A Normalisation-by-Evaluation Program for Typed Lambda Calculus in Agda

Marcelo Fiore
Department of Computer Science and Technology
University of Cambridge

July 2022

Abstract

This note presents a normalisation-by-evaluation program for typed lambda calculus in the dependently-typed functional programming language Agda synthesised from my *Semantic Analysis of Normalisation by Evaluation for Typed Lambda Calculus* (PPDP'02: Proceedings of the 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, October 2002).

Syntax. We consider simple types over a countably infinite set of base types:

```
-- base types
T : Set
T = Nat

-- simple types
data T̃ : Set where
  θ : T → T̃
  1 : T̃
  _*_ : T̃ → T̃ → T̃
  _⇒_ : T̃ → T̃ → T̃
```

Typing contexts are inductively generated by context extension from an empty context:

```
-- typing contexts
data 𝔽↓ : Set → Set where
  · : {𝒯 : Set} → 𝔽↓𝒯
  _::_ : {𝒯 : Set} → 𝔽↓𝒯 → 𝒯 → 𝔽↓𝒯
```

We then have a family of variable indices given as follows:

```
-- variable indices
data V : {𝒯 : Set} → 𝒯 → 𝔽↓𝒯 → Set where
  • : {𝒯 : Set} {τ : 𝒯} {Γ : 𝔽↓𝒯} → V τ (Γ :: τ)
  ↑ : {𝒯 : Set} {τ σ : 𝒯} {Γ : 𝔽↓𝒯} → V σ Γ → V σ (Γ :: τ)
```

for which context renamings are considered:

```
-- context renamings
𝔽↓ : (T̃ : Set) → 𝔽↓T̃ × 𝔽↓T̃ → Set
𝔽↓T̃ (Δ , Γ ) = {τ : T̃} → V τ Δ → V τ Γ
```

The abstract syntax of simply typed terms is implemented by the inductive family below:

```
-- simply typed terms
data ℒ : T̃ → 𝔽↓T̃ → Set where
  var : {τ : T̃} {Γ : 𝔽↓T̃} → V τ Γ → ℒ τ Γ
  unit : {Γ : 𝔽↓T̃} → ℒ 1 Γ
  pair : {τ σ : T̃} {Γ : 𝔽↓T̃} → ℒ τ Γ → ℒ σ Γ → ℒ (τ ∗ σ) Γ
  fst : {τ σ : T̃} {Γ : 𝔽↓T̃} → ℒ (τ ∗ σ) Γ → ℒ τ Γ
  snd : {τ σ : T̃} {Γ : 𝔽↓T̃} → ℒ (τ ∗ σ) Γ → ℒ σ Γ
  abs : {τ σ : T̃} {Γ : 𝔽↓T̃} → ℒ σ (Γ :: τ ) → ℒ (τ ⇛ σ) Γ
  app : {τ σ : T̃} {Γ : 𝔽↓T̃} → ℒ (τ ⇛ σ) Γ → ℒ τ Γ → ℒ σ Γ
```

Analogously, the abstract syntax of neutral and normal terms is implemented by the following mutually-inductive families:

```
-- neutral and normal terms
mutual
  data ℳ : T̃ → 𝔽↓T̃ → Set where
    varₘ : {τ : T̃} {Γ : 𝔽↓T̃} → V τ Γ → ℳ τ Γ
    fstₘ : {τ σ : T̃} {Γ : 𝔽↓T̃} → ℳ (τ ∗ σ) Γ → ℳ τ Γ
    sndₘ : {τ σ : T̃} {Γ : 𝔽↓T̃} → ℳ (τ ∗ σ) Γ → ℳ σ Γ
    appₘ : {τ σ : T̃} {Γ : 𝔽↓T̃} → ℳ (τ ⇛ σ) Γ → 𝒩 τ Γ → ℳ σ Γ

  data 𝒩 : T̃ → 𝔽↓T̃ → Set where
    varₙ : {i : T} {Γ : 𝔽↓T̃} → V (θ i) Γ → 𝒩 (θ i) Γ
    fstₙ : {i : T} {σ : T̃} {Γ : 𝔽↓T̃} → ℳ (θ i ∗ σ) Γ → 𝒩 (θ i) Γ
    sndₙ : {i : T} {τ : T̃} {Γ : 𝔽↓T̃} → ℳ (τ ∗ θ i) Γ → 𝒩 (θ i) Γ
    appₙ : {i : T} {τ : T̃} {Γ : 𝔽↓T̃} → ℳ (τ ⇛ θ i) Γ → 𝒩 τ Γ → 𝒩 (θ i) Γ
    unitₙ : {Γ : 𝔽↓T̃} → 𝒩 1 Γ
    pairₙ : {τ σ : T̃} {Γ : 𝔽↓T̃} → 𝒩 τ Γ → 𝒩 σ Γ → 𝒩 (τ ∗ σ) Γ
    absₙ : {τ σ : T̃} {Γ : 𝔽↓T̃} → 𝒩 σ (Γ :: τ) → 𝒩 (τ ⇛ σ) Γ
```

Their presheaf actions will be needed:

```
-- neutral and normal presheaf actions
mutual
  _[_]ₘ : {τ : T̃} {Δ Γ : 𝔽↓T̃} → ℳ τ Δ → 𝔽↓T̃( Δ , Γ ) → ℳ τ Γ
  varₘ x [ ρ ]ₘ = varₘ ( ρ x )
  fstₘ m [ ρ ]ₘ = fstₘ ( m [ ρ ]ₘ )
  sndₘ m [ ρ ]ₘ = sndₘ ( m [ ρ ]ₘ )
  appₘ m n [ ρ ]ₘ = appₘ ( m [ ρ ]ₘ ) ( n [ ρ ]ₙ )

  _[_]ₙ : {τ : T̃} {Δ Γ : 𝔽↓T̃} → 𝒩 τ Δ → 𝔽↓T̃( Δ , Γ ) → 𝒩 τ Γ
  varₙ x [ ρ ]ₙ = varₙ ( ρ x )
  fstₙ m [ ρ ]ₙ = fstₙ ( m [ ρ ]ₘ )
  sndₙ m [ ρ ]ₙ = sndₙ ( m [ ρ ]ₘ )
  appₙ m n [ ρ ]ₙ = appₙ ( m [ ρ ]ₘ ) ( n [ ρ ]ₙ )
  unitₙ [ ρ ]ₙ = unitₙ
  pairₙ n₁ n₂ [ ρ ]ₙ = pairₙ ( n₁ [ ρ ]ₙ ) ( n₂ [ ρ ]ₙ )
  absₙ n [ ρ ]ₙ = absₙ ( n [ lift ρ ]ₙ )
              where
```

```
lift : {τ : T̃} {Δ Γ : F↓T̃} → F↓T̃( Δ , Γ ) → F↓T̃( Δ :: τ , Γ :: τ )
lift _ • = •
lift ρ (↑ x) = ↑(ρ x)
```

Semantics. We implement the presheaf semantics of types induced by the interpretation of base types as neutral terms.

```
-- type semantics
⟦_⟧ : T̃ → F↓T̃ → Set
⟦ θ i ⟧ Γ = M (θ i) Γ
⟦ 1 ⟧ _ = ⊤
⟦ τ * σ ⟧ Γ = ⟦ τ ⟧ Γ × ⟦ σ ⟧ Γ
⟦ τ ⇒ σ ⟧ Γ = {Δ : F↓T̃} → F↓T̃( Γ , Δ ) → ⟦ τ ⟧ Δ → ⟦ σ ⟧ Δ

_[_] : {τ : T̃} {Δ Γ : F↓T̃} → ⟦ τ ⟧ Δ → F↓T̃( Δ , Γ ) → ⟦ τ ⟧ Γ
_[_] {θ _} = _[_]m
_[_] {1} = _
_[_] {_ * _} ( x1 , x2 ) ρ = ( x1 [ ρ ] , x2 [ ρ ] )
_[_] {_ ⇒ _} f ρ ρ′ = f (ρ′ ∘ ρ)
```

The semantic interpretation of terms follows:

```
-- term semantics
Π : F↓T̃ → (T̃ → F↓T̃ → Set) → F↓T̃ → Set
Π · _ _ = ⊤
Π (Γ :: τ) P Δ = (Π Γ P Δ) × (P τ Δ)

⟦_⟧ : {τ : T̃} {Γ : F↓T̃} → L τ Γ → {Δ : F↓T̃} → Π Γ ⟦_⟧ Δ → ⟦ τ ⟧ Δ
⟦ var x ⟧ ε = ε ⟨ x ⟩
          where
            _⟨_⟩ : {τ : T̃} {Γ Δ : F↓T̃} → Π Γ ⟦_⟧ Δ → V τ Γ → ⟦ τ ⟧ Δ
            ε ⟨ • ⟩ = π2 ε
            ε ⟨ ↑ x ⟩ = π1 ε ⟨ x ⟩
⟦ unit ⟧ _ = _
⟦ pair t1 t2 ⟧ ε = ( ⟦ t1 ⟧ ε , ⟦ t2 ⟧ ε )
⟦ fst t ⟧ ε = π1 ( ⟦ t ⟧ ε )
⟦ snd t ⟧ ε = π2 ( ⟦ t ⟧ ε )
⟦ abs t ⟧ ε f x = ⟦ t ⟧ ( ε [ f ]Π , x )
            where
              _[_]Π : {Ξ Δ Γ : F↓T̃} → Π Ξ ⟦_⟧ Δ → F↓T̃( Δ , Γ ) → Π Ξ ⟦_⟧ Γ
              _[_]Π {·} = _
              _[_]Π {_ :: _} ( ε , x ) ρ = ( ε [ ρ ]Π , x [ ρ ] )
⟦ app t1 t2 ⟧ ε = ⟦ t1 ⟧ ε (λ x → x) ( ⟦ t2 ⟧ ε )
```

Normalisation by evaluation. The unquote and quote functions are implemented:

```
-- unquote and quote
mutual
  u : {τ : T̃} {Γ : F↓T̃} → M τ Γ → ⟦ τ ⟧ Γ
  u {θ _} m = m
  u {1} _ = _
  u {_ * _} m = ( u (fstm m) , u (sndm m) )
```

```
u {_ ⇒ _} m ρ x = u ( appm (m [ ρ ]m) (q x) )

q : {τ : T̃} {Γ : F↓T̃} → ⟦ τ ⟧ Γ → 𝒩 τ Γ
q {θ _} (varm x) = varn x
q {θ _} (fstm m) = fstn m
q {θ _} (sndm m) = sndn m
q {θ _} (appm m n) = appn m n
q {1} _ = unitn
q {_ * _} (x1 , x2) = pairn (q x1) (q x2)
q {_ ⇒ _} f = absn( q( f ↑ (u (varm •)) ) )
```

Finally, the normalisation function is:

```
-- nbe
nf : {τ : T̃} {Γ : F↓T̃} → ℒ τ Γ → 𝒩 τ Γ
nf t = q ( ⟦ t ⟧ ( Π (u ∘ varm) xs ) )
   where
     Π : (f : {τ : T̃} {Δ : F↓T̃} → V τ Δ → ⟦_⟧ τ Δ) {Γ Δ : F↓T̃} → Π Γ V Δ → Π Γ ⟦_⟧ Δ
     Π _ {·} _ = _
     Π f {_ :: _} ( xs , x ) = ( Π f xs , f x )

     xs : {Γ : F↓T̃} → Π Γ V Γ
     xs {·} = _
     xs {_ :: _} = ( xs [ ↑ ]v , • )
            where
              _[_]v : {Ξ Δ Γ : F↓T̃} → Π Ξ V Δ → F↓T̃( Δ , Γ ) → Π Ξ V Γ
              _[_]v {·} = _
              _[_]v {_ :: _} ( xs , x ) ρ = ( xs [ ρ ]v , ρ x )
```