ A + Ne Γ A

PshVal Γ (A ⇀̂ B) ≔ Π {Δ} → Ren Δ Γ → Val Δ A → Val Δ B
PshVal Γ (A ×̂ B) ≔ Val Γ A × Val Γ B
PshVal Γ 𝟙̂       ≔ 𝟙
PshVal Γ 𝟘̂       ≔ 𝟘
```

Note that although we are mixing inductively (i.e. Ne) and recursively (i.e. PshVal) defined type families here, the construction remains well-founded.

Renaming can now be implemented for PshVal Γ A by recursion on the type A, and renaming of values in general, renVal, can merely delagate renaming on PshVal Γ As and Ne Γ As as appropriate.

```
renVal : Ren Δ Γ → Val Γ A → Val Δ A
renPshVal : Π A → Ren Δ Γ → PshVal Γ A → PshVal Δ A
renVal δ (inl t) ≔ inl (renPshVal _ δ t)
renVal δ (inr t) ≔ inr (renNe      δ t)
renPshVal (A ⇀̂ B) δ f      ≔ λ σ t → f (δ ; σ) t
renPshVal (A ×̂ B) δ (t , u) ≔ renVal δ t , renVal δ u
renPshVal 𝟙̂       δ ⟨⟩     ≔ ⟨⟩
```

To implement NbE, we need to define both evaluation from terms to values and "quotation" from values to normal forms. We start with evaluation, which is quite similar to $[\![\_]\!]^{\mathsf{Tm}}$ in section Section 1.3, but needs to deal with the cases for stuck neutrals appropriately.

We start by defining NbE environments, which unlike the standard model are now parameterised by two contexts, similarly to Ren/Sub: first, the context each of the values exist in and second the list of types of the values themselves.

```
data Env : Ctx → Ctx → Type where
  ε   : Env Δ ε
  _,_ : Env Δ Γ → Val Δ A → Env Δ (Γ , A)
```

Note that environments can be renamed by simply folding renVal.

```
renEnv : Ren Θ Δ → Env Δ Γ → Env Θ Γ
```

Evaluation then proceeds by recursion on the target term. The main subtlety is in application of values, where the LHS is neutral. In this case we need to turn quote the RHS back to an Nf via qval to apply $\_\cdot\_ : \mathsf{Ne}\ \Gamma\ (\mathsf{A} \hat{\to} \mathsf{B}) \to \mathsf{Nf}\ \Gamma\ \mathsf{A} \to \mathsf{Ne}\ \Gamma\ \mathsf{B}$ (i.e. evaluation actually depends on quotation).

$$
\begin{aligned}
&\mathsf{qval} : \Pi\ \mathsf{A} \to \mathsf{Val}\ \Gamma\ \mathsf{A} \to \mathsf{Nf}\ \Gamma\ \mathsf{A}\\
&\mathsf{lookupVal} : \mathsf{Var}\ \Gamma\ \mathsf{A} \to \mathsf{Env}\ \Delta\ \Gamma \to \mathsf{Val}\ \Delta\ \mathsf{A}\\
&\mathsf{lookupVal}\ \mathsf{vz}\ \ (\rho\ ,\ \mathsf{x}) \coloneqq \mathsf{x}\\
&\mathsf{lookupVal}\ (\mathsf{vs}\ \mathsf{i})\ (\rho\ ,\ \mathsf{x}) \coloneqq \mathsf{lookupVal}\ \mathsf{i}\ \rho\\
&\mathsf{appVal} : \mathsf{Val}\ \Gamma\ (\mathsf{A} \hat{\to} \mathsf{B}) \to \mathsf{Val}\ \Gamma\ \mathsf{A} \to \mathsf{Val}\ \Gamma\ \mathsf{B}\\
&\mathsf{appVal}\ (\mathsf{inl}\ \mathsf{f})\ \mathsf{x} \coloneqq \mathsf{f}\ \mathsf{id}\ \mathsf{x}\\
&\mathsf{appVal}\ (\mathsf{inr}\ \mathsf{t})\ \mathsf{x} \coloneqq \mathsf{inr}\ (\mathsf{t} \cdot \mathsf{qval}\ \_\ \mathsf{x})\\
&\pi_1\mathsf{Val} : \mathsf{Val}\ \Gamma\ (\mathsf{A} \hat{\times} \mathsf{B}) \to \mathsf{Val}\ \Gamma\ \mathsf{A}\\
&\pi_1\mathsf{Val}\ (\mathsf{inl}\ (\mathsf{t}\ ,\ \mathsf{u})) \coloneqq \mathsf{t}\\
&\pi_1\mathsf{Val}\ (\mathsf{inr}\ \mathsf{t}) \coloneqq \mathsf{inr}\ (\pi_1\ \mathsf{t})\\
&\pi_2\mathsf{Val} : \mathsf{Val}\ \Gamma\ (\mathsf{A} \hat{\times} \mathsf{B}) \to \mathsf{Val}\ \Gamma\ \mathsf{B}\\
&\pi_2\mathsf{Val}\ (\mathsf{inl}\ (\mathsf{t}\ ,\ \mathsf{u})) \coloneqq \mathsf{u}\\
&\pi_2\mathsf{Val}\ (\mathsf{inr}\ \mathsf{t}) \coloneqq \mathsf{inr}\ (\pi_2\ \mathsf{t})\\
&\mathsf{eval} : \mathsf{Tm}\ \Gamma\ \mathsf{A} \to \mathsf{Env}\ \Delta\ \Gamma \to \mathsf{Val}\ \Delta\ \mathsf{A}\\
&\mathsf{eval}\ (\grave{}\ \mathsf{i})\ \ \rho \coloneqq \mathsf{lookupVal}\ \mathsf{i}\ \rho\\
&\mathsf{eval}\ (\hat{\lambda}\ \mathsf{t})\ \ \rho \coloneqq \mathsf{inl}\ \lambda\ \delta\ \mathsf{u} \to \mathsf{eval}\ \mathsf{t}\ (\mathsf{renEnv}\ \delta\ \rho\ ,\ \mathsf{u})\\
&\mathsf{eval}\ (\mathsf{t} \cdot \mathsf{u})\ \rho \coloneqq \mathsf{appVal}\ (\mathsf{eval}\ \mathsf{t}\ \rho)\ (\mathsf{eval}\ \mathsf{u}\ \rho)\\
&\mathsf{eval}\ (\mathsf{t}\ ,\ \mathsf{u})\ \rho \coloneqq \mathsf{inl}\ (\mathsf{eval}\ \mathsf{t}\ \rho\ ,\ \mathsf{eval}\ \mathsf{u}\ \rho)\\
&\mathsf{eval}\ (\pi_1\ \mathsf{t})\ \rho \coloneqq \pi_1\mathsf{Val}\ (\mathsf{eval}\ \mathsf{t}\ \rho)\\
&\mathsf{eval}\ (\pi_2\ \mathsf{t})\ \rho \coloneqq \pi_2\mathsf{Val}\ (\mathsf{eval}\ \mathsf{t}\ \rho)\\
&\mathsf{eval}\ \langle\rangle\ \ \rho \coloneqq \mathsf{inl}\ \langle\rangle
\end{aligned}
$$

To implement qval, we instead proceed by recursion on types. Being able to rename values is critical to quoting back function values, where to inspect their structure, we need to be able to apply them to a fresh variable vz.

$$
\begin{aligned}
&\mathsf{qval}\ \mathsf{A}\ \ (\mathsf{inr}\ \mathsf{t}) \coloneqq \mathsf{ne}\ \mathsf{t}\\
&\mathsf{qval}\ (\mathsf{A} \hat{\to} \mathsf{B})\ (\mathsf{inl}\ \mathsf{f}) \coloneqq \hat{\lambda}\ \mathsf{qval}\ \mathsf{B}\ (\mathsf{f}\ \mathsf{wk}\ (\mathsf{inr}\ (\grave{}\ \mathsf{vz})))\\
&\mathsf{qval}\ (\mathsf{A} \hat{\times} \mathsf{B})\ (\mathsf{inl}\ (\mathsf{t}\ ,\ \mathsf{u})) \coloneqq \mathsf{qval}\ \mathsf{A}\ \mathsf{t}\ ,\ \mathsf{qval}\ \mathsf{B}\ \mathsf{u}\\
&\mathsf{qval}\ \hat{\mathbb{1}}\ \ (\mathsf{inl}\ \langle\rangle) \coloneqq \langle\rangle
\end{aligned}
$$

Normalisation of open terms now only needs a way to construct identity environments (effectively lists of increasing variables):

$$
\begin{aligned}
&\mathsf{idEnv} : \mathsf{Env}\ \Gamma\ \Gamma\\
&\mathsf{idEnv}\ \{\Gamma \coloneqq \varepsilon\} \coloneqq \varepsilon\\
&\mathsf{idEnv}\ \{\Gamma \coloneqq \Gamma\ ,\ \mathsf{A}\} \coloneqq \mathsf{renEnv}\ \mathsf{wk}\ \mathsf{idEnv}\ ,\ \mathsf{inr}\ (\grave{}\ \mathsf{vz})\\
&\mathsf{nbe} : \mathsf{Tm}\ \Gamma\ \mathsf{A} \to \mathsf{Nf}\ \Gamma\ \mathsf{A}\\
&\mathsf{nbe}\ \mathsf{t} \coloneqq \mathsf{qval}\ \_\ (\mathsf{eval}\ \mathsf{t}\ \mathsf{idEnv})
\end{aligned}
$$

We are done! Of course, to verify our normalisation algorithm is correct, we need to do more work, checking soundness and completeness. One way to achieve this is start with a syntax quotiented by conversion (guaranteeing soundness) and refine values into proof-relevant predicates indexed by the unnormalised term, paired with the relevant correctness conditions [3].

[3]: Altenkirch et al. (2017), *Normalisation by evaluation for type theory, in type theory*

1.4.1 NbE for Dependent Types

In the setting of dependent types, the main difference is of course that types may contain terms. [3] implements quotation on non-normalised types,

though extending this approach to more involved type-level computation (their syntax includes only El : Tm Γ U → Ty Γ with no additional equations) requires a bit of extra work. E.g. if we were in a dependent type theory featuring large elimination of booleans and encountered the type if t then A else B while quoting, we must first evaluate t, and potentially recursively quote at type A or B if t turns out to reduce to a closed boolean (if the type ends up a stuck neutral, we are at least guaranteed that the only possible values are neutral).

Luckily, NbE in this project is merely employed as an algorithm for implementing typechecking, in the partial language of Haskell. We can therefore define NbE on an untyped syntax, relying on the external invariant that, in practice, we will only call eval on terms we have already been type-checked. In the next section, we will cover the tweaks we can make to NbE in this partial setting, retaining equivalent operational behaviour but making the algorithm more convenient to implement and more efficient.

1.5 NbE in Haskell: Optimisations and Design Decisions

TODO!

1.5.1 "Inductively"-defined Values

TODO! The general idea is defining values as a non-positive datatype with e.g. constructors like VLam : Ren → Val → Val instead of by recursion on object types (which isn't really possible in a non-dependently-typed setting).

1.5.2 Avoiding Quotation during Evaluation

TODO! The general idea is to define "neutral values", which are also non-positive, but by examining the algorithm we can see that the operational behaviour ends up the same.

Should probably also discuss how it is possible to decide conversion on values directly (i.e. fusing conversion-checking and quoting).

1.5.3 Renamings vs Thinnings

TODO! The general idea is that for quoting, it is actually sufficient for NbE values to form a presheaf on the category of "thinnings" (a subset of renamings where new variables are only inserted between existing ones - i.e. no permuting or duplicating variables etc...). Thinnings (especially the variant where identity is made a constructor) can be applied more efficiently than renamings (check easily check for id and short-circuit).

1.5.4 De Bruijn Levels

TODO! General idea is to represent variables in values with de Bruijn *levels* rather than *indices*, such that variables count the number of binders between their binding site and the root of the term (rather than their binding site and their use). This makes inserting fresh variables (i.e. the presheaf stuff we needed for quoting to work) no longer require a full traversal of the value.

1.5.5 Explicit Closures

TODO! I don't currently plan on implementing this optimisation, but it is still probably worth mentioning. It turns out the operational behaviour of the NbE algorithm can be replicated without meta-language closures entirely! Closures can be represented in a first-order fashion by pairing un-evaluated terms and captured environments. This variation is no longer structurally recursive (we need to `eval` the closure bodies during applications, similarly to naive normalisation) but can be faster on than relying on meta-closures depending on implementation language/runtime.

1.6 NbE in Haskell: Local Equations

TODO! The general idea is just to track a map of neutrals to values and lookup neutrals in the map when necessary. Function values need to be parameterised by updated maps to reduce properly in contexts with new equations.

Bibliography

[1] U Berger and H Schwichtenberg. "An inverse of the evaluation functional for typed lambda-calculus". In: *Proceedings 1991 Sixth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society. 1991, pp. 203–204 (cited on page 1).

[2] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. "Categorical reconstruction of a reduction free normalization proof". In: *Category Theory and Computer Science: 6th International Conference, CTCS'95 Cambridge, United Kingdom, August 7–11, 1995 Proceedings 6*. Springer. 1995, pp. 182–199 (cited on page 1).

[3] Thorsten Altenkirch and Ambrus Kaposi. "Normalisation by evaluation for type theory, in type theory". In: *Logical methods in computer science* 13 (2017) (cited on pages 1, 9).

[4] Ambrus Kaposi and Loïc Pujet. "Type Theory in Type Theory Using a Strictified Syntax". Preprint available at `https://pujet.fr/`. 2025 (cited on page 1).

[5] Nils Anders Danielsson. "A formalisation of a dependently typed language as an inductive-recursive family". In: *International Workshop on Types for Proofs and Programs*. Springer. 2006, pp. 93–109 (cited on page 1).

[6] András Kovács. *TTinTTasSetoid.agda*. 2023. URL: `https://gist.github.com/AndrasKovacs/4fcafec4c97fc1af75210f65c20e624d` (visited on 04/05/2025) (cited on page 1).

[7] András Kovács. *smalltt*. 2023. URL: `https://github.com/AndrasKovacs/smalltt` (visited on 03/07/2025) (cited on page 1).

[8] Thierry Coquand. "An algorithm for type-checking dependent types". In: *Science of computer programming* 26.1-3 (1996), pp. 167–177 (cited on page 1).

[9] Conor McBride. *Mastodon*. 2025. URL: `https://types.pl/@pigworker/114087501391646354` (visited on 03/07/2025) (cited on page 2).

[10] Nicolaas Govert De Bruijn. "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem". In: *Indagationes mathematicae (proceedings)*. Vol. 75. 5. Elsevier. 1972, pp. 381–392 (cited on page 2).

[11] András Kovács. "A machine-checked correctness proof of normalization by evaluation for simply typed lambda calculus". MA thesis. MS thesis, Eötvös Loránd University, Budapest, 2017.[Online]. Available . . ., 2017 (cited on page 2).

[12] Thorsten Altenkirch, Nathaniel Burke, and Philip Wadler. "Substitution without copy and paste". Available at https://github.com/txa/substitution/blob/main. 2025 (cited on page 3).

[13] Gabriel Scherer. "Deciding equivalence with sums and the empty type". In: *ACM SIGPLAN Notices* 52.1 (2017), pp. 374–386 (cited on page 4).

[14] Chantal Keller and Thorsten Altenkirch. "Hereditary substitutions for simple types, formalized". In: *Proceedings of the third ACM SIGPLAN workshop on Mathematically structured functional programming*. 2010, pp. 3–10 (cited on page 4).

[15] Allais Guillaume and Naïm Camille Favier. *Combining Accessibility Proofs and Structural Ordering*. 2024. URL: `https://agda.zulipchat.com/#narrow/channel/238741-general/topic/Combining.20Accessibility.20Proofs.20and.20Structural.20Ordering` (cited on page 4).

[16] The 1Lab Development Team. *Exponential objects in presheaf categories*. 2025. URL: `https://1lab.dev/Cat.Instances.Presheaf.Exponentials.html` (visited on 04/05/2025) (cited on page 7).

[17] Thorsten Altenkirch et al. "Normalization by evaluation for typed lambda calculus with coproducts". In: *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*. IEEE. 2001, pp. 303–310 (cited on page 8).