Nameless, Painless

Nicolas Pouillard

INRIA

ICFP 2011, Tokyo
September 21, 2011

Safe programming with de Bruijn indices

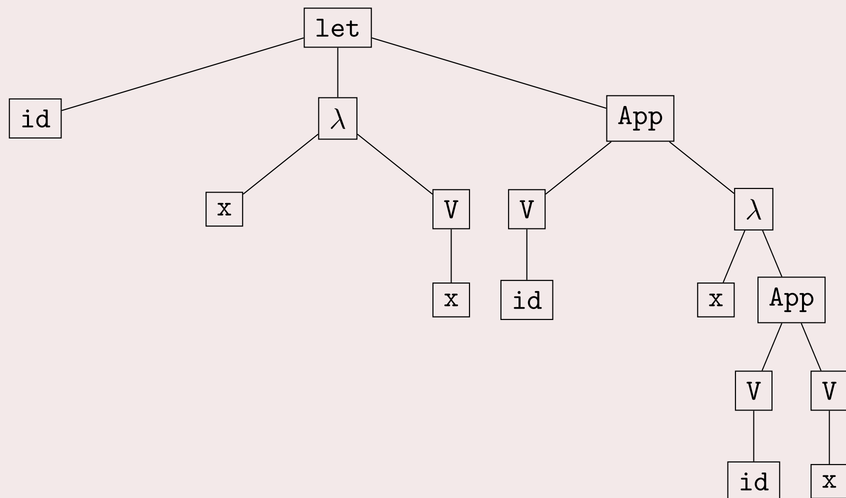
Warm-up

Here is a program in its textual form:

```
let id = λ x → x in
id (λ x → id x)
```

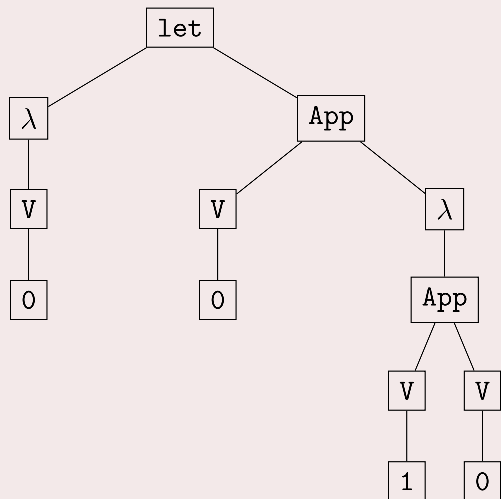
Nominal style

The same program represented as a tree, graphically depicted:



De Bruijn style

Variable are represented by their distance to the binding node:




```
let id = λ x → x in
id (λ x → id x)
```

Bare de Bruijn λ-terms

```
data TmB : Set where
```

Bare de Bruijn λ-terms

```
data TmB : Set where
  V   : (x : ℕ) → TmB
```

Bare de Bruijn λ-terms

```
data Tmᴮ : Set where
  V   : (x : ℕ) → Tmᴮ
  _·_ : (t u : Tmᴮ) → Tmᴮ
```

Bare de Bruijn λ-terms

```
data Tmᴮ : Set where
  V    : (x : ℕ) → Tmᴮ
  _·_  : (t u : Tmᴮ) → Tmᴮ
  ƛ    : (t : Tmᴮ) → Tmᴮ
```

Bare de Bruijn λ-terms

```
data TmB : Set where
  V   : (x : ℕ) → TmB
  _·_ : (t u : TmB) → TmB
  ƛ   : (t : TmB) → TmB
  Let : (t u : TmB) → TmB
```

Bare de Bruijn λ-terms

```
data Tmᴮ : Set where
  V   : (x : ℕ) → Tmᴮ
  _·_ : (t u : Tmᴮ) → Tmᴮ
  ƛ   : (t : Tmᴮ) → Tmᴮ
  Let : (t u : Tmᴮ) → Tmᴮ
```

Examples:

```
apTmᴮ = ƛ (ƛ (V 1 · V 0))       -- λ f → λ x → f x

ididTmᴮ = Let (ƛ (V 0))         -- let id = λ x → x in
              (ƛ (V 1 · V 0)) -- λ x → id x

non-closedᴮ : Tmᴮ
non-closedᴮ = ƛ (V 1 · V 0)    -- λ x → f x
```

A predicate for well-scoped λ-terms

```
is-closed? : Tm^B → Set
is-closed? t = WS 0 t
```

A predicate for well-scoped λ-terms

```
is-closed? : Tmᴮ → Set
is-closed? t = WS 0 t
```

```
data WS (n : ℕ) : Tmᴮ → Set where
  WS-V    : ∀ x →
            x < n →
            WS n (V x)
  WS-·    : ∀ t u →
            WS n t → WS n u →
            WS n (t · u)
  WS-ƛ    : ∀ t →
            WS (1 + n) t →
            WS n (ƛ t)
  WS-Let  : ∀ t u →
            WS n t → WS (1 + n) u →
            WS n (Let t u)
```

The Fin approach

```
data Fin : ℕ → Set where                -- Fin n ⇔ { i | i < n }
  zero : {n : ℕ} → Fin (suc n)
  suc  : {n : ℕ} (i : Fin n) → Fin (suc n)

_ᶠ : ∀ n → Fin (suc n)
```

The Fin approach

```
data Fin : ℕ → Set where              -- Fin n ⇔ { i | i < n }
  zero : {n : ℕ} → Fin (suc n)
  suc  : {n : ℕ} (i : Fin n) → Fin (suc n)

_ᶠ : ∀ n → Fin (suc n)
```

```
data Tmᶠ n : Set where
  V     : (x : Fin n) → Tmᶠ n
  _·_   : (t u : Tmᶠ n) → Tmᶠ n
  ƛ     : (t : Tmᶠ (suc n)) → Tmᶠ n
  Let   : (t : Tmᶠ n) (u : Tmᶠ (suc n)) → Tmᶠ n
```

The Fin approach

```
data Fin : ℕ → Set where                -- Fin n ⇔ { i | i < n }
  zero : {n : ℕ} → Fin (suc n)
  suc  : {n : ℕ} (i : Fin n) → Fin (suc n)

_ᶠ : ∀ n → Fin (suc n)
```

```
data Tmᶠ n : Set where
  V     : (x : Fin n) → Tmᶠ n
  _·_   : (t u : Tmᶠ n) → Tmᶠ n
  ƛ     : (t : Tmᶠ (suc n)) → Tmᶠ n
  Let   : (t : Tmᶠ n) (u : Tmᶠ (suc n)) → Tmᶠ n
```

```
apTmᶠ : Tmᶠ 0
apTmᶠ = ƛ (ƛ (V (1 ᶠ) · V (0 ᶠ)))
non-closedᶠ : Tmᶠ 1
non-closedᶠ = ƛ (V (1 ᶠ) · V (0 ᶠ))
```

The *nested* data type approach

```
data Maybe A : Set where -- nothing | just A
Maybe^ : ℕ → Set → Set
_ᴹ : ∀ {A} n → Maybe^ n A
```

The *nested* data type approach

```
data Maybe A : Set where -- nothing | just A
Maybe^ : ℕ → Set → Set
_ᴹ : ∀ {A} n → Maybe^ n A
```

The type $\mathtt{Tm}^{\mathtt{M}}$ is parameterized by the type of variables:

```
data Tmᴹ A : Set where
  V     : (x : A) → Tmᴹ A
  _·_   : (t u : Tmᴹ A) → Tmᴹ A
  ƛ     : (t : Tmᴹ (Maybe A)) → Tmᴹ A
  Let   : (t : Tmᴹ A) (u : Tmᴹ (Maybe A)) → Tmᴹ A
```

The *nested* data type approach

```
data Maybe A : Set where -- nothing | just A
Maybe^ : ℕ → Set → Set
_^M : ∀ {A} n → Maybe^ n A
```

The type $\mathtt{Tm}^{\mathtt{M}}$ is parameterized by the type of variables:

```
data TmM A : Set where
  V    : (x : A) → TmM A
  _·_  : (t u : TmM A) → TmM A
  ƛ    : (t : TmM (Maybe A)) → TmM A
  Let  : (t : TmM A) (u : TmM (Maybe A)) → TmM A
```

```
apTmM : TmM ⊥
apTmM = ƛ (ƛ (V (1 M) · V (0 M)))
non-closedM : TmM (Maybe ⊥)
non-closedM = ƛ (V (1 M) · V (0 M))
```

Motivations

Safe, yet expressive programming in de Bruijn style:

Motivations

Safe, yet expressive programming in de Bruijn style:

- Safer than bare de Bruijn indices

Motivations

Safe, yet expressive programming in de Bruijn style:

- Safer than bare de Bruijn indices
- Safer than using `Fin` ℓ or `Maybe^` ℓ $\bot$, yet as expressive

Motivations

Safe, yet expressive programming in de Bruijn style:

- Safer than bare de Bruijn indices
- Safer than using `Fin` ℓ or `Maybeˆ` ℓ $\bot$, yet as expressive
- More expressive than using `Maybe` and `FreshLook` approaches

Motivations

Safe, yet expressive programming in de Bruijn style:

- Safer than bare de Bruijn indices
- Safer than using `Fin` ℓ or `Maybe^` ℓ $\bot$, yet as expressive
- More expressive than using `Maybe` and `FreshLook` approaches
- Expressiveness enables more efficient programs

Motivations

Safe, yet expressive programming in de Bruijn style:

- Safer than bare de Bruijn indices
- Safer than using `Fin` ℓ or `Maybe^` ℓ $\bot$, yet as expressive
- More expressive than using `Maybe` and `FreshLook` approaches
- Expressiveness enables more efficient programs

Built for PL programmers, to write:

Motivations

Safe, yet expressive programming in de Bruijn style:

- Safer than bare de Bruijn indices
- Safer than using `Fin` ℓ or `Maybe^` ℓ $\bot$, yet as expressive
- More expressive than using `Maybe` and `FreshLook` approaches
- Expressiveness enables more efficient programs

Built for PL programmers, to write:

- Compilers, evaluators...

Motivations

Safe, yet expressive programming in de Bruijn style:

- Safer than bare de Bruijn indices
- Safer than using `Fin` ℓ or `Maybe^` ℓ $\bot$, yet as expressive
- More expressive than using `Maybe` and `FreshLook` approaches
- Expressiveness enables more efficient programs

Built for PL programmers, to write:

- Compilers, evaluators...
- Static analysers, type-checkers...

Motivations

Safe, yet expressive programming in de Bruijn style:

- Safer than bare de Bruijn indices
- Safer than using `Fin` ℓ or `Maybe^` ℓ $\bot$, yet as expressive
- More expressive than using `Maybe` and `FreshLook` approaches
- Expressiveness enables more efficient programs

Built for PL programmers, to write:

- Compilers, evaluators...
- Static analysers, type-checkers...
- Proof assistants, logic systems...

Motivations

Safe, yet expressive programming in de Bruijn style:

- Safer than bare de Bruijn indices
- Safer than using `Fin` ℓ or `Maybe^` ℓ $\perp$, yet as expressive
- More expressive than using `Maybe` and `FreshLook` approaches
- Expressiveness enables more efficient programs

Built for PL programmers, to write:

- Compilers, evaluators...
- Static analysers, type-checkers...
- Proof assistants, logic systems...
- Code generators, specializers, generic programs...

Motivations

Safe, yet expressive programming in de Bruijn style:

- Safer than bare de Bruijn indices
- Safer than using `Fin` ℓ or `Maybe^` ℓ $\perp$, yet as expressive
- More expressive than using `Maybe` and `FreshLook` approaches
- Expressiveness enables more efficient programs

Built for PL programmers, to write:

- Compilers, evaluators...
- Static analysers, type-checkers...
- Proof assistants, logic systems...
- Code generators, specializers, generic programs...

Built for “name library” writers:

Motivations

Safe, yet expressive programming in de Bruijn style:

- Safer than bare de Bruijn indices
- Safer than using `Fin` ℓ or `Maybe^` ℓ $\perp$, yet as expressive
- More expressive than using `Maybe` and `FreshLook` approaches
- Expressiveness enables more efficient programs

Built for PL programmers, to write:

- Compilers, evaluators...
- Static analysers, type-checkers...
- Proof assistants, logic systems...
- Code generators, specializers, generic programs...

Built for "name library" writers:

- To optimize techniques like *nested* with rewrite rules

Motivations

Safe, yet expressive programming in de Bruijn style:

- Safer than bare de Bruijn indices
- Safer than using `Fin` ℓ or `Maybe^` ℓ $\bot$, yet as expressive
- More expressive than using `Maybe` and `FreshLook` approaches
- Expressiveness enables more efficient programs

Built for PL programmers, to write:

- Compilers, evaluators...
- Static analysers, type-checkers...
- Proof assistants, logic systems...
- Code generators, specializers, generic programs...

Built for “name library” writers:

- To optimize techniques like *nested* with rewrite rules
- To implement lighter interfaces like Locally nameless/named, HOAS

NaPa types

```
record NaPa : Set₁ where
  field
    -- Worlds can be thought of finite sets of numbers:
    World : Set
```

NaPa types

```
record NaPa : Set₁ where
  field
    -- Worlds can be thought of finite sets of numbers:
    World : Set

    -- Worlds are built out of these operations:
    ∅      : World
    -- α +1 ⇔ { x + 1 | x ∈ α }
    _+1    : World → World
    -- α ↑1 ⇔ { 0 } ∪ α +1
    _↑1    : World → World
```

NaPa types

```
record NaPa : Set₁ where
  field
    -- Worlds can be thought of finite sets of numbers:
    World : Set

    -- Worlds are built out of these operations:
    ∅      : World
    -- α +1 ⇔ { x + 1 | x ∈ α }
    _+1    : World → World
    -- α ↑1 ⇔ { 0 } ∪ α +1
    _↑1    : World → World

    -- Names inhabits worlds:
    Name   : World → Set
    -- World inclusion witnesses
    _⊆_    : World → World → Set
```

NaPa types (cont.)

Iterated versions of _+1 and _↑1 are derived:

```
record NaPa : Set₁ where
  ...
  _+ᵂ_ : World → ℕ → World
  α +ᵂ 0     = α
  α +ᵂ suc n = (α +ᵂ n)+1

  _↑_ : World → ℕ → World
  α ↑ 0     = α
  α ↑ suc n = (α ↑ n)↑1
```

NaPa operations

```
record NaPa : Set₁ where
  ...
  field
    zeroᴺ       : ∀ {α} → Name (α ↑1)
```

NaPa operations

```
record NaPa : Set₁ where
  ...
  field
    zeroᴺ      : ∀ {α} → Name (α ↑1)
    addᴺ       : ∀ {α} k → Name α → Name (α +ᵂ k)
```

NaPa operations

```
record NaPa : Set₁ where
  ...
  field
    zeroᴺ     : ∀ {α} → Name (α ↑1)
    addᴺ      : ∀ {α} k → Name α → Name (α +ᵂ k)
    subtractᴺ : ∀ {α} k → Name (α +ᵂ k) → Name α
```

NaPa operations

```
record NaPa : Set₁ where
  ...
  field
    zeroᴺ      : ∀ {α} → Name (α ↑1)
    addᴺ       : ∀ {α} k → Name α → Name (α +ᵂ k)
    subtractᴺ  : ∀ {α} k → Name (α +ᵂ k) → Name α
    cmpᴺ       : ∀ {α} k → Name (α ↑ k) → Name (∅ ↑ k)
                                          ⊎ Name (α +ᵂ k)
```

NaPa operations

```
record NaPa : Set₁ where
  ...
  field
    zeroᴺ       : ∀ {α} → Name (α ↑1)
    addᴺ        : ∀ {α} k → Name α → Name (α +ᵂ k)
    subtractᴺ   : ∀ {α} k → Name (α +ᵂ k) → Name α
    cmpᴺ        : ∀ {α} k → Name (α ↑ k) → Name (∅ ↑ k)
                                            ⊎ Name (α +ᵂ k)

    _==ᴺ_       : ∀ {α} → Name α → Name α → Bool
    coerceᴺ     : ∀ {α β} → (α ⊆ β) → (Name α → Name β)
    ¬Name∅      : Name ∅ → ⊥
```

NaPa operations

```
record NaPa : Set₁ where
  ...
  field
    zeroᴺ      : ∀ {α} → Name (α ↑1)
    addᴺ       : ∀ {α} k → Name α → Name (α +ᵂ k)
    subtractᴺ  : ∀ {α} k → Name (α +ᵂ k) → Name α
    cmpᴺ       : ∀ {α} k → Name (α ↑ k) → Name (∅ ↑ k)
                                           ⊎ Name (α +ᵂ k)

    _==ᴺ_      : ∀ {α} → Name α → Name α → Bool
    coerceᴺ    : ∀ {α β} → (α ⊆ β) → (Name α → Name β)
    ¬Name∅     : Name ∅ → ⊥

  _ᴺ  : ∀ {α} ℓ → Name (α ↑ suc ℓ)
  ℓ ᴺ = coerceᴺ {! omitted !} (addᴺ ℓ zeroᴺ)
```

Terms indexed by worlds

```
data Tmᴰ α : Set where
  V    : (x : Name α) → Tmᴰ α
  _·_  : (t u : Tmᴰ α) → Tmᴰ α
  ƛ    : (t : Tmᴰ (α ↑1)) → Tmᴰ α
  Let  : (t : Tmᴰ α) (u : Tmᴰ (α ↑1)) → Tmᴰ α
```

Terms indexed by worlds

```
data Tmᴰ α : Set where
  V     : (x : Name α) → Tmᴰ α
  _·_   : (t u : Tmᴰ α) → Tmᴰ α
  ƛ     : (t : Tmᴰ (α ↑1)) → Tmᴰ α
  Let   : (t : Tmᴰ α) (u : Tmᴰ (α ↑1)) → Tmᴰ α
```

Examples:

```
apTmᴰ : Tmᴰ ∅
apTmᴰ = ƛ (ƛ (V (1 ᴺ) · V (0 ᴺ)))

non-closedᴰ : Tmᴰ (∅ ↑ 1)
non-closedᴰ = ƛ (V (1 ᴺ) · V (0 ᴺ))
```

Building a renaming function

Derived from $\texttt{add}^{\texttt{N}}$, $\texttt{cmp}^{\texttt{N}}$, $\texttt{subtract}^{\texttt{N}}$, and $\texttt{coerce}^{\texttt{N}}$:

$$\begin{array}{lll} \texttt{suc}^{\texttt{N}}\!\uparrow & : \forall\ \{\alpha\} \to \texttt{Name}\ \alpha & \to\ \texttt{Name}\ (\alpha\ \uparrow 1) \\ \texttt{pred}^{\texttt{N}}? & : \forall\ \{\alpha\} \to \texttt{Name}\ (\alpha\ \uparrow 1) & \to\ \texttt{Maybe}\ (\texttt{Name}\ \alpha) \end{array}$$

Building a renaming function

Derived from $\texttt{add}^{\texttt{N}}$, $\texttt{cmp}^{\texttt{N}}$, $\texttt{subtract}^{\texttt{N}}$, and $\texttt{coerce}^{\texttt{N}}$:

```
sucᴺ↑  : ∀ {α} → Name α      →  Name (α ↑1)
predᴺ? : ∀ {α} → Name (α ↑1) →  Maybe (Name α)
```

Can we shift a function ?

```
protect↑1 : ∀ {α β} → (Name α      → Name β)
                    → (Name (α ↑1) → Name (β ↑1))
protect↑1 f = maybe (sucᴺ↑ ∘ f) zeroᴺ ∘ predᴺ?
```

Building a renaming function

Derived from add^{N}, cmp^{N}, subtract^{N}, and coerce^{N}:

```
sucᴺ↑  : ∀ {α} → Name α      →  Name (α ↑1)
predᴺ? : ∀ {α} → Name (α ↑1) →  Maybe (Name α)
```

Can we shift a function ?

```
protect↑1 : ∀ {α β} → (Name α       → Name β)
                    → (Name (α ↑1) → Name (β ↑1))
protect↑1 f = maybe (sucᴺ↑ ∘ f) zeroᴺ ∘ predᴺ?
```

We use protect↑1 to move under name-abstractions:

```
renameTmᴰ : ∀ {α β} → (Name α → Name β)
                    → (Tmᴰ α  → Tmᴰ β)
renameTmᴰ f (V x)     = V (f x)
renameTmᴰ f (t · u)   = renameTmᴰ f t · renameTmᴰ f u
renameTmᴰ f (ƛ t)     = ƛ (renameTmᴰ (protect↑1 f) t)
```

Derived operations and advanced examples

- We can build a lot more: free-variables, comparison, NBE...

Derived operations and advanced examples

- We can build a lot more: free-variables, comparison, NBE...
- `traverseTm`$^{\mathtt{D}}$ generalizes `renameTm`$^{\mathtt{D}}$ in three ways !

Derived operations and advanced examples

- We can build a lot more: free-variables, comparison, NBE...
- `traverseTm`$^{\mathtt{D}}$ generalizes `renameTm`$^{\mathtt{D}}$ in three ways !
- Applying `traverseTm`$^{\mathtt{D}}$ yields many functions...

Derived operations and advanced examples

- We can build a lot more: free-variables, comparison, NBE...
- `traverseTm`$^{\mathtt{D}}$ generalizes `renameTm`$^{\mathtt{D}}$ in three ways!
- Applying `traverseTm`$^{\mathtt{D}}$ yields many functions...
- ... including shifting and substitution functions

Derived operations and advanced examples

- We can build a lot more: free-variables, comparison, NBE...
- `traverseTm`$^{\mathtt{D}}$ generalizes `renameTm`$^{\mathtt{D}}$ in three ways !
- Applying `traverseTm`$^{\mathtt{D}}$ yields many functions...
- ... including shifting and substitution functions

All the details are in the paper !

Soundness properties

Logical relations save the day !

$\mathtt{e} \quad : \quad \tau$

Logical relations save the day !

```
e   :   τ                -- for each program well-typed
```

Logical relations save the day !

```
  e   :   τ                      -- for each program well-typed
⟦ e ⟧ : ⟦ τ ⟧ e e
```

Logical relations save the day !

```
  e    :   τ                    -- for each program well-typed
⟦ e ⟧ : ⟦ τ ⟧ e e               -- one theorem for free
```

Logical relations save the day !

```
  e   :  τ                              -- for each program well-typed
⟦ e ⟧ : ⟦ τ ⟧ e e                       -- one theorem for free
```

```
⟦ℕ⟧ x₁ x₂ = x₁ ≡ x₂
```

Logical relations save the day !

```
  e   :   τ                                -- for each program well-typed
⟦ e ⟧ : ⟦ τ ⟧ e e                          -- one theorem for free
```

```
⟦ℕ⟧ x₁ x₂ = x₁ ≡ x₂                        -- Relations indexed by types
```

Logical relations save the day !

```
  e   :   τ                  -- for each program well-typed
⟦ e ⟧ : ⟦ τ ⟧ e e            -- one theorem for free
```

```
⟦ℕ⟧ x₁ x₂ = x₁ ≡ x₂          -- Relations indexed by types

-- Extensionality on functions
(A_r ⟦→⟧ B_r) f₁ f₂ = ∀ {x₁ x₂} (x_r : A_r x₁ x₂)
                      → B_r (f₁ x₁) (f₂ x₂)
```

Logical relations save the day !

```
  e   :   τ                  -- for each program well-typed

⟦ e ⟧ : ⟦ τ ⟧ e e            -- one theorem for free
```

```
⟦ℕ⟧ x₁ x₂ = x₁ ≡ x₂          -- Relations indexed by types

-- Extensionality on functions
(Aᵣ ⟦→⟧ Bᵣ) f₁ f₂ = ∀ {x₁ x₂} (xᵣ : Aᵣ x₁ x₂)
                    → Bᵣ (f₁ x₁) (f₂ x₂)

-- Extends nicely to dependent types
(⟦Π⟧ Aᵣ Bᵣ) f₁ f₂ = ∀ {x₁ x₂} (xᵣ : Aᵣ x₁ x₂)
                    → Bᵣ xᵣ (f₁ x₁) (f₂ x₂)
```

Logical relations save the day !

```
  e    :    τ                    -- for each program well-typed
⟦ e ⟧ : ⟦ τ ⟧ e e                -- one theorem for free
```

```
⟦ℕ⟧ x₁ x₂ = x₁ ≡ x₂              -- Relations indexed by types

-- Extensionality on functions
(Aᵣ ⟦→⟧ Bᵣ) f₁ f₂ = ∀ {x₁ x₂} (xᵣ : Aᵣ x₁ x₂)
                   → Bᵣ (f₁ x₁) (f₂ x₂)

-- Extends nicely to dependent types
(⟦Π⟧ Aᵣ Bᵣ) f₁ f₂ = ∀ {x₁ x₂} (xᵣ : Aᵣ x₁ x₂)
                   → Bᵣ xᵣ (f₁ x₁) (f₂ x₂)

⟦Set⟧ A₁ A₂ = A₁ → A₂ → Set
```

Free theorems for library clients

`c : (lib : NomPa)` $\rightarrow \tau$

Free theorems for library clients

```
c : (lib : NomPa) → τ

c : ∀ World Name _==ᴺ_ ...  → τ
```

Free theorems for library clients

```
c : (lib : NomPa) → τ

c : ∀ World Name _==ᴺ_ ...  → τ

⟦c⟧ : ⟦ ∀ World Name _==ᴺ_ ...  → τ ⟧ c c
```

Free theorems for library clients

```
c : (lib : NomPa) → τ

c : ∀ World Name _==ᴺ_ ...  → τ

⟦c⟧ : ⟦ ∀ World Name _==ᴺ_ ...  → τ ⟧ c c

⟦c⟧ : (∀⟨ ⟦World⟧ : ⟦Set⟧                        ⟩⟦→⟧
       ∀⟨ ⟦Name⟧  : ⟦World⟧ ⟦→⟧ ⟦Set⟧ ⟩⟦→⟧
       ∀⟨ _⟦==ᴺ⟧_  : ...                          ⟩⟦→⟧
       ...  ⟦→⟧ ⟦ τ ⟧) c c
```

Free theorems for library clients

```
c : (lib : NomPa) → τ

c : ∀ World Name _==ᴺ_ ...  → τ

⟦c⟧ : ⟦ ∀ World Name _==ᴺ_ ...  → τ ⟧ c c

⟦c⟧ : (∀⟨ ⟦World⟧ : ⟦Set⟧                  ⟩⟦→⟧
       ∀⟨ ⟦Name⟧  : ⟦World⟧ ⟦→⟧ ⟦Set⟧ ⟩⟦→⟧
       ∀⟨ _⟦==ᴺ⟧_ : ...                    ⟩⟦→⟧
       ...  ⟦→⟧ ⟦ τ ⟧) c c

⟦c⟧ : ∀ {W₁ W₂}    (⟦World⟧  : W₁ → W₂ → Set)
        {N₁ N₂}    (⟦Name⟧   : ... N₁ N₂)
        {==₁ ==₂}  (_⟦==ᴺ⟧_  : ... ==₁ ==₂)
      ... → (⟦ τ ⟧ (c ... ) (c ... ))
```

NaPa soundness, modularly

```
module ⟦NomPa⟧ where
  record ⟦World⟧ (α₁ α₂ : World) : Set₁ where
    constructor _,_
    field ℛ          : Name α₁ → Name α₂ → Set
    field ℛ-pres-≡ : ∀ x₁ y₁ x₂ y₂ → ℛ x₁ x₂ → ℛ y₁ y₂
                                       → x₁ ≡ y₁ ↔ x₂ ≡ y₂
```

NaPa soundness, modularly

```
module ⟦NomPa⟧ where
  record ⟦World⟧ (α₁ α₂ : World) : Set₁ where
    constructor _,_
    field ℛ           : Name α₁ → Name α₂ → Set
    field ℛ-pres-≡ : ∀ x₁ y₁ x₂ y₂ → ℛ x₁ x₂ → ℛ y₁ y₂
                                     → x₁ ≡ y₁ ↔ x₂ ≡ y₂

  ⟦Name⟧ (ℛ , _) x₁ x₂ = ℛ x₁ x₂
```

NaPa soundness, modularly

```
module ⟦NomPa⟧ where
  record ⟦World⟧ (α₁ α₂ : World) : Set₁ where
    constructor _,_
    field ℛ          : Name α₁ → Name α₂ → Set
    field ℛ-pres-≡ : ∀ x₁ y₁ x₂ y₂ → ℛ x₁ x₂ → ℛ y₁ y₂
                                   → x₁ ≡ y₁ ↔ x₂ ≡ y₂

  ⟦Name⟧ (ℛ , _) x₁ x₂ = ℛ x₁ x₂

  ⟦zeroᴺ⟧ : ⟦ ∀ {α} → Name (α ↑1) ⟧ zeroᴺ zeroᴺ
```

NaPa soundness, modularly

```
module ⟦NomPa⟧ where
  record ⟦World⟧ (α₁ α₂ : World) : Set₁ where
    constructor _,_
    field ℛ          : Name α₁ → Name α₂ → Set
    field ℛ-pres-≡ : ∀ x₁ y₁ x₂ y₂ → ℛ x₁ x₂ → ℛ y₁ y₂
                                   → x₁ ≡ y₁ ↔ x₂ ≡ y₂

  ⟦Name⟧ (ℛ , _) x₁ x₂ = ℛ x₁ x₂

  ⟦zeroᴺ⟧ : ⟦ ∀ {α} → Name (α ↑1) ⟧ zeroᴺ zeroᴺ

  -- all our functions inhabit the logical relation
  ⟦==ᴺ⟧, ⟦addᴺ⟧, ...
```

NaPa soundness, modularly

```
module ⟦NomPa⟧ where
  record ⟦World⟧ (α₁ α₂ : World) : Set₁ where
    constructor _,_
    field ℛ          : Name α₁ → Name α₂ → Set
    field ℛ-pres-≡ : ∀ x₁ y₁ x₂ y₂ → ℛ x₁ x₂ → ℛ y₁ y₂
                                      → x₁ ≡ y₁ ↔ x₂ ≡ y₂

  ⟦Name⟧ (ℛ , _) x₁ x₂ = ℛ x₁ x₂

  ⟦zeroᴺ⟧ : ⟦ ∀ {α} → Name (α ↑1) ⟧ zeroᴺ zeroᴺ

  -- all our functions inhabit the logical relation
  ⟦==ᴺ⟧, ⟦addᴺ⟧, ...
```

Wrong functions get rejected, though ! (`isEven?`, `_≤_`, ...)

Free theorems for library clients (cont.)

$\texttt{f} : \forall\ \{\alpha\} \rightarrow \texttt{Name}\ \alpha \rightarrow \texttt{Bool}$

Free theorems for library clients (cont.)

$$\texttt{f} : \forall\ \{\alpha\} \rightarrow \texttt{Name}\ \alpha \rightarrow \texttt{Bool}$$

$$\texttt{f}_r : (\forall\langle\ \alpha_r : [\![\texttt{World}]\!]\ \rangle[\![\rightarrow]\!]\ [\![\texttt{Name}]\!]\ \alpha_r\ [\![\rightarrow]\!]\ [\![\texttt{Bool}]\!])\ \texttt{f}\ \texttt{f}$$

Free theorems for library clients (cont.)

$$\texttt{f} : \forall \{\alpha\} \rightarrow \texttt{Name}\ \alpha \rightarrow \texttt{Bool}$$

$$\texttt{f}_r : (\forall\langle\ \alpha_r : [\![\texttt{World}]\!]\ \rangle[\![\rightarrow]\!]\ [\![\texttt{Name}]\!]\ \alpha_r\ [\![\rightarrow]\!]\ [\![\texttt{Bool}]\!])\ \texttt{f}\ \texttt{f}$$

$$\texttt{f-const} : \forall\ \texttt{x}_1\ \texttt{x}_2 \rightarrow \texttt{f}\ \texttt{x}_1 \equiv \texttt{f}\ \texttt{x}_2$$

Free theorems for library clients (cont.)

```
f : ∀ {α} → Name α → Bool

fᵣ : (∀⟨ αᵣ : ⟦World⟧ ⟩⟦→⟧ ⟦Name⟧ αᵣ ⟦→⟧ ⟦Bool⟧) f f

f-const : ∀ x₁ x₂ → f x₁ ≡ f x₂
```

```
Ren : (α β : World) → Set
⟨_⟩  : ∀ {α β} → Ren α β → Tmᴰ α → Tmᴰ β
```

Free theorems for library clients (cont.)

$$\texttt{f} : \forall \{\alpha\} \rightarrow \texttt{Name}\ \alpha \rightarrow \texttt{Bool}$$

$$\texttt{f}_r : (\forall\langle\ \alpha_r : [\![\texttt{World}]\!]\ \rangle[\![\rightarrow]\!]\ [\![\texttt{Name}]\!]\ \alpha_r\ [\![\rightarrow]\!]\ [\![\texttt{Bool}]\!])\ \texttt{f}\ \texttt{f}$$

$$\texttt{f-const} : \forall\ \texttt{x}_1\ \texttt{x}_2 \rightarrow \texttt{f}\ \texttt{x}_1 \equiv \texttt{f}\ \texttt{x}_2$$

$$\texttt{Ren} : (\alpha\ \beta : \texttt{World}) \rightarrow \texttt{Set}$$
$$\langle_\rangle : \forall \{\alpha\ \beta\} \rightarrow \texttt{Ren}\ \alpha\ \beta \rightarrow \texttt{Tm}^{\texttt{D}}\ \alpha \rightarrow \texttt{Tm}^{\texttt{D}}\ \beta$$

$$\texttt{f} : \forall \{\alpha\} \rightarrow \texttt{Tm}^{\texttt{D}}\ \alpha \rightarrow \texttt{Tm}^{\texttt{D}}\ \alpha$$

Free theorems for library clients (cont.)

```
f : ∀ {α} → Name α → Bool

fᵣ : (∀⟨ αᵣ : ⟦World⟧ ⟩⟦→⟧ ⟦Name⟧ αᵣ ⟦→⟧ ⟦Bool⟧) f f

f-const : ∀ x₁ x₂ → f x₁ ≡ f x₂
```

```
Ren : (α β : World) → Set
⟨_⟩ : ∀ {α β} → Ren α β → Tmᴰ α → Tmᴰ β

f : ∀ {α} → Tmᴰ α → Tmᴰ α

fᵣ : (∀⟨ αᵣ : ⟦World⟧ ⟩⟦→⟧ ⟦Tmᴰ⟧ αᵣ ⟦→⟧ ⟦Tmᴰ⟧ αᵣ) f f
```

Free theorems for library clients (cont.)

```
f : ∀ {α} → Name α → Bool

f_r : (∀⟨ α_r : ⟦World⟧ ⟩⟦→⟧ ⟦Name⟧ α_r ⟦→⟧ ⟦Bool⟧) f f

f-const : ∀ x₁ x₂ → f x₁ ≡ f x₂
```

```
Ren : (α β : World) → Set
⟨_⟩ : ∀ {α β} → Ren α β → Tmᴰ α → Tmᴰ β

f : ∀ {α} → Tmᴰ α → Tmᴰ α

f_r : (∀⟨ α_r : ⟦World⟧ ⟩⟦→⟧ ⟦Tmᴰ⟧ α_r ⟦→⟧ ⟦Tmᴰ⟧ α_r) f f

f-comm-ren : ∀ {α β} (Φ : Ren α β) → ⟨ Φ ⟩ ∘ f ≗ f ∘ ⟨ Φ ⟩
```

NaPa is a subset of NomPa

- The NomPa interface have a few more functions

NaPa is a subset of NomPa

- The NomPa interface have a few more functions
- ... including a wide set of world inclusions witnesses

NaPa is a subset of NomPa

- The NomPa interface have a few more functions
- ... including a wide set of world inclusions witnesses
- not only de Bruijn style binders

NaPa is a subset of NomPa

- The NomPa interface have a few more functions
- . . . including a wide set of world inclusions witnesses
- not only de Bruijn style binders
- Nominal style binders

NaPa is a subset of NomPa

- The NomPa interface have a few more functions
- ... including a wide set of world inclusions witnesses
- not only de Bruijn style binders
- Nominal style binders
- de Bruijn levels

NaPa is a subset of NomPa

- The NomPa interface have a few more functions
- . . . including a wide set of world inclusions witnesses
- not only de Bruijn style binders
- Nominal style binders
- de Bruijn levels
- Combinations of these different styles

NaPa is a subset of NomPa

- The NomPa interface have a few more functions
- ... including a wide set of world inclusions witnesses
- not only de Bruijn style binders
- Nominal style binders
- de Bruijn levels
- Combinations of these different styles
- Many generic operations and examples

NaPa is a subset of NomPa

- The NomPa interface have a few more functions
- . . . including a wide set of world inclusions witnesses
- not only de Bruijn style binders
- Nominal style binders
- de Bruijn levels
- Combinations of these different styles
- Many generic operations and examples
- Encoding of various other binding techniques

Conclusion

- Safety through *abstract* types on names

Conclusion

- Safety through *abstract* types on names
- `_+1` is a new operation on worlds. `Fin` was not enough !

Conclusion

- Safety through *abstract* types on names
- `_+1` is a new operation on worlds. `Fin` was not enough !
- Safe bulk operations (`+k`, $\uparrow$`k`)

Conclusion

- Safety through *abstract* types on names
- `_+1` is a new operation on worlds. `Fin` was not enough !
- Safe bulk operations (`+k`, $\uparrow$`k`)
- All in AGDA, code, formalization, and proofs

Conclusion

- Safety through *abstract* types on names
- `_+1` is a new operation on worlds. `Fin` was not enough !
- Safe bulk operations (`+k`, `↑k`)
- All in AGDA, code, formalization, and proofs
- Free theorems available on-line

Bonus tracks

- Building world inclusion witnesses
- Shifting versus adding example
- Connecting approaches with type-isomorphisms
- The `Fin` approach is too concrete
- Traverse
- Traverse (cont.)
- Normalization by evaluation
- Shifting is: renaming with an addition
- Shifting a name
- Renaming without `protect↑`
- How dangerous this kind of programming is ?
- Shifting gets a finer type
- . . . and a faster implementation
- How precise are worlds ? Singletons worlds !

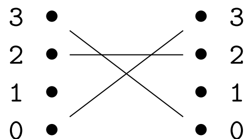
Building world inclusion witnesses

```
_⊆_ : World → World → Set
```

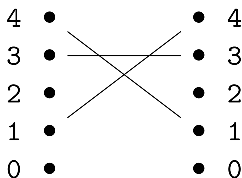
```
⊆-refl   : Reflexive _⊆_
⊆-trans  : Transitive _⊆_
⊆-∅      : ∀ {α} → ∅ ⊆ α

⊆-∅+1    : ∅ +1 ⊆ ∅
⊆-↑1-↑1  : ∀{α β}→ α ⊆ β ↔ α ↑1 ⊆ β ↑1
⊆-+1-+1  : ∀{α β}→ α ⊆ β ↔ α +1 ⊆ β +1
⊆-+1-↑1  : ∀{α}→ α +1 ⊆ α ↑1
```

◂ Back

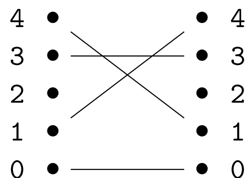
Shifting versus adding example



α_r



$\alpha_r \; [\![+1]\!]$



$\alpha_r \; [\![\uparrow 1]\!]$

Back

Connecting approaches with type-isomorphisms

$\forall\ \ell \rightarrow$ `Maybe (Fin` ℓ`)` $\Leftrightarrow$ `Fin (suc` ℓ`)`

Connecting approaches with type-isomorphisms

$\forall\ \ell \rightarrow$ `Maybe (Fin` ℓ`)` $\Leftrightarrow$ `Fin (suc` ℓ`)`

$\forall\ \alpha \rightarrow$ `Maybe (Name` α`)` $\Leftrightarrow$ `Name (`$\alpha \uparrow$`1)`

Connecting approaches with type-isomorphisms

$$\forall\ \ell \rightarrow \texttt{Maybe}\ (\texttt{Fin}\ \ell) \Leftrightarrow \texttt{Fin}\ (\texttt{suc}\ \ell)$$

$$\forall\ \alpha \rightarrow \texttt{Maybe}\ (\texttt{Name}\ \alpha) \Leftrightarrow \texttt{Name}\ (\alpha \uparrow 1)$$

$$\begin{aligned} \forall\ \ell \rightarrow \texttt{Maybe\^{}}\ \ell\ \bot &\Leftrightarrow \texttt{Fin}\ \ell \\ \ldots &\Leftrightarrow \texttt{Name}\ (\emptyset \uparrow \ell) \end{aligned}$$

Connecting approaches with type-isomorphisms

$\forall\ \ell \rightarrow$ `Maybe` $($`Fin` $\ell) \Leftrightarrow$ `Fin` $($`suc` $\ell)$

$\forall\ \alpha \rightarrow$ `Maybe` $($`Name` $\alpha) \Leftrightarrow$ `Name` $(\alpha \uparrow 1)$

$\forall\ \ell \rightarrow$ `Maybeˆ` $\ell\ \bot \Leftrightarrow$ `Fin` ℓ
$\quad\quad \ldots \Leftrightarrow$ `Name` $(\emptyset \uparrow \ell)$

$\forall\ \alpha \rightarrow$ `Name` $\alpha \Leftrightarrow$ `Name` $(\alpha$ `+1`$)$

Connecting approaches with type-isomorphisms

$\forall\ \ell \to \texttt{Maybe}\ (\texttt{Fin}\ \ell) \Leftrightarrow \texttt{Fin}\ (\texttt{suc}\ \ell)$

$\forall\ \alpha \to \texttt{Maybe}\ (\texttt{Name}\ \alpha) \Leftrightarrow \texttt{Name}\ (\alpha \uparrow 1)$

$$\forall\ \ell \to \texttt{Maybe\^{}}\ \ell\ \bot \Leftrightarrow \texttt{Fin}\ \ell$$
$$\ldots \Leftrightarrow \texttt{Name}\ (\emptyset \uparrow \ell)$$

$\forall\ \alpha \to \texttt{Name}\ \alpha \Leftrightarrow \texttt{Name}\ (\alpha + 1)$

$\forall\ \alpha \to \texttt{Tm}^{\texttt{M}}\ (\texttt{Name}\ \alpha) \Leftrightarrow \texttt{Tm}^{\texttt{D}}\ \alpha$

Connecting approaches with type-isomorphisms

$\forall\ \ell \to$ `Maybe (Fin` ℓ`)` $\Leftrightarrow$ `Fin (suc` ℓ`)`

$\forall\ \alpha \to$ `Maybe (Name` α`)` $\Leftrightarrow$ `Name (`α $\uparrow$`1)`

$\forall\ \ell \to$ `Maybeˆ` $\ell\ \bot \Leftrightarrow$ `Fin` ℓ
... $\Leftrightarrow$ `Name (`$\emptyset \uparrow \ell$`)`

$\forall\ \alpha \to$ `Name` $\alpha \Leftrightarrow$ `Name (`α `+1)`

$\forall\ \alpha \to$ $\mathtt{Tm}^{\mathtt{M}}$ `(Name` α`)` $\Leftrightarrow$ $\mathtt{Tm}^{\mathtt{D}}\ \alpha$

$\forall\ \ell \to$ $\mathtt{Tm}^{\mathtt{f}}\ \ell \Leftrightarrow \mathtt{Tm}^{\mathtt{M}}$ `(Fin` ℓ`)`
... $\Leftrightarrow \mathtt{Tm}^{\mathtt{M}}$ `(Name (`$\emptyset \uparrow \ell$`))`
... $\Leftrightarrow \mathtt{Tm}^{\mathtt{D}}$ `(`$\emptyset \uparrow \ell$`)`

Back

The `Fin` approach is too concrete

A function can analyse its argument `n`:

```
fᶠ : ∀ {n} → Tmᶠ n → ...
```

Back

Traverse

```
module TraverseTmᴰ
          {E}  (E-app  : Applicative E) {α β}
          (trName : ∀ ℓ → Name (α ↑ ℓ)
                        → E (Tmᴰ (β ↑ ℓ)))
  where
  open Applicative E-app

  tr : ∀ ℓ → Tmᴰ (α ↑ ℓ) → E (Tmᴰ (β ↑ ℓ))
  tr ℓ (V x)     = trName ℓ x
  tr ℓ (t · u)   = pure _·_ ⊛ tr ℓ t ⊛ tr ℓ u
  tr ℓ (ƛ t)     = pure ƛ   ⊛ tr (suc ℓ) t

  trTmᴰ : Tmᴰ α → E (Tmᴰ β)
  trTmᴰ = tr 0
```

◂ Back

Traverse (cont.)

```
traverseTmᴰ :
  ∀ {E} (E-app : Applicative E) {α β}
    (trName : ∀ ℓ → Name (α ↑ ℓ) → E (Tmᴰ (β ↑ ℓ)))
    Tmᴰ α → E (Tmᴰ β)
```

Traverse (cont.)

```
traverseTm^D :
  ∀ {E} (E-app : Applicative E) {α β}
    (trName : ∀ ℓ → Name (α ↑ ℓ) → E (Tm^D (β ↑ ℓ)))
    Tm^D α → E (Tm^D β)
```

Traverse three main points:

- Delegates all the name handling to `trName`

Back

Traverse (cont.)

```
traverseTm^D :
  ∀ {E} (E-app : Applicative E) {α β}
    (trName : ∀ ℓ → Name (α ↑ ℓ) → E (Tm^D (β ↑ ℓ)))
    Tm^D α → E (Tm^D β)
```

Traverse three main points:

- Delegates all the name handling to `trName`
- `trName` includes substitutions as well

◂ Back

Traverse (cont.)

```
traverseTm^D :
  ∀ {E} (E-app : Applicative E) {α β}
    (trName : ∀ ℓ → Name (α ↑ ℓ) → E (Tm^D (β ↑ ℓ)))
    Tm^D α → E (Tm^D β)
```

Traverse three main points:

- Delegates all the name handling to `trName`
- `trName` includes substitutions as well
- `trName` can have any applicative effect

◂ Back

Normalization by evaluation

```
module M (Abs : (World → Set) → World → Set) where
  data T α : Set where
    V     : Name α      → T α
    ƛ     : Abs T α     → T α
    _·_   : T α → T α → T α
```

◂ Back

Normalization by evaluation

```
SynAbsᴺ : (World → Set) → World → Set
SynAbsᴺ F α = ∃[ b ](F (b ◁ α))

open M SynAbsᴺ renaming (T to Term)

SemAbs : (World → Set) → World → Set
SemAbs F α = ∀ {β} → α ⊆ β → F β → F β

open M SemAbs renaming (T to Sem)
```

◂ Back

Normalization by evaluation

```
coerceSem : ∀ {α β} → α ⊆ β → Sem α → Sem β
coerceSem pf (V a)     = V (coerce^N pf a)
coerceSem pf (ƛ f)     = ƛ (λ pf′ v → f (⊆-trans pf pf′) v)
coerceSem pf (t · u)   = coerceSem pf t · coerceSem pf u

EvalEnv : (α β : World) → Set
EvalEnv α β = Name α → Sem β
-- α is the inner world
-- β is the outer world

_,_↦_ : ∀ {α β} (Γ : EvalEnv α β) b → Sem β → EvalEnv (b ◁
_,_↦_ Γ b v = exportWith v Γ
-- b ↦ v
-- x ↦ Γ x
```

Back

Normalization by evaluation

```
coerceEnv : ∀ {α β₁ β₂} → β₁ ⊆ β₂ → EvalEnv α β₁ → EvalEnv α /
coerceEnv pf Γ = coerceSem pf ∘ Γ


eval : ∀ {α β} → EvalEnv α β → Term α → Sem β
eval Γ (ƛ (a , t)) = ƛ (λ pf v → eval (coerceEnv pf Γ , a ↦
eval Γ (V x)       = Γ x
eval Γ (t · u)     = app (eval Γ t) (eval Γ u) where
  app : ∀ {α} → Sem α → Sem α → Sem α
  app (ƛ f)  v = f ⊆-refl v
  app n      v = n · v
```

Back

Normalization by evaluation

```
reify : ∀ {α} → Supply α → Sem α → Term α
reify s          (V a)   = V a
reify s          (n · v) = reify s n · reify s v
reify (sᴮ , s#) (ƛ f)   =
    ƛ (sᴮ , reify (sucˢ (sᴮ , s#)) (f (⊆-# s#) (V (nameᴮ sᴮ))))

nf : ∀ {α} → Supply α → Term α → Term α
nf supply = reify supply ∘ eval V
```

◂ Back

Shifting is: renaming with an addition

The traditional shifting can be expressed:

```
shiftTmᴰ↑ : ∀ {α} k → Tmᴰ α → Tmᴰ (α ↑ k)
shiftTmᴰ↑ = renameTmᴰ ∘ addᴺ↑
```

Shifting is: renaming with an addition

The traditional shifting can be expressed:

```
shiftTmᴰ↑ : ∀ {α} k → Tmᴰ α → Tmᴰ (α ↑ k)
shiftTmᴰ↑ = renameTmᴰ ∘ addᴺ↑
```

Is it the complexity we want for $\texttt{shiftTm}^{\texttt{D}}\!\uparrow$?

Shifting is: renaming with an addition

The traditional shifting can be expressed:

```
shiftTmᴰ↑ : ∀ {α} k → Tmᴰ α → Tmᴰ (α ↑ k)
shiftTmᴰ↑ = renameTmᴰ ∘ addᴺ↑
```

Is it the complexity we want for $\mathtt{shiftTm}^{\mathtt{D}}{\uparrow}$?

This piles up a chain of predecessor/successor...

Back

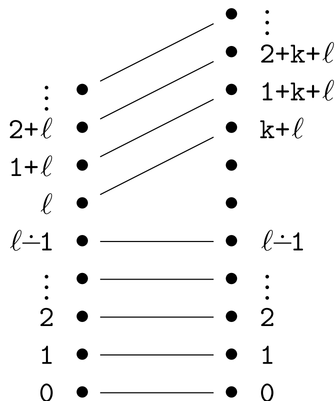
Shifting a name

… amounts to a protected addition:

```
shiftᴺ↑ : ∀ {α} k ℓ
            → Name (α ↑ ℓ)
            → Name (α ↑ k ↑ ℓ)
shiftᴺ↑ = protect↑ ∘ addᴺ↑
```

Shifting a name

The function has the following graph:



… amounts to a protected addition:

```
shiftᴺ↑ : ∀ {α} k ℓ
          → Name (α ↑ ℓ)
          → Name (α ↑ k ↑ ℓ)
shiftᴺ↑ = protect↑ ∘ addᴺ↑
```

Back

Renaming without protect↑

Here, ℓ is use to count crossed binders:

```
renameTmᴰ↑ : ∀ {α β} → (∀ ℓ → Name (α ↑ ℓ) → Name (β ↑ ℓ))
                     → (Tmᴰ α → Tmᴰ β)
renameTmᴰ↑ {α} {β} f = go 0 where
  go : ∀ ℓ → Tmᴰ (α ↑ ℓ) → Tmᴰ (β ↑ ℓ)
  go ℓ (V x)      = V (f ℓ x)
  go ℓ (t · u)    = go ℓ t · go ℓ u
  go ℓ (ƛ t)      = ƛ (go (suc ℓ) t)
```

Renaming without protect↑

Here, ℓ is use to count crossed binders:

```
renameTmᴰ↑ : ∀ {α β} → (∀ ℓ → Name (α ↑ ℓ) → Name (β ↑ ℓ))
                     → (Tmᴰ α → Tmᴰ β)
renameTmᴰ↑ {α} {β} f = go 0 where
  go : ∀ ℓ → Tmᴰ (α ↑ ℓ) → Tmᴰ (β ↑ ℓ)
  go ℓ (V x)      = V (f ℓ x)
  go ℓ (t · u)    = go ℓ t · go ℓ u
  go ℓ (ƛ t)      = ƛ (go (suc ℓ) t)
```

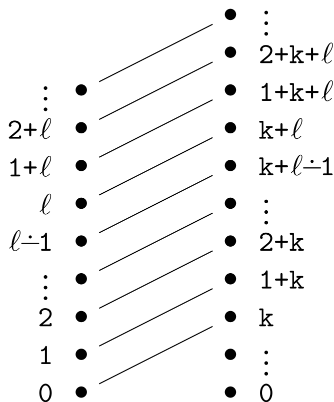
```
shiftTmᴰ↑′ : ∀ {α} k → Tmᴰ α → Tmᴰ (α ↑ k)
shiftTmᴰ↑′ = renameTmᴰ↑ ∘ shiftᴺ↑
```

◂ Back

How dangerous this kind of programming is ?

Sadly the unprotected addition can be coerced to this type:

```
unprotectedAdd^N↑ :
  ∀ {α} k ℓ
  → Name (α ↑ ℓ)
  → Name (α ↑ k ↑ ℓ)
unprotectedAdd^N↑ k ℓ =
  {! coercion omitted !}
  ∘ add^N↑ k
  -- this is ok since
  -- (α ↑ k) ↑ ℓ ≡ (α ↑ ℓ) ↑ k
```



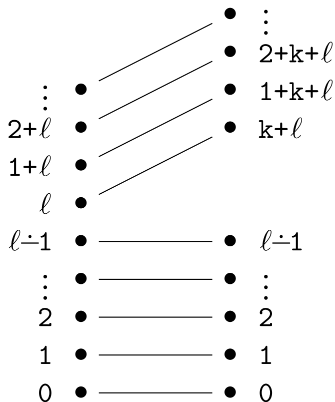
Back

Shifting gets a finer type

```
shiftᴺ : ∀ {α} k ℓ → Name (α ↑ ℓ) → Name (α +ᵂ k ↑ ℓ)
shiftᴺ = protect↑ ∘ addᴺ
```

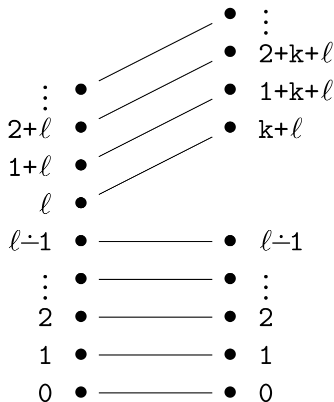
Shifting gets a finer type

```
shiftᴺ : ∀ {α} k ℓ → Name (α ↑ ℓ) → Name (α +ᵂ k ↑ ℓ)
shiftᴺ = protect↑ ∘ addᴺ
```



Shifting gets a finer type

```
shiftᴺ : ∀ {α} k ℓ → Name (α ↑ ℓ) → Name (α +ᵂ k ↑ ℓ)
shiftᴺ = protect↑ ∘ addᴺ
```



Ruling out any unprotected attempt !

◂ Back

... and a faster implementation

```
shiftᴺ : ∀ {α} k ℓ → Name (α ↑ ℓ) → Name (α +ᵂ k ↑ ℓ)
shiftᴺ k ℓ x
    with subtractᴺ ℓ x
... | inj₁ x′ = x′          -- coercion omitted
... | inj₂ x′ = addᴺ k n′ -- coercion omitted
```

Back

How precise are worlds ? Singletons worlds !

```
Worldˢ : ℕ → World
Worldˢ n = ∅ ↑1 +ᵂ n

Nameˢ : ℕ → Set
Nameˢ = Name ∘ Worldˢ

_ᴺˢ : ∀ n → Nameˢ n
_ᴺˢ n = zeroᴺ +ᴺ n
```

```
addˢ : ∀ {n} k → Nameˢ n → Nameˢ (k +ℕ n)
addˢ {n} k x = addᴺ k x ⟨-because ⊆-assoc-+ ⊆-refl n k -⟩

subtractˢ : ∀ {n} k → Nameˢ (k +ℕ n) → Nameˢ n
subtractˢ {n} k x = subtractᴺ k x
                    ⟨-because ⊆-assoc-+′ ⊆-refl n k -⟩
```

Back