A fresh look at programming with names and binders

Nicolas Pouillard and François Pottier

INRIA

ICFP 2010, Baltimore
September 28, 2010

Program representation should stay well-typed and well-scoped

What do we need to manipulate data structures with names and binders?

A type for names

```
Name : Set
```

A type for names

```
Name : Set
```

An equality test for names

```
_≡_ : Name → Name → Bool
```

A type for names

```
Name : Set
```

An equality test for names

```
_≡_ : Name → Name → Bool
```

An infinite set of distinct names (gensym, stream, whatever)

```
x, y, z, ... : Name
```

Example: Data type for the untyped lambda calculus

```
data Tm : Set where
 V     : Name → Tm
 _·_   : Tm → Tm → Tm
 ƛ     : Name → Tm → Tm
 Let   : Name → Tm → Tm → Tm
```

Examples using this data type

Building terms

```
x, y :: Name
idTerm, apTerm :: Tm
idTerm = ƛ x (V x)
apTerm = ƛ x (ƛ y (V x · V y))
```

Analysing terms

```
size : Tm → ℕ
size (V _)          = 1
size (t · u)        = 1 + size t + size u
size (ƛ _ t)        = 1 + size t
size (Let _ t u)    = 1 + size t + size u
```

Nothing more!

Something less?

$\lambda\, x\, e \cdot V\, y$

Shall we compare x and y?

$\lambda\, x\, e \cdot V\, y$

Shall we compare x and y?

No!

Thou Shalt Not Discriminate α-equivalent terms!

Comparison is World Homogeneous

$$\_\equiv\_ : \forall\ \{\alpha\} \to \texttt{Name}\ \alpha \to \texttt{Name}\ \alpha \to \texttt{Bool}$$

Names inhabit worlds

```
World  : Set
Name   : World → Set
∅      : World
```

Worlds will be either ∅ or abstract variables (like α,β)

$$(\lambda\, \mathtt{x}_{\beta}\ (\mathsf{V}\, \mathtt{z}_{\beta}))\ \cdot \mathsf{V}\, \mathtt{y}_{\alpha}$$

$$(\lambda\, \mathtt{x}_{\beta}\ (\mathsf{V}\, \mathtt{z}_{\beta}))\ \cdot \mathsf{V}\, \mathtt{y}_{\alpha}$$

- Comparing $\mathtt{x}$ and $\mathtt{y}$ is now forbidden!

$$(\lambda\, \mathtt{x}_{\beta}\ (\mathsf{V}\, \mathtt{z}_{\beta}))\ \cdot \mathsf{V}\, \mathtt{y}_{\alpha}$$

- Comparing $\mathtt{x}$ and $\mathtt{y}$ is now forbidden!
- Comparing $\mathtt{x}$ and $\mathtt{z}$ is allowed.

$$(\lambda\, \mathtt{x}_{\beta}\ (\mathsf{V}\, \mathtt{z}_{\beta}))\ \cdot \mathsf{V}\, \mathtt{y}_{\alpha}$$

- Comparing $\mathtt{x}$ and $\mathtt{y}$ is now forbidden!
- Comparing $\mathtt{x}$ and $\mathtt{z}$ is allowed.
- One could expect to be able to compare $\mathtt{z}$ and $\mathtt{y}$ when $\mathtt{x}$ and $\mathtt{z}$ are distinct!

$$(\lambda\, \mathtt{x}_{\beta}\ (\mathsf{V}\, \mathtt{z}_{\beta}))\ \cdot \mathsf{V}\, \mathtt{y}_{\alpha}$$

- Comparing $\mathtt{x}$ and $\mathtt{y}$ is now forbidden!
- Comparing $\mathtt{x}$ and $\mathtt{z}$ is allowed.
- One could expect to be able to compare $\mathtt{z}$ and $\mathtt{y}$ when $\mathtt{x}$ and $\mathtt{z}$ are distinct!

We need links between worlds!

A type for binders

$$\texttt{Binder}\ \alpha\ \beta : \texttt{Set}$$

- A more precise type for a Name β
- Where α is the outer world
- And β is the inner world

```
nameOfBinder : ∀ {α β} → Binder α β → Name β
```

Example: Data type for the untyped lambda calculus

```
data Tm : Set where
 V     : Name → Tm
 _·_   : Tm → Tm → Tm
 ƛ     : Name → Tm → Tm
 Let   : Name → Tm → Tm → Tm
```

Example: Data type for the untyped lambda calculus

```
data Tm α : Set where
 V     : Name α → Tm α
 _·_   : Tm α → Tm α → Tm α
 ƛ     : ∀{β} → Binder α β → Tm β → Tm α
 Let   : ∀{β} → Binder α β → Tm α → Tm β → Tm α
```

Name abstraction is existential quantification over scopes

```
Abs : (World → Set) → World → Set
Abs F α = ∃ β, Binder α β × F β
```

Example: Data type for the untyped lambda calculus

```
data Tm α : Set where
 V    : Name α → Tm α
 _·_  : Tm α → Tm α → Tm α
 ƛ    : Abs Tm α → Tm α
 Let  : Abs (λ β → Tm α × Tm β) α → Tm α
```

Computing the size of a term

```
size : Tm → ℕ
size (V _)          = 1
size (t · u)        = 1 + size t + size u
size (ƛ _ t)        = 1 + size t
size (Let _ t u)    = 1 + size t + size u
```

Computing the size of a term

```
size : ∀ {α} → Tm α → ℕ
size (V _)          = 1
size (t · u)        = 1 + size t + size u
size (ƛ _ t)        = 1 + size t
size (Let _ t u)    = 1 + size t + size u
```

Computing the size of a term

```
size : ∀ {α} → Tm α → ℕ
size (V _)          = 1
size (t · u)        = 1 + size t + size u
size (ƛ _ t)        = 1 + size t
size (Let _ t u)    = 1 + size t + size u
```

- Polymorphic recursion at work!

Computing the size of a term

```
size : ∀ {α} → Tm α → ℕ
size (V _)          = 1
size (t · u)        = 1 + size t + size u
size (ƛ _ t)        = 1 + size t
size (Let _ t u)    = 1 + size t + size u
```

- Polymorphic recursion at work!
- No freshening! (neither explicit nor implicit)

Computing the size of a term

```
size : ∀ {α} → Tm α → ℕ
size (V _)          = 1
size (t · u)        = 1 + size t + size u
size (ƛ _ t)        = 1 + size t
size (Let _ t u)    = 1 + size t + size u
```

- Polymorphic recursion at work!
- No freshening! (neither explicit nor implicit)
- No shifting!

Exporting a name through a binder

```
exportBinder : ∀ {α β} (x : Binder α β) (y : Name β)
               → Maybe (Name α)
```

- Returns `just y` when `y` and `nameOfBinder x` are distinct

Exporting a name through a binder

```
exportBinder : ∀ {α β} (x : Binder α β) (y : Name β)
               → Maybe (Name α)
```

- Returns `just y` when `y` and `nameOfBinder x` are distinct
- Performs a type refinement on success

Exporting a name through a binder

```
exportBinder : ∀ {α β} (x : Binder α β) (y : Name β)
               → Maybe (Name α)
```

- Returns `just y` when `y` and `nameOfBinder x` are distinct
- Performs a type refinement on success
- Partial!

Computing free variables

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

Computing free variables

```
rm : Name → List Name → List Name
rm _  []                      = []
rm x  (y :: ys)  with x ≡ y
...              | false      = y :: rm x ys
...              | true       = rm x ys
```

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

Computing free variables

```
rm : Name → List Name → List Name
rm _  []                      = []
rm x  (y :: ys)  with x ≡ y
...              | false      = y :: rm x ys
...              | true       = rm x ys
```

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

Computing free variables

```
rm : ∀ {α β} → Binder α β → List (Name β) → List (Name α)
rm _  []                        = []
rm x  (y :: ys)  with exportBinder x y
...              | just y   = y :: rm x ys
...              | nothing  = rm x ys
```

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

Two implementations

Two implementations

Nominal

$$
\begin{aligned}
\texttt{World} &\overset{\text{def}}{=} \texttt{Pred}\ \mathbb{N} \\
\texttt{Name}\ \beta &\overset{\text{def}}{=} \{\ \mathsf{a} : \mathbb{N} \mid \mathsf{a} \in \beta\ \} \\
\texttt{Binder}\ \alpha\ \beta &\overset{\text{def}}{=} \{\ \mathsf{a} : \mathbb{N} \mid \beta = \{\mathsf{a}\} \cup \alpha\ \} \\
\_\equiv\_ &\overset{\text{def}}{=} \_\equiv_{\mathbb{N}}\_ \\
\texttt{exportBinder}\ \mathsf{a}\ \mathsf{b} &\overset{\text{def}}{=} \text{if}\ \mathsf{a} \equiv_{\mathbb{N}} \mathsf{b}\ \text{then}\ \texttt{nothing}\ \text{else}\ \texttt{just}\ \mathsf{b} \\
\texttt{nameOfBinder}\ \mathsf{a} &\overset{\text{def}}{=} \mathsf{a}
\end{aligned}
$$

Two implementations

De Bruijn indices

$$
\begin{aligned}
\texttt{World} &\overset{\text{def}}{=} \mathbb{N} \\
\texttt{Name}\ \beta &\overset{\text{def}}{=} \texttt{Fin}\ \beta \\
&\quad \text{(i.e. } [0,\beta)\text{)} \\
\texttt{Binder}\ \alpha\ \beta &\overset{\text{def}}{=} \beta = (\alpha + 1) \\
\_\equiv\_ &\overset{\text{def}}{=} \_\equiv_{\mathbb{N}}\_ \\
\text{exportBinder}\ ()\ 0 &\overset{\text{def}}{=} \texttt{nothing} \\
\text{exportBinder}\ ()\ (\texttt{suc}\ i) &\overset{\text{def}}{=} \texttt{just}\ i \\
\texttt{nameOfBinder}\ () &\overset{\text{def}}{=} 0
\end{aligned}
$$

Well-typed programs preserve α-equivalence!

What do we mean by α-equivalence?

- One builds a logical relation indexed by types
- It models observational equivalence between programs
- It coincides with traditional α-equivalence at type $\mathtt{Tm}$ (for instance)
- It extends α-equivalence at richer types

Free theorems

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

- Every name that occurs free in `f x` must occur free in `x`
- Furthermore, `f` commutes with any renaming
- Using well-scoped De Bruijn indices would not yield the latter guarantee

The paper:

- describes more primitive machinery to safely import or generate names
- shows how to lift operations on names (equality test, import, export...) to types like `Tm`
- offers an advanced example (NBE)
- explores a HOAS-like form of abstraction:

$$\texttt{FunAbs F } \alpha \texttt{ = } \forall\ \{\beta\} \rightarrow \alpha \subseteq \beta \rightarrow \texttt{F } \beta \rightarrow \texttt{F } \beta$$

Conclusion

- Explicit scopes using world indices
- No freshening when pattern-matching abstractions
- Name abstraction as existential quantification
- Expressiveness close to a manual model with names
- Two concrete and safe implementations of the interface

Future Work: How to reason about programs using this model

Bonus tracks

- Capture avoiding substitution
- `occurs`
- `Star`
- `⋆Binder`
- `export⋆Binder`
- `fv'`
- De Bruijn indices canonicity: precisely
- Packaging
- `app`
- Contexts
- Patterns
- `nameEqH`

Example: Data type for the untyped lambda calculus

```
data Tm
 =  V     Name
 |  _·_   Tm Tm
 |  ƛ      Name Tm
 |  Let   Name Tm Tm
```

Example: Data type for the untyped lambda calculus

```
data Tm : Set where
 V    : Name → Tm
 _·_  : Tm → Tm → Tm
 ƛ    : Name → Tm → Tm
 Let  : Name → Tm → Tm → Tm
```

Example: Data type for the untyped lambda calculus

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

Example: Data type for the untyped lambda calculus

```
data Tm α
 =          V     (Name α)
 |          _·_   (Tm α) (Tm α)
 |  ∃ β.   ƛ     (Binder α β)   (Tm β)
 |  ∃ β.   Let   (Binder α β)   (Tm α) (Tm β)
```

Example: Data type for the untyped lambda calculus

```
data Tm α : Set where
 V     : Name α → Tm α
 _·_   : Tm α → Tm α → Tm α
 ƛ     : ∀{β} → Binder α β → Tm β → Tm α
 Let   : ∀{β} → Binder α β → Tm α → Tm β → Tm α
```

Example: Data type for the untyped lambda calculus

```
data Tm α : Set where
 V    : Name α → Tm α
 _·_  : Tm α → Tm α → Tm α
 ƛ    : Abs Tm α → Tm α
 Let  : Abs (λ β → Tm α × Tm β) α → Tm α
```

Example: Data type for the untyped lambda calculus

```
data Tm α : Set where
 V    : (x : Name α) → Tm α
 _·_  : (t u : Tm α) → Tm α
 ƛ    : ∀{β} (x : Binder α β) (t : Tm β) → Tm α
 Let  : ∀{β} (x : Binder α β) (t : Tm α) (u : Tm β) → Tm α
```

Capture avoiding substitution

```
module Substitution where
  Fresh×Name→F : (F : World → Set) (α β : World) → Set
  Fresh×Name→F F α β = Fresh β × (Name α → F β)

  commθ : ∀ {F} → Import⊆ F → (∀ {α} → Name α → F α)
                → Comm (Fresh×Name→F F)
  commθ imp var x (y , f)
    = (_ , (nextOf y , f' ∘ exportBinder x) , weakOf y)
    where
      open FreshPack
      f' = maybe (imp (⊆Of y) ∘ f) (var (nameOf y))
```

◂ Back

Capture avoiding substitution (part 2)

```
open Substitution
substTm : ∀ {α β} → Fresh×Name→F Tm α β → Tm α → Tm β
substTm θ (V x) = proj₂ θ x
substTm θ (t · u)
  = substTm θ t · substTm θ u
substTm θ (ƛ x t)
    with commθ impTm V x θ
... | (_ , θ' , x') = ƛ x' (substTm θ' t)
substTm θ (Let x t u)
    with commθ impTm V x θ
... | (_ , θ' , x') = Let x' (substTm θ t) (substTm θ' u)
```

Occurs check

```
occurs : ∀ {α} → Name α → Tm α → Bool
occurs x₀ = occ (λ y → x₀ ≡ y)
  where
    OccEnv  : World → Set
    OccEnv  α = Name α → Bool
    extend  : ∀ {α β} → Binder α β → OccEnv α → OccEnv β
    extend  x Γ y = maybe Γ false (exportBinder x y)

    occ : ∀ {α} → OccEnv α → Tm α → Bool
    occ Γ  (V y)          = Γ y
    occ Γ  (t · u)        = occ Γ t ∨ occ Γ u
    occ Γ  (ƛ x t)        = occ (extend x Γ) t
    occ Γ  (Let x t u)    = occ Γ t ∨ occ (extend x Γ) u
```

◂ Back

Reflexive and transitive closure of a relation

```
Rel A = A → A → Set
```

```
data Star {I : Set} (T : Rel I) : Rel I where
  ε     : ∀ {i} → Star T i i
  _◁_   : ∀ {i j k} → T i j → Star T j k → Star T i k
```

```
Star⁻¹  : ∀ {I : Set} → Rel I → Rel I
Star⁻¹  = flip ∘ Star ∘ flip
```

◂ Back

Environments as a chain of binders

```
⋆Binder  : Rel World
⋆Binder  = Star⁻¹ Binder
```

```
  ϵ   : ∀ {α} → ⋆Binder α α
_◁_   : ∀ {α β γ}  (x : Binder β γ) (Γ : ⋆Binder α β)
                →  ⋆Binder α γ
```

◂ Back

Exporting through a chain of binders

```
export⋆Binder  : ∀ {α β}  → ⋆Binder α β
                          → Name β → Maybe (Name α)
export⋆Binder  ϵ          y
  = just y
export⋆Binder  (x ◁ Γ)  y
  = exportBinder x y >>= export⋆Binder Γ
  where open MaybeMonad
```

◂ Back

Free variables, continued...

```
fv'  : ∀ {β α} → ⋆Binder α β → Tm β → List (Name α)
fv'  Γ  (V x)          = List.fromMaybe (export⋆Binder Γ x)
fv'  Γ  (t · u)        = fv' Γ t ++ fv' Γ u
fv'  Γ  (ƛ x t)        = fv' (x ◁ Γ) t
fv'  Γ  (Let x t u)    = fv' Γ t ++ fv' (x ◁ Γ) u
```

◂ Back

De Bruijn indices canonicity: precisely

- Bound and fresh names uniquely chosen.
- Free names are freely chosen, though!
- Consequence: α-equivalence is not so simple!
- α-equivalences are not automatically preserved!

Packaging up...

- `binderOf`
- `nameOf`
- `nextOf`
- `importWith`

◂ Back

Constructing terms

```
app  : Tm ∅
app  = ƛ  (binderOf x) (ƛ (binderOf y)
          (V (importWith y (nameOf x)) · V (nameOf y)))
  where  open FreshPack
         x  = fresh∅
         y  = nextOf x
```

◂ Back

Towards elaborate uses of worlds

```
data C α : (β : World) → Set where
  Hole   : C α α
  _·1_   : ∀ {β} (c : C α β) (t : Tm α) → C α β
  _·2_   : ∀ {β} (t : Tm α) (c : C α β) → C α β
  ƛ      : ∀ {β γ}  (x : Binder α β) (τ : Ty)
                    (c : C β γ) → C α γ
  Let₁   : ∀ {β γ}  (x : Binder α β) (c : C α γ)
                    (t : Tm β) → C α γ
  Let₂   : ∀ {β γ}  (x : Binder α β) (t : Tm α)
                    (c : C β γ) → C α γ
```

◂ Back

Patterns types

```
data Pa α β : Set where
  PVar   : Binder α β → Pa α β
  PPair  : ∀{γ} → Pa α γ → Pa γ β → Pa α β
```

◂ Back

Safe heterogeneous comparison!

```
nameEqH : ∀{α β}→ Name α → Name β → Binder α β → Bool
```

```
nameEqH x y lnk with exportBinder lnk y
...                | just y' = x ≡ y'
...                | nothing = false
```

◂ Back