

Principles of Recursion and Induction for Nominal Lambda Calculus.

March 29, 2015

All the code shown is compiled in the last Agda's version 2.4.2.2 and 0.9 standard library, and can be fully accessed at:

<https://github.com/ernius/formalmetatheory-nominal>

1 Infrastructure

```
data  $\Lambda$  : Set where
  v      : Atom  $\rightarrow$   $\Lambda$ 
   $\cdot$     :  $\Lambda \rightarrow \Lambda \rightarrow \Lambda$ 
   $\lambda$    : Atom  $\rightarrow \Lambda \rightarrow \Lambda$ 
```

Figure 1: Terms

```
data  $\_ \# \_$  (a : Atom) :  $\Lambda \rightarrow$  Set where
  #v      : {b : Atom}  $\rightarrow b \neq a \rightarrow a \# v b$ 
  # $\cdot$     : {M N :  $\Lambda$ }  $\rightarrow a \# M \rightarrow a \# N \rightarrow a \# M \cdot N$ 
  # $\lambda \equiv$  : {M :  $\Lambda$ }  $\rightarrow a \# \lambda a M$ 
  # $\lambda$     : {b : Atom} {M :  $\Lambda$ }  $\rightarrow a \# M \rightarrow a \# \lambda b M$ 

( $\_ \bullet \_$ )a : Atom  $\rightarrow$  Atom  $\rightarrow$  Atom  $\rightarrow$  Atom
( $a \bullet b$ )a c with c  $\stackrel{?}{=}_a$  a
... | yes  _ = b
... | no   _ with c  $\stackrel{?}{=}_a$  b
...       | yes _ = a
...       | no  _ = c

( $\_ \bullet \_$ )a : Atom  $\rightarrow$  Atom  $\rightarrow \Lambda \rightarrow \Lambda$ 
( $a \bullet b$ )a v c = v (( $a \bullet b$ )a c)
( $a \bullet b$ )a M · N = (( $a \bullet b$ )a M) · (( $a \bullet b$ )a N)
( $a \bullet b$ )a  $\lambda$  c M =  $\lambda$  (( $a \bullet b$ )a c) (( $a \bullet b$ )a M)
```

```

 $\_ \bullet_a \_ : \Pi \rightarrow \text{Atom} \rightarrow \text{Atom}$ 
 $\pi \bullet_a a = \text{foldr } (\lambda s b \rightarrow ( \text{proj}_1 s \bullet \text{proj}_2 s )_a b) a \pi$ 

 $\_ \bullet \_ : \Pi \rightarrow \Lambda \rightarrow \Lambda$ 
 $\pi \bullet M = \text{foldr } (\lambda s M \rightarrow ( \text{proj}_1 s \bullet \text{proj}_2 s ) M) M \pi$ 

data  $\_ \sim_\alpha \_ : \Lambda \rightarrow \Lambda \rightarrow \text{Set}$  where
   $\sim_\alpha \vee : \{a : \text{Atom}\} \rightarrow \vee a \sim_\alpha \vee a$ 
   $\sim_\alpha \cdot : \{M M' N N' : \Lambda\} \rightarrow M \sim_\alpha M' \rightarrow N \sim_\alpha N' \rightarrow M \cdot N \sim_\alpha M' \cdot N'$ 
   $\sim_\alpha \lambda : \{M N : \Lambda\} \{a b : \text{Atom}\} (xs : \text{List Atom}) \rightarrow ((c : \text{Atom}) \rightarrow c \notin xs \rightarrow (a \bullet c) M \sim_\alpha (b \bullet c) N) \rightarrow \lambda a M \sim_\alpha \lambda b N$ 

```

2 Induction Principles

Primitive induction over terms presented in figure 1:

```

TermPrimInd : (P :  $\Lambda \rightarrow \text{Set}$ )
   $\rightarrow (\forall a \rightarrow P (\vee a))$ 
   $\rightarrow (\forall M N \rightarrow P M \rightarrow P N \rightarrow P (M \cdot N))$ 
   $\rightarrow (\forall M b \rightarrow P M \rightarrow P (\lambda b M))$ 
   $\rightarrow \forall M \rightarrow P M$ 

```

Figure 2: Primitive Induction

Next, we introduce another induction principle over terms with a stronger inductive hypothesis for the abstraction case, which says the property holds for all permutation of names of the body of the abstraction. This principle is proved using previous primitive induction principle.

```

TermIndPerm : (P :  $\Lambda \rightarrow \text{Set}$ )
   $\rightarrow (\forall a \rightarrow P (\vee a))$ 
   $\rightarrow (\forall M N \rightarrow P M \rightarrow P N \rightarrow P (M \cdot N))$ 
   $\rightarrow (\forall M b \rightarrow (\forall \pi \rightarrow P (\pi \bullet M)) \rightarrow P (\lambda b M))$ 
   $\rightarrow \forall M \rightarrow P M$ 

```

Figure 3: Permutation Induction Principle

α -compatible predicates:

```

 $\alpha\text{CompatiblePred} : (\Lambda \rightarrow \text{Set}) \rightarrow \text{Set}$ 
 $\alpha\text{CompatiblePred } P = \{M N : \Lambda\} \rightarrow M \sim_\alpha N \rightarrow P M \rightarrow P N$ 

```

If the predicate is α -compatible then we can prove the following induction principle using previous induction principle. This new principle enables us to

choose the variable of the abstraction case different from a finite list of variables, in this way this principle allow us to emulate Barendregt Variable Convention (BVC).

$$\begin{aligned}
& \text{Term}\alpha\text{PrimInd} : (P : \Lambda \rightarrow \text{Set}) \\
& \rightarrow \alpha\text{CompatiblePred } P \\
& \rightarrow (\forall a \rightarrow P (\text{v } a)) \\
& \rightarrow (\forall M N \rightarrow P M \rightarrow P N \rightarrow P (M \cdot N)) \\
& \rightarrow \exists (\lambda vs \rightarrow (\forall M b \rightarrow b \notin' vs \rightarrow P M \rightarrow P (\text{x } b M))) \\
& \rightarrow \forall M \rightarrow P M
\end{aligned}$$

Again assuming an α -compatible predicate, we can prove the following principle using again the induction principle of figure 3:

$$\begin{aligned}
& \text{Term}\alpha\text{IndPerm} : \forall P \rightarrow \alpha\text{CompatiblePred } P \\
& \rightarrow (\forall a \rightarrow P (\text{v } a)) \\
& \rightarrow (\forall M N \rightarrow P M \rightarrow P N \rightarrow P (M \cdot N)) \\
& \rightarrow \exists (\lambda as \rightarrow (\forall M b \rightarrow b \notin' as \rightarrow (\forall \pi \rightarrow P (\pi \bullet M)) \rightarrow P (\text{x } b M))) \\
& \rightarrow \forall M \rightarrow P M
\end{aligned}$$

Figure 4: Permutation α Induction

3 Iteration and Recursion Principles

We want to define strong α -compatible functions, that is, functions over the α -equivalence class of terms. This functions can not depend on the abstraction variables of a term. We can resume this concept in the following definition:

$$\begin{aligned}
& \text{strong}\sim\alpha\text{Compatible} : \{A : \text{Set}\} \rightarrow (\Lambda \rightarrow A) \rightarrow \Lambda \rightarrow \text{Set} \\
& \text{strong}\sim\alpha\text{Compatible } f M = \forall N \rightarrow M \sim\alpha N \rightarrow f M \equiv f N
\end{aligned}$$

We define an iteration principle over raw terms which always produces α -compatible functions. This is granted because abstraction variables are given by the induction principle, hiding the specific abstraction variables of the given term, in this way the result of a function defined with this iterator has no way to extract any information from abstracted variables. This principle also allow us to give a list of variables from where the abstractions variables will not to be choosen, this will be usefull to define the no capture substitution operation latter. This iteration principle is derived from the last presented induction principle in figure 4.

$$\begin{aligned}
& \Lambda\text{It} : (A : \text{Set}) \\
& \rightarrow (\text{Atom} \rightarrow A) \\
& \rightarrow (A \rightarrow A \rightarrow A) \\
& \rightarrow \text{List Atom} \times (\text{Atom} \rightarrow A \rightarrow A) \\
& \rightarrow \Lambda \rightarrow A
\end{aligned}$$

The following result establish the strong compatibility of previous iteration principle. This result is proved using the induction principle in figure 4.

```

lemma  $\alpha$ StrongCompatible : (A : Set)
  → (hv : Atom → A)
  → (h· : A → A → A)
  → (vs : List Atom)
  → (hλ : Atom → A → A)
  → (M :  $\Lambda$ ) → strong~ $\alpha$ Compatible ( $\Lambda$ lt A hv h· (vs , hλ)) M

```

Figure 5: Strong α Compatibility of the Iteration Principle

4 Iterator Application

Next we show several iteration principle applications.

4.1 Free Variables

We implement the function that returns the free variables of a term.

```

fv :  $\Lambda$  → List Atom
fv =  $\Lambda$ lt (List Atom) [] _+_ _ ([],  $\lambda$  v r → r - v)

```

As a direct consequence of strong α compatibility of the iteration principle we obtain that α compatible terms have equal free variables.

The relation $_ \ast _$ holds when a variables occurs free in a term.

```

data  $\_ \ast \_$  : Atom →  $\Lambda$  → Set where
  *v : {x : Atom} → x * v x
  *.l : {x : Atom} {M N :  $\Lambda$ } → x * M → x * (M · N)
  *.r : {x : Atom} {M N :  $\Lambda$ } → x * N → x * (M · N)
  *.λ : {x y : Atom} {M :  $\Lambda$ } → x * M → y  $\neq$  x → x * ( $\lambda$  y M)

```

We can use the last induction principle (fig. 4) to prove the following result predicate:

```

Pfv* : Atom →  $\Lambda$  → Set
Pfv* a M = a  $\in$ ' fv M → a * M

```

4.2 Substitution

We implement the no capture substitution operation. We give the substituted variable and free variables of the replaced term as variables to not to choose as abstractions to avoid any variable capture.

```

hvar : Atom →  $\Lambda$  → Atom →  $\Lambda$ 
hvar x N y with x  $\stackrel{?}{=}_a$  y

```

$\dots \mid \text{yes } _ = N$
 $\dots \mid \text{no } _ = \vee y$
 $-$
 $_ [_ := _] : \Lambda \rightarrow \text{Atom} \rightarrow \Lambda \rightarrow \Lambda$
 $M [a := N] = \text{Alt } \Lambda (\text{hvar } a N) _ \cdot _ (a :: \text{fv } N, \lambda) M$

Again because of the strong α compability of the iteration principle we obtain the following result for free:

$\text{lemmaSubst1} : \{M N : \Lambda\} (P : \Lambda) (a : \text{Atom})$
 $\rightarrow M \sim_{\alpha} N \rightarrow M [a := P] \equiv N [a := P]$

Using the induction principle in figure 3 we prove:

$\text{lemmaSubst2} : \forall \{N\} \{P\} M x$
 $\rightarrow N \sim_{\alpha} P \rightarrow M [x := N] \sim_{\alpha} M [x := P]$

Finally, from the two previous result we directly obtain next substitution lemma.

$\text{lemmaSubst} : \{M N P Q : \Lambda\} (a : \text{Atom})$
 $\rightarrow M \sim_{\alpha} N \rightarrow P \sim_{\alpha} Q$
 $\rightarrow M [a := P] \sim_{\alpha} N [a := Q]$
 $\text{lemmaSubst } \{M\} \{N\} \{P\} \{Q\} a M \sim N P \sim Q$
 $= \text{begin}$
 $\quad M [a := P]$
 $\quad \approx \langle \text{lemmaSubst1 } P a M \sim N \rangle$
 $\quad N [a := P]$
 $\quad \sim \langle \text{lemmaSubst2 } N a P \sim Q \rangle$
 $\quad N [a := Q]$
 \square

Remarkably all this result are directly derived from the first primitive induction principle, and no need of induction on the length of terms or accesible predicates were needed in all of this formalization.