Formalisation in Constructive Type Theory of Barendregt's Variable Convention for Generic Structures with Binders

Ernesto Copello [1] Nora Szasz [2] Álvaro Tasistro [2]

[1]Department of Computer Science
The University of Iowa, USA

[2]Facultad de Ingeniería
Universidad ORT Uruguay

June 22, 2018

Outline

- We introduce a universe of regular datatypes with variable binding information with:
  - a first-order named syntax interpretion
  - usual formation and elimination operators
  - operations and predicates specific to variables (swapping, free variables, fresh binders, etc)
  - an α-equivalence relation based on name-swapping.
  - iteration and induction principles which capture the *Barendregt's Variable Convention*
- We instantiate λ-Calculus and System F, deriving:
  - almost free substitution operations and α-conversion lemmas
  - substitution composition lemma
- The whole work is carried out in Constructive Type Theory and machine-checked by the system Agda.

Regular Trees Types with Binders

- Functor datatype: introduces the codes of functors.
- ⟦_⟧ function: gives the interpretation of codes.
- μ datatype: represents the fixpoint of some given F functor.

```
data Functor : Set₁ where
  |1|   :                                   Functor
  |R|   :                                   Functor
  |E|   : Set                             → Functor
  |Ef|  : Functor                         → Functor
  _|+|_ : Functor → Functor → Functor
  _|x|_ : Functor → Functor → Functor
  |v|   : Sort                            → Functor
  |B|   : Sort      → Functor → Functor
```

```
mutual
  ⟦_⟧ : Functor → Set → Set
  ⟦ |1|        ⟧ _ = ⊤
  ⟦ |R|        ⟧ A = A
  ⟦ |E|    B   ⟧ _ = B
  ⟦ |Ef|   F   ⟧ _ = μ F
  ⟦ F |+|  G   ⟧ A = ⟦ F ⟧ A ⊎ ⟦ G ⟧ A
  ⟦ F |x|  G   ⟧ A = ⟦ F ⟧ A × ⟦ G ⟧ A
  ⟦ |v| S      ⟧ _ = V
  ⟦ |B| S  G   ⟧ A = V         × ⟦ G ⟧ A

  data μ (F : Functor) : Set where
    ⟨_⟩ : ⟦ F ⟧ (μ F) → μ F
```

Regular Trees Types with Binders

- Functor datatype: introduces the codes of functors.
- ⟦_⟧ function: gives the interpretation of codes.
- μ datatype: represents the fixpoint of some given F functor.

```
data Functor : Set₁ where
  |1|   :                              Functor
  |R|   :                              Functor
  |E|   : Set                        → Functor
  |Ef|  : Functor                    → Functor
  _|+|_ : Functor → Functor → Functor
  _|x|_ : Functor → Functor → Functor
  |v|   : Sort                       → Functor
  |B|   : Sort    → Functor → Functor
```

```
mutual
  ⟦_⟧ : Functor → Set → Set
  ⟦ |1|       ⟧ _ = ⊤
  ⟦ |R|       ⟧ A = A
  ⟦ |E|    B  ⟧ _ = B
  ⟦ |Ef|   F  ⟧ _ = μ F
  ⟦ F |+|  G  ⟧ A = ⟦ F ⟧ A ⊎ ⟦ G ⟧ A
  ⟦ F |x|  G  ⟧ A = ⟦ F ⟧ A × ⟦ G ⟧ A
  ⟦ |v|  S    ⟧ _ = V
  ⟦ |B|  S  G ⟧ A = V       × ⟦ G ⟧ A
```

```
data μ (F : Functor) : Set where
  ⟨_⟩ : ⟦ F ⟧ (μ F) → μ F
```

Regular Trees Types with Binders

- Functor datatype: introduces the codes of functors.
- ⟦_⟧ function: gives the interpretation of codes.
- μ datatype: represents the fixpoint of some given *F* functor.

```
data Functor : Set₁ where
  |1|    :                                   Functor
  |R|    :                                   Functor
  |E|    : Set                            →  Functor
  |Ef|   : Functor                        →  Functor
  _|+|_  : Functor  →  Functor            →  Functor
  _|x|_  : Functor  →  Functor            →  Functor
  |v|    : Sort                           →  Functor
  |B|    : Sort     →  Functor            →  Functor
```

```
mutual
  ⟦_⟧ : Functor → Set → Set
  ⟦ |1|         ⟧ _  = ⊤
  ⟦ |R|         ⟧ A  = A
  ⟦ |E|      B  ⟧ _  = B
  ⟦ |Ef|     F  ⟧ _  = μ F
  ⟦ F |+|    G  ⟧ A  = ⟦ F ⟧ A  ⊎  ⟦ G ⟧ A
  ⟦ F |x|    G  ⟧ A  = ⟦ F ⟧ A  ×  ⟦ G ⟧ A
  ⟦ |v|  S      ⟧ _  = V
  ⟦ |B|  S   G  ⟧ A  = V        ×  ⟦ G ⟧ A

  data μ (F : Functor) : Set where
    ⟨_⟩ : ⟦ F ⟧ (μ F) → μ F
```

Regular Trees Types with Binders

- Functor datatype: introduces the codes of functors.
- ⟦_⟧ function: gives the interpretation of codes.
- μ datatype: represents the fixpoint of some given F functor.

```
data Functor : Set₁ where
  |1|   :                                 Functor
  |R|   :                                 Functor
  |E|   : Set                          →  Functor
  |Ef|  : Functor                      →  Functor
  _|+|_ : Functor  →  Functor          →  Functor
  _|x|_ : Functor  →  Functor          →  Functor
  |v|   : Sort                         →  Functor
  |B|   : Sort     →  Functor          →  Functor

mutual
  ⟦_⟧ : Functor → Set → Set
  ⟦ |1|        ⟧ _ = ⊤
  ⟦ |R|        ⟧ A = A
  ⟦ |E|     B  ⟧ _ = B
  ⟦ |Ef|    F  ⟧ _ = μ F
  ⟦ F |+|   G  ⟧ A = ⟦ F ⟧ A ⊎ ⟦ G ⟧ A
  ⟦ F |x|   G  ⟧ A = ⟦ F ⟧ A × ⟦ G ⟧ A
  ⟦ |v|  S     ⟧ _ = V
  ⟦ |B|  S  G  ⟧ A = V       × ⟦ G ⟧ A

data μ (F : Functor) : Set where
  ⟨_⟩ : ⟦ F ⟧ (μ F) → μ F
```

Lambda Calculus Example

λF : Functor - M,N :-
λF = |v| SortλTermVars - x
|+| |R| |x| |R| - | M N
|+| |B| SortλTermVars |R| - | λ x . M

λTerm : Set
λTerm = μ λF

v : V → λTerm
v = ⟨_⟩ ∘ inj_1

_·_ : λTerm → λTerm → λTerm
M · *N* = ⟨ inj_2 (inj_1 (*M* , *N*)) ⟩

ƛ : V → λTerm → λTerm
ƛ *n* *M* = ⟨ inj_2 (inj_2 (*n* , *M*)) ⟩

System F Example

```
tyF : Functor                                  -- t,r :-
tyF = |v| SortFTypeVars                        -- α
   |+| |R| |x| |R|                             -- | t → r
   |+| |B| SortFTypeVars |R|                   -- | ∀ α . t

tF : Functor                                   -- M,N :-
tF =  |v| SortFTermVars                        -- x
   |+| |R| |x| |R|                             -- | M N
   |+| |Ef| tyF |x| |B| SortFTermVars |R|      -- | λ x : t . M
   |+| |R| |x| |Ef| tyF                        -- | M t
   |+| |B| SortFTypeVars |R|                   -- | Λ α . M

FType : Set
FType = μ tyF

FTerm : Set
FTerm = μ tF
```

System F Example

```
tyF : Functor                                   - t,r :-
tyF = |v| SortFTypeVars                         - α
   |+| |R| |x| |R|                              - | t → r
   |+| |B| SortFTypeVars |R|                    - | ∀ α . t

tF : Functor                                    - M,N :-
tF = |v| SortFTermVars                          - x
   |+| |R| |x| |R|                              - | M N
   |+| |Ef| tyF |x| |B| SortFTermVars |R|       - | λ x : t . M
   |+| |R| |x| |Ef| tyF                         - | M t
   |+| |B| SortFTypeVars |R|                    - | Λ α . M

FType : Set
FType = μ tyF

FTerm : Set
FTerm = μ tF
```

System F Example

```
tyF : Functor                                   -- t,r :-
tyF =  |v| SortFTypeVars                        -- α
   |+|  |R| |x| |R|                             -- | t → r
   |+|  |B| SortFTypeVars |R|                   -- | ∀ α .  t

tF : Functor                                    -- M,N :-
tF =    |v| SortFTermVars                       -- x
   |+|  |R| |x| |R|                             -- | M N
   |+|  |Ef| tyF |x| |B| SortFTermVars |R|      -- | λ x :  t .  M
   |+|  |R| |x| |Ef| tyF                        -- | M t
   |+|  |B| SortFTypeVars |R|                   -- | Λ α .  M

FType : Set
FType = μ tyF

FTerm : Set
FTerm = μ tF
```

Lambda Calculus Fold Instantiation Example

$$\begin{aligned}
&\text{varsaux} : [\![\, \lambda\text{F} \,]\!]\ \mathbb{N} \to \mathbb{N} \\
&\text{varsaux}\ (\text{inj}_1\ _) = 1 \\
&\text{varsaux}\ (\text{inj}_2\ (\text{inj}_1\ (m\ ,\ n))) = m + n \\
&\text{varsaux}\ (\text{inj}_2\ (\text{inj}_2\ (_\ ,\ m))) = m
\end{aligned}$$

$$\begin{aligned}
&\lambda\text{F} : \text{Functor} && \text{-- M,N :-} \\
&\lambda\text{F} = \ |\text{v}|\ \text{Sort}\lambda\text{TermVars} && \text{-- x} \\
&\quad |{+}|\ \ |\text{R}|\ |\text{x}|\ |\text{R}| && \text{-- | M N} \\
&\quad |{+}|\ \ |\text{B}|\ \text{Sort}\lambda\text{TermVars}\ |\text{R}| && \text{-- | } \lambda \text{ x . M}
\end{aligned}$$

$$\begin{aligned}
&\text{vars} : \mu\ \lambda\text{F} \to \mathbb{N} \\
&\text{vars} = \text{fold}\ \lambda\text{F varsaux}
\end{aligned}$$

vars function could also be defined generically (for any functor).

Fold with Context(μ C) and a Functorial Return Type(μ H)

Fold instance

- adds a c extra argument of type μ C, used by the folded function f as an explicit invariant context through the entire fold operation
- the μ H type of the result is an instance of our universe (instead of an arbitrary set as in fold).

$$
\begin{aligned}
\mathsf{foldCtx} \;:\; & \{C\ H : \mathsf{Functor}\}(F : \mathsf{Functor}) \\
\rightarrow\; & (\mu\ C \rightarrow [\![\ F\ ]\!]\ (\mu\ H) \rightarrow \mu\ H) \\
\rightarrow\; & \mu\ C \rightarrow \mu\ F \rightarrow \mu\ H
\end{aligned}
$$

$$\mathsf{foldCtx}\ F\ f\ c = \mathsf{fold}\ F\ (f\ c)$$

Lambda Calculus Example: Naive Substitution

We derive the naive substitution for the λ-calculus from previous fold instance. Using the cF functor descriptor for the context argument, representing the pair formed by the variable to be replaced and the substituted term.

$$\text{cF} = |\text{v}|\ \text{Sort}\lambda\text{TermVars}\ |\text{x}|\ |\text{Ef}|\ \lambda\text{F}$$

$$
\begin{aligned}
&\text{substaux} : \mu\ \text{cF} \to [\![\ \lambda\text{F}\ ]\!]\ (\mu\ \lambda\text{F}) \to \mu\ \lambda\text{F} \\
&\text{substaux}\ \_ \quad (\text{inj}_2\ (\text{inj}_1\ (t_1\ ,\ t_2))) = t_1 \cdot t_2 \\
&\text{substaux}\ \_ \quad (\text{inj}_2\ (\text{inj}_2\ (y\ ,\ t))) = \bar{\lambda}\ y\ t \\
&\text{substaux}\ \langle\ x\ ,\ N\ \rangle\ (\text{inj}_1\ y)\ \text{with}\ x \stackrel{?}{=}\text{v}\ y \\
&\ldots\ |\ \text{yes}\ \_ \quad = N \\
&\ldots\ |\ \text{no}\ \_ \quad = \text{v}\ y
\end{aligned}
$$

$$
\begin{aligned}
&\_[\_{:=}\_]_n : \lambda\text{Term} \to \text{V} \to \lambda\text{Term} \to \lambda\text{Term} \\
&M\ [\ x := N\ ]_n = \text{foldCtx}\ \lambda\text{F}\ \text{substaux}\ \langle\ x\ ,\ N\ \rangle\ M
\end{aligned}
$$

Primitive Induction

fih function receives a predicate $P : \mu\ F \rightarrow Set$,and returns a predicate $[\![G]\!](\mu\ F) \rightarrow Set$, representing P holding in all $\mu\ F$ recursive positions in an element of type $[\![G]\!](\mu\ F)$.

$$
\begin{array}{llll}
\text{fih} & : \{F : \text{Functor}\}(G : \text{Functor})(P : \mu\ F \rightarrow \text{Set}) \rightarrow [\![\ G\ ]\!]\ (\mu\ F) \rightarrow \text{Set} & & \\
\text{fih}\ |1| & P\ \text{tt} & = & \top \\
\text{fih}\ |R| & P\ e & = & P\ e \\
\text{fih}\ (|E|\ \ B) & P\ e & = & \top \\
\text{fih}\ (|Ef|\ \ G) & P\ e & = & \top \\
\text{fih}\ (G_1\ |{+}|\ \ G_2) & P\ (\text{inj}_1\ \ e) & = & \text{fih}\ G_1\ \ P\ e \\
\text{fih}\ (G_1\ |{+}|\ \ G_2) & P\ (\text{inj}_2\ \ e) & = & \text{fih}\ G_2\ \ P\ e \\
\text{fih}\ (G_1\ |x|\ \ G_2) & P\ (e_1\ ,\ \ e_2) & = & \text{fih}\ G_1\ \ P\ e_1 \times \text{fih}\ G_2\ P\ e_2 \\
\text{fih}\ (|v|\ \ S) & P\ x & = & \top \\
\text{fih}\ (|B|\ S\ \ G) & P\ (x\ ,\ \ e) & = & \text{fih}\ G\ \ P\ e
\end{array}
$$

Primitive Induction

```
foldmapFh : {F : Functor}(G : Functor)(P : μ F → Set)
            → ((e : ⟦ F ⟧ (μ F)) → fih F P e →  P ⟨ e ⟩)
            → (x : ⟦ G ⟧ (μ F)) → fih G P x
foldmapFh |1|             P hi tt          =  tt
foldmapFh {F} |R|         P hi ⟨ e ⟩       =  hi e (foldmapFh {F} F P hi e)
foldmapFh (|E|   B)       P hi b           =  tt
foldmapFh (|Ef|  F)       P hi b           =  tt
foldmapFh (G1 |+|  G2)    P hi (inj1  e)   =  foldmapFh G1  P hi e
foldmapFh (G1 |+|  G2)    P hi (inj2  e)   =  foldmapFh G2  P hi e
foldmapFh (G1 |x|  G2)    P hi (e1  , e2)  =  foldmapFh G1  P hi e1 , foldmapFh G2
foldmapFh (|v|   S)       P hi n           =  tt
foldmapFh (|B| S   G)     P hi (x   , e)   =  foldmapFh G   P hi e

foldInd  :   (F : Functor)(P : μ F → Set)
         →  ((e : ⟦ F ⟧ (μ F)) → fih F P e →  P ⟨ e ⟩)
         →  (e : μ F) → P e
foldInd F P hi e = foldmapFh {F} |R| P hi e
```

Primitive Induction

$$
\begin{array}{l}
\mathsf{foldmapFh} : \{F : \mathsf{Functor}\}(G : \mathsf{Functor})(P : \mu\ F \to \mathsf{Set}) \\
\quad \to ((e : [\![\ F\ ]\!]\ (\mu\ F)) \to \mathsf{fih}\ F\ P\ e \to P\ \langle\ e\ \rangle) \\
\quad \to (x : [\![\ G\ ]\!]\ (\mu\ F)) \to \mathsf{fih}\ G\ P\ x \\
\mathsf{foldmapFh}\ |1|\ P\ hi\ \mathsf{tt} = \mathsf{tt} \\
\mathsf{foldmapFh}\ \{F\}\ |\mathsf{R}|\ P\ hi\ \langle\ e\ \rangle = hi\ e\ (\mathsf{foldmapFh}\ \{F\}\ F\ P\ hi\ e) \\
\mathsf{foldmapFh}\ (|\mathsf{E}|\ B)\ P\ hi\ b = \mathsf{tt} \\
\mathsf{foldmapFh}\ (|\mathsf{Ef}|\ F)\ P\ hi\ b = \mathsf{tt} \\
\mathsf{foldmapFh}\ (G_1\ |{+}|\ G_2)\ P\ hi\ (\mathsf{inj}_1\ e) = \mathsf{foldmapFh}\ G_1\ P\ hi\ e \\
\mathsf{foldmapFh}\ (G_1\ |{+}|\ G_2)\ P\ hi\ (\mathsf{inj}_2\ e) = \mathsf{foldmapFh}\ G_2\ P\ hi\ e \\
\mathsf{foldmapFh}\ (G_1\ |\mathsf{x}|\ G_2)\ P\ hi\ (e_1\ ,\ e_2) = \mathsf{foldmapFh}\ G_1\ P\ hi\ e_1\ ,\ \mathsf{foldmapFh}\ G_2 \\
\mathsf{foldmapFh}\ (|\mathsf{v}|\ S)\ P\ hi\ n = \mathsf{tt} \\
\mathsf{foldmapFh}\ (|\mathsf{B}|\ S\ G)\ P\ hi\ (x\ ,\ e) = \mathsf{foldmapFh}\ G\ P\ hi\ e
\end{array}
$$

$$
\begin{array}{l}
\mathsf{foldInd} : (F : \mathsf{Functor})(P : \mu\ F \to \mathsf{Set}) \\
\quad \to ((e : [\![\ F\ ]\!]\ (\mu\ F)) \to \mathsf{fih}\ F\ P\ e \to P\ \langle\ e\ \rangle) \\
\quad \to (e : \mu\ F) \to P\ e \\
\mathsf{foldInd}\ F\ P\ hi\ e = \mathsf{foldmapFh}\ \{F\}\ |\mathsf{R}|\ P\ hi\ e
\end{array}
$$

Primitive Induction

$$
\begin{array}{l}
\mathsf{foldmapFh} : \{F : \mathsf{Functor}\}(G : \mathsf{Functor})(P : \mu\ F \to \mathsf{Set}) \\
\qquad \to ((e : [\![\ F\ ]\!]\ (\mu\ F)) \to \mathsf{fih}\ F\ P\ e \to\ P\ \langle\ e\ \rangle) \\
\qquad \to (x : [\![\ G\ ]\!]\ (\mu\ F)) \to \mathsf{fih}\ G\ P\ x \\
\mathsf{foldmapFh}\ |1|\quad P\ hi\ \mathsf{tt} = \mathsf{tt} \\
\mathsf{foldmapFh}\ \{F\}\ |\mathsf{R}|\quad P\ hi\ \langle\ e\ \rangle = hi\ e\ (\mathsf{foldmapFh}\ \{F\}\ F\ P\ hi\ e) \\
\mathsf{foldmapFh}\ (|\mathsf{E}|\ \ B)\quad P\ hi\ b = \mathsf{tt} \\
\mathsf{foldmapFh}\ (|\mathsf{Ef}|\ \ F)\quad P\ hi\ b = \mathsf{tt} \\
\mathsf{foldmapFh}\ (G_1\ |{+}|\ \ G_2)\ P\ hi\ (\mathsf{inj}_1\ \ e) = \mathsf{foldmapFh}\ G_1\ \ P\ hi\ e \\
\mathsf{foldmapFh}\ (G_1\ |{+}|\ \ G_2)\ P\ hi\ (\mathsf{inj}_2\ \ e) = \mathsf{foldmapFh}\ G_2\ \ P\ hi\ e \\
\mathsf{foldmapFh}\ (G_1\ |\mathsf{x}|\ \ G_2)\ P\ hi\ (e_1\ \ ,\ e_2) = \mathsf{foldmapFh}\ G_1\ \ P\ hi\ e_1\ ,\ \mathsf{foldmapFh}\ G_2 \\
\mathsf{foldmapFh}\ (|\mathsf{v}|\ \ S)\quad P\ hi\ n = \mathsf{tt} \\
\mathsf{foldmapFh}\ (|\mathsf{B}|\ S\ \ G)\ P\ hi\ (x\ \ ,\ e) = \mathsf{foldmapFh}\ G\ \ P\ hi\ e
\end{array}
$$

$$
\begin{array}{l}
\mathsf{foldInd} : (F : \mathsf{Functor})(P : \mu\ F \to \mathsf{Set}) \\
\qquad \to ((e : [\![\ F\ ]\!]\ (\mu\ F)) \to \mathsf{fih}\ F\ P\ e \to\ P\ \langle\ e\ \rangle) \\
\qquad \to (e : \mu\ F) \to P\ e \\
\mathsf{foldInd}\ F\ P\ hi\ e = \mathsf{foldmapFh}\ \{F\}\ |\mathsf{R}|\ P\ hi\ e
\end{array}
$$

Primitive Induction

$$
\begin{array}{l}
\mathsf{foldmapFh} : \{F : \mathsf{Functor}\}(G : \mathsf{Functor})(P : \mu\ F \to \mathsf{Set}) \\
\quad \to ((e : [\![\ F\ ]\!]\ (\mu\ F)) \to \mathsf{fih}\ F\ P\ e \to\ P\ \langle\ e\ \rangle) \\
\quad \to (x : [\![\ G\ ]\!]\ (\mu\ F)) \to \mathsf{fih}\ G\ P\ x \\
\mathsf{foldmapFh}\ |1|\quad P\ hi\ \mathsf{tt} = \mathsf{tt} \\
\mathsf{foldmapFh}\ \{F\}\ |R|\quad P\ hi\ \langle\ e\ \rangle = hi\ e\ (\mathsf{foldmapFh}\ \{F\}\ F\ P\ hi\ e) \\
\mathsf{foldmapFh}\ (|E|\ \ B)\quad P\ hi\ b = \mathsf{tt} \\
\mathsf{foldmapFh}\ (|Ef|\ \ F)\quad P\ hi\ b = \mathsf{tt} \\
\mathsf{foldmapFh}\ (G_1\ |{+}|\ \ G_2)\quad P\ hi\ (\mathsf{inj}_1\ \ e) = \mathsf{foldmapFh}\ G_1\ \ P\ hi\ e \\
\mathsf{foldmapFh}\ (G_1\ |{+}|\ \ G_2)\quad P\ hi\ (\mathsf{inj}_2\ \ e) = \mathsf{foldmapFh}\ G_2\ \ P\ hi\ e \\
\mathsf{foldmapFh}\ (G_1\ |\mathsf{x}|\ \ G_2)\quad P\ hi\ (e_1\ \ ,\ e_2) = \mathsf{foldmapFh}\ G_1\ \ P\ hi\ e_1\ ,\ \mathsf{foldmapFh}\ G_2 \\
\mathsf{foldmapFh}\ (|\mathsf{v}|\ \ S)\quad P\ hi\ n = \mathsf{tt} \\
\mathsf{foldmapFh}\ (|B|\ S\ \ G)\quad P\ hi\ (x\ \ ,\ e) = \mathsf{foldmapFh}\ G\ \ P\ hi\ e
\end{array}
$$

$$
\begin{array}{l}
\mathsf{foldInd}\ :\ (F : \mathsf{Functor})(P : \mu\ F \to \mathsf{Set}) \\
\quad \to ((e : [\![\ F\ ]\!]\ (\mu\ F)) \to \mathsf{fih}\ F\ P\ e \to\ P\ \langle\ e\ \rangle) \\
\quad \to (e : \mu\ F) \to P\ e \\
\mathsf{foldInd}\ F\ P\ hi\ e = \mathsf{foldmapFh}\ \{F\}\ |R|\ P\ hi\ e
\end{array}
$$

Lambda Calculus Induction Instantiation Example

We use the presented induction principle to prove that the application of the function vars application is always greater than zero (Pvars predicate). The auxiliary lemma plus>0 states that the sum of two positive numbers is also positive.

```
PVars : μ λF → Set
PVars M = vars M > 0

proof : (e : ⟦ λF ⟧ (μ λF)) → fih λF PVars e → PVars ⟨ e ⟩
proof (inj₁ x)               tt          = s≤s z≤n
proof (inj₂ (inj₁ (M , N)))  (ihM , ihN) = plus>0 ihM ihN
proof (inj₂ (inj₂ (_ , M)))  ihM         = ihM

provePVars : (M : μ λF) → PVars M
provePVars = foldInd λF PVars proof
```

Name-Swapping

Swaps names occurrences (either free, bound or binding) of some sort.

$$\begin{array}{llll}
\text{swapF} & : \{F : \text{Functor}\}(G : \text{Functor}) \to \text{Sort} \to \text{V} \to \text{V} \to [\![G]\!] (\mu\ F) \to [\![G]\!] (\mu\ F) & & \\
\text{swapF}\ |1| & S\ a\ b\ \text{tt} & = & \text{tt} \\
\text{swapF}\ \{F\}\ |\text{R}| & S\ a\ b\ \langle\ e\ \rangle & = & \langle\ \text{swapF}\ F\ S\ a\ b\ e\ \rangle \\
\text{swapF}\ (|\text{E}|\ _) & S\ a\ b\ e & = & e \\
\text{swapF}\ (|\text{Ef}|\ G) & S\ a\ b\ \langle\ e\ \rangle & = & \langle\ \text{swapF}\ G\ S\ a\ b\ e\ \rangle \\
\text{swapF}\ (G_1\ |+|\ G_2) & S\ a\ b\ (\text{inj}_1\ e) & = & \text{inj}_1\ (\text{swapF}\ G_1\ S\ a\ b\ e) \\
\text{swapF}\ (G_1\ |+|\ G_2) & S\ a\ b\ (\text{inj}_2\ e) & = & \text{inj}_2\ (\text{swapF}\ G_2\ S\ a\ b\ e) \\
\text{swapF}\ (G_1\ |\text{x}|\ G_2) & S\ a\ b\ (e_1\ ,\ e_2) & = & \text{swapF}\ G_1\ S\ a\ b\ e_1\ ,\ \text{swapF}\ G_2\ S\ a\ b\ e_2 \\
\text{swapF}\ (|\text{v}|\ S') & S\ a\ b\ c\ \text{with}\ S' \stackrel{?}{=} \text{S}\ S & & \\
\ldots\ |\ \text{yes}\ _ & & = & (\ a \bullet b\ )_a\ c \\
\ldots\ |\ \text{no}\ _ & & = & c \\
\text{swapF}\ (|\text{B}|\ S'\ G) & S\ a\ b\ (c\ ,\ e)\ \text{with}\ S' \stackrel{?}{=} \text{S}\ S & & \\
\ldots\ |\ \text{yes}\ _ = & & & (\ a \bullet b\ )_a\ c\ ,\ \text{swapF}\ G\ S\ a\ b\ e \\
\ldots\ |\ \text{no}\ _ = & & & c\ ,\ \text{swapF}\ G\ S\ a\ b\ e
\end{array}$$

Interaction between name-swapping and the iteration principle

Definition (function f is *equivariant*)

$$\text{swap}\ a\ b\ (f(x)) = f(\text{swap}\ a\ b\ x)$$

The fold and its instance with context information are equivariant, given that the folded operation is equivariant.

$$
\begin{array}{l}
\text{lemmaSwapFoldCtxEquiv} : \{C\ H\ F\ : \text{Functor}\}\{S : \text{Sort}\}\{x\ y : \text{V}\} \\
\qquad \{e : \mu\ F\}\{f : \mu\ C \rightarrow [\![\ F\ ]\!]\ (\mu\ H) \rightarrow \mu\ H\}\{c : \mu\ C\} \\
\quad \rightarrow\ (\{c : \mu\ C\}\{S\ : \text{Sort}\}\{x\ y : \text{V}\}\{e : [\![\ F\ ]\!]\ (\mu\ H)\} \\
\qquad\qquad\qquad \rightarrow f\ (\text{swap}\ S\ x\ y\ c)\ (\text{swapF}\ F\ S\ x\ y\ e) \equiv \text{swap}\ S\ x\ y\ (f\ c\ e)) \\
\quad \rightarrow\ \text{foldCtx}\ F\ f\ (\text{swap}\ \{C\}\ S\ x\ y\ c)\ (\text{swap}\ \{F\}\ S\ x\ y\ e) \\
\qquad \equiv \\
\qquad \text{swap}\ \{H\}\ S\ x\ y\ (\text{foldCtx}\ F\ f\ c\ e)
\end{array}
$$

Example: Lambda Calculus

We derive that substitution is equivariant by direct use of last lemma. We use a direct auxiliary lemma lemma-substauxSwap stating that the function substaux, used to define substitution, is equivariant.

```
(_•_)_ = swap {λF} SortλTermVars

lemma-[]Swap : {x y z : V}{M N : λTerm}
  → (( y • z ) M) [ ( y • z )_a x := ( y • z ) N ]_n ≡ ( y • z ) (M [ x := N ]_n)
lemma-[]Swap {x} {y} {z} {M} {⟨ N ⟩}
  = lemmaSwapFoldCtxEquiv {cF} {λF} {λF} {SortλTermVars} {y} {z} {M}
     {substaux} {⟨ x , ⟨ N ⟩ ⟩}
     (λ {c} {S} {x} {y} {e} → lemma-substauxSwap {c} {S} {x} {y} {e})
```

```
data ∼αF {F : Functor} : (G : Functor) → ⟦ G ⟧ (μ F) →  ⟦ G ⟧ (μ F) → Set where
  ∼α1   :                            ∼αF |1|          tt         tt
  ∼αR   :  {e e' : ⟦ F ⟧ (μ F)}
        →  ∼αF F e e'          →  ∼αF |R|          ⟨ e ⟩      ⟨ e' ⟩
  ∼αE   :  {B : Set}{b : B}    →  ∼αF (|E| B)      b          b
  ∼αEf  :  {G : Functor}{e e' : ⟦ G ⟧ (μ G)}
        →  ∼αF G e e'          →  ∼αF (|Ef| G)     ⟨ e ⟩      ⟨ e' ⟩
  ∼α+₁  :  {F₁ F₂ : Functor}{e e' : ⟦ F₁ ⟧ (μ F)}
        →  ∼αF F₁ e e'         →  ∼αF (F₁ |+| F₂)  (inj₁ e)   (inj₁ e')
  ∼α+₂  :  {F₁ F₂ : Functor}{e e' : ⟦ F₂ ⟧ (μ F)}
        →  ∼αF F₂ e e'         →  ∼αF (F₁ |+| F₂)  (inj₂ e)   (inj₂ e')
  ∼αx   :  {F₁ F₂ : Functor}{e₁ e₁' : ⟦ F₁ ⟧ (μ F)}
           {e₂ e₂' : ⟦ F₂ ⟧ (μ F)}
        →  ∼αF F₁ e₁ e₁'       →  ∼αF F₂ e₂ e₂'
                               →  ∼αF (F₁ |x| F₂)  (e₁ , e₂)  (e₁' , e₂')
  ∼αV   :  {x : V}{S : Sort}   →  ∼αF (|v| S)      x          x
  ∼αB   :  (xs : List V){S : Sort}{x y : V}{G : Functor}{e e' : ⟦ G ⟧ (μ F)}
        →  ((z : V) → z ∉ xs  →  ∼αF G  (swapF G S x z e) (swapF G S y z e'))
                               →  ∼αF (|B| S G)    (x , e)    (y  , e')

_∼α_ : {F : Functor} → μ F → μ F → Set
_∼α_ = ∼αF |R|
```

$$\frac{\exists xs, \forall z \notin xs, (x\ z)e \sim\alpha\ (y\ z)e'}{\lambda x.e \sim\alpha\ \lambda y.e'}$$

$$\begin{aligned}
\sim\alpha\text{B} \quad : \quad & (xs : \text{List V})\{S : \text{Sort}\}\{x\ y : \text{V}\}\{G : \text{Functor}\}\{e\ e' : [\![\ G\ ]\!]\ (\mu\ F)\} \\
\rightarrow \quad & ((z : \text{V}) \rightarrow z \notin xs\ \rightarrow\ \sim\alpha\text{F}\ G\ \ (\text{swapF}\ G\ S\ x\ z\ e)\ (\text{swapF}\ G\ S\ y\ z\ e')) \\
& \rightarrow\ \sim\alpha\text{F}\ (|\text{B}|\ S\ G)\quad (x\ ,\ e)\quad (y\ ,\ e')
\end{aligned}$$

Alpha

Properties

- Equivalence relation.
- Equivariant (preserved under swapping operation)

Definition (α-compatible strong α-compatible)

For all e, e' such that $e \sim\alpha\ e'$,

- a f function is
  - α-compatible iff $f(e) \sim\alpha\ f(e')$.
  - strong α-compatible iff $f(e) \equiv f(e')$.
- a P predicate is α-compatible iff $P(e) \Leftrightarrow P(e')$.

Fold Property

Fold's application is α-convertible when applied to an α-compatible function.

$$\begin{array}{l}\text{lemma-fold-alpha} : \{F\ H : \text{Functor}\}\{f\ f' : [\![\ F\ ]\!]\ (\mu\ H) \rightarrow \mu\ H\} \\ \quad \rightarrow (\{e\ e' : [\![\ F\ ]\!]\ (\mu\ H)\} \rightarrow \sim\!\alpha\text{F}\ F\ e\ e' \rightarrow f\ e \sim\!\alpha\ f'\ e') \\ \quad \rightarrow (e : \mu\ F) \rightarrow \text{fold}\ F\ f\ e \sim\!\alpha\ \text{fold}\ F\ f'\ e\end{array}$$

As a direct corollary fold with context instance is α-compatible in its context argument if the folded function is α-compatible.

$$\begin{array}{l}\text{lemma-foldCtx-alpha-Ctx} : \{F\ H\ C : \text{Functor}\}\{f : \mu\ C \rightarrow [\![\ F\ ]\!]\ (\mu\ H) \rightarrow \mu\ H\}\{c\ c' : \mu\ C\} \\ \quad \rightarrow (\{e\ e' : [\![\ F\ ]\!]\ (\mu\ H)\}\{c\ c' : \mu\ C\} \rightarrow c \sim\!\alpha\ c' \rightarrow \sim\!\alpha\text{F}\ F\ e\ e' \rightarrow f\ c\ e \sim\!\alpha\ f\ c'\ e') \\ \quad \rightarrow c \sim\!\alpha\ c' \rightarrow (e : \mu\ F) \rightarrow \text{foldCtx}\ F\ f\ c\ e \sim\!\alpha\ \text{foldCtx}\ F\ f\ c'\ e \\ \text{lemma-foldCtx-alpha-Ctx}\ \{F\}\ \{f = f\}\ \{c\}\ \{c'\}\ p\ c{\sim}c'\ e = \text{lemma-fold-alpha}\ (p\ c{\sim}c')\ e\end{array}$$

Generic Variable Framework

We generically introduce several functions, relations and properties over our universe in a similar way as we have done for the swap function and α-equivalence relation.
Some of them:

- fv free variables function.
- ListNotOccurBind relation: which holds if all the variables in a given list do not occur in any binder position (associated with any sort) in a term.

Fold Property

Fold with context is also α-compatible in the argument being folded, given that:

- the f folded function is α-compatible and equivariant.
- the free variables in c, c' are respectively not binders in e, e'.

$$
\begin{aligned}
&\text{lemma-foldCtx-alpha} : \{F\ H\ C : \text{Functor}\} \\
&\quad \{f : \mu\ C \to [\![\ F\ ]\!]\ (\mu\ H) \to \mu\ H\}\{c\ c' : \mu\ C\}\{e\ e' : \mu\ F\} \\
&\to (\{e\ e' :\ [\![\ F\ ]\!]\ (\mu\ H)\}\{c\ c' : \mu\ C\} \to c \sim\alpha\ c' \to \sim\alpha\text{F}\ F\ e\ e' \to f\ c\ e \sim\alpha\ f\ c'\ e') \\
&\to (\{c : \mu\ C\}\{S : \text{Sort}\}\{x\ y : \text{V}\}\{e : [\![\ F\ ]\!]\ (\mu\ H)\} \\
&\qquad\qquad \to f\ (\text{swap}\ S\ x\ y\ c)\ (\text{swapF}\ F\ S\ x\ y\ e) \equiv \text{swap}\ S\ x\ y\ (f\ c\ e)) \\
&\to \text{ListNotOccurBind}\ (\text{fv}\ c)\ \ e \to \text{ListNotOccurBind}\ (\text{fv}\ c')\ \ e' \\
&\to c \sim\alpha\ c' \to e \sim\alpha\ e' \\
&\to \text{foldCtx}\ F\ f\ c\ e \sim\alpha\ \text{foldCtx}\ F\ f\ c'\ e'
\end{aligned}
$$

Fold Alpha

bindersFreeElem : {*F* : Functor}(*xs* : List V)(*e* : μ *F*)
→ ∃ (λ *e'* → ListNotOccurBind {*F*} *xs* *e'*)

Properties

lemma-bindersFreeαAlpha :
{*F* : Functor}(*xs* : List V)(*e* : μ *F*)
→ proj_1 (bindersFreeElem *xs* *e*) $\sim\alpha$ *e*

lemma-bindersFreeElem :
{*F* : Functor}(*xs* : List V)(*e* *e'* : μ *F*)
→ *e* $\sim\alpha$ *e'*
→ bindersFreeElem *xs* *e* ≡ bindersFreeElem *xs* *e'*

Fold Alpha

$$\begin{aligned}\textsf{foldCtx-alpha} \;:\;& \{C\ H : \textsf{Functor}\}(F : \textsf{Functor})\\ &\rightarrow (\mu\ C \rightarrow [\![\ F\ ]\!]\ (\mu\ H) \rightarrow \mu\ H)\\ &\rightarrow \mu\ C \rightarrow \mu\ F \rightarrow \mu\ H\end{aligned}$$

$$\textsf{foldCtx-alpha}\ F\ f\ c\ e = \textsf{foldCtx}\ F\ f\ c\ (\textsf{proj}_1\ (\textsf{bindersFreeElem}\ (\textsf{fv}\ c)\ e))$$

Properties

$$\begin{aligned}\textsf{strong}{\sim}\alpha\textsf{Compatible} \;:\;& \{A : \textsf{Set}\}\{F : \textsf{Functor}\}\\ &\rightarrow (\mu\ F \rightarrow A) \rightarrow \mu\ F \rightarrow \textsf{Set}\end{aligned}$$

$$\textsf{strong}{\sim}\alpha\textsf{Compatible}\ f\ M = \forall\ N \rightarrow M \sim\alpha\ N \rightarrow f\ M \equiv f\ N$$

As a direct consequence of lemma lemma-bindersFreeαElem, this fold instance is strong α-compatible.

$$\begin{aligned}&\textsf{lemma-foldCtx}\alpha\textsf{-Strong}\alpha\textsf{Compatible} :\\ &\quad \{C\ H\ F : \textsf{Functor}\}\{f : \mu\ C \rightarrow [\![\ F\ ]\!]\ (\mu\ H) \rightarrow \mu\ H\}\{c : \mu\ C\}\{e : \mu\ F\}\\ &\quad \rightarrow \textsf{strong}{\sim}\alpha\textsf{Compatible}\ (\textsf{foldCtx-alpha}\ F\ f\ c)\ e\end{aligned}$$

Fold Alpha Properties

It is also α-compatible in its context argument as a direct consequence of fold with context being α-compatible in this context argument.

$$
\begin{array}{l}
\text{lemma-foldCtxalpha-cxtalpha} \;:\; \{F\ H\ C : \text{Functor}\} \\
\quad \{f : \mu\ C \to [\![\ F\ ]\!]\ (\mu\ H) \to \mu\ H\}\{c\ c' : \mu\ C\} \\
\to\ (\{e\ e' :\ [\![\ F\ ]\!]\ (\mu\ H)\}\{c\ c' : \mu\ C\} \\
\qquad\qquad \to c \sim\!\alpha\ c' \to \sim\!\alpha\text{F}\ F\ e\ e' \\
\qquad\qquad \to f\ c\ e \sim\!\alpha\ f\ c'\ e') \\
\to\ c \sim\!\alpha\ c' \\
\to\ (e : \mu\ F) \to \text{foldCtx-alpha}\ F\ f\ c\ e \sim\!\alpha\ \text{foldCtx-alpha}\ F\ f\ c'\ e
\end{array}
$$

System F Example: (Strong) α-Compatible Substitution

```
_[_:=_] : FTerm → V → FTerm → FTerm
M [ x := N ]  = foldCtx-alpha  tF substaux ⟨ x , N ⟩ M


lemma-subst-alpha :  {M M′ N : FTerm}{x : V}
      → M ∼α M′ → M [ x := N ] ≡ M′ [ x := N ]
lemma-subst-alpha {M} {M′} M∼M′
   = lemma-foldCtxα-StrongαCompatible {cF} {tF} {tF} {substaux} M′ M∼M′


lemma-substα′ :  {x : V}{M N N′ : FTerm}
                 → N ∼α N′ → M [ x := N ] ∼α M [ x := N′ ]
lemma-substα′ {x} {M} (∼αR N∼N′)
   = lemma-foldCtxalpha-cxtalpha
      lemma-substaux
      (∼αR (∼αx ∼αV (∼αEf N∼N′)))
      M
```

Fold Alpha Properties

Given that the folded function f is:

- α-compatible
- equivariant
- the free variables in c are not binders in e.

Then the fold with context function is α-equivalent to the fold alpha.

$$
\begin{aligned}
&\text{lemma-foldCtxAlpha-foldCtx} : \{C\ H : \text{Functor}\}(F : \text{Functor}) \\
&\quad \{f : \mu\ C \to [\![\ F\ ]\!]\ (\mu\ H) \to \mu\ H\}\{c : \mu\ C\}\{e : \mu\ F\} \\
&\to (\{e\ e' : [\![\ F\ ]\!]\ (\mu\ H)\}\{c\ c' : \mu\ C\} \to c \sim\alpha\ c' \to \sim\alpha\text{F}\ F\ e\ e' \to f\ c\ e \sim\alpha\ f\ c'\ e') \\
&\to (\{c : \mu\ C\}\{S : \text{Sort}\}\{x\ y : \text{V}\}\{e : [\![\ F\ ]\!]\ (\mu\ H)\} \\
&\qquad \to f\ (\text{swap}\ S\ x\ y\ c)\ (\text{swapF}\ F\ S\ x\ y\ e) \equiv \text{swap}\ S\ x\ y\ (f\ c\ e)) \\
&\to \text{ListNotOccurBind}\ (\text{fv}\ c)\ e \\
&\to \text{foldCtx-alpha}\ F\ f\ c\ e \sim\alpha\ \text{foldCtx}\ F\ f\ c\ e
\end{aligned}
$$

System F Example: Relation Between Naive and Correct Substitution

We can directly apply last lemma to derive when the naive and the correct substitution operations are α-equivalent.

```
lemmaSubsts : {z : V}{M N : FTerm}
              → ListNotOccurBind (z :: fv N) M
              → M [ z := N ] ∼α M [ z := N ]ₙ
lemmaSubsts {z} {M} {N} nb
= lemma-foldCtxAlpha-foldCtx
    {cF} {tF} tF {substaux} {⟨ z , N ⟩} {M}
    lemma-substaux
    (λ {c} {S} {x} {y} {e} → lemma-substauxSwap {c} {S} {x} {y} {e})
    (fv2ctx {z} {M} {N} nb)
```

Alpha Induction Principle

$$\text{fihalpha} : \{F : \text{Functor}\}(G : \text{Functor})(P : \mu\ F \to \text{Set}) \to \text{List V} \to [\![\ G\ ]\!]\ (\mu\ F) \to \text{Set}$$

$$\begin{array}{llll}
\ldots & \ldots & \ldots & \\
\text{fihalpha}\ |\text{R}| & P\ xs\ e & = & P\ e \quad \times (\forall\ a \to a \in xs \to a\ \text{notOccurBind}\ e) \\
\text{fihalpha}\ (|\text{B}|\ S\ \ G) & P\ xs\ (x\ ,\ e) & = & x \notin xs \times \text{fihalpha}\ G\ \ P\ xs\ e
\end{array}$$

$$\begin{array}{ll}
\text{alphaPrimInd} : \{F : \text{Functor}\} & \\
\quad (P : \mu\ F \to \text{Set}) & \\
\quad (xs : \text{List V}) & \\
\to\ \alpha\text{CompatiblePred}\ P & \\
\to\ ((e : [\![\ F\ ]\!]\ (\mu\ F)) \to \text{fihalpha}\ F\ P\ xs\ e \to P\ \langle\ e\ \rangle) & \\
\to\ (e : \mu\ F) \to P\ e &
\end{array}$$

Barendregt's Variable Convention

Barendregt's Variable Convention [Bar84](Page 26)

If $M_1, \ldots, M_n$ occur in a certain mathematical context (e.g. definition, proof), then in these terms all bound variables are chosen to be different from the free variables.

$$
\begin{aligned}
&\mathsf{alphaProof} : \{F : \mathsf{Functor}\} \\
&\qquad (P : \mu\ F \to \mathsf{Set}) \\
&\qquad (xs : \mathsf{List\ V}) \\
&\quad \to\ \alpha\mathsf{CompatiblePred}\ P \\
&\quad \to\ ((e : \mu\ F)\ \to \mathsf{ListNotOccurBind}\ xs\ e \to \mathsf{ListNotOccurBind}\ (\mathsf{fv}\ e)\ e \to P\ e\ ) \\
&\quad \to\ (e : \mu\ F) \to P\ e
\end{aligned}
$$

Barendregt's Variable Convention

Barendregt's Variable Convention [Bar84](Page 26)

If $M_1, \ldots, M_n$ occur in a certain mathematical context (e.g. definition, proof), then in these terms all bound variables are chosen to be different from the free variables.

$$
\begin{aligned}
&\mathsf{alphaProof} : \{F : \mathsf{Functor}\} \\
&\qquad (P : \mu\ F \to \mathsf{Set}) \\
&\qquad (xs : \mathsf{List\ V}) \\
&\quad \to\ \alpha\mathsf{CompatiblePred}\ P \\
&\quad \to\ ((e : \mu\ F)\ \to \mathsf{ListNotOccurBind}\ xs\ e \to \mathsf{ListNotOccurBind}\ (\mathsf{fv}\ e)\ e \to P\ e\ ) \\
&\quad \to\ (e : \mu\ F) \to P\ e
\end{aligned}
$$

Barendregt's Variable Convention

Barendregt's Variable Convention [Bar84](Page 26)

If $M_1, \ldots, M_n$ occur in a certain mathematical context (e.g. definition, proof), then in these terms all bound variables are chosen to be different from the free variables.

```
alphaProof : {F : Functor}
      (P : μ F → Set)
      (xs : List V)
   → αCompatiblePred P
   → ((e : μ F)  → ListNotOccurBind xs e → ListNotOccurBind (fv e) e → P e )
   → (e : μ F) → P e
```

Barendregt's Variable Convention

Barendregt's Variable Convention [Bar84](Page 26)

If $M_1, \ldots, M_n$ occur in a certain mathematical context (e.g. definition, proof), then in these terms all bound variables are chosen to be different from the free variables.

```
alphaProof : {F : Functor}
      (P : μ F → Set)
      (xs : List V)
  → αCompatiblePred P
  → ((e : μ F)  → ListNotOccurBind xs e → ListNotOccurBind (fv e) e → P e)
  → (e : μ F) → P e
```

Not an induction principle over terms, and thus applicable in more cases, as the BVC.

System F: Naive Substitution Composition Lemma

First we prove the substitution composition lemma for the **naive substitution** operation by a direct induction on terms.

PSCn : {x y : V}{L : FTerm} → FTerm → FTerm → Set
PSCn {x} {y} {L} N M = $x \notin y$ :: fv L → x notOccurBind L
→ $(M[\, x := N \,]_n)[\, y := L \,]_n \sim\alpha (M[\, y := L \,]_n)[\, x := N[\, y := L \,]_n \,]_n$

lemma-substCompositionN : {x y : V}{M N L : FTerm} → PSCn {x} {y} {L} N M
lemma-substCompositionN {x} {y} {M} {N} {L}
= foldInd tF (PSCn {x} {y} {L} N) lemma-substCompositionNAux M

System F: Naive Substitution Composition Lemma

The abstraction proof case is proved as usually done in pen-and-paper proofs.

```
lemma-substCompositionNAux  (inj₂ (inj₂ (inj₁ (t , z , M))))
                            (_ , hiM)
                            xnotInyfvL
                            xnotBL =
begin
  (ƛ z t M) [ x ≔ N ]ₙ [ y ≔ L ]ₙ
≈⟨ refl ⟩
  ƛ z t (M [ x ≔ N ]ₙ [ y ≔ L ]ₙ)
∼⟨ ∼αR (∼α+₂ (∼α+₂ (∼α+₁
  (∼αx ϱF (lemma∼+B (hiM xnotInyfvL xnotBL)))))) ⟩
  ƛ z t (M [ y ≔ L ]ₙ [ x ≔ N [ y ≔ L ]ₙ ]ₙ)
≈⟨ refl ⟩
  (ƛ z t M) [ y ≔ L ]ₙ [ x ≔ N [ y ≔ L ]ₙ ]ₙ
∎
```

System F: Naive Substitution Composition Lemma

The abstraction proof case is proved as usually done in pen-and-paper proofs.

```
lemma-substCompositionNAux  (inj₂ (inj₂ (inj₁ (t , z , M))))
                            (_ , hiM)
                            xnotInyfvL
                            xnotBL =
begin
   (ƛ z t M) [ x ≔ N ]ₙ [ y ≔ L ]ₙ
≈⟨ refl  ⟩
   ƛ z t (M [ x ≔ N ]ₙ [ y ≔ L ]ₙ)
∼⟨ ∼αR (∼α+₂  (∼α+₂ (∼α+₁
   (∼αx  ρF (lemma∼+B (hiM xnotInyfvL xnotBL)))))) ⟩
   ƛ z t (M [ y ≔ L ]ₙ [ x ≔ N [ y ≔ L ]ₙ ]ₙ)
≈⟨ refl  ⟩
   (ƛ z t M) [ y ≔ L ]ₙ [ x ≔ N [ y ≔ L ]ₙ ]ₙ
∎
```

System F: Naive Substitution Composition Lemma

The abstraction proof case is proved as usually done in pen-and-paper proofs.

```
lemma-substCompositionNAux  (inj₂ (inj₂ (inj₁ (t , z , M))))
                            (_ , hiM)
                            xnotInyfvL
                            xnotBL =
begin
  (ƛ z t M) [ x ≔ N ]ₙ [ y ≔ L ]ₙ
≈⟨ refl ⟩
  ƛ z t (M [ x ≔ N ]ₙ [ y ≔ L ]ₙ)
∼⟨ ∼αR (∼α+₂ (∼α+₂ (∼α+₁
  (∼αx ϱF (lemma∼+B (hiM xnotInyfvL xnotBL)))))) ⟩
  ƛ z t (M [ y ≔ L ]ₙ [ x ≔ N [ y ≔ L ]ₙ ]ₙ)
≈⟨ refl ⟩
  (ƛ z t M) [ y ≔ L ]ₙ [ x ≔ N [ y ≔ L ]ₙ ]ₙ
∎
```

System F: Naive Substitution Composition Lemma

The abstraction proof case is proved as usually done in pen-and-paper proofs.

```
lemma-substCompositionNAux  (inj₂ (inj₂ (inj₁ (t , z , M))))
                            (_ , hiM)
                            xnotInyfvL
                            xnotBL =
begin
   (ƛ z t M) [ x := N ]ₙ [ y := L ]ₙ
≈⟨ refl  ⟩
   ƛ z t (M [ x := N ]ₙ [ y := L ]ₙ)
∼⟨ ∼αR (∼α+₂  (∼α+₂ (∼α+₁
   (∼αx  ϱF (lemma∼+B (hiM xnotInyfvL xnotBL)))))) ⟩
   ƛ z t (M [ y := L ]ₙ [ x := N [ y := L ]ₙ ]ₙ)
≈⟨ refl  ⟩
   (ƛ z t M) [ y := L ]ₙ [ x := N [ y := L ]ₙ ]ₙ
■
```

System F: Substitution Composition Lemma

We now prove the substitution composition lemma for the **correct substitution** using the alpha proof principle.

TreeFTermF = |Ef| tF |x| |Ef| tF |x| |Ef| tF
TreeFTerm = μ TreeFTermF

PSComp : {*x* *y* : V} → TreeFTerm → Set
PSComp {*x*} {*y*} ⟨ *M* , *N* , *L* ⟩ = *x* ∉ *y* :: fv *L*
→ (*M* [*x* := *N*]) [*y* := *L*] ∼α (*M* [*y* := *L*])[*x* := *N* [*y* := *L*]]

```
= begin
    (M' [ x := N' ]) [ y := L' ]
  ≈⟨ cong (λ z → z [ y := L' ]) (lemma-subst-alpha (σ M~M'))           ⟩
    (M  [ x := N' ]) [ y := L' ]
  ≈⟨ lemma-subst-alpha {M [ x := N' ]} {M [ x := N ]}
                       (lemma-substα′ {x} {M} (σ N~N'))  ⟩
    (M  [ x := N  ]) [ y := L' ]
  ∼⟨ lemma-substα′ {y} {M [ x := N ]} (σ L~L')                          ⟩
    (M  [ x := N  ]) [ y := L  ]
  ∼⟨ PMs x∉y:fvL                                                        ⟩
    (M  [ y := L  ]) [ x := N  [ y := L ] ]
  ≈⟨ cong (λ P → P [ x := N [ y := L ] ]) (lemma-subst-alpha M~M')      ⟩
    (M' [ y := L  ]) [ x := N  [ y := L ] ]
  ≈⟨ lemma-subst-alpha {M' [ y := L ]} {M' [ y := L' ]} {N [ y := L ]} {x}
                    (lemma-substα′ {y} {M'} L~L')                       ⟩
    (M' [ y := L' ]) [ x := N  [ y := L ] ]
  ≈⟨ cong (λ P → (M' [ y := L' ]) [ x := P ]) (lemma-subst-alpha N~N')  ⟩
    (M' [ y := L' ]) [ x := N' [ y := L ] ]
  ∼⟨ lemma-substα′ {x} {M' [ y := L' ]} {N' [ y := L ]}
                    (lemma-substα′ {y} {N'} L~L')                       ⟩
    (M' [ y := L' ]) [ x := N' [ y := L' ] ]
  ■
```

```
= begin
    (M' [ x := N' ]) [ y := L' ]
  ≈⟨ cong (λ z → z [ y := L' ]) (lemma-subst-alpha (σ M∼M'))                         ⟩
    (M  [ x := N' ]) [ y := L' ]
  ≈⟨ lemma-subst-alpha {M [ x := N' ]} {M [ x := N ]}
                                    (lemma-substα' {x} {M} (σ N∼N'))  ⟩
    (M  [ x := N  ]) [ y := L' ]
  ∼⟨ lemma-substα' {y} {M [ x := N ]} (σ L∼L')                                       ⟩
    (M  [ x := N  ]) [ y := L  ]
  ∼⟨ PMs x∉y:fvL                                                                    ⟩
    (M  [ y := L  ]) [ x := N  [ y := L ] ]
  ≈⟨ cong (λ P → P [ x := N [ y := L ] ]) (lemma-subst-alpha M∼M')                  ⟩
    (M' [ y := L  ]) [ x := N  [ y := L ] ]
  ≈⟨ lemma-subst-alpha {M' [ y := L ]} {M' [ y := L' ]} {N [ y := L ]} {x}
                               (lemma-substα' {y} {M'} L∼L')                       ⟩
    (M' [ y := L' ]) [ x := N  [ y := L ] ]
  ≈⟨ cong (λ P → (M' [ y := L' ]) [ x := P ]) (lemma-subst-alpha N∼N')            ⟩
    (M' [ y := L' ]) [ x := N' [ y := L ] ]
  ∼⟨ lemma-substα' {x} {M' [ y := L' ]} {N' [ y := L ]}
                               (lemma-substα' {y} {N'} L∼L')                       ⟩
    (M' [ y := L' ]) [ x := N' [ y := L' ] ]
  ■
```

```
= begin
    (M'  [ x := N' ])  [ y := L'  ]
  ≈⟨ cong (λ z → z [ y := L' ]) (lemma-subst-alpha (σ M∼M'))                  ⟩
    (M   [ x := N' ])  [ y := L'  ]
  ≈⟨ lemma-subst-alpha  {M [ x := N' ]} {M [ x := N ]}
                                  (lemma-substα′ {x} {M} (σ N∼N'))  ⟩
    (M   [ x := N  ])  [ y := L'  ]
  ∼⟨ lemma-substα′ {y} {M [ x := N  ]} (σ L∼L')                                ⟩
    (M   [ x := N  ])  [ y := L   ]
  ∼⟨ PMs x∉y:fvL                                                                ⟩
    (M   [ y := L  ])  [ x := N  [ y := L  ] ]
  ≈⟨ cong  (λ P → P [ x := N [ y := L ] ]) (lemma-subst-alpha M∼M')          ⟩
    (M'  [ y := L  ])  [ x := N  [ y := L  ] ]
  ≈⟨ lemma-subst-alpha  {M' [ y := L ]} {M' [ y := L' ]} {N [ y := L ]} {x}
                             (lemma-substα′ {y} {M'} L∼L')                      ⟩
    (M'  [ y := L'  ])  [ x := N  [ y := L  ] ]
  ≈⟨ cong  (λ P → (M' [ y := L' ])  [ x := P ]) (lemma-subst-alpha N∼N')     ⟩
    (M'  [ y := L'  ])  [ x := N'  [ y := L  ] ]
  ∼⟨ lemma-substα′  {x} {M'  [ y := L' ]} {N' [ y := L ]}
                             (lemma-substα′ {y} {N'} L∼L')                      ⟩
    (M'  [ y := L'  ])  [ x := N'  [ y := L'  ] ]
  ■
```

```
= begin
    (M'  [ x := N'  ])  [ y := L'   ]
  ≈⟨ cong (λ z → z [ y := L' ]) (lemma-subst-alpha (σ M~M'))          ⟩
    (M   [ x := N'  ])  [ y := L'   ]
  ≈⟨ lemma-subst-alpha  {M [ x := N' ]} {M [ x := N ]}
                        (lemma-substα′ {x} {M} (σ N~N'))  ⟩
    (M   [ x := N   ])  [ y := L'   ]
  ~⟨ lemma-substα′ {y} {M [ x := N  ]} (σ L~L')                        ⟩
    (M   [ x := N   ])  [ y := L    ]
  ~⟨ PMs x∉y:fvL                                                       ⟩
    (M   [ y := L   ])  [ x := N   [ y := L  ] ]
  ≈⟨ cong  (λ P → P [ x := N [ y := L ] ]) (lemma-subst-alpha M~M')   ⟩
    (M'  [ y := L   ])  [ x := N   [ y := L  ] ]
  ≈⟨ lemma-subst-alpha  {M' [ y := L ]} {M' [ y := L' ]} {N [ y := L ]} {x}
                     (lemma-substα′ {y} {M'} L~L')                     ⟩
    (M'  [ y := L'  ])  [ x := N   [ y := L  ] ]
  ≈⟨ cong  (λ P → (M' [ y := L' ])  [ x := P ]) (lemma-subst-alpha N~N')  ⟩
    (M'  [ y := L'  ])  [ x := N'  [ y := L  ] ]
  ~⟨ lemma-substα′  {x} {M'  [ y := L' ]} {N' [ y := L ]}
                     (lemma-substα′ {y} {N'} L~L')                     ⟩
    (M'  [ y := L'  ])  [ x := N'  [ y := L' ] ]
  ■
```

= begin

$(M\ [\ x := N\ ])\ \ [\ y := L\ ]$

≈⟨ lemma-subst-alpha {M [$x := N$]} (lemmaSubsts {x} {M} {N} x:fvN-NB-M) ⟩

$M\ \ [\ x := N\ ]_n\ \ [\ y := L\ ]$

∼⟨ lemmaSubsts {y} {M [$x := N$]$_n$} {L} y:fvL-NB-M[x:=N]$_n$ ⟩

$M\ \ [\ x := N\ ]_n\ \ [\ y := L\ ]_n$

∼⟨ lemma-substCompositionN {x} {y} {M} {N} {L} *xnIny:fvL* x-NB-L ⟩

$M\ \ [\ y := L\ ]_n\ \ [\ x := N\ [\ y := L\ ]_n\ \ ]_n$

∼⟨ lemma-substn-alpha {x} {M [$y := L$]$_n$} (σ (lemmaSubsts {y} {N} y:fvL-NB-N)) ⟩

$M\ \ [\ y := L\ ]_n\ \ [\ x := N\ [\ y := L\ ]\ \ ]_n$

∼⟨ σ (lemmaSubsts {x} {M [$y := L$]$_n$} {N [$y := L$]} x:fvN[y:=L]-NB-M[y:=L]$_n$) ⟩

$M\ \ [\ y := L\ ]_n\ \ [\ x := N\ [\ y := L\ ]\ \ ]$

≈⟨ lemma-subst-alpha (σ (lemmaSubsts {y} {M} {L} y:fvL-NB-M)) ⟩

$(M\ [\ y := L\ ])\ \ [\ x := N\ [\ y := L\ ]\ \ ]$

■

= begin

$(M\ [\ x := N\ ])\ [\ y := L\ ]$

$\approx\langle$ lemma-subst-alpha $\{M\,[\ x := N\ ]\}$ (lemmaSubsts $\{x\}$ $\{M\}$ $\{N\}$ x:fvN-NB-M) $\rangle$

$M\ [\ x := N\ ]_n\ [\ y := L\ ]$

$\sim\langle$ lemmaSubsts $\{y\}$ $\{M\,[\ x := N\ ]_n\}$ $\{L\}$ y:fvL-NB-M[x:=N]$_n$ $\rangle$

$M\ [\ x := N\ ]_n\ [\ y := L\ ]_n$

$\sim\langle$ lemma-substCompositionN $\{x\}$ $\{y\}$ $\{M\}$ $\{N\}$ $\{L\}$ *xnIny*:*fvL* x-NB-L $\rangle$

$M\ [\ y := L\ ]_n\ [\ x := N\,[\ y := L\ ]_n\ ]_n$

$\sim\langle$ lemma-substn-alpha $\{x\}$ $\{M\,[\ y := L\ ]_n\}$ (σ (lemmaSubsts $\{y\}$ $\{N\}$ y:fvL-NB-N)) $\rangle$

$M\ [\ y := L\ ]_n\ [\ x := N\,[\ y := L\ ]\ ]_n$

$\sim\langle$ σ (lemmaSubsts $\{x\}$ $\{M\,[\ y := L\ ]_n\}$ $\{N\,[\ y := L\ ]\}$ x:fvN[y:=L]-NB-M[y:=L]$_n$) $\rangle$

$M\ [\ y := L\ ]_n\ [\ x := N\,[\ y := L\ ]\ ]$

$\approx\langle$ lemma-subst-alpha (σ (lemmaSubsts $\{y\}$ $\{M\}$ $\{L\}$ y:fvL-NB-M)) $\rangle$

$(M\ [\ y := L\ ])\ [\ x := N\,[\ y := L\ ]\ ]$

■

= begin

$(M\ [\ x := N\ ])\ [\ y := L\ ]$

$\approx\langle$ lemma-subst-alpha $\{M\ [\ x := N\ ]\}$ (lemmaSubsts $\{x\}$ $\{M\}$ $\{N\}$ x:fvN-NB-M) $\rangle$

$M\ [\ x := N\ ]_n\ [\ y := L\ ]$

$\sim\langle$ lemmaSubsts $\{y\}$ $\{M\ [\ x := N\ ]_n\}$ $\{L\}$ y:fvL-NB-M[x:=N]$_n$ $\rangle$

$M\ [\ x := N\ ]_n\ [\ y := L\ ]_n$

$\sim\langle$ lemma-substCompositionN $\{x\}$ $\{y\}$ $\{M\}$ $\{N\}$ $\{L\}$ *xnIny*:*fvL* x-NB-L $\rangle$

$M\ [\ y := L\ ]_n\ [\ x := N\ [\ y := L\ ]_n\ ]_n$

$\sim\langle$ lemma-substn-alpha $\{x\}$ $\{M\ [\ y := L\ ]_n\}$ (σ (lemmaSubsts $\{y\}$ $\{N\}$ y:fvL-NB-N)) $\rangle$

$M\ [\ y := L\ ]_n\ [\ x := N\ [\ y := L\ ]\ ]_n$

$\sim\langle$ σ (lemmaSubsts $\{x\}$ $\{M\ [\ y := L\ ]_n\}$ $\{N\ [\ y := L\ ]\}$ x:fvN[y:=L]-NB-M[y:=L]$_n$) $\rangle$

$M\ [\ y := L\ ]_n\ [\ x := N\ [\ y := L\ ]\ ]$

$\approx\langle$ lemma-subst-alpha (σ (lemmaSubsts $\{y\}$ $\{M\}$ $\{L\}$ y:fvL-NB-M)) $\rangle$

$(M\ [\ y := L\ ])\ [\ x := N\ [\ y := L\ ]\ ]$

■

= begin

$(M\ [\ x := N\ ])\ [\ y := L\ ]$

$\approx\langle$ lemma-subst-alpha $\{M\ [\ x := N\ ]\}$ (lemmaSubsts $\{x\}$ $\{M\}$ $\{N\}$ x:fvN-NB-M) $\rangle$

$M\ [\ x := N\ ]_n\ [\ y := L\ ]$

$\sim\langle$ lemmaSubsts $\{y\}$ $\{M\ [\ x := N\ ]_n\}$ $\{L\}$ y:fvL-NB-M[x:=N]$_n$ $\rangle$

$M\ [\ x := N\ ]_n\ [\ y := L\ ]_n$

$\sim\langle$ lemma-substCompositionN $\{x\}$ $\{y\}$ $\{M\}$ $\{N\}$ $\{L\}$ *xnIny:fvL* x-NB-L $\rangle$

$M\ [\ y := L\ ]_n\ [\ x := N\ [\ y := L\ ]_n\ ]_n$

$\sim\langle$ lemma-substn-alpha $\{x\}$ $\{M\ [\ y := L\ ]_n\}$ (σ (lemmaSubsts $\{y\}$ $\{N\}$ y:fvL-NB-N)) $\rangle$

$M\ [\ y := L\ ]_n\ [\ x := N\ [\ y := L\ ]\ ]_n$

$\sim\langle$ σ (lemmaSubsts $\{x\}$ $\{M\ [\ y := L\ ]_n\}$ $\{N\ [\ y := L\ ]\}$ x:fvN[y:=L]-NB-M[y:=L]$_n$) $\rangle$

$M\ [\ y := L\ ]_n\ [\ x := N\ [\ y := L\ ]\ ]$

$\approx\langle$ lemma-subst-alpha (σ (lemmaSubsts $\{y\}$ $\{M\}$ $\{L\}$ y:fvL-NB-M)) $\rangle$

$(M\ [\ y := L\ ])\ [\ x := N\ [\ y := L\ ]\ ]$

■

Thanks.

- Hendrik Barendregt.
  The λ-calculus Its Syntax and Semantics, volume 103 of *Studies in Logic and the Foundations of Mathematics*.
  North Holland, revised edition, 1984.

- Marcin Benke, Peter Dybjer, and Patrik Jansson.
  Universes for generic programs and proofs in dependent type theory.
  Nordic Journal of Computing, 10(4):265–289, December 2003.

- Ulf Norell.
  Dependently typed programming in agda.
  In *Proceedings of the 6th International Conference on Advanced Functional Programming*, AFP'08, Berlin, 2009. Springer-Verlag.