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

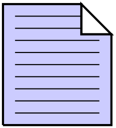
Jury composé de :

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

Outline

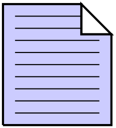
- First steps: programming with binders
- The NomPa library: interface and usage
- Safety of the approach: logical relations and parametricity

What is a program?



Web browsers, software (word processing, image processing, accounting, management, development), operating systems, drivers, games, and so forth...

What is a program?

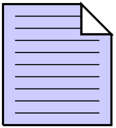


Web browsers, software (word processing, image processing, accounting, management, development), operating systems, drivers, games, and so forth...

At a first sight it is a text, such as:

```
print "Hello! 2 times 21 is equal to " >>
print (show (2 * 21))
```

What is a program?



Web browsers, software (word processing, image processing, accounting, management, development), operating systems, drivers, games, and so forth...

At a first sight it is a text, such as:

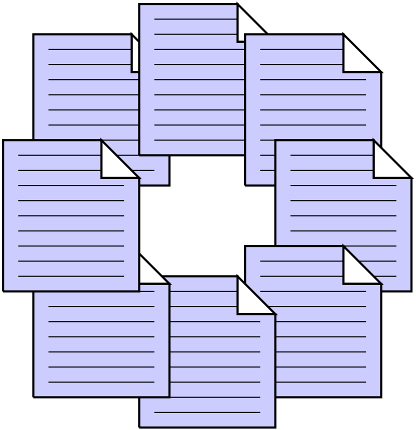
```
print "Hello! 2 times 21 is equal to " >>
print (show (2 * 21))
```

Data processing: an essential activity of programs

- Simple data: numbers, texts...
- Complex data: music, images, videos, presentations..
- Structured data: lists, arrays, trees, graphs...

What is a programming language?

Examples of languages: Java, C, C++, Ruby, Python, OCaml, Haskell, Agda...



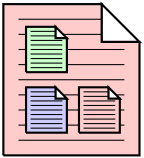
A language is defined by rules:

- To select possible programs
- To give them a meaning

Rules for safety:

- Scopes of variables
- Strong and static typing
- Formal specifications (correctness proofs)

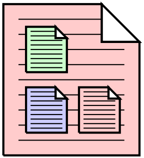
Programs as data...



Definition

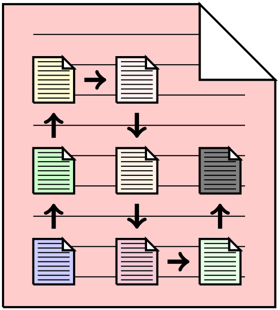
Meta-program: a program processing programs.

Programs as data...



Definition

Meta-program: a program processing programs.



For instance a compiler is a meta-program.
A compiler automatically translate programs from one language to another passing through intermediate languages.

We can *object* language (resp. *object* program) languages and programs that a meta-program process.

λ-abstractions and variables scope

Function definition: "λ-abstraction"

Definition

In the construct λ `x` $\rightarrow$ `e`, the *binder* `x` *scopes* over the expression `e` and represent the function argument.

```
f : ℕ → ℕ
f = λ x → 3 * x + 3
```

```
(λ x → λ y → x + y) y 21
```

```
f 13
```

λ-abstractions and variables scope

Function definition: "λ-abstraction"

Definition

In the construct `λ x → e`, the *binder* `x` *scopes* over the expression `e` and represent the function argument.

```
f : ℕ → ℕ
f = λ x → 3 * x + 3
```

```
(λ x → λ y → x + y) y 21
```

```
f 13
⇝ (λ x → 3 * x + 3) 13
```

λ-abstractions and variables scope

Function definition: "λ-abstraction"

Definition

In the construct `λ x → e`, the *binder* `x` *scopes* over the expression `e` and represent the function argument.

```
f : ℕ → ℕ
f = λ x → 3 * x + 3
```

```
(λ x → λ y → x + y) y 21
```

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

λ-abstractions and variables scope

Function definition: "λ-abstraction"

Definition

In the construct `λ x → e`, the *binder* `x` *scopes* over the expression `e` and represent the function argument.

```
f : ℕ → ℕ
f = λ x → 3 * x + 3
```

```
(λ x → λ y → x + y) y 21
```

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

λ-abstractions and variables scope

Function definition: "λ-abstraction"

Definition

In the construct λ `x` $\to$ `e`, the *binder* `x` *scopes* over the expression `e` and represent the function argument.

```
f : ℕ → ℕ
f = λ x → 3 * x + 3
```

```
(λ x → λ y → x + y) y 21
```

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

λ-abstractions and variables scope

Function definition: "λ-abstraction"

Definition

In the construct `λ x → e`, the *binder* `x` *scopes* over the expression `e` and represent the function argument.

```
f : ℕ → ℕ
f = λ x → 3 * x + 3
```

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

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

λ-abstractions and variables scope

Function definition: "λ-abstraction"

Definition

In the construct `λ x → e`, the *binder* `x` *scopes* over the expression `e` and represent the function argument.

```
f : ℕ → ℕ
f = λ x → 3 * x + 3
```

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

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

λ-abstractions and variables scope

Function definition: "λ-abstraction"

Definition

In the construct `λ x → e`, the *binder* `x` *scopes* over the expression `e` and represent the function argument.

```
f : ℕ → ℕ
f = λ x → 3 * x + 3
```

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

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

λ-abstractions and variables scope

Function definition: "λ-abstraction"

Definition

In the construct `λ x → e`, the *binder* `x` *scopes* over the expression `e` and represent the function argument.

```
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
WRONG
```

λ-abstractions and variables scope

Function definition: "λ-abstraction"

Definition

In the construct `λ x → e`, the *binder* `x` *scopes* over the expression `e` and represent the function argument.

```
f : ℕ → ℕ
f = λ x → 3 * x + 3
```

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

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

```
(λ x → λ y → x + y) y 21
```

λ-abstractions and variables scope

Function definition: "λ-abstraction"

Definition

In the construct `λ x → e`, the *binder* `x` *scopes* over the expression `e` and represent the function argument.

```
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
WRONG
```

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

λ-abstractions and variables scope

Function definition: "λ-abstraction"

Definition

In the construct `λ x → e`, the *binder* `x` *scopes* over the expression `e` and represent the function argument.

```
f : ℕ → ℕ
f = λ x → 3 * x + 3
```

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

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

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

λ-abstractions and variables scope

Function definition: "λ-abstraction"

Definition

In the construct `λ x → e`, the *binder* `x` *scopes* over the expression `e` and represent the function argument.

```
f : ℕ → ℕ
f = λ x → 3 * x + 3
```

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

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

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

Data types and nominal style

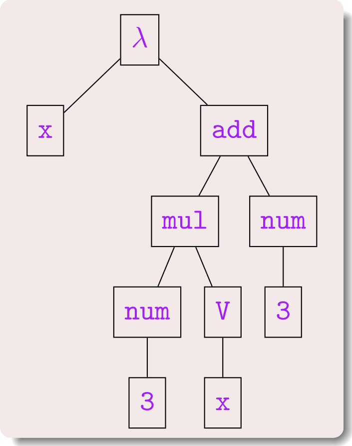
Meta-programming is made easier by the introduction of data types to represent programming languages.

```
λ x → 3 * x + 3
```

Data types and nominal style

Meta-programming is made easier by the introduction of data types to represent programming languages.

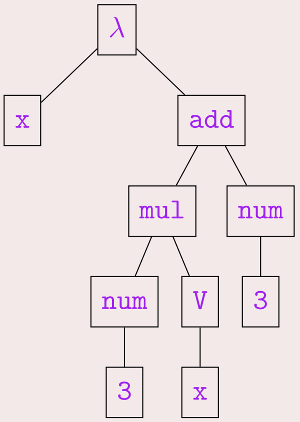
λ `x` $\rightarrow$ `3 * x + 3`



Data types and nominal style

Meta-programming is made easier by the introduction of data types to represent programming languages.

```
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))
```

Closed terms and well-formed terms

A closed term:

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

An open term (non-closed):

λ `x` $\rightarrow$ `f x`

Definition

A term is well-formed when all variables are either bound by a binder of the term either bound in the *environment*.

Ill-formed:

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

Well-formed in the environment containing `f`:

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

Definition

A term is closed if and only if it is well-formed in the empty environment.

Goal 1: To guarantee that we manipulate only well-scoped terms

α-equivalence & α-purity

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

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

α-equivalence & α-purity

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

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

α-purity of functions:

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

Definition

A function is α-pure if and only if it returns α-equivalent results when given α-equivalent inputs.

α-equivalence & α-purity

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

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

α-purity of functions:

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

Definition

A function is α-pure if and only if it returns α-equivalent results when given α-equivalent inputs.

What about this function?

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

Goal 2: Computation should preserve α-equivalence

NomPa: interface and examples

Nominal terms with 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
```

Nominal terms with NomPa

```
-- 1: Cleanning...
```

```
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
```

Nominal terms with NomPa

```
-- 1: Cleanning...
```

```
data Tm : Set where




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

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

Nominal terms with NomPa

```
-- 1: Cleanning...
```

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

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

Nominal terms with NomPa

```
-- 1: Cleanning...
```

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

```
record NomPa : Set₁ where
  field

    Name :             Set
```

Nominal terms with NomPa

```
-- 2: Separating names and binders...
```

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

```
record NomPa : Set₁ where
  field

    Name :              Set
```

Nominal terms with NomPa

```
-- 2: Separating names and binders...
```

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

```
record NomPa : Set₁ where
  field

    Name :             Set
    Binder : Set
```

Nominal terms with NomPa

```
-- 3: Indexing of names and terms...
```

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

```
record NomPa : Set₁ where
  field

    Name :               Set
    Binder : Set

```

Nominal terms with NomPa

```
-- 3: Indexing of names and terms...
```

```
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
```

Nominal terms with NomPa

```
-- 4: By abstract worlds...
```

```
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
```

Nominal terms with NomPa

```
-- 4: By abstract worlds...
```

```
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
```

Nominal terms with NomPa

```
-- Intuition: a world can thought of a list of binders
```

```
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
```

Nominal terms with NomPa

```
-- 5: Naming the binder...
```

```
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
```

Nominal terms with NomPa

```
-- 5: Naming the binder...
```

```
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
```

Nominal terms with NomPa

```
-- 6: Scope of the binder '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
```

Nominal terms with NomPa

```
-- 6: Scope of the binder ‘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
```

Nominal terms with NomPa

```
-- Remark: nothing is packaging the binder with the subterm
```

```
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
```

The NomPa interface (part 1)

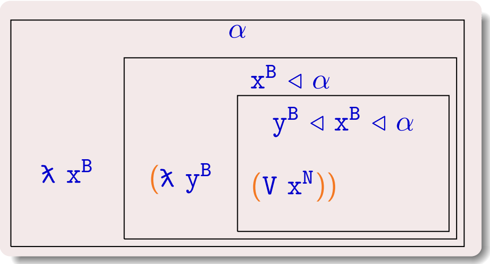
```
record NomPa : Set₁ where
  field
    World : Set
    Name : World → Set
    Binder : Set
    _◁_ : Binder → World → World

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

    ...
```

The NomPa interface (part 1)

$$\mathtt{export}^{\mathtt{N}}? : \forall\ \{\mathtt{b}\ \alpha\} \to \mathtt{Name}\ (\mathtt{b} \triangleleft \alpha) \to \mathtt{Maybe}\ (\mathtt{Name}\ \alpha)$$



Example: Collecting free-variables

```
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)
```

Example: Collecting free-variables

```
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)
```

Example: Collecting free-variables

```
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)
```

Example: Collecting free-variables

```
rm : ∀ {α} b → List (Name (b ◁ α)) → List (Name α)
rm b [] = []
rm b (x :: xs)       with export^N? {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)
```

- We cannot forget to remove b.
- No hidden execution cost.
- By parametricity we will obtain that returned names comes from the input term.

The NomPa interface (2nd part)

```
record NomPa : Set₁ where
  field
    ...
    -- The empty world
    ∅      : World
    -- An infinite set of binders
    zeroᴮ : Binder
    sucᴮ  : Binder → Binder
    -- From binders one builds names
    nameᴮ : ∀ {α} b → Name (b ◁ α)
    ...
```

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

Generic traversal and traversal kits

```
-- Here is the non-effectful traversal:
module Traverseᵀᵐ {Env} (trKit : TrKit Env Tm) where
  open TrKit trKit
  trᵀᵐ : ∀ {α β} → Env α β → Tm α → Tm β
  trᵀᵐ Δ (V x)   = trName Δ x
  trᵀᵐ Δ (t · u) = trᵀᵐ Δ t · trᵀᵐ Δ u
  trᵀᵐ Δ (ƛ b t) = ƛ _ (trᵀᵐ (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)
```

Generic traversal and traversal kits

```
-- Here is the non-effectful traversal:
module Traverseᵀᵐ {Env} (trKit : TrKit Env Tm) where
  open TrKit trKit
  trᵀᵐ : ∀ {α β} → Env α β → Tm α → Tm β
  trᵀᵐ Δ (V x)   = trName Δ x
  trᵀᵐ Δ (t · u) = trᵀᵐ Δ t · trᵀᵐ Δ u
  trᵀᵐ Δ (ƛ b t) = ƛ _ (trᵀᵐ (extEnv b Δ) t)
```

```
-- Here is the skeleton of the renaming kit:
RenameEnv : (α β : World) → Set
RenameEnv α β = (Name α → Name β) × ...

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

Based on the generic traversal

A single traversal function enables to lift effectful functions from on names (Name α → E (Name β)) to effectful functions on terms (Tm α → E (Tm β)).

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

Based on the generic traversal

A single traversal function enables to lift effectful functions from on names ($\texttt{Name}\ \alpha \to \texttt{E}\ (\texttt{Name}\ \beta)$) to effectful functions on terms ($\texttt{Tm}\ \alpha \to \texttt{E}\ (\texttt{Tm}\ \beta)$).

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

As a second step, one can do the same with functions from names to terms ($\texttt{Name}\ \alpha \to \texttt{E}\ (\texttt{Tm}\ \beta)$). Capture avoiding substitution can thus be derived from this traversal and many other functions as well.

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

NomPa: interface and usage

- The interface:
  - A notion of worlds to index names and terms.
  - Worlds start empty and are extended by binders.
  - Names are comparable and exportable under some conditions.
  - In addition: world inclusions, add/subtract/compare operations on names.
- Operations on terms:
  - Standard functions such as `fv` and `rm` are uncluttered.
  - In addition: term comparison, Normalization By Evaluation, ...
- Traversals and kits:
  - Generic traversals: most of the structure preserving, term to term, functions as a single function.
  - In addition: effectful traversals with applicative functors, more kits and traversals.

We want α-purity and thus want computations to preserve a relation...

Logical relations and parametricity!

Logical relation primer

```
τ : Set -- τ a type
⟦ τ ⟧ : τ → τ → Set -- ⟦ τ ⟧ a relation
```

```
(Aᵣ ⟦→⟧ Bᵣ) f₁ f₂ =
              ∀ {x₁ x₂} → Aᵣ x₁ x₂
                        → Bᵣ (f₁ x₁) (f₂ x₂)
```

```
(⟦Π⟧ Aᵣ Bᵣ) f₁ f₂ = ∀ {x₁ x₂} (xᵣ : Aᵣ x₁ x₂)
                    → Bᵣ xᵣ (f₁ x₁) (f₂ x₂)
```

```
⟦Set⟧ : Set → Set → Set₁
⟦Set⟧ A₁ A₂ = A₁ → A₂ → Set
```

⟦Bool⟧ and ⟦ℕ⟧ are identity relations

Parametricity primer

We want the logical relation to be the α-equivalence.

However, the language here (Agda) is fixed. The solution is parametricity.

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

Parametricity primer

We want the logical relation to be the α-equivalence.

However, the language here (Agda) is fixed. The solution is parametricity.

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

Parametricity primer

We want the logical relation to be the α-equivalence.

However, the language here (Agda) is fixed. The solution is parametricity.

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

Safety goals

In the end we get α-purity because $[\![\_]\!]$ is the α-equivalence.

```
-- In particular at type Tm.
α-equivalence ⇔ ⟦Tm⟧ ⟦∅⟧
```

We remark that our definition equips all types with α-equivalence.

```
-- α-purity implies that α-equivalent terms
-- terms are not distinguishable.
f : ∀ {α} → Tm α → Bool
f-lemma : ∀ t₁ t₂ → α-equivalence t₁ t₂ → f t₁ ≡ f t₂
```

Functions of type $\forall\ \{\alpha\} \to \texttt{Tm}\ \alpha \to \texttt{Tm}\ \alpha$ are insensitive to any renaming of the free names in their input. Identity of free names makes no importance.

Free theorems for library clients

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

$$[\![\texttt{c}]\!] : [\![\ (\texttt{lib} : \texttt{NomPa}) \to \ldots\ ]\!]\ \texttt{c}\ \texttt{c}$$

Free theorems for library clients

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

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

Free theorems for library clients

```
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 ...)
```

Free theorems for library clients

```
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 ...)
```

Free theorems for library clients

```
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 ...)
```

Free theorems for library clients

```
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 ...)
```

We are looking for definitions for ⟦World⟧, ⟦Name⟧, ... which maximize the usefulness of the resulting theorem.

NomPa soundness, modularly

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

NomPa soundness, modularly

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

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
```

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

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
```

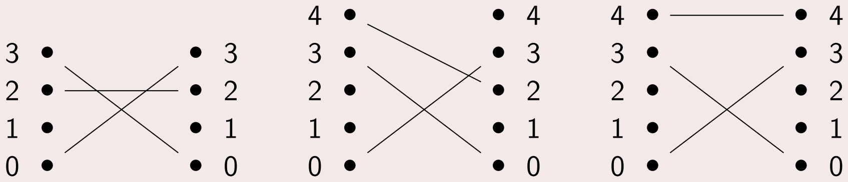
```
⟦Name⟧ : ∀ {α₁ α₂} → ⟦World⟧ α₁ α₂ → Name α₁ → Name α₂ → Set
⟦Name⟧ (ℛ , _) x₁ x₂ = ℛ x₁ x₂
```

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
```

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

NomPa soundness, modularly



α_r $\quad\quad$ $\langle 4,2\rangle[\![\triangleleft]\!]\ \alpha_r$ $\quad\quad$ $\langle 4,4\rangle[\![\triangleleft]\!]\ \langle 4,2\rangle[\![\triangleleft]\!]\ \alpha_r$

```
_⟦◁⟧_ : (⟦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
```

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
```

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

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
```

```
_⟦==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 !}
```

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
```

In the end the relation $[\![\_]\!]$ corresponds to α-equivalence.

NomPa: a multi-style library for names and binders

- The NomPa interface has a few more functions.
- And other types such as world inclusion witnesses.
- Not only nominal style bindings.
- de Bruijn style bindings (indices and levels) and computations on names.
- Combinations of these different styles.
- Many generic operations and examples.
- Encoding of various other binding techniques.

Conclusion

- Computation preserves α-equivalence.
- Thus, we manipulate only well-scoped terms.
- Names and terms indexed by worlds.
- Safety through *abstract* types on base types.
- Names are separated from binders.
- Finer grained than FreshML and HOAS (no hidden costs).
- All in Agda: code, formalization, and proofs.
- Free theorems available on-line: `http://nicolaspouillard.fr/`

Perspectives

- Improve the meta-programming support of AGDA to:
  - Infer the inclusion witnesses.
  - Provide a support for the $[\![\_]\!]$ operation.
- How NOMPA could be used in meta-theory?
- NOMPA as a target explicit language for more high-level languages (pure FreshML or the *nested* approach).
- Study the interactions with references.
- Look for other uses of parametricity as a safety proof.