Namely, Painless

Nicolas Pouillard

INRIA

Marburg
October 12, 2011

Safe programming with names and binders

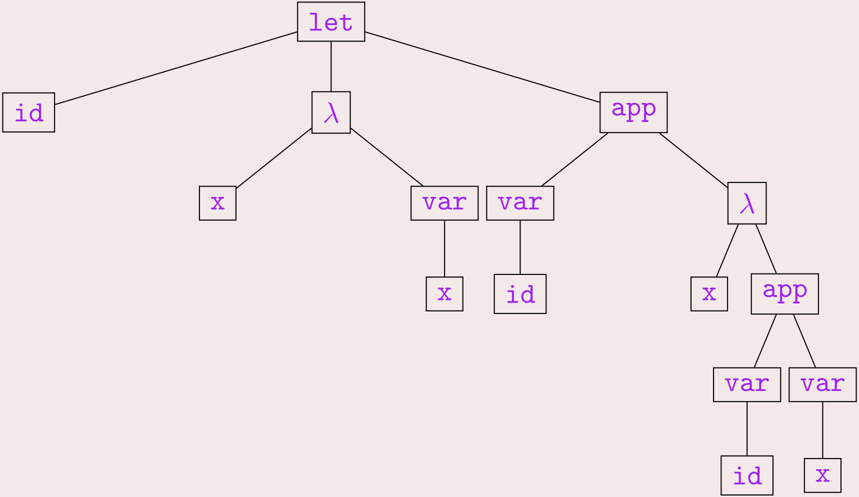
Warm-up

Here is a program in its textual form:

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

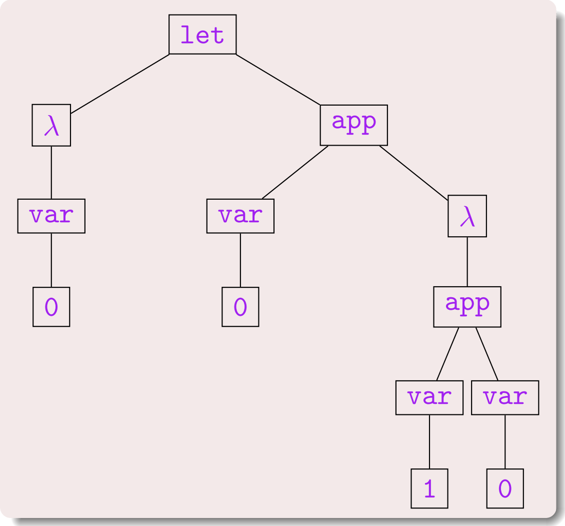
Nominal style

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)
```

The bare Nominal approach: Atoms

```
-- A set of atoms (could be ℕ)
Atom : Set

-- Atom is countably infinite; here are some atoms:
-- x,y,z... could be represented by 0,1,2...
x y z f g ... : Atom

-- The equality test on atoms
_==-Atom_ : (x y : Atom) → Bool
```

The bare Nominal approach: Terms

```
data Tm : Set where
  V   : (x : Atom) → Tm
  _·_ : (t u : Tm) → Tm
  ƛ   : (b : Atom) (t : Tm) → Tm
```

```
-- λx. x
idTm : Tm
idTm = ƛ x (V x)
```

```
-- λf. λx. f x
apTm : Tm
apTm = ƛ f (ƛ x (V f · V x))
```

```
-- a non-closed term: λx. f x
ncTm : Tm
ncTm = ƛ x (V f · V x)
```

Collecting free-variables

```
rm : Atom → List Atom → List Atom
rm _ []          = []
rm x (y ∷ ys) =
  if x ==-Atom y then rm x ys
                 else y ∷ rm x ys
```

```
fv : Tm → List Atom
fv (V x)    = [ x ]
fv (t · u) = fv t ++ fv u
fv (ƛ x t) = rm x (fv t)
```

Goal 1: How to guarantee that we manipulate only well-scoped terms?

Environments and Membership

```
data Env : Set where
  ϵ   : Env
  _,_ : (Γ : Env) (x : Atom) → Env
```

```
data _∈_ x : (Γ : Env) → Set where

  here  :               -------------
                         x ∈ (Γ , x)

  there : ∀ {y}       →   x ∈ Γ
                      →  -------------
                         x ∈ (Γ , y)
```

Well-scoped judgments

```
data _⊢_ Γ : Tm → Set where
  V : ∀ {x}
          →  x ∈ Γ
          → ---------
             Γ ⊢ V x

  _·_ : ∀ {t u}
          →  Γ ⊢ t
          →  Γ ⊢ u
          → -----------
             Γ ⊢ t · u

  ƛ : ∀ {t b}
          →  Γ , b ⊢ t
          → -----------
             Γ ⊢ ƛ b t
```

```
⊢id : ϵ ⊢ idᵀᵐ
⊢id = ƛ (V here)
```

```
⊢ap : ϵ ⊢ apᵀᵐ
⊢ap = ƛ (ƛ (V x₁ · V x₀))
  where x₀ = here
        x₁ = there here
```

```
f⊢nc : (ϵ , f) ⊢ ncᵀᵐ
f⊢nc = ƛ (V f₁ · V x₀))
  where x₀ = here
        f₁ = there here
```

Can we integrate this property into terms?

Well-scoped terms

```
data Tmʷˢ Γ : Set where
  V    : ∀ {x} → x ∈ Γ → Tmʷˢ Γ
  _·_  : Tmʷˢ Γ → Tmʷˢ Γ → Tmʷˢ Γ
  ƛ    : ∀ b → Tmʷˢ (Γ , b) → Tmʷˢ Γ
```

Well-scoped terms

```
data Tmʷˢ Γ : Set where
  V   : ∀ {x} → x ∈ Γ → Tmʷˢ Γ
  _·_ : Tmʷˢ Γ → Tmʷˢ Γ → Tmʷˢ Γ
  ƛ   : ∀ b → Tmʷˢ (Γ , b) → Tmʷˢ Γ
```

```
idʷˢ : Tmʷˢ ϵ
idʷˢ = ƛ x (V here)
```

```
apʷˢ : Tmʷˢ ϵ
apʷˢ = ƛ f (ƛ x (V (there here) · V here))
```

```
ncʷˢ : Tmʷˢ (ϵ , f)
ncʷˢ = ƛ (V (there here) · V here))
```

```
-- λx. x
idˣ : Tm
idˣ = ƛ x (V x)
```

```
-- λy. y
idʸ : Tm
idʸ = ƛ y (V y)
```

```
-- λx. x
idˣ : Tm
idˣ = ƛ x (V x)
```

```
-- λy. y
idʸ : Tm
idʸ = ƛ y (V y)
```

$$\forall\ f \to f\ \mathtt{id}^{x} \equiv f\ \mathtt{id}^{y}$$

```
-- λx. x
idˣ : Tm
idˣ = ƛ x (V x)
```

```
-- λy. y
idʸ : Tm
idʸ = ƛ y (V y)
```

$$\forall\ f \to f\ \mathtt{id}^{x} \equiv f\ \mathtt{id}^{y}$$

Goal 2: Computation modulo α-equivalence

```
f : ∀ {Γ} → Tmʷˢ Γ → Tmʷˢ Γ
```

```
f : ∀ {Γ} → Tm^ws Γ → Tm^ws Γ
```

This may leak information!

$$f : \forall \{\Gamma\} \to Tm^{ws}\ \Gamma \to Tm^{ws}\ \Gamma$$

This may leak information!

Goal 3: Leak-free abstraction!

Motivations

Safe, yet expressive programming with names and binders:

Motivations

Safe, yet expressive programming with names and binders:

- Safer than the bare Nominal approach

Motivations

Safe, yet expressive programming with names and binders:

- Safer than the bare Nominal approach
- More abstract than well-scoped terms

Motivations

Safe, yet expressive programming with names and binders:

- Safer than the bare Nominal approach
- More abstract than well-scoped terms
- No cost at name-abstraction

Motivations

Safe, yet expressive programming with names and binders:

- Safer than the bare Nominal approach
- More abstract than well-scoped terms
- No cost at name-abstraction
- Expressiveness enables more efficient programs

Motivations

Safe, yet expressive programming with names and binders:

- Safer than the bare Nominal approach
- More abstract than well-scoped terms
- No cost at name-abstraction
- Expressiveness enables more efficient programs

Built for PL programmers, to write:

Motivations

Safe, yet expressive programming with names and binders:

- Safer than the bare Nominal approach
- More abstract than well-scoped terms
- No cost at name-abstraction
- Expressiveness enables more efficient programs

Built for PL programmers, to write:

- Compilers, evaluators...

Motivations

Safe, yet expressive programming with names and binders:

- Safer than the bare Nominal approach
- More abstract than well-scoped terms
- No cost at name-abstraction
- Expressiveness enables more efficient programs

Built for PL programmers, to write:

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

Motivations

Safe, yet expressive programming with names and binders:

- Safer than the bare Nominal approach
- More abstract than well-scoped terms
- No cost at name-abstraction
- Expressiveness enables more efficient programs

Built for PL programmers, to write:

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

Motivations

Safe, yet expressive programming with names and binders:

- Safer than the bare Nominal approach
- More abstract than well-scoped terms
- No cost at name-abstraction
- Expressiveness enables more efficient programs

Built for PL programmers, to write:

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

Motivations

Safe, yet expressive programming with names and binders:

- Safer than the bare Nominal approach
- More abstract than well-scoped terms
- No cost at name-abstraction
- Expressiveness enables more efficient programs

Built for PL programmers, to write:

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

Built for "name library" writers:

Motivations

Safe, yet expressive programming with names and binders:

- Safer than the bare Nominal approach
- More abstract than well-scoped terms
- No cost at name-abstraction
- Expressiveness enables more efficient programs

Built for PL programmers, to write:

- Compilers, evaluators...
- Static analyzers, 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 with names and binders:

- Safer than the bare Nominal approach
- More abstract than well-scoped terms
- No cost at name-abstraction
- Expressiveness enables more efficient programs

Built for PL programmers, to write:

- Compilers, evaluators...
- Static analyzers, 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

The NomPa interface

```
record NomPa : Set₁ where
  field
    -- minimal kit to define types
    World  : Set
    Name   : World → Set
    Binder : Set
    _◁_    : Binder → World → World

    -- An infinite set of binders
    zeroᴮ : Binder
    sucᴮ  : Binder → Binder

    -- Converting names and binders back and forth
    nameᴮ   : ∀ {α} b → Name (b ◁ α)
    binderᴺ : ∀ {α} → Name α → Binder
```

The NomPa interface (cont.)

```
...
    -- There is no name in the empty world
    ∅           : World
    ¬Name∅      : ¬ (Name ∅)

    -- Names are comparable and exportable
    _==ᴺ_    : ∀ {α} (x y : Name α) → Bool
    exportᴺ : ∀ {α b} → Name (b ◁ α)
                       → Name (b ◁ ∅) ⊎ Name α

    -- The fresh-for relation
    _#_   : Binder → World → Set
    _#∅   : ∀ b → b # ∅
    suc#  : ∀ {α b} → b # α → (sucᴮ b) # (b ◁ α)
```

The NomPa interface (cont.)

```
...
    -- inclusion between worlds
    _⊆_        : World → World → Set
    coerceᴺ    : ∀ {α β} → (α ⊆ β) → (Name α → Name β)
    ⊆-refl     : Reflexive _⊆_
    ⊆-trans    : Transitive _⊆_
    ⊆-∅        : ∀ {α} → ∅ ⊆ α
    ⊆-◁        : ∀ {α β} b → α ⊆ β → (b ◁ α) ⊆ (b ◁ β)
    ⊆-#        : ∀ {α b} → b # α → α ⊆ (b ◁ α)
```

```
_ᴮ : ℕ → Binder
zero    ᴮ = zeroᴮ
(suc n) ᴮ = sucᴮ (n ᴮ)
```

```
exportᴺ? : ∀ {b α} → Name (b ◁ α) → Maybe (Name α)
```

Nominal terms with NomPa

```
data Tm α : Set where
  V   : Name α → Tm α
  _·_ : Tm α → Tm α → Tm α
  ƛ   : ∀ b → Tm (b ◁ α) → Tm α
```

```
-- Tm ∅ works as well
idTm : ∀ {α} → Tm α
idTm = ƛ x (V (nameB x))
       where x = 0 B
```

```
falseTm : ∀ {α} → Tm α
falseTm = ƛ x (ƛ x (V (nameB x)))
          where x = 0 B


-- this does not type-check
trueTm ÷ ∀ {α} → Tm α
trueTm = ƛ x (ƛ y (V (nameB x)))
   where x = 0 B
         y = 1 B
```

Weakening

```
trueᵀᵐ : ∀ {α} → Tm α
trueᵀᵐ {α} = ƛ x (ƛ y xᵀ)
  where open ⊆-Reasoning
        x   = 0 ᴮ
        y   = 1 ᴮ
        -- _⟨-because_-⟩ is coerceᴺ
        xᵀ  = V (nameᴮ x ⟨-because pf -⟩)
        pf  = x ◁ ∅
            ⊆⟨ ⊆-# (suc# (x #∅)) ⟩
              y ◁ x ◁ ∅
            ⊆⟨ ⊆-◁ y (⊆-◁ x ⊆-∅) ⟩∎
              y ◁ x ◁ α
            ∎
```

Weakening (shorter)

```
name◁... : ∀ {α} k x → Name ((k + x) ◁... α)
name◁... = {! omitted !}

V... : ∀ {α} k x → Tm ((k + x) ◁... α)
V... k x = V (name◁... k x)

true^Tm : ∀ {α} → Tm α
true^Tm = ƛ (x ᴮ) (ƛ (y ᴮ) (V... 1 x)) where x = 0
                                             y = 1
nc^Tm : Tm^D (0 ᴮ ◁ ∅)
nc^Tm = ƛ (x ᴮ) (V... 1 f · V... 0 x) where f = 0
                                            x = 1
ap^Tm : ∀ {α} → Tm α
ap^Tm = ƛ (f ᴮ) (ƛ (x ᴮ) (V... 1 f · V... 0 x)) where f = 0
                                                      x = 1
```

Collecting free-variables

```
rm : ∀ {α} b → List (Name (b ◁ α)) → List (Name α)
rm b [] = []
rm b (x ∷ xs)        with exportᴺ? x -- b is implicit
... {- bound: x≡b -} | nothing    = rm b xs
... {- free:  x≢b -} | just x′    = x′  ∷ rm b xs
```

```
fv : ∀ {α} → Tm α → List (Name α)
fv (V x)          = [ x ]
fv (fct · arg)    = fv fct ++ fv arg
fv (ƛ b t)        = rm b (fv t)
```

```
|Cmp| F i j = F i → F j → Bool
```

```
extendNameCmp : ∀ {α₁ α₂ b₁ b₂} → |Cmp| Name α₁ α₂
                                → |Cmp| Name (b₁ ◁ α₁) (b₂ ◁ α₂)
extendNameCmp f x₁ x₂
  with exportᴺ? x₁   | exportᴺ? x₂
...    | just x₁′     | just x₂′    = f x₁′ x₂′
...    | nothing      | nothing     = true
...    | _            | _           = false
```

```
cmpᵀᵐ : ∀ {α₁ α₂}→ |Cmp| Name α₁ α₂ → Tm α₁ → Tm α₂ → Bool
cmpᵀᵐ Γ (V x₁)       (V x₂)      = Γ x₁ x₂
cmpᵀᵐ Γ (t₁ · u₁)    (t₂ · u₂)   = cmpᵀᵐ Γ t₁ t₂ ∧ cmpᵀᵐ Γ u₁ u₂
cmpᵀᵐ Γ (ƛ _ t₁)     (ƛ _ t₂)    = cmpᵀᵐ (extendNameCmp Γ) t₁ t₂
cmpᵀᵐ _ _            _           = false
```

```
_==ᵀᵐ_ : ∀ {α} → Tm α → Tm α → Bool
_==ᵀᵐ_ = cmpᵀᵐ _==ᴺ_
```

Generic traversal

```
module Traverse^Tm {E}   (E-app : Applicative E)
                   {Env} (trKit : TrKit Env (E ∘ Tm)) where

  open Applicative E-app
  open TrKit trKit

  tr^Tm : ∀ {α β} → Env α β → Tm α → E (Tm β)
  tr^Tm Δ (V x)   = trName Δ x
  tr^Tm Δ (t · u) = pure _·_ ⊛ tr^Tm Δ t ⊛ tr^Tm Δ u
  tr^Tm Δ (ƛ b t) = pure (ƛ _) ⊛ tr^Tm (extEnv b Δ) t
```

Logical relations and parametricity!

Logical relation primer

$$
\begin{aligned}
(A_r \;[\![\rightarrow]\!]\; B_r)\; f_1\; f_2 = &\\
\forall\; \{x_1\; x_2\} &\rightarrow A_r\; x_1\; x_2\\
&\rightarrow B_r\; (f_1\; x_1)\; (f_2\; x_2)
\end{aligned}
$$

$$
\begin{aligned}
(\,[\![\Pi]\!]\; A_r\; B_r)\; f_1\; f_2 = \forall\; \{x_1\; x_2\}\; (x_r : A_r\; x_1\; x_2)&\\
\rightarrow B_r\; x_r\; (f_1\; x_1)\; (f_2\; x_2)&
\end{aligned}
$$

$$
\begin{aligned}
&[\![Set_0]\!] : Set_0 \rightarrow Set_0 \rightarrow Set_1\\
&[\![Set_0]\!]\; A_1\; A_2 = A_1 \rightarrow A_2 \rightarrow Set_0
\end{aligned}
$$

Applying the relation manually

```
-- What we would like to write but cannot:
⟦ ℕ → ℕ → Bool ⟧ =

-- What we write instead:
⟦ℕ⟧ ⟦→⟧ ⟦ℕ⟧ ⟦→⟧ ⟦Bool⟧ =

-- What this means:
λ f₁ f₂ →
  ∀ {x₁ x₂} (xᵣ : ⟦ℕ⟧ x₁ x₂)
    {y₁ y₂} (yᵣ : ⟦ℕ⟧ y₁ y₂)
  → ⟦Bool⟧ (f₁ x₁ y₁) (f₂ x₂ y₂)
```

Applying the relation manually (cont.)

```
-- What we would like to write but cannot:
⟦ (A : Set₀) → A → A ⟧ =

-- What we write instead:
⟦Π⟧ ⟦Set₀⟧ (λ Aᵣ → Aᵣ ⟦→⟧ Aᵣ) =

-- What this means:
λ f₁ f₂ →
  ∀ {A₁ A₂} (Aᵣ : A₁ → A₂ → Set₀)
    {x₁ x₂} (xᵣ : Aᵣ x₁ x₂)
  → Aᵣ (f₁ A₁ x₁) (f₂ A₂ x₂)
```

Applying the relation manually (cont.)

```
-- What we would like to write but cannot:
⟦ (A : Set₀) → List A ⟧ =

-- What we write instead (using a notation):
⟨ Aᵣ : ⟦Set₀⟧ ⟩⟦→⟧ ⟦List⟧ Aᵣ =

-- What this means:
λ l₁ l₂ →
  ∀ {A₁ A₂} (Aᵣ : A₁ → A₂ → Set₀)

  → ⟦List⟧ Aᵣ (l₁ A₁) (l₂ A₂)
```

Example Boolean values as numbers

```
B : Set
B = ℕ
```

```
false : B
false = 0
```

```
true : B
true = 1
```

```
_∨_ : B → B → B
m ∨ n = m + n
```

```
is42? : B → B
is42? 42 = true
is42? _  = false
```

Boolean example soundness

```
data ⟦B⟧ : B → B → Set where
  ⟦false⟧ : ⟦B⟧ 0 0
  ⟦true⟧  : ∀ {m n} → ⟦B⟧ (suc m) (suc n)
```

```
_⟦∨⟧_ : (⟦B⟧ ⟦→⟧ ⟦B⟧ ⟦→⟧ ⟦B⟧) _∨_ _∨_
```

Boolean example soundness

```
data ⟦B⟧ : B → B → Set where
  ⟦false⟧ : ⟦B⟧ 0 0
  ⟦true⟧  : ∀ {m n} → ⟦B⟧ (suc m) (suc n)
```

```
_⟦∨⟧_ : (⟦B⟧ ⟦→⟧ ⟦B⟧ ⟦→⟧ ⟦B⟧) _∨_ _∨_
```

```
_⟦∨⟧_ : ∀ {x₁ x₂} (xᵣ : ⟦B⟧ x₁ x₂)
          {y₁ y₂} (yᵣ : ⟦B⟧ y₁ y₂)
        → ⟦B⟧ (x₁ ∨ y₁) (x₂ ∨ y₂)
```

Boolean example soundness

```
data ⟦B⟧ : B → B → Set where
  ⟦false⟧ : ⟦B⟧ 0 0
  ⟦true⟧  : ∀ {m n} → ⟦B⟧ (suc m) (suc n)
```

```
_⟦∨⟧_ : (⟦B⟧ ⟦→⟧ ⟦B⟧ ⟦→⟧ ⟦B⟧) _∨_ _∨_
```

```
_⟦∨⟧_ : ∀ {x₁ x₂} (xᵣ : ⟦B⟧ x₁ x₂)
          {y₁ y₂} (yᵣ : ⟦B⟧ y₁ y₂)
        → ⟦B⟧ (x₁ ∨ y₁) (x₂ ∨ y₂)
```

```
⟦false⟧  ⟦∨⟧  x  =  x
⟦true⟧   ⟦∨⟧  _  =  ⟦true⟧
```

Boolean example soundness

```
data ⟦B⟧ : B → B → Set where
  ⟦false⟧ : ⟦B⟧ 0 0
  ⟦true⟧  : ∀ {m n} → ⟦B⟧ (suc m) (suc n)
```

```
_⟦∨⟧_ : (⟦B⟧ ⟦→⟧ ⟦B⟧ ⟦→⟧ ⟦B⟧) _∨_ _∨_
```

```
_⟦∨⟧_ : ∀ {x₁ x₂} (xᵣ : ⟦B⟧ x₁ x₂)
          {y₁ y₂} (yᵣ : ⟦B⟧ y₁ y₂)
        → ⟦B⟧ (x₁ ∨ y₁) (x₂ ∨ y₂)
```

```
⟦false⟧  ⟦∨⟧  x  =  x
⟦true⟧   ⟦∨⟧  _  =  ⟦true⟧
```

```
¬⟦is42?⟧ : ¬((⟦B⟧ ⟦→⟧ ⟦B⟧) is42? is42?)
¬⟦is42?⟧ ⟦is42?⟧ with ⟦is42?⟧ {42} {27} ⟦true⟧
...                    | () -- absurd
```

Soundness of our library

Free theorems for library clients

$\texttt{c} : (\texttt{lib} : \texttt{NomPa}) \rightarrow \tau$

Free theorems for library clients

$\texttt{c} : (\texttt{lib} : \texttt{NomPa}) \to \tau$

$\texttt{c} : \forall\ \texttt{World}\ \texttt{Name}\ \texttt{\_==}^{\texttt{N}}\texttt{\_}\ \texttt{...} \to \tau$

Free theorems for library clients

$$\texttt{c} : (\texttt{lib} : \texttt{NomPa}) \to \tau$$

$$\texttt{c} : \forall\ \texttt{World Name \_==}^{\texttt{N}}\texttt{\_ \dots} \to \tau$$

$$[\![\texttt{c}]\!] : [\![\ \forall\ \texttt{World Name \_==}^{\texttt{N}}\texttt{\_ \dots} \to \tau\ ]\!]\ \texttt{c c}$$

Free theorems for library clients

$$c : (\mathtt{lib} : \mathtt{NomPa}) \to \tau$$

$$c : \forall\ \mathtt{World}\ \mathtt{Name}\ \_{==}^{\mathtt{N}}\_\ \ldots \to \tau$$

$$[\![c]\!] : [\![\ \forall\ \mathtt{World}\ \mathtt{Name}\ \_{==}^{\mathtt{N}}\_\ \ldots \to \tau\ ]\!]\ c\ c$$

$$\begin{aligned}[\![c]\!] : (&\forall\langle\ [\![\mathtt{World}]\!] : [\![\mathtt{Set}]\!]\ \rangle[\![\to]\!] \\ &\forall\langle\ [\![\mathtt{Name}]\!] : [\![\mathtt{World}]\!]\ [\![\to]\!]\ [\![\mathtt{Set}]\!]\ \rangle[\![\to]\!] \\ &\forall\langle\ \_[\![{==}^{\mathtt{N}}]\!]\_ : \ldots\ \rangle[\![\to]\!] \\ &\ldots\ [\![\to]\!]\ [\![\ \tau\ ]\!]\ )\ c\ c\end{aligned}$$

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 ...))
```

NomPa soundness, modularly

```
--       ⟦World⟧ : ⟦Set₁⟧ World World
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⟧ : (⟦World⟧ ⟦→⟧ ⟦Set₀⟧) Name Name
    -- : ∀ {α₁ α₂} → ⟦World⟧ α₁ α₂ → Name α₁ → Name α₂ → Set₀
⟦Name⟧ (ℛ , _) x₁ x₂ = ℛ x₁ x₂

⟦Binder⟧ : ⟦Set₀⟧ Binder Binder
      -- : Binder → Binder → Set₀
⟦Binder⟧ _ _ = ⊤
```

NomPa soundness, modularly (cont.)

```
_⟦#⟧_ : (⟦Binder⟧ ⟦→⟧ ⟦World⟧ ⟦→⟧ ⟦Set₀⟧) _#_ _#_
  -- : ∀ {b₁ b₂} → ⟦Binder⟧ b₁ b₂ → ∀ {α₁ α₂} → ⟦World⟧ α₁ α₂
  --             → b₁ # α₁ → b₂ # α₂ → Set₀
_⟦#⟧_ _ _ _ _ = ⊤

_⟦⊆⟧_ : (⟦World⟧ ⟦→⟧ ⟦World⟧ ⟦→⟧ ⟦Set₀⟧)
          _⊆_ _⊆_
  -- : ∀ {α₁ α₂} → ⟦World⟧ α₁ α₂ →
  --   ∀ {β₁ β₂} → ⟦World⟧ β₁ β₂ →
  --     α₁ ⊆ β₁ → α₂ ⊆ β₂ → Set₀
_⟦⊆⟧_ αᵣ βᵣ α₁⊆β₁ α₂⊆β₂
  = (⟦Name⟧ αᵣ ⟦→⟧ ⟦Name⟧ βᵣ) (coerceᴺ α₁⊆β₁) (coerceᴺ α₂⊆β₂)
```

Free theorems for library clients (cont.)

```
f : ∀ {α} → Name α → 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

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

f-const : ∀ x₁ x₂ → f x₁ ≡ f x₂
```

```
⟦Tm⟧ -- α-equivalence on terms

Ren : (α β : World) → Set -- includes a name supply
⟨_⟩  : ∀ {α β} → Ren α β → Tm α → Tm β
```

Free theorems for library clients (cont.)

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

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

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

$[\![\texttt{Tm}]\!]$ -- α-equivalence on terms

Ren : $(\alpha\ \beta$: World$) \rightarrow$ Set -- includes a name supply

$\langle\_\rangle$: $\forall\ \{\alpha\ \beta\} \rightarrow$ Ren $\alpha\ \beta \rightarrow$ Tm $\alpha \rightarrow$ Tm β

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

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
```

```
⟦Tm⟧ -- α-equivalence on terms

Ren : (α β : World) → Set -- includes a name supply
⟨_⟩ : ∀ {α β} → Ren α β → Tm α → Tm β

f : ∀ {α} → Tm α → Tm α

f_r : (∀⟨ α_r : ⟦World⟧ ⟩⟦→⟧ ⟦Tm⟧ α_r ⟦→⟧ ⟦Tm⟧ α_r) 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
```

```
⟦Tm⟧ -- α-equivalence on terms

Ren : (α β : World) → Set -- includes a name supply
⟨_⟩  : ∀ {α β} → Ren α β → Tm α → Tm β

f : ∀ {α} → Tm α → Tm α

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

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

NomPa: a multi-style library for names and binders

- The NomPa interface have a few more functions

NomPa: a multi-style library for names and binders

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

NomPa: a multi-style library for names and binders

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

NomPa: a multi-style library for names and binders

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

NomPa: a multi-style library for names and binders

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

NomPa: a multi-style library for names and binders

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

NomPa: a multi-style library for names and binders

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

NomPa: a multi-style library for names and binders

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

Conclusion

- Well-scoped terms only

Conclusion

- Well-scoped terms only
- Computation modulo α-equivalence

Conclusion

- Well-scoped terms only
- Computation modulo α-equivalence
- Leak-free abstraction

Conclusion

- Well-scoped terms only
- Computation modulo α-equivalence
- Leak-free abstraction
- Names and terms indexed by worlds

Conclusion

- Well-scoped terms only
- Computation modulo α-equivalence
- Leak-free abstraction
- Names and terms indexed by worlds
- Safety through *abstract* types on base types

Conclusion

- Well-scoped terms only
- Computation modulo α-equivalence
- Leak-free abstraction
- Names and terms indexed by worlds
- Safety through *abstract* types on base types
- Separates names from binders

Conclusion

- Well-scoped terms only
- Computation modulo α-equivalence
- Leak-free abstraction
- Names and terms indexed by worlds
- Safety through *abstract* types on base types
- Separates names from binders
- All in Agda: code, formalization, and proofs

Conclusion

- Well-scoped terms only
- Computation modulo α-equivalence
- Leak-free abstraction
- Names and terms indexed by worlds
- Safety through *abstract* types on base types
- Separates names from binders
- All in AGDA: code, formalization, and proofs
- Free theorems available on-line