A fresh look at programming with names and binders

Nicolas Pouillard and François Pottier

{Nicolas.Pouillard,Francois.Pottier}@inria.fr

Gallium Seminar, May 2010

Towards safer and more expressive languages for meta-programming

Program representation should stay well-typed and **well-scoped**

What do we need to program with names and binders?

A type for names

A type for names

```
Name : Set
```

A type for names

```
Name : Set
```

An equality test for names

A type for names

```
Name : Set
```

An equality test for names

```
_≟Name_ : Name → Name → Bool
```

A type for names

```
Name : Set
```

An equality test for names

```
_≟Name_ : Name → Name → Bool
```

A generator of fresh names

A type for names

```
Name : Set
```

An equality test for names

```
_≟Name_ : Name → Name → Bool
```

A generator of fresh names

```
fresh : () → Name
```

A type for names

```
Name : Set
```

An equality test for names

```
_≟Name_ : Name → Name → Bool
```

A generator of fresh names

```
fresh : () → Name
```

Alternatively: An initial fresh name

A type for names

```
Name : Set
```

An equality test for names

```
_≟Name_ : Name → Name → Bool
```

A generator of fresh names

```
fresh : () → Name
```

Alternatively: An initial fresh name

```
init : Name
```

A type for names

```
Name : Set
```

An equality test for names

```
_≟Name_ : Name → Name → Bool
```

A generator of fresh names

```
fresh : () → Name
```

Alternatively: An initial fresh name

```
init : Name
```

And: How to pick a new one

A type for names

```
Name : Set
```

An equality test for names

```
_≟Name_ : Name → Name → Bool
```

A generator of fresh names

```
fresh : () → Name
```

Alternatively: An initial fresh name

```
init : Name
```

And: How to pick a new one

```
next : Name → Name
```

Data type for the untyped lambda calculus

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

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
```

Nothing more!

Something less?

$$(\lambda \ \mathtt{x} \rightarrow \mathtt{e}) \ \mathtt{x}$$

can we compare the two '`x`'s?

Comparison is World Homogeneous

$_\stackrel{?}{=}\texttt{Name}_ : \forall\ \{\alpha\} \to \texttt{Name}\ \alpha \to \texttt{Name}\ \alpha \to \texttt{Bool}$

Names inhabit worlds

- `World` : `Set`
- `Name` : `World` $\rightarrow$ `Set`
- $\emptyset$: `World`

Example:

α : `World`
`x` : `Name` α

$$\neg\texttt{name}\emptyset \;:\; \texttt{Name}\ \emptyset \to \bot$$

$$(\lambda\ \mathtt{x}_{\beta}.\ \ \mathtt{e}_{\beta})\ \mathtt{x}_{\alpha}$$

$$(\lambda\ \mathtt{x}_{\beta}.\ \ \mathtt{e}_{\beta})\ \ \mathtt{x}_{\alpha}$$

$$(\lambda\ \mathtt{x}_{\beta}.\ \ \mathtt{y}_{\beta})\ \ \mathtt{y}_{\alpha}$$

$$(\lambda\ \mathtt{x}_{\beta}.\ \ \mathtt{e}_{\beta})\ \mathtt{x}_{\alpha}$$

$$(\lambda\ \mathtt{x}_{\beta}.\ \ \mathtt{y}_{\beta})\ \mathtt{y}_{\alpha}$$

We need links between worlds!

A type for binders

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

- α outer world / β inner world

A type for binders

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

- α outer world / β inner world
- A more precise type for a Name β

A type for binders

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

- α outer world / β inner world
- A more precise type for a Name β
- `nameOfBinder` : $\forall\ \{\alpha\ \beta\} \to$ `Binder` $\alpha\ \beta \to$ `Name` β

A type for binders

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

- α outer world / β inner world
- A more precise type for a Name β
- `nameOfBinder` $: \forall\ \{\alpha\ \beta\} \rightarrow \texttt{Binder}\ \alpha\ \beta \rightarrow \texttt{Name}\ \beta$
- Using `Binder`, name shadowing is permitted

Name abstraction as existential quantification

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
```

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
```

Computing the size of a term

```
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 freshning!

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 freshning!
- No shifting!

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

- Compares `y` and `nameOfBinder x`

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

- Compares `y` and `nameOfBinder x`
- Performs a type refinement on success

$$\texttt{exportBinder :}\ \forall\ \{\alpha\ \beta\}\ \texttt{(x : Binder}\ \alpha\ \beta\texttt{) (y : Name}\ \beta\texttt{)}$$
$$\rightarrow \texttt{Maybe (Name}\ \alpha\texttt{)}$$

- Compares y and `nameOfBinder` x
- Performs a type refinement on success
- Partial!

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

- Compares y and `nameOfBinder` x
- Performs a type refinement on success
- Partial!
- Injective!

What about moving names the other way around?

What about moving names the other way around? No!

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 ≟Name 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 ≟Name 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)
```

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)
```

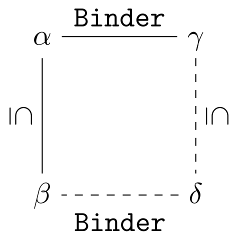
`fv` enjoys a free theorem!

Importing names

```
_⊆_           : Rel World
⊆-refl        : ∀ {α} → α ⊆ α
⊆-trans       : ∀ {α β γ} → α ⊆ β → β ⊆ γ → α ⊆ γ
import⊆       : ∀ {α β} → α ⊆ β → Name α → Name β
∅-bottom-⊆    : ∀ {α} → ∅ ⊆ α
```

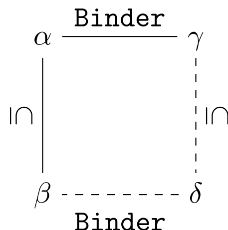
Importing names

```
_⊆_           : Rel World
⊆-refl        : ∀ {α} → α ⊆ α
⊆-trans       : ∀ {α β γ} → α ⊆ β → β ⊆ γ → α ⊆ γ
import⊆       : ∀ {α β} → α ⊆ β → Name α → Name β
∅-bottom-⊆    : ∀ {α} → ∅ ⊆ α
```



Importing names

```
_⊆_          : Rel World
⊆-refl       : ∀ {α} → α ⊆ α
⊆-trans      : ∀ {α β γ} → α ⊆ β → β ⊆ γ → α ⊆ γ
import⊆      : ∀ {α β} → α ⊆ β → Name α → Name β
∅-bottom-⊆   : ∀ {α} → ∅ ⊆ α
```




```
_Binder-commute-⊆_ : ∀ {α β γ} → Binder α γ → α ⊆ β
                     → ∃ λ δ   → γ ⊆ δ × Binder β δ
```

Strong binders

```
BINDER     : Rel World
weaken     : ∀ {α β} → BINDER α β → Binder α β
dropName   : ∀ {α β} → BINDER α β → α ⊆ β
```

Generating names

```
Fresh        : World → Set
Fresh α      = ∃ λ β → BINDER α β
fresh∅       : Fresh ∅
nextBINDER   : ∀ {α β γ}  → Binder α β → BINDER α γ
                          → Fresh β
```

Safety Properties

Well-typed programs...

- ... do not get stuck

Safety Properties

Well-typed programs...

- ... do not get stuck
- ... preserve α-equivalence

Safety Properties

Well-typed programs...

- ... do not get stuck
- ... preserve α-equivalence
- ... do not break name abstraction

Safety Properties

Well-typed programs...

- ... do not get stuck
- ... preserve α-equivalence
- ... do not break name abstraction
- ... do not mix names with different scopes

How it works?

Nominal implementation

$$\texttt{World} \stackrel{\mathsf{def}}{=} \texttt{Pred}\,\mathbb{N}$$

Nominal implementation

$$\begin{aligned} \texttt{World} &\stackrel{\text{def}}{=} \texttt{Pred}\,\mathbb{N} \\ \texttt{Name}\,\beta &\stackrel{\text{def}}{=} \{\, \texttt{a} : \mathbb{N} \mid \texttt{a} \in \beta \,\} \end{aligned}$$

Nominal implementation

$$
\begin{aligned}
\texttt{World} &\stackrel{\text{def}}{=} \texttt{Pred}\ \mathbb{N} \\
\texttt{Name}\ \beta &\stackrel{\text{def}}{=} \{\ \mathsf{a} : \mathbb{N} \mid \mathsf{a} \in \beta\ \} \\
\texttt{Binder}\ \alpha\ \beta &\stackrel{\text{def}}{=} \{\ \mathsf{a} : \mathbb{N} \mid \beta \equiv \{\mathsf{a}\} \cup \alpha\ \}
\end{aligned}
$$

Nominal implementation

$$
\begin{aligned}
\texttt{World} &\stackrel{\text{def}}{=} \texttt{Pred}\,\mathbb{N} \\
\texttt{Name}\,\beta &\stackrel{\text{def}}{=} \{\ \mathsf{a} : \mathbb{N} \mid \mathsf{a} \in \beta\ \} \\
\texttt{Binder}\,\alpha\,\beta &\stackrel{\text{def}}{=} \{\ \mathsf{a} : \mathbb{N} \mid \beta \equiv \{\mathsf{a}\} \cup \alpha\ \} \\
\texttt{BINDER}\,\alpha\,\beta &\stackrel{\text{def}}{=} \{\ \mathsf{a} : \mathbb{N} \mid \beta \equiv \{\mathsf{a}\} \cup \alpha \wedge \mathsf{a} > \alpha\ \}
\end{aligned}
$$

Nominal implementation

$$\begin{aligned}
\texttt{World} &\stackrel{\text{def}}{=} \texttt{Pred}\ \mathbb{N} \\
\texttt{Name}\ \beta &\stackrel{\text{def}}{=} \{\ \mathsf{a} : \mathbb{N} \mid \mathsf{a} \in \beta\ \} \\
\texttt{Binder}\ \alpha\ \beta &\stackrel{\text{def}}{=} \{\ \mathsf{a} : \mathbb{N} \mid \beta \equiv \{\mathsf{a}\} \cup \alpha\ \} \\
\texttt{BINDER}\ \alpha\ \beta &\stackrel{\text{def}}{=} \{\ \mathsf{a} : \mathbb{N} \mid \beta \equiv \{\mathsf{a}\} \cup \alpha \wedge \mathsf{a} > \alpha\ \} \\
\alpha \subseteq \beta &\stackrel{\text{def}}{=} \forall\ \mathsf{a},\ \mathsf{a} \in \alpha \Rightarrow \mathsf{a} \in \beta
\end{aligned}$$

Nominal implementation

$$
\begin{aligned}
\texttt{World} &\stackrel{\text{def}}{=} \texttt{Pred}\,\mathbb{N} \\
\texttt{Name}\,\beta &\stackrel{\text{def}}{=} \{\ \mathsf{a} : \mathbb{N} \mid \mathsf{a} \in \beta\ \} \\
\texttt{Binder}\,\alpha\,\beta &\stackrel{\text{def}}{=} \{\ \mathsf{a} : \mathbb{N} \mid \beta \equiv \{\mathsf{a}\} \cup \alpha\ \} \\
\texttt{BINDER}\,\alpha\,\beta &\stackrel{\text{def}}{=} \{\ \mathsf{a} : \mathbb{N} \mid \beta \equiv \{\mathsf{a}\} \cup \alpha \wedge \mathsf{a} > \alpha\ \} \\
\alpha \subseteq \beta &\stackrel{\text{def}}{=} \forall\ \mathsf{a},\ \mathsf{a} \in \alpha \Rightarrow \mathsf{a} \in \beta \\
_\stackrel{?}{=}_{\texttt{Name}}_ &\stackrel{\text{def}}{=} _\stackrel{?}{=}_{\mathbb{N}}_
\end{aligned}
$$

Nominal implementation

$$
\begin{aligned}
\texttt{World} &\stackrel{\text{def}}{=} \texttt{Pred}\ \mathbb{N} \\
\texttt{Name}\ \beta &\stackrel{\text{def}}{=} \{\ \mathsf{a} : \mathbb{N} \mid \mathsf{a} \in \beta\ \} \\
\texttt{Binder}\ \alpha\ \beta &\stackrel{\text{def}}{=} \{\ \mathsf{a} : \mathbb{N} \mid \beta \equiv \{\mathsf{a}\} \cup \alpha\ \} \\
\texttt{BINDER}\ \alpha\ \beta &\stackrel{\text{def}}{=} \{\ \mathsf{a} : \mathbb{N} \mid \beta \equiv \{\mathsf{a}\} \cup \alpha \land \mathsf{a} > \alpha\ \} \\
\alpha \subseteq \beta &\stackrel{\text{def}}{=} \forall\ \mathsf{a},\ \mathsf{a} \in \alpha \Rightarrow \mathsf{a} \in \beta \\
_ \stackrel{?}{=}_{\texttt{Name}} _ &\stackrel{\text{def}}{=} _ \stackrel{?}{=}_{\mathbb{N}} _ \\
\texttt{exportBinder}\ \mathsf{a}\ \mathsf{b} &\stackrel{\text{def}}{=} \text{if } \mathsf{a} \stackrel{?}{=}_{\mathbb{N}} \mathsf{b} \text{ then } \texttt{nothing} \text{ else } \texttt{just}\ \mathsf{b}
\end{aligned}
$$

Nominal implementation

$$
\begin{aligned}
\texttt{World} &\stackrel{\text{def}}{=} \texttt{Pred}\ \mathbb{N} \\
\texttt{Name}\ \beta &\stackrel{\text{def}}{=} \{\ \mathsf{a} : \mathbb{N} \mid \mathsf{a} \in \beta\ \} \\
\texttt{Binder}\ \alpha\ \beta &\stackrel{\text{def}}{=} \{\ \mathsf{a} : \mathbb{N} \mid \beta \equiv \{\mathsf{a}\} \cup \alpha\ \} \\
\texttt{BINDER}\ \alpha\ \beta &\stackrel{\text{def}}{=} \{\ \mathsf{a} : \mathbb{N} \mid \beta \equiv \{\mathsf{a}\} \cup \alpha \wedge \mathsf{a} > \alpha\ \} \\
\alpha \subseteq \beta &\stackrel{\text{def}}{=} \forall\ \mathsf{a},\ \mathsf{a} \in \alpha \Rightarrow \mathsf{a} \in \beta \\
_\stackrel{?}{=}_{\texttt{Name}}_ &\stackrel{\text{def}}{=} _\stackrel{?}{=}_{\mathbb{N}}_ \\
\texttt{exportBinder}\ \mathsf{a}\ \mathsf{b} &\stackrel{\text{def}}{=} \mathsf{if}\ \mathsf{a} \stackrel{?}{=}_{\mathbb{N}} \mathsf{b}\ \mathsf{then}\ \texttt{nothing}\ \mathsf{else}\ \texttt{just}\ \mathsf{b} \\
\texttt{nextBINDER}\ \mathsf{a}\ \mathsf{b} &\stackrel{\text{def}}{=} \mathsf{b} \sqcup (\mathsf{a} + 1)
\end{aligned}
$$

Nominal implementation

$$
\begin{aligned}
\texttt{World} &\stackrel{\text{def}}{=} \texttt{Pred}\ \mathbb{N} \\
\texttt{Name}\ \beta &\stackrel{\text{def}}{=} \{\ \mathsf{a} : \mathbb{N} \mid \mathsf{a} \in \beta\ \} \\
\texttt{Binder}\ \alpha\ \beta &\stackrel{\text{def}}{=} \{\ \mathsf{a} : \mathbb{N} \mid \beta \equiv \{\mathsf{a}\} \cup \alpha\ \} \\
\texttt{BINDER}\ \alpha\ \beta &\stackrel{\text{def}}{=} \{\ \mathsf{a} : \mathbb{N} \mid \beta \equiv \{\mathsf{a}\} \cup \alpha \wedge \mathsf{a} > \alpha\ \} \\
\alpha \subseteq \beta &\stackrel{\text{def}}{=} \forall\ \mathsf{a},\ \mathsf{a} \in \alpha \Rightarrow \mathsf{a} \in \beta \\
_ \stackrel{?}{=}_{\texttt{Name}} _ &\stackrel{\text{def}}{=} _ \stackrel{?}{=}_{\mathbb{N}} _ \\
\texttt{exportBinder}\ \mathsf{a}\ \mathsf{b} &\stackrel{\text{def}}{=} \mathsf{if}\ \mathsf{a} \stackrel{?}{=}_{\mathbb{N}} \mathsf{b}\ \mathsf{then}\ \texttt{nothing}\ \mathsf{else}\ \texttt{just}\ \mathsf{b} \\
\texttt{nextBINDER}\ \mathsf{a}\ \mathsf{b} &\stackrel{\text{def}}{=} \mathsf{b} \sqcup (\mathsf{a} + 1)
\end{aligned}
$$

`import⊆`, `nameOfBinder`, `weaken`, `dropName`, `Binder-commute-⊆`, `⊆-trans`, `⊆-refl` are computation free.

Why is this safe?

Logical relations recap

$$f\ (\tau \to \sigma)\ g \quad \stackrel{\text{def}}{=} \quad \forall\, v\ w,\ v\ (\tau)\ w \Rightarrow f\ v\ (\sigma)\ g\ w$$

Logical relations recap

$$
\begin{aligned}
f\ (\tau \to \sigma)\ g &\stackrel{\text{def}}{=} \forall\ v\ w,\ v\ (\tau)\ w \Rightarrow f\ v\ (\sigma)\ g\ w \\
(x_1,\ y_1)\ (\tau \times \sigma)\ (x_2,\ y_2) &\stackrel{\text{def}}{=} x_1\ (\tau)\ x_2 \wedge y_1\ (\sigma)\ y_2
\end{aligned}
$$

Logical relations recap

$$\begin{aligned}
f\ (\tau \to \sigma)\ g &\overset{\text{def}}{=} \forall\ v\ w,\ v\ (\tau)\ w \Rightarrow f\ v\ (\sigma)\ g\ w \\
(x_1,\ y_1)\ (\tau \times \sigma)\ (x_2,\ y_2) &\overset{\text{def}}{=} x_1\ (\tau)\ x_2 \wedge y_1\ (\sigma)\ y_2 \\
i_1\ (\mathbb{N})\ i_2 &\overset{\text{def}}{=} i_1 \equiv i_2
\end{aligned}$$

Worlds as atom bijections

Shadowing extension $\mathsf{a}_1\ ((\mathsf{b}_1,\mathsf{b}_2) \boxplus \alpha)\ \mathsf{a}_2$ holds if and only if either

Worlds as atom bijections

Shadowing extension $a_1 \; ((b_1,b_2) \boxplus \alpha) \; a_2$ holds if and only if either

- $a_1 \equiv b_1 \wedge a_2 \equiv b_2$ or,

Worlds as atom bijections

Shadowing extension $a_1\ ((b_1,b_2) \boxplus \alpha)\ a_2$ holds if and only if either

- $a_1 \equiv b_1 \wedge a_2 \equiv b_2$ or,
- $a_1 \not\equiv b_1 \wedge a_2 \not\equiv b_2 \wedge a_1\ (\alpha)\ a_2$

Worlds as atom bijections

Shadowing extension $\mathsf{a}_1\ ((\mathsf{b}_1,\mathsf{b}_2) \boxplus \alpha)\ \mathsf{a}_2$ holds if and only if either

- $\mathsf{a}_1 \equiv \mathsf{b}_1 \wedge \mathsf{a}_2 \equiv \mathsf{b}_2$ or,
- $\mathsf{a}_1 \not\equiv \mathsf{b}_1 \wedge \mathsf{a}_2 \not\equiv \mathsf{b}_2 \wedge \mathsf{a}_1\ (\alpha)\ \mathsf{a}_2$

Fresh extension $(\mathsf{b}_1,\mathsf{b}_2)\alpha$ is

Worlds as atom bijections

Shadowing extension $a_1 \; ((b_1, b_2) \boxplus \alpha) \; a_2$ holds if and only if either

- $a_1 \equiv b_1 \wedge a_2 \equiv b_2$ or,
- $a_1 \not\equiv b_1 \wedge a_2 \not\equiv b_2 \wedge a_1 \; (\alpha) \; a_2$

Fresh extension $(b_1, b_2)\alpha$ is

- defined only if $b_1 > \mathtt{dom}\,(\alpha)$ and $b_2 > \mathtt{codom}\,(\alpha)$

Worlds as atom bijections

Shadowing extension $\mathsf{a}_1 \; ((\mathsf{b}_1,\mathsf{b}_2) \boxplus \alpha) \; \mathsf{a}_2$ holds if and only if either

- $\mathsf{a}_1 \equiv \mathsf{b}_1 \wedge \mathsf{a}_2 \equiv \mathsf{b}_2$ or,
- $\mathsf{a}_1 \not\equiv \mathsf{b}_1 \wedge \mathsf{a}_2 \not\equiv \mathsf{b}_2 \wedge \mathsf{a}_1 \; (\alpha) \; \mathsf{a}_2$

Fresh extension $(\mathsf{b}_1,\mathsf{b}_2)\alpha$ is

- defined only if $\mathsf{b}_1 > \mathtt{dom}(\alpha)$ and $\mathsf{b}_2 > \mathtt{codom}(\alpha)$
- equal to $\{(\mathsf{b}_1,\mathsf{b}_2)\} \cup \alpha$ when defined

Nominal logical relations

$$\mathsf{a}_1 \ (\mathtt{Name}\ \beta) \ \mathsf{a}_2 \quad \stackrel{\mathsf{def}}{=} \quad \mathsf{a}_1 \ (\beta) \ \mathsf{a}_2$$

Nominal logical relations

$$
\begin{aligned}
\mathsf{a}_1\ (\mathtt{Name}\ \beta)\ \mathsf{a}_2 &\overset{\text{def}}{=} \mathsf{a}_1\ (\beta)\ \mathsf{a}_2 \\
\mathsf{a}_1\ (\mathtt{Binder}\ \alpha\ \beta)\ \mathsf{a}_2 &\overset{\text{def}}{=} \beta \equiv (\mathsf{a}_1,\mathsf{a}_2) \boxplus \alpha
\end{aligned}
$$

Nominal logical relations

$$
\begin{aligned}
\mathsf{a}_1\,(\mathtt{Name}\;\beta)\,\mathsf{a}_2 &\stackrel{\text{def}}{=} \mathsf{a}_1\,(\beta)\,\mathsf{a}_2 \\
\mathsf{a}_1\,(\mathtt{Binder}\;\alpha\;\beta)\,\mathsf{a}_2 &\stackrel{\text{def}}{=} \beta \equiv (\mathsf{a}_1,\mathsf{a}_2) \boxplus \alpha \\
\mathsf{a}_1\,(\mathtt{BINDER}\;\alpha\;\beta)\,\mathsf{a}_2 &\stackrel{\text{def}}{=} \beta \equiv (\mathsf{a}_1,\mathsf{a}_2)\;\alpha
\end{aligned}
$$

Nominal logical relations

$$
\begin{aligned}
\mathsf{a}_1\ (\texttt{Name}\ \beta)\ \mathsf{a}_2 &\stackrel{\text{def}}{=} \mathsf{a}_1\ (\beta)\ \mathsf{a}_2 \\
\mathsf{a}_1\ (\texttt{Binder}\ \alpha\ \beta)\ \mathsf{a}_2 &\stackrel{\text{def}}{=} \beta \equiv (\mathsf{a}_1,\mathsf{a}_2) \boxplus \alpha \\
\mathsf{a}_1\ (\texttt{BINDER}\ \alpha\ \beta)\ \mathsf{a}_2 &\stackrel{\text{def}}{=} \beta \equiv (\mathsf{a}_1,\mathsf{a}_2)\ \alpha \\
()\ (\alpha \subseteq \beta)\ () &\stackrel{\text{def}}{=} \alpha \subseteq \beta
\end{aligned}
$$

Soundness theorem for nominal logical relations

Theorem

Every primitive operation p of type τ is related to itself at type τ

Come on, De Bruijn indices are much simpler

De Bruijn implementation (types)

$$\texttt{World} \stackrel{\text{def}}{=} \mathbb{N}$$

De Bruijn implementation (types)

$$\begin{aligned} \texttt{World} &\stackrel{\text{def}}{=} \mathbb{N} \\ \texttt{Name}\,\beta &\stackrel{\text{def}}{=} \texttt{Fin}\,\beta \Leftrightarrow [0,\beta) \end{aligned}$$

De Bruijn implementation (types)

$$
\begin{aligned}
\texttt{World} &\overset{\text{def}}{=} \mathbb{N} \\
\texttt{Name}\,\beta &\overset{\text{def}}{=} \texttt{Fin}\,\beta \Leftrightarrow [0,\beta) \\
\texttt{Binder}\,\alpha\,\beta &\overset{\text{def}}{=} \beta \equiv (\alpha + 1)
\end{aligned}
$$

De Bruijn implementation (types)

$$
\begin{aligned}
\texttt{World} &\overset{\text{def}}{=} \mathbb{N} \\
\texttt{Name}\ \beta &\overset{\text{def}}{=} \texttt{Fin}\ \beta \Leftrightarrow [0,\beta) \\
\texttt{Binder}\ \alpha\ \beta &\overset{\text{def}}{=} \beta \equiv (\alpha + 1) \\
\texttt{BINDER} &\overset{\text{def}}{=} \texttt{Binder}
\end{aligned}
$$

De Bruijn implementation (types)

$$
\begin{aligned}
\texttt{World} &\stackrel{\mathsf{def}}{=} \mathbb{N} \\
\texttt{Name}\ \beta &\stackrel{\mathsf{def}}{=} \texttt{Fin}\ \beta \Leftrightarrow [0,\beta) \\
\texttt{Binder}\ \alpha\ \beta &\stackrel{\mathsf{def}}{=} \beta \equiv (\alpha + 1) \\
\texttt{BINDER} &\stackrel{\mathsf{def}}{=} \texttt{Binder} \\
\alpha \subseteq \beta &\stackrel{\mathsf{def}}{=} \{\ \mathsf{k} : \mathbb{N} \mid \beta \equiv \alpha + \mathsf{k}\ \}
\end{aligned}
$$

De Bruijn implementation (primitives)

$$- \stackrel{?}{=}_{\texttt{Name}} - \quad \stackrel{\text{def}}{=} \quad - \stackrel{?}{=}_{\mathbb{N}} -$$

De Bruijn implementation (primitives)

$$_\overset{?}{=}_{\texttt{Name}}_ \overset{\text{def}}{=} _\overset{?}{=}_{\mathbb{N}}_$$

$$\text{exportBinder}\ ()\ 0 \overset{\text{def}}{=} \texttt{nothing}$$

De Bruijn implementation (primitives)

$$
\begin{aligned}
_\stackrel{?}{=}_{\texttt{Name}}_ &\stackrel{\text{def}}{=} _\stackrel{?}{=}_{\mathbb{N}}_ \\
\text{exportBinder}\ ()\ 0 &\stackrel{\text{def}}{=} \texttt{nothing} \\
\text{exportBinder}\ ()\ (\texttt{suc}\ i) &\stackrel{\text{def}}{=} \texttt{just}\ i
\end{aligned}
$$

De Bruijn implementation (primitives)

$$
\begin{aligned}
_\overset{?}{=}_{\texttt{Name}}_ &\overset{\text{def}}{=} _\overset{?}{=}_{\mathbb{N}}_ \\
\text{exportBinder}\ ()\ 0 &\overset{\text{def}}{=} \texttt{nothing} \\
\text{exportBinder}\ ()\ (\texttt{suc}\ i) &\overset{\text{def}}{=} \texttt{just}\ i \\
\texttt{import}\subseteq\ k\ i &\overset{\text{def}}{=} (k + i)
\end{aligned}
$$

De Bruijn implementation (primitives)

$$
\begin{aligned}
_\overset{?}{=}_{\texttt{Name}}_ &\overset{\text{def}}{=} _\overset{?}{=}_{\mathbb{N}}_ \\
\text{exportBinder}\ ()\ 0 &\overset{\text{def}}{=} \texttt{nothing} \\
\text{exportBinder}\ ()\ (\texttt{suc}\ i) &\overset{\text{def}}{=} \texttt{just}\ i \\
\texttt{import}\subseteq\ k\ i &\overset{\text{def}}{=} (k + i) \\
\subseteq\texttt{-trans}\ k\ k' &\overset{\text{def}}{=} (k + k')
\end{aligned}
$$

De Bruijn implementation (primitives)

$$
\begin{aligned}
_\overset{?}{=}_{\texttt{Name}}_ &\overset{\text{def}}{=} _\overset{?}{=}_{\mathbb{N}}_ \\
\text{exportBinder}\ ()\ 0 &\overset{\text{def}}{=} \texttt{nothing} \\
\text{exportBinder}\ ()\ (\texttt{suc}\ i) &\overset{\text{def}}{=} \texttt{just}\ i \\
\texttt{import}\subseteq\ k\ i &\overset{\text{def}}{=} (k + i) \\
\subseteq\texttt{-trans}\ k\ k' &\overset{\text{def}}{=} (k + k') \\
\texttt{nameOfBinder}\ () &\overset{\text{def}}{=} 0
\end{aligned}
$$

De Bruijn implementation (primitives)

$$
\begin{aligned}
_\overset{?}{=}_{\texttt{Name}}_ &\overset{\text{def}}{=} _\overset{?}{=}_{\mathbb{N}}_ \\
\text{exportBinder}\ ()\ 0 &\overset{\text{def}}{=} \texttt{nothing} \\
\text{exportBinder}\ ()\ (\texttt{suc}\ i) &\overset{\text{def}}{=} \texttt{just}\ i \\
\texttt{import}\subseteq\ k\ i &\overset{\text{def}}{=} (k + i) \\
\subseteq\texttt{-trans}\ k\ k' &\overset{\text{def}}{=} (k + k') \\
\texttt{nameOfBinder}\ () &\overset{\text{def}}{=} 0 \\
\texttt{dropName}\ () &\overset{\text{def}}{=} 1
\end{aligned}
$$

De Bruijn implementation (primitives)

$$
\begin{aligned}
_\overset{?}{=}_{\texttt{Name}}_ &\overset{\text{def}}{=} _\overset{?}{=}_{\mathbb{N}}_ \\
\text{exportBinder}\ ()\ 0 &\overset{\text{def}}{=} \texttt{nothing} \\
\text{exportBinder}\ ()\ (\texttt{suc}\ i) &\overset{\text{def}}{=} \texttt{just}\ i \\
\texttt{import}\subseteq\ k\ i &\overset{\text{def}}{=} (k + i) \\
\subseteq\texttt{-trans}\ k\ k' &\overset{\text{def}}{=} (k + k') \\
\texttt{nameOfBinder}\ () &\overset{\text{def}}{=} 0 \\
\texttt{dropName}\ () &\overset{\text{def}}{=} 1 \\
\subseteq\texttt{-refl} &\overset{\text{def}}{=} 0
\end{aligned}
$$

De Bruijn implementation (primitives)

$$
\begin{aligned}
_\overset{?}{=}_{\texttt{Name}}_ &\overset{\text{def}}{=} _\overset{?}{=}_{\mathbb{N}}_ \\
\text{exportBinder}\ ()\ 0 &\overset{\text{def}}{=} \texttt{nothing} \\
\text{exportBinder}\ ()\ (\texttt{suc}\ i) &\overset{\text{def}}{=} \texttt{just}\ i \\
\texttt{import}\!\subseteq k\ i &\overset{\text{def}}{=} (k + i) \\
\subseteq\!\texttt{-trans}\ k\ k' &\overset{\text{def}}{=} (k + k') \\
\texttt{nameOfBinder}\ () &\overset{\text{def}}{=} 0 \\
\texttt{dropName}\ () &\overset{\text{def}}{=} 1 \\
\subseteq\!\texttt{-refl} &\overset{\text{def}}{=} 0
\end{aligned}
$$

`nextBINDER`, `weaken`, and `Binder-commute-⊆` are computation free.

(well-scoped) De Bruijn indices advantages

Canonical representation

(well-scoped) De Bruijn indices advantages

Canonical representation

- α-equivalence made simple!

(well-scoped) De Bruijn indices advantages

Canonical representation

- α-equivalence made simple!
- Structural induction!

(well-scoped) De Bruijn indices advantages

Canonical representation

- α-equivalence made simple!
- Structural induction!

Guaranteed well-formness

(well-scoped) De Bruijn indices advantages

Canonical representation

- α-equivalence made simple!
- Structural induction!

Guaranteed well-formness

- The `Fin` *n* technique is well known for that

(well-scoped) De Bruijn indices advantages

Canonical representation

- α-equivalence made simple!
- Structural induction!

Guaranteed well-formness

- The `Fin` *n* technique is well known for that
- Every variable has to be smaller than *n*

It is trivially safe, right?

It is trivially safe, right? No!

De Bruijn indices canonicity: precisely

- Bound and fresh names uniquely chosen.

De Bruijn indices canonicity: precisely

- Bound and fresh names uniquely chosen.
- Free names are freely chosen, though!

De Bruijn indices canonicity: precisely

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

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!

Worlds as bijections again

Shifting $\alpha\uparrow \stackrel{\mathsf{def}}{=} \{(0,0)\} \cup \{(\mathsf{i}_1{+}1,\mathsf{i}_2{+}1) \mid (\mathsf{i}_1,\mathsf{i}_2) \in \alpha\}$

Worlds as bijections again

Shifting $\alpha\uparrow \stackrel{\text{def}}{=} \{(0,0)\} \cup \{(\mathsf{i}_1{+}1,\mathsf{i}_2{+}1) \mid (\mathsf{i}_1,\mathsf{i}_2) \in \alpha\}$

Iterated shifting We write $\alpha\uparrow^k$ for the result of shifting the world α k times

De Bruijn logical relations

$$i_1 \; (\texttt{Name} \; \beta) \; i_2 \;\; \stackrel{\text{def}}{=} \;\; i_1 \; (\beta) \; i_2$$

De Bruijn logical relations

$$\begin{aligned} i_1 \; (\texttt{Name}\ \beta) \; i_2 &\stackrel{\text{def}}{=} i_1 \; (\beta) \; i_2 \\ \texttt{()} \; (\texttt{Binder}\ \alpha\ \beta) \; \texttt{()} &\stackrel{\text{def}}{=} \beta \equiv \alpha \uparrow \end{aligned}$$

De Bruijn logical relations

$$
\begin{aligned}
i_1 \; (\texttt{Name}\;\beta) \; i_2 &\stackrel{\mathsf{def}}{=} i_1 \; (\beta) \; i_2 \\
\texttt{()} \; (\texttt{Binder}\;\alpha\;\beta) \; \texttt{()} &\stackrel{\mathsf{def}}{=} \beta \equiv \alpha \uparrow \\
\texttt{()} \; (\texttt{BINDER}\;\alpha\;\beta) \; \texttt{()} &\stackrel{\mathsf{def}}{=} \beta \equiv \alpha \uparrow
\end{aligned}
$$

De Bruijn logical relations

$$
\begin{aligned}
i_1\ (\texttt{Name}\ \beta)\ i_2 &\stackrel{\text{def}}{=} i_1\ (\beta)\ i_2 \\
()\ (\texttt{Binder}\ \alpha\ \beta)\ () &\stackrel{\text{def}}{=} \beta \equiv \alpha\uparrow \\
()\ (\texttt{BINDER}\ \alpha\ \beta)\ () &\stackrel{\text{def}}{=} \beta \equiv \alpha\uparrow \\
k\ (\alpha \subseteq \beta)\ k &\stackrel{\text{def}}{=} \beta \equiv \alpha\uparrow^k
\end{aligned}
$$

Soundness theorem for De Bruijn logical relations

Theorem

Every primitive operation p of type τ is related to itself at type τ

Listings of some advanced examples

- Deciding α-equivalence on types like `Tm`

Listings of some advanced examples

- Deciding α-equivalence on types like `Tm`
- Generic export through a binder

Listings of some advanced examples

- Deciding α-equivalence on types like `Tm`
- Generic export through a binder
- Generic import through a world inclusion witness

Listings of some advanced examples

- Deciding α-equivalence on types like `Tm`
- Generic export through a binder
- Generic import through a world inclusion witness
- Normalisation by evaluation

Challenges and future work

- More powerful logical relations (relating to an abstract model)

Challenges and future work

- More powerful logical relations (relating to an abstract model)
- How to carry information on binders (types, specs...)

Challenges and future work

- More powerful logical relations (relating to an abstract model)
- How to carry information on binders (types, specs...)
- More formalisation of the logical relations

Challenges and future work

- More powerful logical relations (relating to an abstract model)
- How to carry information on binders (types, specs...)
- More formalisation of the logical relations
- More advanced usages

Conclusion

- Explicit scopes using world indices

Conclusion

- Explicit scopes using world indices
- Non-freshening opening

Conclusion

- Explicit scopes using world indices
- Non-freshening opening
- Name abstraction as existential quantification

Conclusion

- Explicit scopes using world indices
- Non-freshening opening
- Name abstraction as existential quantification
- Expressiveness close to a manual model with names

Conclusion

- Explicit scopes using world indices
- Non-freshening opening
- Name abstraction as existential quantification
- Expressiveness close to a manual model with names
- Two concrete and safe implementations of the interface

Questions?

Bonus track

- `occurs`
- `Star`
- `⋆Binder`
- `export⋆Binder`
- `fv'`
- Packaging
- `app`
- Contexts
- Patterns
- `nameEqH`

Data type for the untyped lambda calculus

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

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
```

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
```

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)
```

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
```

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
```

Occurs check

```
occurs : ∀ {α} → Name α → Tm α → Bool
occurs x₀ = occ (λ y → x₀ ≟Name 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

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