Une approche unifiante pour programmer sûrement avec de la syntaxe du premier ordre contenant des lieurs

Nicolas Pouillard

INRIA

Soutenance de thèse
13 Janvier 2012

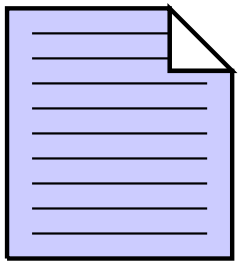
Jury composé de :

Président	M. Roberto Di Cosmo
Rapporteurs	M. Andrew Pitts
	M. Dale Miller
Examinateurs	M. Daniel Hirschkoff
	M. Conor McBride
Directeur	M. François Pottier

Plan

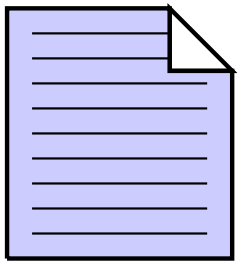
- Préliminaires : programmation avec des lieurs
- La bibliothèque NomPa : interface et utilisation
- Sûreté de l'approche : relations logiques et paramétricité

Qu'est-ce qu'un programme ?



Navigateurs Web, logiciels (traitement de texte, retouche photo, comptabilité, gestion, développement), systèmes d'exploitations, pilotes matériels, jeux, et tant d'autres.

Qu'est-ce qu'un programme ?

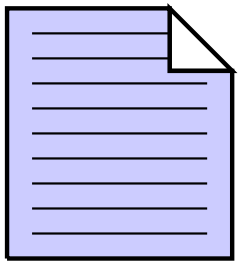


Navigateurs Web, logiciels (traitement de texte, retouche photo, comptabilité, gestion, développement), systèmes d'exploitations, pilotes matériels, jeux, et tant d'autres.

Au premier abord c'est un texte, comme :

```
print "Bonjour ! 2 fois 21 est égal à " >>
print (show (2 * 21))
```

Qu'est-ce qu'un programme ?



Navigateurs Web, logiciels (traitement de texte, retouche photo, comptabilité, gestion, développement), systèmes d'exploitations, pilotes matériels, jeux, et tant d'autres.

Au premier abord c'est un texte, comme :

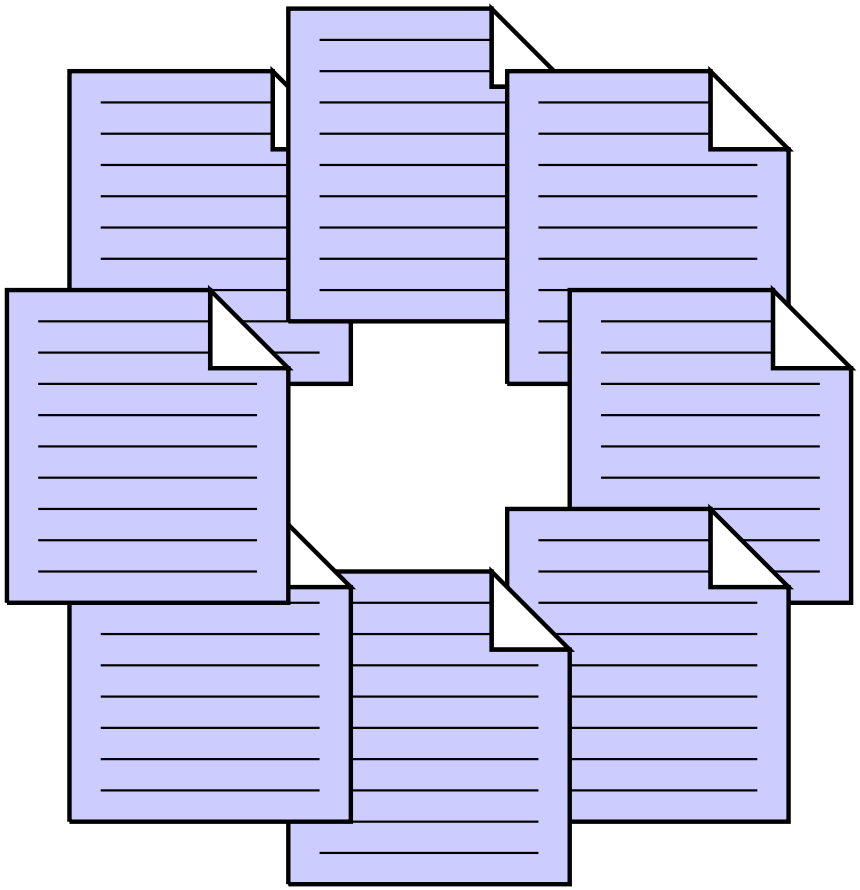
```
print "Bonjour ! 2 fois 21 est égal à " >>
print (show (2 * 21))
```

Traitement des données : une activité essentielle des programmes

- Les données simples : nombres, textes ...
- Les données complexes : musiques, images, vidéos, présentations ...
- Les données structurées : listes, tableaux, arbres, graphes ...

Qu'est-ce qu'un langage de programmation ?

Exemples de langages : Java, C, C++, Ruby, Python, OCaml, Haskell, Agda ...



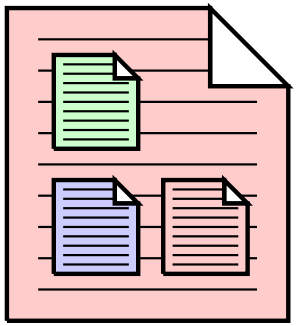
Un langage est défini par des règles :

- Pour sélectionner les programmes possibles
- Pour donner un sens aux programmes

Règles pour la sûreté :

- Portée des variables
- Typage fort et statique
- Spécifications formelles (preuves de correction)

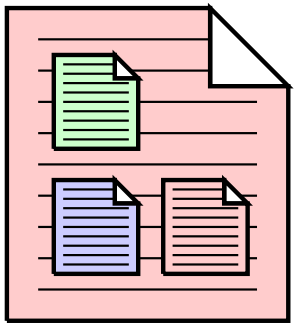
Quand les programmes sont des données ...



Définition

Méta-programme : un programme manipulant des programmes.

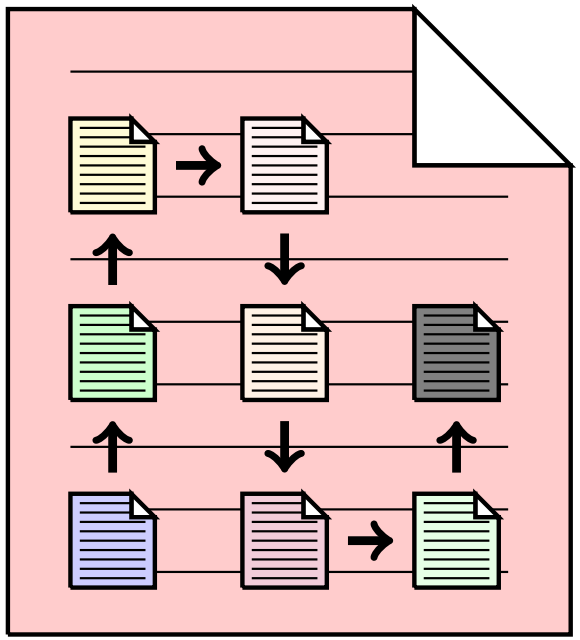
Quand les programmes sont des données ...



Définition

Méta-programme : un programme manipulant des programmes.

Par exemple un compilateur est un méta-programme.
Un compilateur traduit automatiquement des programmes d'un langage vers un autre en passant par des langages intermédiaires.



On appelle langage *objet* (resp. programme *objet*) les langages et programmes manipulés par un méta-programme.

λ-abstractions et portée des variables

Définition de fonctions : "λ-abstraction"

Définition

Dans la construction λ `x` $\rightarrow$ `e`, le *lieur* `x` *porte* dans l'expression `e` et représente l'argument de la fonction.

Exemple

```
f : ℕ → ℕ
f = λ x → 3 * x + 3
```

```
f 13
```

```
(λ x → λ y → x + y) y 21
```

λ-abstractions et portée des variables

Définition de fonctions : "λ-abstraction"

Définition

Dans la construction λ `x` $\rightarrow$ `e`, le *lieur* `x` *porte* dans l'expression `e` et représente l'argument de la fonction.

Exemple

```
f : ℕ → ℕ
f = λ x → 3 * x + 3
```

```
f 13
⇝ (λ x → 3 * x + 3) 13
```

```
(λ x → λ y → x + y) y 21
```

λ-abstractions et portée des variables

Définition de fonctions : "λ-abstraction"

Définition

Dans la construction $\lambda\ \mathtt{x} \to \mathtt{e}$, le *lieur* $\mathtt{x}$ *porte* dans l'expression $\mathtt{e}$ et représente l'argument de la fonction.

Exemple

$$\mathtt{f} : \mathbb{N} \to \mathbb{N}$$
$$\mathtt{f} = \lambda\ \mathtt{x} \to 3 * \mathtt{x} + 3$$

$$\mathtt{f}\ 13 \rightsquigarrow (\lambda\ \mathtt{x} \to 3 * \mathtt{x} + 3)\ 13 \rightsquigarrow 3 * 13 + 3$$

$$(\lambda\ \mathtt{x} \to \lambda\ \mathtt{y} \to \mathtt{x} + \mathtt{y})\ \mathtt{y}\ 21$$

λ-abstractions et portée des variables

Définition de fonctions : "λ-abstraction"

Définition

Dans la construction $\lambda\ \mathtt{x} \to \mathtt{e}$, le *lieur* $\mathtt{x}$ *porte* dans l'expression $\mathtt{e}$ et représente l'argument de la fonction.

Exemple

$$\mathtt{f} : \mathbb{N} \to \mathbb{N}$$
$$\mathtt{f} = \lambda\ \mathtt{x} \to 3 * \mathtt{x} + 3$$

$$\mathtt{f}\ 13 \rightsquigarrow (\lambda\ \mathtt{x} \to 3 * \mathtt{x} + 3)\ 13 \rightsquigarrow 3 * 13 + 3 \rightsquigarrow 39 + 3$$

$$(\lambda\ \mathtt{x} \to \lambda\ \mathtt{y} \to \mathtt{x} + \mathtt{y})\ \mathtt{y}\ 21$$

λ-abstractions et portée des variables

Définition de fonctions : "λ-abstraction"

Définition

Dans la construction $\lambda\ \mathtt{x} \to \mathtt{e}$, le *lieur* $\mathtt{x}$ *porte* dans l'expression $\mathtt{e}$ et représente l'argument de la fonction.

Exemple

$$\mathtt{f} : \mathbb{N} \to \mathbb{N}$$
$$\mathtt{f} = \lambda\ \mathtt{x} \to 3 * \mathtt{x} + 3$$

$$\mathtt{f}\ 13 \rightsquigarrow (\lambda\ \mathtt{x} \to 3 * \mathtt{x} + 3)\ 13 \rightsquigarrow 3 * 13 + 3 \rightsquigarrow 39 + 3 \rightsquigarrow 42$$

$$(\lambda\ \mathtt{x} \to \lambda\ \mathtt{y} \to \mathtt{x} + \mathtt{y})\ \mathtt{y}\ 21$$

λ-abstractions et portée des variables

Définition de fonctions : "λ-abstraction"

Définition

Dans la construction $\lambda\ \mathtt{x} \to \mathtt{e}$, le *lieur* $\mathtt{x}$ *porte* dans l'expression $\mathtt{e}$ et représente l'argument de la fonction.

Exemple

$$\mathtt{f} : \mathbb{N} \to \mathbb{N}$$
$$\mathtt{f} = \lambda\ \mathtt{x} \to 3 * \mathtt{x} + 3$$

$$\begin{aligned}
&\mathtt{f}\ 13 \\
&\rightsquigarrow (\lambda\ \mathtt{x} \to 3 * \mathtt{x} + 3)\ 13 \\
&\rightsquigarrow 3 * 13 + 3 \\
&\rightsquigarrow 39 + 3 \\
&\rightsquigarrow 42
\end{aligned}$$

$$\begin{aligned}
&(\lambda\ \mathtt{x} \to \lambda\ \mathtt{y} \to \mathtt{x} + \mathtt{y})\ \mathtt{y}\ 21 \\
&\rightsquigarrow (\lambda\ \mathtt{y} \to \mathtt{y} + \mathtt{y})\ 21
\end{aligned}$$

λ-abstractions et portée des variables

Définition de fonctions : "λ-abstraction"

Définition

Dans la construction λ `x` $\to$ `e`, le *lieur* `x` *porte* dans l'expression `e` et représente l'argument de la fonction.

Exemple

```
f : ℕ → ℕ
f = λ x → 3 * x + 3
```

```
f 13
⇝ (λ x → 3 * x + 3) 13
⇝ 3 * 13 + 3
⇝ 39 + 3
⇝ 42
```

```
(λ x → λ y → x + y) y 21
⇝ (λ y → y + y) 21
⇝ 21 + 21
```

λ-abstractions et portée des variables

Définition de fonctions : "λ-abstraction"

Définition

Dans la construction λ `x` $\rightarrow$ `e`, le *lieur* `x` *porte* dans l'expression `e` et représente l'argument de la fonction.

Exemple

```
f : ℕ → ℕ
f = λ x → 3 * x + 3
```

```
f 13
⇝ (λ x → 3 * x + 3) 13
⇝ 3 * 13 + 3
⇝ 39 + 3
⇝ 42
```

```
(λ x → λ y → x + y) y 21
⇝ (λ y → y + y) 21
⇝ 21 + 21
⇝ 42
```

λ-abstractions et portée des variables

Définition de fonctions : "λ-abstraction"

Définition

Dans la construction λ `x` $\rightarrow$ `e`, le *lieur* `x` *porte* dans l'expression `e` et représente l'argument de la fonction.

Exemple

```
f : ℕ → ℕ
f = λ x → 3 * x + 3
```

```
f 13
⇝ (λ x → 3 * x + 3) 13
⇝ 3 * 13 + 3
⇝ 39 + 3
⇝ 42
```

```
(λ x → λ y → x + y) y 21
⇝ (λ y → y + y) 21
⇝ 21 + 21
⇝ 42
FAUX
```

λ-abstractions et portée des variables

Définition de fonctions : "λ-abstraction"

Définition

Dans la construction λ `x` $\to$ `e`, le *lieur* `x` *porte* dans l'expression `e` et représente l'argument de la fonction.

Exemple

```
f : ℕ → ℕ
f = λ x → 3 * x + 3
```

```
f 13
⇝ (λ x → 3 * x + 3) 13
⇝ 3 * 13 + 3
⇝ 39 + 3
⇝ 42
```

```
(λ x → λ y → x + y) y 21
⇝ (λ y → y + y) 21
⇝ 21 + 21
⇝ 42
FAUX
```

```
(λ x → λ y → x + y) y 21
```

λ-abstractions et portée des variables

Définition de fonctions : "λ-abstraction"

Définition

Dans la construction λ `x` $\to$ `e`, le *lieur* `x` *porte* dans l'expression `e` et représente l'argument de la fonction.

Exemple

```
f : ℕ → ℕ
f = λ x → 3 * x + 3
```

```
f 13
⇝ (λ x → 3 * x + 3) 13
⇝ 3 * 13 + 3
⇝ 39 + 3
⇝ 42
```

```
(λ x → λ y → x + y) y 21
⇝ (λ y → y + y) 21
⇝ 21 + 21
⇝ 42
FAUX
```

```
(λ x → λ y → x + y) y 21
⇝ (λ x → λ z → x + z) y 21
```

λ-abstractions et portée des variables

Définition de fonctions : "λ-abstraction"

Définition

Dans la construction λ x $\to$ e, le *lieur* x *porte* dans l'expression e et représente l'argument de la fonction.

Exemple

```
f : ℕ → ℕ
f = λ x → 3 * x + 3
```

```
f 13
⇝ (λ x → 3 * x + 3) 13
⇝ 3 * 13 + 3
⇝ 39 + 3
⇝ 42
```

```
(λ x → λ y → x + y) y 21
⇝ (λ y → y + y) 21
⇝ 21 + 21
⇝ 42
FAUX
```

```
(λ x → λ y → x + y) y 21
⇝ (λ x → λ z → x + z) y 21
⇝ (λ z → y + z) 21
```

λ-abstractions et portée des variables

Définition de fonctions : "λ-abstraction"

Définition

Dans la construction λ `x` $\to$ `e`, le *lieur* `x` *porte* dans l'expression `e` et représente l'argument de la fonction.

Exemple

```
f : ℕ → ℕ
f = λ x → 3 * x + 3
```

```
f 13
⇝ (λ x → 3 * x + 3) 13
⇝ 3 * 13 + 3
⇝ 39 + 3
⇝ 42
```

```
(λ x → λ y → x + y) y 21
⇝ (λ y → y + y) 21
⇝ 21 + 21
⇝ 42
FAUX
```

```
(λ x → λ y → x + y) y 21
⇝ (λ x → λ z → x + z) y 21
⇝ (λ z → y + z) 21
⇝ y + 21
```

Types de données et style nominal

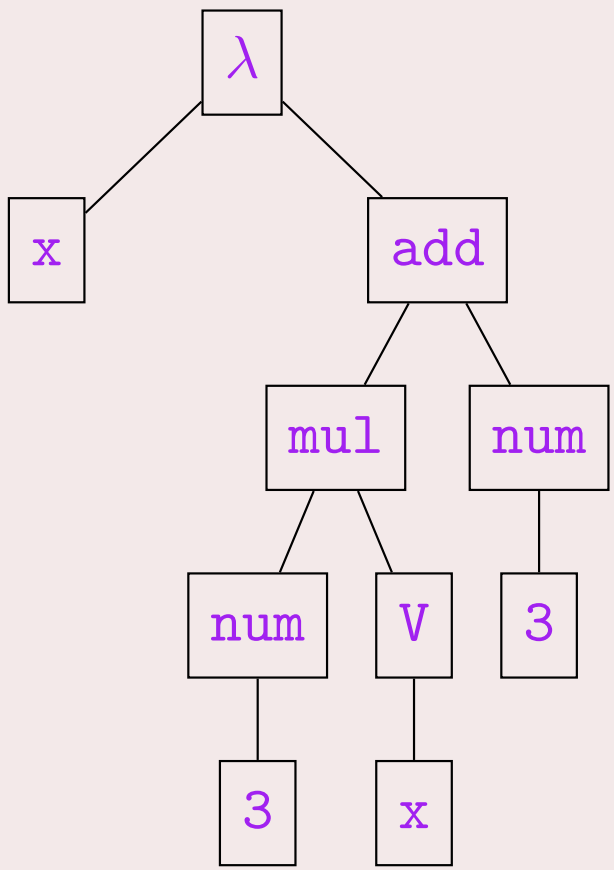
La méta-programmation est facilitée par l'introduction de types de données pour représenter les langages de programmation.

```
λ x → 3 * x + 3
```

Types de données et style nominal

La méta-programmation est facilitée par l'introduction de types de données pour représenter les langages de programmation.

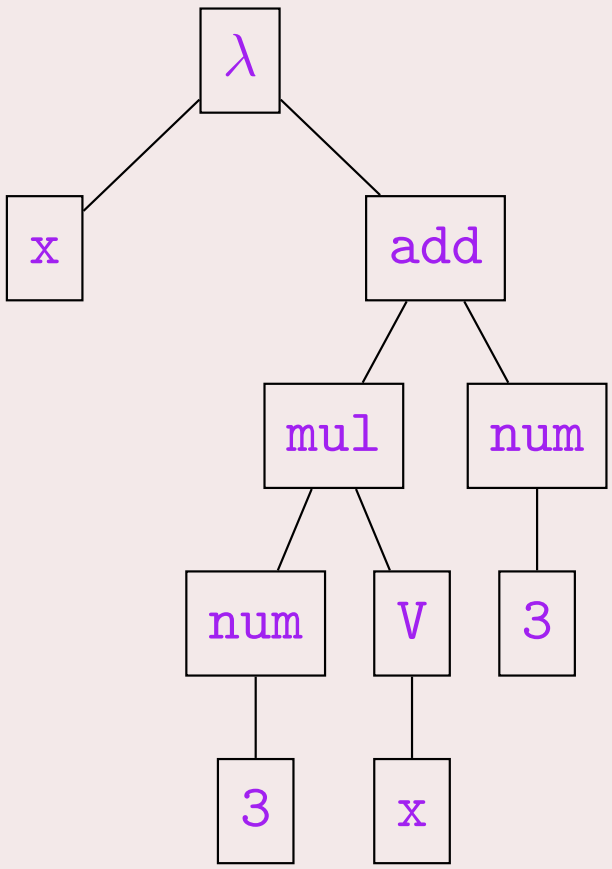
```
λ x → 3 * x + 3
```



Types de données et style nominal

La méta-programmation est facilitée par l'introduction de types de données pour représenter les langages de programmation.

```
λ x → 3 * x + 3
```




```
Name : Set
xᴺ yᴺ ... : Name

data Tm : Set where
  num : ℕ → Tm
  add : Tm → Tm → Tm
  mul : Tm → Tm → Tm
  V   : Name → Tm
  ƛ   : Name → Tm → Tm
  _·_ : Tm → Tm → Tm

ex₁ : Tm
ex₁ = ƛ xᴺ (add (mul (num 3) (V xᴺ))
                (num 3))
```

Termes clos et termes bien formés

Un terme clos :

λ f $\rightarrow$ λ x $\rightarrow$ f x

Un terme ouvert (non clos) :

λ x $\rightarrow$ f x

Définition

Un terme est bien formé lorsque toutes les variables sont soit liées par un lieur du terme soit liées dans *l'environnement*.

Mal formé :

$\epsilon \vdash \lambda$ x $\rightarrow$ f x

Bien formé dans l'environnement contenant f :

f $\vdash \lambda$ x $\rightarrow$ f x

Définition

Un terme est clos si et seulement si il est bien formé dans l'environnement vide.

Objectif 1 : Garantir que l'on ne manipule que des termes bien formés

α-équivalence & α-pureté

```
-- λ x → x
id^x : Tm
id^x = ƛ x^N (V x^N)
```

```
-- λ y → y
id^y : Tm
id^y = ƛ y^N (V y^N)
```

α-équivalence & α-pureté

```
-- λ x → x
idˣ : Tm
idˣ = ƛ xᴺ (V xᴺ)
```

```
-- λ y → y
idʸ : Tm
idʸ = ƛ yᴺ (V yᴺ)
```

α-pureté des fonctions :

```
∀ (f : Tm → Bool) →
  f idˣ ≡ f idʸ
```

Définition

Une fonction est α-pure si et seulement si elle renvoie des résultats α-équivalents pour des arguments α-équivalents.

α-équivalence & α-pureté

```
-- λ x → x
idˣ : Tm
idˣ = ƛ xᴺ (V xᴺ)
```

```
-- λ y → y
idʸ : Tm
idʸ = ƛ yᴺ (V yᴺ)
```

α-pureté des fonctions :

```
∀ (f : Tm → Bool) →
  f idˣ ≡ f idʸ
```

Définition

Une fonction est α-pure si et seulement si elle renvoie des résultats α-équivalents pour des arguments α-équivalents.

Que penser de cette fonction ?

```
compare-bound-atoms : Tm → Bool
compare-bound-atoms (ƛ z _) = z ==ᴺ xᴺ
compare-bound-atoms _       = false
```

Objectif 2 : Tout calcul devrait préserver l'α-équivalence

NomPa : interface et exemples

Termes en style nominal avec NomPa

```
data Tm : Set where
  num : ℕ → Tm
  add : Tm → Tm → Tm
  mul : Tm → Tm → Tm
  _·_ : Tm → Tm → Tm
  V   : Name → Tm
  ƛ   : Name → Tm → Tm
```

```
record NomPa : Set₁ where
  field
    Name : Set
```

Termes en style nominal avec NomPa

```
-- 1) Nettoyage ...
```

```
data Tm : Set where
  num : ℕ → Tm
  add : Tm → Tm → Tm
  mul : Tm → Tm → Tm
  _·_ : Tm → Tm → Tm
  V   : Name → Tm
  ƛ   : Name → Tm → Tm
```

```
record NomPa : Set₁ where
  field
    Name : Set
```

Termes en style nominal avec NomPa

```
-- 1) Nettoyage ...
```

```
data Tm : Set where

  _·_ : Tm → Tm → Tm
  V   : Name → Tm
  ƛ   : Name → Tm → Tm
```

```
record NomPa : Set₁ where
  field
    Name : Set
```

Termes en style nominal avec NomPa

```
-- 1) Nettoyage ...
```

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

```
record NomPa : Set₁ where
  field
    Name : Set
```

Termes en style nominal avec NomPa

```
-- 1) Nettoyage ...
```

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

```
record NomPa : Set₁ where
  field

    Name :           Set
```

Termes en style nominal avec NomPa

```
-- 2) Distinguons les noms et les lieurs ...
```

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

```
record NomPa : Set₁ where
  field

    Name :           Set
```

Termes en style nominal avec NomPa

```
-- 2) Distinguons les noms et les lieurs ...
```

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

```
record NomPa : Set₁ where
  field

    Name :          Set
    Binder : Set
```

Termes en style nominal avec NomPa

```
-- 3) Indiçons les noms et les termes ...
```

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

```
record NomPa : Set₁ where
  field

    Name :          Set
    Binder : Set
```

Termes en style nominal avec NomPa

```
-- 3) Indiçons les noms et les termes ...
```

```
data Tm (α :  ?  ) : Set where
  _·_ : Tm α → Tm α → Tm α
  V   : Name α → Tm α
  ƛ   :      Binder  → Tm (  ?  ) → Tm α
```

```
record NomPa : Set₁ where
  field

    Name :  ?   → Set
    Binder : Set
```

Termes en style nominal avec NomPa

```
-- 4) Par des mondes abstraits ...
```

```
data Tm (α :  ?  ) : Set where
  _·_ : Tm α → Tm α → Tm α
  V   : Name α → Tm α
  ƛ   :      Binder  → Tm (  ?  ) → Tm α
```

```
record NomPa : Set₁ where
  field

    Name :  ?   → Set
    Binder : Set
```

Termes en style nominal avec NomPa

```
-- 4) Par des mondes abstraits ...
```

```
data Tm (α : World) : Set where
  _·_ : Tm α → Tm α → Tm α
  V   : Name α → Tm α
  ƛ   :      Binder  → Tm (  ?  ) → Tm α
```

```
record NomPa : Set₁ where
  field
    World : Set
    Name : World → Set
    Binder : Set
```

Termes en style nominal avec NomPa

```
-- Intuition : un monde est vu comme une liste de lieurs
```

```
data Tm (α : World) : Set where
  _·_ : Tm α → Tm α → Tm α
  V   : Name α → Tm α
  ƛ   :      Binder  → Tm (  ?  ) → Tm α
```

```
record NomPa : Set₁ where
  field
    World : Set
    Name : World → Set
    Binder : Set
```

Termes en style nominal avec NomPa

```
-- 5) Nommons le lieur ...
```

```
data Tm (α : World) : Set where
  _·_ : Tm α → Tm α → Tm α
  V   : Name α → Tm α
  ƛ   :      Binder  → Tm (  ?  ) → Tm α
```

```
record NomPa : Set₁ where
  field
    World : Set
    Name : World → Set
    Binder : Set
```

Termes en style nominal avec NomPa

```
-- 5) Nommons le lieur ...
```

```
data Tm (α : World) : Set where
  _·_ : Tm α → Tm α → Tm α
  V   : Name α → Tm α
  ƛ   : (b : Binder) → Tm (  ?  ) → Tm α
```

```
record NomPa : Set₁ where
  field
    World : Set
    Name : World → Set
    Binder : Set
```

Termes en style nominal avec NomPa

```
-- 6) Portée du lieur ‘b’ ...
```

```
data Tm (α : World) : Set where
  _·_ : Tm α → Tm α → Tm α
  V   : Name α → Tm α
  ƛ   : (b : Binder) → Tm (  ?  ) → Tm α
```

```
record NomPa : Set₁ where
  field
    World : Set
    Name : World → Set
    Binder : Set
```

Termes en style nominal avec NomPa

```
-- 6) Portée du lieur 'b' ...
```

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

```
record NomPa : Set₁ where
  field
    World : Set
    Name : World → Set
    Binder : Set
    _◁_ : Binder → World → World
```

Termes en style nominal avec NomPa

```
-- Remarque : rien n'empaquette le lieur avec le sous-terme
```

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

```
record NomPa : Set₁ where
  field
    World : Set
    Name : World → Set
    Binder : Set
    _◁_ : Binder → World → World
```

L'interface NomPa (1ère partie)

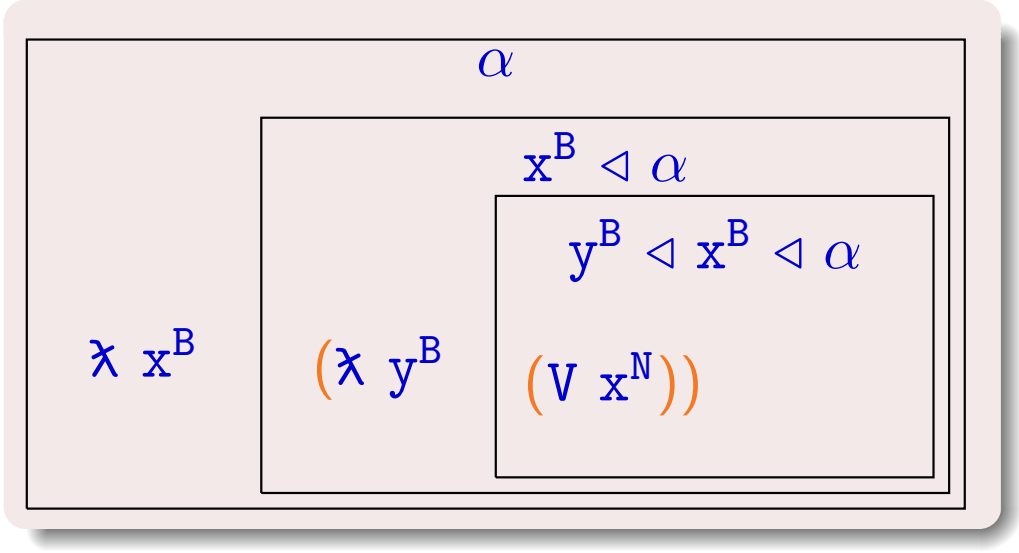
```
record NomPa : Set₁ where
  field
    World : Set
    Name : World → Set
    Binder : Set
    _◁_ : Binder → World → World

    _==ᴺ_     : ∀ {α} (x y : Name α) → Bool
    exportᴺ? : ∀ {b α} → Name (b ◁ α) → Maybe (Name α)

    ...
```

L'interface NomPa (1ère partie)

```
export^N? : ∀ {b α} → Name (b ◁ α) → Maybe (Name α)
```



Exemple : Collecter les variables libres

```
rm : Name → List Name → List Name
rm b [] = []
rm b (x :: xs)        with x ==ᴺ b
... {- bound: x≡b -} | true       = rm b xs
... {- free:  x≢b -} | false      = x  :: rm b xs
```

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

Exemple : Collecter les variables libres

```
rm : ∀ {α} b → List (Name (b ◁ α)) → List (Name α)
rm b [] = []
rm b (x :: xs)        with x ==ᴺ b
... {- bound: x≡b -} | true       = rm b xs
... {- free:  x≢b -} | false      = x  :: rm b xs
```

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

Exemple : Collecter les variables libres

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

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

Exemple : Collecter les variables libres

```
rm : ∀ {α} b → List (Name (b ◁ α)) → List (Name α)
rm b [] = []
rm b (x :: xs)       with exportᴺ? {b} x
... {- 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)
```

- On ne peut pas oublier d'enlever `b`.
- Pas de coûts cachés.
- Par paramétricité on obtiendra que les noms retournés viennent du terme d'entrée.

L'interface NomPa (2ème partie)

```
record NomPa : Set₁ where
  field
    ...
    -- Le monde vide
    ∅      : World
    -- Un ensemble infini de lieurs
    zeroᴮ : Binder
    sucᴮ  : Binder → Binder
    -- Un ensemble infini de noms
    nameᴮ : ∀ {α} b → Name (b ◁ α)
    ...
```

```
-- λ x → x
idᵀᵐ : ∀ {α} → Tm α
idᵀᵐ = ƛ x (V (nameᴮ x))
  where x = zeroᴮ
```

Traversée générique et kits de traversée

```
-- Voici la traversée pure (sans effets) :
module Traverse^Tm {Env} (trKit : TrKit Env Tm) where
  open TrKit trKit
  tr^Tm : ∀ {α β} → Env α β → Tm α → Tm β
  tr^Tm Δ (V x)   = trName Δ x
  tr^Tm Δ (t · u) = tr^Tm Δ t · tr^Tm Δ u
  tr^Tm Δ (ƛ b t) = ƛ _ (tr^Tm (extEnv b Δ) t)
```

```
record TrKit (Env : (α β : World) → Set)
             (Res : World → Set) : Set where
  field
    trName   : ∀ {α β} → Env α β → Name α → Res β
    trBinder : ∀ {α β} → Env α β → Binder → Binder
    extEnv   : ∀ {α β} b (Δ : Env α β)
                       → Env (b ◁ α) (trBinder Δ b ◁ β)
```

Traversée générique et kits de traversée

```
-- Voici la traversée pure (sans effets) :
module Traverse^Tm {Env} (trKit : TrKit Env Tm) where
  open TrKit trKit
  tr^Tm : ∀ {α β} → Env α β → Tm α → Tm β
  tr^Tm Δ (V x)   = trName Δ x
  tr^Tm Δ (t · u) = tr^Tm Δ t · tr^Tm Δ u
  tr^Tm Δ (ƛ b t) = ƛ _ (tr^Tm (extEnv b Δ) t)
```

```
-- Voici le squelette du kit de renommage (sans effets) :
RenameEnv : (α β : World) → Set
RenameEnv α β = (Name α → Name β) × ...

renameKit : TrKit RenameEnv Name
renameKit = ...
```

À partir de la traversée générique

Une fonction de traversée permet de relever les fonctions sur les noms et avec effets (`Name` $\alpha \to$ `E` (`Name` β)) vers des fonctions sur les termes (`Tm` $\alpha \to$ `E` (`Tm` β)).

```
exportTm? : ∀ {b α} → Supply α → Tm (b ◁ α) →? Tm α
```

À partir de la traversée générique

Une fonction de traversée permet de relever les fonctions sur les noms et avec effets (`Name` $\alpha \to$ `E` (`Name` β)) vers des fonctions sur les termes (`Tm` $\alpha \to$ `E` (`Tm` β)).

```
exportTm? : ∀ {b α} → Supply α → Tm (b ◁ α) →? Tm α
```

Dans un second temp, on peut faire de même avec les fonctions des noms dans les termes (`Name` $\alpha \to$ `E` (`Tm` β)). On obtient beaucoup de fonctions, dont la substitution sans captures, à partir de cette fonction de traversée.

```
substTm    : ∀ {α β} → Supply β → (Name α → Tm β)
                                →   Tm α   → Tm β
```

NomPa : interface et utilisation

- L'interface :
  - Les noms et les termes sont indicés par une notion de monde.
  - Les mondes, vides au départ, sont étendus par des lieurs.
  - Les noms sont comparables et exportables sous certaines conditions.
  - En plus : inclusions de mondes, opérations addition/soustraction/comparaison sur les noms.
- Opérations sur les termes :
  - Les fonctions standard comme `fv` et `rm` sont peu altérées.
  - En plus : comparaison de termes, Normalisation Par Évaluation ...
- Traversées et kits :
  - Traversées génériques : la plupart des fonctions, termes vers termes, préservant la structure, en une seule fonction.
  - En plus : traversées avec effet à l'aide de foncteurs applicatifs, plus de kits et traversées.

On veut l'α-pureté et donc que le calcul préserve une relation ...

Relations logiques et paramétricité !

Rappels sur les relations logiques

```
τ : Set -- τ un type
⟦ τ ⟧ : τ → τ → Set -- ⟦ τ ⟧ une relation
```

```
(A_r ⟦→⟧ B_r) f_1 f_2 =
              ∀ {x_1 x_2} → A_r x_1 x_2
                          → B_r (f_1 x_1) (f_2 x_2)
```

```
(⟦Π⟧ A_r B_r) f_1 f_2 = ∀ {x_1 x_2} (x_r : A_r x_1 x_2)
                        → B_r x_r (f_1 x_1) (f_2 x_2)
```

```
⟦Set⟧ : Set → Set → Set_1
⟦Set⟧ A_1 A_2 = A_1 → A_2 → Set
```

⟦Bool⟧ et ⟦ℕ⟧ sont des relations identité

La paramétricité en deux mots

On veut que la relation logique corresponde à l'α-équivalence.

Cependant ici le langage est fixé (AGDA). La paramétricité est la solution.

```
e   :   τ        -- pour chaque programme bien typé
```

La paramétricité en deux mots

On veut que la relation logique corresponde à l'α-équivalence.

Cependant ici le langage est fixé (Agda). La paramétricité est la solution.

```
        e   :   τ        -- pour chaque programme bien typé
            ⇓
    ⟦ e ⟧ : ⟦ τ ⟧ e e -- un théorème gratuit
```

La paramétricité en deux mots

On veut que la relation logique corresponde à l'α-équivalence.

Cependant ici le langage est fixé (AGDA). La paramétricité est la solution.

```
  Γ   ⊢   e   :   τ        -- pour chaque programme bien typé
                ⇓
⟦ Γ ⟧ ⊢ ⟦ e ⟧ : ⟦ τ ⟧ e e -- un théorème gratuit
```

Objectifs de sûreté

Finalement on obtient l'α-pureté car $[\![\_]\!]$ est l'α-équivalence.

```
-- En particulier pour le type Tm.
α-equivalence ⇔ ⟦Tm⟧ ⟦∅⟧
```

On remarque que notre définition munit tous les types d'une α-équivalence.

```
-- L'α-pureté implique que les termes α-équivalents
-- sont indistinguables.
f : ∀ {α} → Tm α → Bool
f-lemma : ∀ t₁ t₂ → α-equivalence t₁ t₂ → f t₁ ≡ f t₂
```

Les fonctions de type $\forall\ \{\alpha\} \to \mathtt{Tm}\ \alpha \to \mathtt{Tm}\ \alpha$ sont insensibles aux renommages des noms libres de leur entrée. L'identité des noms libres n'a pas d'importance.

Théorèmes gratuits pour les clients de bibliothèques

```
c : (lib : NomPa) → ...
```

```
⟦c⟧ : ⟦ (lib : NomPa) → ... ⟧ c c
```

Théorèmes gratuits pour les clients de bibliothèques

```
c : (World : Set)
    (Name  : World → Set)
    (==ᴺ   : ...) ... → ...

⟦c⟧ : ⟦ (World : Set)
        (Name  : World → Set)
        (==ᴺ   : ...) ... → ... ⟧ c c
```

Théorèmes gratuits pour les clients de bibliothèques

```
c : (World : Set)
    (Name  : World → Set)
    (==ᴺ   : ...) ... → ...

⟦c⟧ : ∀{World₁ World₂}(⟦World⟧ : ⟦Set⟧ World₁ World₂ )
      {Name₁ Name₂}  (⟦Name⟧  : ⟦ World → Set ⟧ Name₁ Name₂ )
      {==ᴺ₁ ==ᴺ₂}      (⟦==ᴺ⟧   : ...) ...
      → ⟦ ... ⟧ (c World₁ ...) (c World₂ ...)
```

Théorèmes gratuits pour les clients de bibliothèques

```
c : (World : Set)
    (Name  : World → Set)
    (==ᴺ   : ...) ... → ...

⟦c⟧ : ∀{World      }(⟦World⟧ : ⟦Set⟧ World  World  )
      {Name       } (⟦Name⟧  : ⟦ World → Set ⟧ Name  Name  )
      {==ᴺ        }      (⟦==ᴺ⟧   : ...) ...
      → ⟦ ... ⟧ (c World ...) (c World ...)
```

Théorèmes gratuits pour les clients de bibliothèques

```
c : (World : Set)
    (Name  : World → Set)
    (==ᴺ   : ...) ... → ...

⟦c⟧ : ∀{World} (⟦World⟧ : ⟦Set⟧ World World)
      {Name}  (⟦Name⟧  : ⟦ World → Set ⟧ Name Name)
      {==ᴺ}   (⟦==ᴺ⟧   : ...) ...
      → ⟦ ... ⟧ (c World ...) (c World ...)
```

Théorèmes gratuits pour les clients de bibliothèques

```
c : (World : Set)
    (Name  : World → Set)
    (==ᴺ   : ...) ... → ...

⟦c⟧ : ∀{World} (⟦World⟧ : ⟦Set⟧ World World)
      {Name}  (⟦Name⟧  : ⟦ World → Set ⟧ Name Name)
      {==ᴺ}   (⟦==ᴺ⟧   : ...) ...
      → ⟦ ... ⟧ (c World ...) (c World ...)
```

On cherche des définitions pour ⟦World⟧, ⟦Name⟧, ... qui maximisent l'utilité du théorème restant.

La sûreté de NomPa, de façon modulaire

```
⟦Binder⟧ : ⟦Set⟧ Binder Binder
⟦Binder⟧ _ _ = ⊤
```

La sûreté de NomPa, de façon modulaire

```
⟦Binder⟧ : Binder → Binder → Set
⟦Binder⟧ _ _ = ⊤
```

La sûreté de NomPa, de façon modulaire

```
--      ⟦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
```

La sûreté de NomPa, de façon modulaire

```
--      ⟦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⟧ ⟦→⟧ ⟦Set⟧) Name Name
⟦Name⟧ (ℛ , _) x₁ x₂ = ℛ x₁ x₂
```

La sûreté de NomPa, de façon modulaire

```
--      ⟦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
```

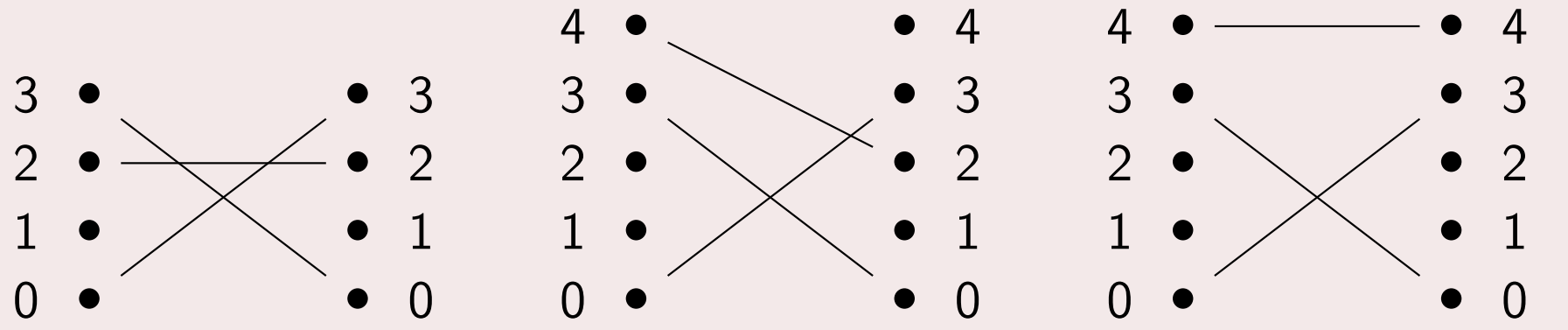
```
⟦Name⟧ : ∀ {α₁ α₂} → ⟦World⟧ α₁ α₂ → Name α₁ → Name α₂ → Set
⟦Name⟧ (ℛ , _) x₁ x₂ = ℛ x₁ x₂
```

La sûreté de NomPa, de façon modulaire

```
--      ⟦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
```

```
⟦∅⟧ : ⟦World⟧ ∅ ∅
⟦∅⟧ = (λ _ _ → ⊥) , {! proof omitted !}
```

La sûreté de NomPa, de façon modulaire




```
_⟦◁⟧_ : (⟦Binder⟧ ⟦→⟧ ⟦World⟧ ⟦→⟧ ⟦World⟧) _◁_ _◁_
_⟦◁⟧_ {b₁} {b₂} _ {α₁} {α₂} (αᵣ , _) = _ℛ_ , {!proof omitted!}
 where
  data _ℛ_ x y : Set where
   here  : binderᴺ x ≡ b₁ → binderᴺ y ≡ b₂ → x ℛ y
   there : binderᴺ x ≢ b₁ → binderᴺ y ≢ b₂ → αᵣ x y → x ℛ y
```

La sûreté de NomPa, de façon modulaire

```
--      ⟦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
```

```
_⟦==ᴺ⟧_ : (∀⟨ αᵣ : ⟦World⟧ ⟩⟦→⟧
             ⟦Name⟧ αᵣ ⟦→⟧
             ⟦Name⟧ αᵣ ⟦→⟧
             ⟦Bool⟧) _==ᴺ_ _==ᴺ_
_⟦==ᴺ⟧_ αᵣ xᵣ yᵣ = {! proof omitted !}
```

La sûreté de NomPa, de façon modulaire

```
--      ⟦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
```

```
_⟦==ᴺ⟧_ : ∀ {α₁ α₂} (αᵣ : ⟦World⟧ α₁ α₂)
            {x₁ x₂} (xᵣ : ⟦Name⟧ αᵣ x₁ x₂)
            {y₁ y₂} (yᵣ : ⟦Name⟧ αᵣ y₁ y₂)
          → ⟦Bool⟧ (x₁ ==ᴺ y₁) (x₂ ==ᴺ y₂)
_⟦==ᴺ⟧_ αᵣ xᵣ yᵣ = {! proof omitted !}
```

La sûreté de NomPa, de façon modulaire

```
--      ⟦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
```

Finalement, la relation $[\![\_]\!]$ correspond bien à l'α-équivalence.

NomPa : une bibliothèque multi-style pour les noms et les lieurs

- L'interface NomPa a d'autres fonctions.
- Et d'autres types comme les témoins d'inclusion de monde.
- Pas seulement des liaisons en style nominal.
- Liaisons en style de Bruijn (indices et niveaux) et calcul sur les noms.
- Combinaisons de ces différents styles.
- De nombreux exemples et opérations.
- Codage de diverses techniques de liaisons.

Conclusion

- Tout calcul préserve l'α-équivalence.
- Donc on ne manipule que des termes bien formés.
- Les noms et les termes sont indicés par des mondes.
- Sûreté via des types de base *abstraits*.
- Les noms sont séparés des lieurs.
- Grain plus fin que FreshML et HOAS (pas de coûts cachés).
- Tout en Agda : code, formalisation et preuves.
- Théorèmes disponibles en ligne : `http://nicolaspouillard.fr/`

Perspectives

- Améliorer la méta-programmation en Agda pour :
  - Inférer les témoins d'inclusion.
  - Ajouter l'opération $[\![\_]\!]$.
- Comment NomPa pourrait être utilisé en méta-théorie ?
- NomPa en tant que langage cible explicite pour des langages de plus haut niveau (pure FreshML ou l'approche avec des types non-réguliers).
- Étudier les interactions avec les références.
- Regarder d'autres utilisations de la paramétricité comme preuve de sûreté.