Alpha-Structural Induction and Recursion for the Lambda Calculus in Constructive Type Theory

Ana Bove [1]

Chalmers University of Technology
Gothenburg, Sweden

Maribel Fernandez [2]

King's College London
London, England

Álvaro Tasistro [3] Nora Szasz [4] Ernesto Copello [5]

Universidad ORT Uruguay
Montevideo, Uruguay

Abstract

We formulate principles of induction and recursion for a variant of lambda calculus with bound names where α-conversion is based upon name swapping as in nominal abstract syntax. The principles allow to work modulo alpha-conversion and implement the Barendregt variable convention. We derive them all from the simple structural induction principle on concrete terms and work out applications to some fundamental meta-theoretical results, such as the substitution lemma for alpha-conversion and the lemma on substitution composition. The whole work is implemented in Agda.

Keywords: Formal Metatheory, Lambda Calculus, Constructive Type Theory

1 Introduction

We are interested in methods for formalising in constructive type theory the meta-theory of the lambda-calculus. The main reason for this is that the lambda calculus is both a primigenial programming language and a prime test bed for formal reasoning on tree structures that feature (name) binding.

[1] Email: bove@chalmers.se
[2] Email: Maribel.Fernandez@kcl.ac.uk
[3] Email: tasistro@ort.edu.uy
[4] Email: szasz@ort.edu.uy
[5] Email: copello@ort.edu.uy

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Specifically concerning the latter, the informal procedure consists to begin with in "identifying terms up to α-conversion". However, this is not simply carried out when functions are defined by recursion and properties proven by induction. The problem has to do with the fact that the consideration of the α-equivalence classes is actually conducted through the use of convenient representatives thereof. These are chosen by the so-called Barendregt Variable Convention (BVC): each term representing its α-class is chosen so that its bound names are all different and different from all names free in the current context. Now a general validity criterion determines that this procedure ought to be accompanied in all cases by the verification that the proofs and results of functions depend only on the α-class and do not vary with the particular choice of the representative in question. Such verification is seldom accomplished but yet it is not the main difficulty concerning the validity of the constructions so performed. The main point is that e.g. inductive proofs are often carried out employing the structural principle for concrete terms —and then it may well happen that an induction step corresponding to functional abstractions can be carried out for a conveniently chosen bound name but not for an arbitrary one as the principle requires.

The problem can be avoided by the use of de Bruijn's nameless syntax [**?**] or its more up-to-date version *locally nameless* syntax [**?**,**?**], which uses names for the free or global variables and the indices counting up to the binding abstractor for the occurrences of local parameters. But these methods are not without overhead in the form of several operations or well-formedness predicates. As a result, there certainly is a relief in not having to consider α-conversion; but, at the same time, the nameless syntax seriously affects the connection between actual terms and their manipulation on the one hand and the informal statements concerning them on the other, so that one can easily get things wrong. The same has to be said of the map representation introduced in [**?**].

A different alternative is to replace the (as explained above, problematic) use of structural induction and recursion principles on concrete terms by that of so-called *alpha*-structural principles working directly on the α-equivalence classes. This means providing principles that allow to prove properties by induction and defining functions by recursion by direct use of the BVC, somewhat easying the burden associated to the verification of the validity of the procedure.

A first attempt in this direction is [**?**], which gives an axiomatic description of lambda terms in which equality embodies α-conversion and that provides a method of definition of functions by recursion on such type of objects. This solution ultimately rests upon the use of higher-order abstract syntax within the HOL system, and a theoretical model using de Bruijn's nameless syntax is sketched to show the soundness of the system of axioms. In [**?**,**?**,**?**], models of syntax with binders are introduced which formulate the basic concepts of abstraction, α-equivalence and a name being "sufficiently fresh" in a mathematical object, on the basis of the simple operation of name swapping. This theory —which has become known as *nominal abstract syntax*— provides a framework of (first-order) languages with binding with associated principles of α-structural recursion and induction that are based on the verification of the non-dependence of the mathematical objects in the current context, as well as of the results of recursively defined functions, on the bound names

chosen for the representatives of the α-classes involved. Implementations of this approach have been tried in Isabelle/HOL [?] and Coq [?]. In the first case the solution rests upon a weak version of higher-order abstract syntax, whereas the second one is an axiomatisation in which —similarly to [?] cited above— equality is postulated as embodying α-conversion and a model of the system based on locally nameless syntax has been constructed.

Yet another approach to the formulation of the alpha-structural principles originates in the observation that, if the property to be tried is α-compatible —i.e., it is actually a property of the α-classes and not just of the concrete terms— then induction on the *size* of terms can be used to bridge over the possible gap pointed out above in proofs by induction that confine themselves to convenient choice of bound names. Indeed a step f

.

All the code referred to in this work is compiled in the last Agda's version 2.4.2.2 and 0.9 standard library, and can be accessed at:

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

1.1 Related Work

There exist several developments in the direction of our work, all of them based the Isabelle/HOL proof assistant. Gordon [?] constructs a similar BVC induction principle over a variation of de Bruijn syntax. The syntax used in Gordons's work was already suggested by de Bruijn [?], in which "free variables have names but the bound variables are nameless". In this representation α-convertible terms are syntactically equal, but invalid terms appears, so a well-formed predicate is needed to eliminate the incorrect terms. Because this last issue, every introduced function must be proved to be closed under well-formed terms. On the other hand, the main adventage of this mixed strategy is that theorems can be expressed in convetional form, without de Bruijn encoding, and in spite of this, the renaming of bound variables is still supported in proofs, because syntactical equality is up to α-conversion. He hide this explicit renaming from proofs introducing an induction principle for decidable predicates, which is proved by induction on the length of terms. As the BVC convention, this induction principle enable us to choose the abstraction variables fresh enough from the context in a similar way as we will do in this work. Although, as Gordon point outs, we belive name-carrying syntax up to literal equality would be needed to represent language definitions, such as that of standard ML, for instance, where syntax is not identified up to α-conversion. De Bruijn notation has been used to implement several theorem provers, where syntax is internally represented in De Bruijn but human interacting uses a name-carrying notation. But this is different to use also this internal notation at a logic level.

Previous approach is *first-order* in the sense that the variable-binding operations of the embedded syntax is distinct form variable-binding at the host proof assistant language. In [?], Gordon and Melham began to explore a *second-order* approach ...

1.2 A brief introduction to Agda

The Agda system [**?**] can be seen both as a programming language with dependent types and as an interactive proof assistant. It implements Martin-Löf's (intentional) type theory [**?**] and it extends this theory with a number of features that makes programming more convenient. In order to guarantee logical consistency, only functions that can syntactically be checked total can be defined in the system.

The syntax of Agda resembles that of Haskell and provides many standard programming constructor such as modules, datatypes and case-expressions, signatures and records, and let- and where-expressions. It also allows defining functions by pattern matching on one or several of the function's arguments, and supports a very flexible way of naming expressions and types, including the possibility of having mixfix names (the positions of the arguments in the name are indicated by the character _) and of using Unicode in the names.

Agda supports the possibility of abstracting over the result of an expression when defining a function. This abstraction is performed with the constructor with, which effectively adds another argument to the function to be defined that can then be matched in the usual way.

It is possible to omit terms that the type checker can figure out for itself by replacing them by _. Agda also allows the possibility of defining certain arguments as "implicit", which is done by using brackets in the declarations of the arguments. Implicit arguments do not need to be provided, so only arguments that the system can deduce on its own should be defined as implicit. To give an implicit argument explicit one should embrace the corresponding expression with brackets.

2 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
```

3 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
```

Fig. 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
```

Fig. 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.

$$\begin{aligned}&\mathsf{Term\alpha PrimInd} : \{l : \mathsf{Level}\}(P : \Lambda \to \mathsf{Set}\ l) \to \mathsf{\alpha CompatiblePred}\ P\\ &\quad \to (\forall\ a \to P\ (\mathsf{v}\ a))\\ &\quad \to (\forall\ M\ N \to P\ M \to P\ N \to P\ (M \cdot N))\\ &\quad \to \exists\ (\lambda\ vs \to (\forall\ M\ b \to b \notin vs \to P\ M \to P\ (\mathsf{\lambda}\ b\ M)))\\ &\quad \to \forall\ M \to P\ M\end{aligned}$$

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.

$$\begin{aligned}&\mathsf{Term\alpha IndPerm} : \{l : \mathsf{Level}\}(P : \Lambda \to \mathsf{Set}\ l) \to \mathsf{\alpha CompatiblePred}\ P\\ &\quad \to (\forall\ a \to P\ (\mathsf{v}\ a))\\ &\quad \to (\forall\ M\ N \to P\ M \to P\ N \to\ \ P\ (M \cdot N))\\ &\quad \to \exists\ (\lambda\ as \to (\forall\ M\ b \to b \notin as \to (\forall\ \pi \to\ \ P\ (\pi \bullet M)) \to P\ (\mathsf{\lambda}\ b\ M)))\\ &\quad \to \forall\ M \to P\ M\end{aligned}$$

Fig. 3. Permutation α-Induction Principle

4 The Choice Function

Our recursion principle will proceed by recursion on the structure of terms. The interesting case is of course that of the λ-abstractions, in which the principle ought to satisfy two requirements:

(i) To allow defining the function in question by assuming that the abstraction is $\mathsf{\lambda}\,\chi\,N$ where χ is a name not belonging to a given finite set of names. (This is Barendregt's convention —in practical cases the set of names to be avoided is determined by a certain predicate but it is always finite.)

(ii) To equate (i.e. not distinguish) $\sim_\alpha$-equivalent terms while at the same time avoiding a too coarse identification. (A principle yielding e.g. only constant functions would indeed identify $\sim_\alpha$-equivalent terms, but it would be devoid of any interest.)

It would then seem that we need to know the full definition of $\sim_\alpha$ in order to implement our recursor. But such is not the case, as what we need in fact is only to determine how to proceed in the case of λ-abstractions, whose behavior with respect to $\sim_\alpha$ is determined by the simple expedient of "changing the bound atom". So suppose the procedure is receiving an abstraction $\mathsf{\lambda}\,x\,M$. In order to implement Barendregt's convention, the finite set of names to be avoided has to be provided to

the method, which we may represent as a given list vs of atoms. Further, we notice that:

$$\chi \# \lambda\!\!\lambda x\, M \Rightarrow \lambda\!\!\lambda x\, M \sim_\alpha \lambda\!\!\lambda \chi\, (x \bullet \chi)\, M$$

So we obtain an implementation of the required procedure by choosing a name χ such that:

(i) $\chi \notin vs$

(ii) $\chi \# \lambda\!\!\lambda x\, M$

and then stating the value of the function which is being defined which corresponds to input $\lambda\!\!\lambda \chi\, (x \bullet \chi)\, M$. For this, the result of the function on $(x \bullet \chi)\, M$ can be (recursively) used. Therefore we are implementing a form of inductive/recursive reasoning on, so to speak, $\sim_\alpha$-equivalence classes: The value of the function that is being defined for any abstraction is determined as the value of a convenient representative of its $\sim_\alpha$-equivalence class. Actually, for the defined correspondence to be indeed a function, it is important that the name χ is chosen in a deterministic way. This can be easily achieved, for instance by determining χ as the *first* name satisfying the requirements above in a fixed enumeration of the type of names, which must of course exist. We implement such determinism coding the following function:

```
χ : List Atom → Λ → Atom
```

This functions has the desired previously enumerated properties. As a consequence of the imposed fresh property, the application of χ function to the same list of variables and $\sim_\alpha$-equivalent terms should return the same atom, because $\sim_\alpha$-equivalent terms have the same fresh atoms.

At this point our development seems to diverge from the nominal one. Our χ function is reduced to an auxiliary one with the following signature:

```
χ' : List V → V
```

which is not *finite supported*, while nominal theory requires functions to be finitely supported. A function $f : X \to Y$, where X, Y are nominal sets, is finitely supported if there exists a finite set of atoms A that for all atoms $a, a' \notin A$ and any $x \in X$ $(a \ \bullet a')(f((a\ a')x)) = f(x)$. Back to our choice χ', it can be seen that there no exists such fixed set of atoms such that for any list of atoms in the image, swapping some atoms in it, then applying our choice function, and then again applying the same swapping have no effect in the result.

5 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
```

Next result make explicit the iterator behaviour in the abstracion case.

```
ΛItƛ  : {l : Level}(A : Set l)
   → (hv : Atom → A)
   → (h· : A → A → A)
   → (vs : List Atom)
   → (hƛ : Atom → A → A)
   → ∀ a M
   → ΛIt A hv h· (vs , hƛ) (ƛ a M) ≡
     hƛ  (χ vs (ƛ a M))
         (ΛIt A hv h· (vs , hƛ) ([ a , (χ vs (ƛ a M))] • M))
```

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
```

Fig. 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
```

6 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 λ-calculus theory. In the appendix A we also apply our iteration/recursion principle

to the examples of functions over terms presented in [?]. This work presents a sequence of increasing complexity functions, providinging this way a set of functions to test recursion principles over λ-calculus terms. Each of the defined functions respects the α-equivalence relation, that is, are strong compatible functions by being implemented over the previously introduced itaration/recursion principles.

6.1 Free Variables

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

```
fv : Λ → List Atom
fv = Λlt (List Atom) [_] _++_ ([] , λ v r → r - v)
```

As a direct consequence of strong α-compatibility of the iteration principle we have 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 λ- abstraction case of the induction proof, we can exclude the variable a from the abstraction variables of the term where the induction is done, simplifying this proof. We have to prove that $\forall b \not\equiv a, a \in \mathsf{fv}\,(\text{ƛ}\,b\,M) \Rightarrow a * \text{ƛ}\,b\,M$, knowing by inductive hypothesis that $\forall \pi, a \in \mathsf{fv}\,(\pi \bullet M) \Rightarrow a * (\pi \bullet M)$. Using the lemma Λltƛ, about the behaviour of the iterator for the abstraction case, we know $\mathsf{fv}\,(\text{ƛ}\,b\,M) = \mathsf{fv}\,((b \bullet \chi)\ M) - \chi$ where $\chi = \chi\,[\,]\ \text{ƛ}\,b\,M$, so we can infer that $a \in \mathsf{fv}\,((b \bullet \chi)\ M)$ and $a \not\equiv \chi$. We can use the inductive hypothesis with $\pi = [(b, \chi)]$ and previous result to obtain that $a * ((b \bullet \chi) \bullet M)$, which using that $b \not\equiv a$ and $a \not\equiv \chi$ we can obtain that $a * M$. Finlally, we are able to apply the constructor ƛ* of the relation * to previous result and $b \not\equiv a$ to obtain the desired result.

The flexibility to exclude variable a comes with an extra cost, we need to prove that the predicate $\forall a$, Pfv*a is α-compatible in order to use the choosen induction principle. This α-compatible proof is direct once we prove that * is an α-compatible relation and the fv function is strong α-compatible. The last property is direct because we implemented fv with the iteration principle, so the extra cost is just the proof that * is α-compatible.

Another approach where the last proof can be automatically obtained, as we freely obtainded that fv is strong α-complatible, is to define the free relation using our iteration principle, and not a data type as previously done.

```
_free_ : Atom → Λ → Set
(_free_) a = Λlt 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 predicate 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.

6.2 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 ] = Λlt Λ (hvar a N) _·_ (a :: fv N , ϰ) 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 over α-equivalence classes of terms 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 [**?**]. 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*.

$$\begin{aligned} isVar &: \Lambda \to \quad Maybe\ (Variable) \\ isVar\ &(v\ x) \quad = Just \\ isVar\ &(M \cdot N) = Nothing \\ isVar\ &(\lambda x M) \quad = Nothing \end{aligned}$$

$$\begin{aligned} isApp &: \Lambda \to \quad Maybe\ (\Lambda \times \Lambda) \\ isApp\ &(v\ x) \quad = Nothing \\ isApp\ &(M \cdot N) = Just(M, N) \\ isApp\ &(\lambda x M) \quad = Nothing \end{aligned}$$

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

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.

$$
\begin{aligned}
size : \Lambda \rightarrow \quad & \mathbb{N} \\
size\ (v\ x) \quad & = 1 \\
size\ (M \cdot N) & = size(M) + size(N) + 1 \\
size\ (\lambda x M) & = size(M) + 1
\end{aligned}
$$

```
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.

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

```
_⇒_ : 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.

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

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 = Λlt (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.

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

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
```