Principles of Recursion and Induction for Nominal Lambda Calculus.

April 9, 2015

All the shown code 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 Λ : Set where
  v      : Atom → Λ
  _·_    : Λ → Λ → Λ
  ƛ      : Atom → Λ → Λ

data _#_ (a : Atom) : Λ → Set where
  #v    : {b : Atom}             → b ≢ a           → a # v b
  #·    : {M N : Λ }             → a # M → a # N   → a # M · N
  #ƛ≡   : {M : Λ}                                  → a # ƛ a M
  #ƛ    : {b : Atom}{M : Λ}  → a # M               → a # ƛ b M

(_•_)ₐ_ : Atom → Atom → Atom → Atom
( a • b )ₐ c  with c ≟ₐ a
... | yes _              = b
... | no  _  with c ≟ₐ b
...          | yes _   = a
...          | no  _   = c

(_•_)_ : Atom → Atom → Λ → Λ
( a • b ) v c      = v (( a • b )ₐ c)
( a • b ) M · N  = (( a • b ) M) ·  (( a • b ) N)
( a • b ) ƛ c M  = ƛ (( a • b )ₐ c) (( a • b ) M)

_•ₐ_ : Π → Atom → Atom
π •ₐ a = foldr (λ s b → ( proj₁ s • proj₂ s )ₐ b) a π
```

```
_•_ : Π → Λ → Λ
π • M = foldr (λ s M → ( proj₁ s • proj₂ s ) M) M π

data _∼α_ : Λ → Λ → Set where
  ∼αv  : {a : Atom} → v a ∼α v a
  ∼α·  : {M M' N N' : Λ} → M ∼α M' → N ∼α N'
       → M · N ∼α M' · N'
  ∼αƛ  : {M N : Λ}{a b : Atom}(xs : List Atom)
       → ((c : Atom) → c ∉ xs → ( a • c ) M ∼α ( b • c ) N)
       → ƛ a M ∼α ƛ b N
```

2 Induction Principles

Primitive induction over Λ terms.

```
TermPrimInd : {l : Level}(P : Λ → Set l)
  → (∀ a → P (v a))
  → (∀ M N → P M → P N → P (M · N))
  → (∀ M b → P M → P (ƛ b M))
  → ∀ M → P M
```

Figure 1: Primitive Induction Principle

Next term induction principle provides a strong inductive hypothesis for the lambda abstraction case, which gives the property for all permutation of names of the body of the abstraction. This induction principle is proved using the previous introduced principle.

```
TermIndPerm : {l : Level}(P : Λ → Set l)
  → (∀ a → P (v a))
  → (∀ M N → P M → P N →  P (M · N))
  → (∀ M b → (∀ π → P (π • M)) → P (ƛ b M))
  → ∀ M → P M
```

Figure 2: Permutation Induction Principle

A predicate is α-compatible if it holds for a given term it also holds for all α-equivalent terms.

```
αCompatiblePred : {l : Level} → (Λ → Set l) → Set l
αCompatiblePred P = {M N : Λ} → M ∼α N → P M → P N
```

If a predicate is α-compatible then we can prove the following induction principle using previously introduced one. This new principle enables us to choose the variable of the abstraction case different from a given finite list of variables. In this way this principle allow us to emulate Barendregt Variable

Convention (BVC), and assume enough fresh variables in a proof, that is, doing proofs over α-equivalence classes of terms. Our aim is use this principle whenever is possible, previous and next principles are usefull to internaly deal with swap operation, but we want to hide this operation from our proofs as much as possible.

```
TermαPrimInd : {l : Level}(P : Λ → Set l) → αCompatiblePred P
  → (∀ a → P (v a))
  → (∀ M N → P M → P N → P (M · N))
  → ∃ (λ vs → (∀ M b → b ∉ vs → P M → P (ƛ b M)))
  → ∀ M → P M
```

Again assuming an α-compatible predicate, and using the second induction principle (figure 2), we can prove the following induction principle which combines the two previous principles characteristics.

```
TermαIndPerm : {l : Level}(P : Λ → Set l) → αCompatiblePred P
  → (∀ a → P (v a))
  → (∀ M N → P M → P N →  P (M · N))
  → ∃ (λ as → (∀ M b → b ∉ as → (∀ π →  P (π • M)) → P (ƛ b M)))
  → ∀ M → P M
```

Figure 3: Permutation α-Induction Principle

3 Iteration and Recursion Principles

We want to define strong α-compatible functions, that is, functions over the α-equivalence class of terms. So this functions should not depend on the abstraction variables of a term and return the same result between α-equivalent terms.

```
strong∼αCompatible : {l : Level}{A : Set l} → (Λ → A) → Λ → Set l
strong∼αCompatible f M = ∀ N → M ∼α N → f M ≡ f N
```

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, hidding the specific abstraction variables of the inspected 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 3.

```
ΛIt  : {l : Level}(A : Set l)
→ (Atom → A)
→ (A → A → A)
→ List Atom × (Atom → A → A)
→ Λ → A
```

The following lemma says our iteration principle always return strong compatibility functions. This result is proved using the induction principle in figure 3.

```
lemmaΛItStrongαCompatible : {l : Level}(A : Set l)
   → (hv : Atom → A)
   → (h· : A → A → A)
   → (vs : List Atom)
   → (hƛ : Atom → A → A )
   → (M : Λ) → strong∼αCompatible (ΛIt A hv h· (vs , hƛ)) M
```

Figure 4: Strong α Compatibility of the Iteration Principle

From this iteration principle we directly derive the next recursion principle over terms, which also generates strong α-compatible functions.

```
ΛRec  : {l : Level}(A : Set l)
      → (Atom → A)
      → (A → A → Λ → Λ → A)
      → List Atom × (Atom → A → Λ → A)
      → Λ → A
```

4 Iterator Application

We present several applications of the iteration/recursive principle defined in previous section. In the following two sub-sections we implement two classic examples of ƛ − $c$alculus theory. While in the appendix A we successfully apply our iteration/recursion principle to all the examples presented in [1]. This work introduces a sequence of increasing complexity functions definitions to provide a test for any principle of function definition. Each of the defined functions respects the α-equivalence relation, that is, are strong compatible functions by being implemented over the previously introduced itaration principle.

4.1 Free Variables

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

```
fv : Λ → List Atom
fv = ΛIt (List Atom) [_] _++_ ([] , λ 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 _*_ holds when a variables ocurrs free in a term.

```
data _*_ : Atom → Λ → Set where
   *v  : {x : Atom}                          → x * v x
```

```
*·l  :  {x : Atom}{M N : Λ}  → x * M          → x * (M · N)
*·r  :  {x : Atom}{M N : Λ}  → x * N          → x * (M · N)
*ƛ   :  {x y : Atom}{M : Λ}  → x * M → y ≢ x  → x * (ƛ y M)
```

We can use the last induction principle (fig. 3) to prove the following proposition:

```
Pfv* : Atom → Λ → Set
Pfv* a M = a ∈ fv M → a * M
```

In the lambda abstraction obligation proof of the induction principle used, we can exclude the variable a from the abstraction variables we need to prove, simplifying in this way the required proof. We have to prove that $\forall b \not\equiv a, a \in$ fv (ƛ $b\,M$) $\Rightarrow a$ * ƛ $b\,M$, knowing by inductive hypothesis that $\forall \pi, a \in$ fv $(\pi \bullet M) \Rightarrow a$ * $(\pi \bullet M)$. So $a \in$ fv (ƛ $b\,M$) and $b \not\equiv a$ then we can derive that $a \in$ fv M holds. Now, instantiting the inductive hypothesis with an empty permtutation and the previous result, we have that a * M, using again that $b \not\equiv a$, we can then conclude the desired result: a * ƛ $b\,M$.

This flexibility comes with the extra cost that we need to prove that the predicate $\forall a,$ Pfv*a is α-compatible in order to use this induction principle. This α-compatible proof is direct because * is an α-compatible relation and the fv function is strong α-compatible. So the extra cost is just the proof that * is α-compatible.

Another approach where the last proof is automatically obtained is to define the free relation using our iteration principle and not a data type as previously done.

```
_free_ : Atom → Λ → Set
(_free_) a = ΛIt Set (λ b → a ≡ b) _∨_ ([ a ] , const id)
```

For the variable case we return the type of the propositional equality, inhabited only if the searched variable is equal to the term variable. The application case is the disjoint union of the types returned by the recursive calls, that is, the union of the variable free ocurrence evidence in the applied terms. Finally, in the abstraction case we can choose the abstraction variable to be different from the searchead one, so we can ignore the abstraction variable, and return just the recursive call, containing the evidence of any variable free ocurrence in the abstraction body.

This free predicete impementation is strong compatible by construction because we build it from our iterator principle, so given any variable a and two α-compatible terms M, N, the returned set should be the same. So is direct that if the predicate holds for M (which means that the returned set is inhabited for M), then the predicate should also hold for N.

From this point we can do an analog proof of Pfv* proposition, but now using this new free predicate definition which is α-compatible by construction. This give us a framework where we can define strong compatible functions and also α-compatible predicates over terms, and then prove properties about theses functions and predicates using our induction principle that provides us with the BVC.

4.2 Substitution

We implement the no capture substitution operation, we avoid any variable capture giving as variables to not to choose from as variable abtractions: the substituted variable and the free variables of the replaced term.

```
hvar : Atom → Λ → Atom → Λ
hvar x N y with x ≟ₐ y
... | yes _ = N
... | no  _ = v y
-
_[_≔_] : Λ → Atom → Λ → Λ
M [ a ≔ N ] = ΛIt Λ (hvar a N) _·_ (a ∷ fv N , ƛ) M
```

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

```
lemmaSubst1 : {M N : Λ}(P : Λ)(a : Atom)
  → M ∼α N → M [ a ≔ P ] ≡ N [ a ≔ P ]
```

Using the induction principle in figure 2 we prove:

```
lemmaSubst2 : ∀ {N} {P} M x
  → N ∼α P → M [ x ≔ N ] ∼α M [ x ≔ P ]
```

From the two previous result we directly obtain next α-compatibility substitution lemma .

```
lemmaSubst : {M N P Q : Λ}(a : Atom)
  → M ∼α N → P ∼α Q
  → M [ a ≔ P ] ∼α N [ a ≔ Q ]
lemmaSubst {M} {N} {P} {Q} a M∼N P∼Q
  = begin
      M [ a ≔ P ]
    ≈⟨ lemmaSubst1 P a M∼N ⟩
      N [ a ≔ P ]
    ∼⟨ lemmaSubst2 N a P∼Q  ⟩
      N [ a ≔ Q ]
    □
```

With previous result we can derive that our substitution operation is α-equivalent with a naive one for fresh enough abstraction variables.

```
lemmaƛ∼[] : ∀ {a b P} M → b ∉ a ∷ fv P
  → ƛ b M [ a ≔ P ] ∼α  ƛ b (M [ a ≔ P ])
```

We can combine this last result with the TermαPrimInd principle which emulates BVC convention, and mimic in this way a pen and pencil inductive proofs about substitution operation. As an example we show next substitution composition predicate:

```
PSC : ∀ {x y L} N → Λ → Set
PSC {x} {y} {L} N M = x ≢ y → x ∉ fv L
  → (M [ x ≔ N ]) [ y ≔ L ] ∼α (M [ y ≔ L ])[ x ≔ N [ y ≔ L ] ]
```

Next we give a direct equational proof that PSC predicate is α-compatible:

```
αCompatiblePSC : ∀ {x y L} N → αCompatiblePred (PSC {x} {y} {L} N)
αCompatiblePSC {x} {y} {L} N {M} {P} M∼P PM x≢y x∉fvL
  = begin
        (P [ x ≔ N ]) [ y ≔ L ]
      - Strong α compability of inner substitution operation
      ≈⟨ cong (λ z → z [ y ≔ L ]) (lemmaSubst1 N x (σ M∼P)) ⟩
        (M [ x ≔ N ]) [ y ≔ L ]
      - We apply that we know the predicate holds for M
      ∼⟨ PM x≢y x∉fvL ⟩
        (M [ y ≔ L ]) [ x ≔ N [ y ≔ L ] ]
      - Strong α compability of inner substitution operation
      ≈⟨ cong (λ z → z [ x ≔ N [ y ≔ L ] ]) (lemmaSubst1 L y (M∼P))  ⟩
        (P [ y ≔ L ]) [ x ≔ N [ y ≔ L ] ]
      □
```

For the interesting abstraction case of the α-structural induction over the lambda term, we asume the abstraction variables in the term are not in the substituted variables nor the subsituted terms. In this way the substitution operations are α-compatible to naive substitutions, then the inductive hipothesis allow us to complete the the inductive proof in a direct maner. Next we show the code fragment correspondint to this proof:

```
begin
  (ƛ b M [ x ≔ N ])   [ y ≔ L ]
- Inner substitution is α equivalent
- to a naive one because b ∉ x :: fv N
≈⟨ lemmaSubst1 L y (lemmaƛ∼[] M b∉x::fvN)  ⟩
  (ƛ b (M [ x ≔ N ])) [ y ≔ L ]
- Outer substitution is α equivalent
- to a naive one because b ∉ y :: fv L
∼⟨ lemmaƛ∼[] (M [ x ≔ N ]) b∉y::fvL ⟩
  ƛ b ((M [ x ≔ N ])  [ y ≔ L ])
- We can now apply our inductive hypothesis
∼⟨ lemma∼αƛ (IndHip x≢y x∉fvL) ⟩
  ƛ b ((M [ y ≔ L ])  [ x ≔ N [ y ≔ L ] ])
- Outer substitution is α equivalent
- to a naive one because b ∉ x :: fv N [y ≔ L]
∼⟨ σ (lemmaƛ∼[] (M [ y ≔ L ]) b∉x::fvN[y:=L]) ⟩
```

```
   (ƛ b (M [ y ≔ L ])) [ x ≔ N [ y ≔ L ] ]
- Inner substitution is α equivalent
- to a naive one because b ∉ y ∷ fv L
≈⟨ sym (lemmaSubst1 (N [ y ≔ L ])  x (lemmaƛ∼[] M b∉y::fvL))  ⟩
   (ƛ b M [ y ≔ L ])    [ x ≔ N [ y ≔ L ] ]
□
```

Remarkably theses results 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.

A Iteration/Recursion Applications

In the following sections we successfully apply our iteration/recursion principle to all the examples from [1]. This work presents a sequence of increasing complexity functions definitions to provide a test for any principle of function definition, where each of the given functions respects the α-equivalence relation.

A.1 Case Analysis and Examining Constructor Arguments

The following family of functions distinguishes between constructors returning the constructor components, giving in a sense a kind of *pattern-matching*.

$$\begin{array}{lll}
isVar & :\Lambda\rightarrow & Maybe\ (Variable)\\
isVar & (v\ x) & = Just\\
isVar & (M\cdot N) & = Nothing\\
isVar & (\lambda x M) & = Nothing
\end{array}
\qquad
\begin{array}{lll}
isApp & :\Lambda\rightarrow & Maybe\ (\Lambda\times\Lambda)\\
isApp & (v\ x) & = Nothing\\
isApp & (M\cdot N) & = Just(M,N)\\
isApp & (\lambda x M) & = Nothing
\end{array}$$

$$\begin{array}{lll}
isAbs & :\Lambda\rightarrow & Maybe\ (Variable\times\Lambda)\\
isAbs & (v\ x) & = Nothing\\
isAbs & (M\cdot N) & = Nothing\\
isAbs & (\lambda x M) & = Just(x,M)
\end{array}$$

Next we present the corresponding codifications using our iteration/recursion principle:

```
isVar : Λ → Maybe Atom
isVar = ΛIt  (Maybe Atom)
             just
             (λ _ _ → nothing)
             ([] , λ _ _ → nothing)
-
isApp : Λ → Maybe (Λ × Λ)
isApp = ΛRec  (Maybe (Λ × Λ))
              (λ _ → nothing)
              (λ _ _  M N → just (M , N))
              ([] , λ _ _ _ → nothing)
```

```
-
isAbs : Λ → Maybe (Atom × Λ)
isAbs = ΛRec  (Maybe (Atom × Λ))
              (λ _ → nothing) (λ _ _ _ _ → nothing)
              ([] , λ a _ M → just (a , M))
```

A.2 Simple recursion

The size function returns a numeric measurement of the size of a term.

$$\begin{array}{lll} size & :\Lambda\to & \mathbb{N} \\ size & (v\ x) & =1 \\ size & (M\cdot N) & =size(M)+size(N)+1 \\ size & (\lambda x M) & =size(M)+1 \end{array}$$

```
size : Λ → ℕ
size = ΛIt ℕ (const 1) (λ n m → suc n + m) ( [] , λ _ n → suc n)
```

A.3 Alfa Equality

Next functions decides the α-equality relation between two terms.

```
equal : Λ → Λ → Bool
equal = ΛIt (Λ → Bool) vareq appeq ([] , abseq)
  where
  vareq : Atom → Λ → Bool
  vareq a M with isVar M
  ... | nothing  = false
  ... | just b   = ⌊ a ≟ₐ b ⌋
  appeq : (Λ → Bool) → (Λ → Bool) → Λ → Bool
  appeq fM fN P with isApp P
  ... | nothing          = false
  ... | just (M' , N')   = fM M' ∧ fN N'
  abseq : Atom → (Λ → Bool) → Λ → Bool
  abseq a fM N with isAbs N
  ... | nothing = false
  ... | just (b , P) = ⌊ a ≟ₐ b ⌋ ∧ fM P
```

Observe that isAbs function also normalises N, so it is correct in last line to ask if the two binders are equal.

A.4 Recursion Mentioning a Bound Variable

The enf function is true of a term if it is in η-normal form, the fv function returns the set of a term's free variables and was previously defined.

$$\begin{array}{lll}
enf & : \Lambda \to & Bool \\
enf & (v\ x) & = True \\
enf & (M \cdot N) & = enf(M) \wedge enf(N) + 1 \\
enf & (\lambda x M) & = enf(M) \wedge (\exists N, x / isApp(M) == Just(N, v\ x) \Rightarrow x \in fv(N))
\end{array}$$

```
_⇒_ : Bool → Bool → Bool
false  ⇒ b = true
true   ⇒ b = b
-
enf : Λ → Bool
enf = ΛRec Bool (const true) (λ b1 b2 _ _ → b1 ∧ b2) ([] , absenf)
  where
  absenf : Atom → Bool → Λ → Bool
  absenf a b M with isApp M
  ... | nothing = b
  ... | just (P , Q) = b ∧ (equal Q (v a) ⇒ a ∈b (fv P))
```

A.5 Recursion with an Additional Parameter

Given the ternary type of possible directions to follow when passing through a term (Lt, Rt, In), corresponding to the two sub-terms of an application constructor and the body of an abstraction, return the set of paths (lists of directions) to the occurrences of the given free variable in a term, where *cons* insert an element in front of a list.

$$\begin{array}{rlll}
 & vposns & : Variable \times \Lambda \to & List\ (List\ Direction) \\
 & vposns & (x, v\ y) & = if\ (x == y)\ then\ [[]]\ else\ [] \\
 & vposns & (x, M \cdot N) & = map\ (cons\ Lt)\ (vposns\ x\ M) +\!\!+ \\
 & & & map\ (cons\ Rt)\ (vposns\ x\ N) \\
x \neq y \Rightarrow & vposns & (x, \lambda y M) & = map\ (cons\ In)\ (vposns\ x\ M)
\end{array}$$

Notice how the condition guard of the abstraction case is translated to the list of variables from where not to chose from the abstraction variable.

```
data Direction : Set where
  Lt Rt In : Direction
-
vposns : Atom → Λ → List (List Direction)
vposns a = ΛIt (List (List Direction)) varvposns appvposns ([ a ] , absvposns)
  where
  varvposns : Atom → List (List Direction)
  varvposns b with a ≟a b
  ... | yes _ = [ [] ]
  ... | no  _ = []
  appvposns : List (List Direction) → List (List Direction)
```

```
    → List (List Direction)
appvposns l r = map (_∷_ Lt) l ++ map (_∷_ Rt) r
absvposns : Atom → List (List Direction) → List (List Direction)
absvposns a r = map (_∷_ In) r
```

A.6 Recursion with Varying Parameters and Terms as Range

A variant of the substitution function, which substitutes a term for a variable, but further adjusts the term being substituted by wrapping it in one application of the variable named "0" per traversed binder.

$$
\begin{array}{llll}
 & sub' & : \Lambda \times Variable \times \Lambda & \to \Lambda \\
 & sub' & (P, x, v\ y) & = if\ (x == y)\ then\ P\ else\ (v\ y) \\
 & sub' & (P, x, M \cdot N) & = (sub'(P, x, M)) \cdot (sub'(P, x, N)) \\
\left.\begin{array}{r} y \neq x \wedge \\ y \neq 0 \ \ \wedge \\ y \notin fv(P) \end{array}\right\} \Rightarrow & sub' & (P, x, \lambda y M) & = \lambda y(sub'((v\ 0) \cdot M, x, M))
\end{array}
$$

To implement this function with our iterator principle we must change the parameters order, so our iterator principle now returns a function that is waiting the term to be substituted. In this way we manage to vary the parameter through the iteration.

```
hvar : Atom → Atom → Λ → Λ

hvar x y with x ≟ₐ y
... | yes _ = id
... | no  _ = λ _ → (v y)
-

sub' : Atom → Λ → Λ → Λ
sub' x M P = ΛIt  (Λ → Λ)
                  (hvar x)
                  (λ f g N →  f N · g N)
                  (x ∷ 0 ∷ fv P , λ a f N → ƛ a (f ((v 0) · N)))
                  M P
```

References

[1] Michael Norrish. Recursive function definition for types with binders. In *In Seventeenth International Conference on Theorem Proving in Higher Order Logics*, pages 241–256, 2004.