Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 323 (2016) 109–124

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

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

Ernesto Copello [1] Álvaro Tasistro [2] Nora Szasz [3]

Universidad ORT Uruguay
Montevideo, Uruguay

Ana Bove [4]

Chalmers University of Technology
Gothenburg, Sweden

Maribel Fernández [5]

King's College London
London, England

Abstract

We formulate principles of induction and recursion for a variant of lambda calculus in its original syntax (i.e., with only one sort of names) where α-conversion is based upon name swapping as in nominal abstract syntax. The principles allow to work modulo α-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 α-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

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

http://dx.doi.org/10.1016/j.entcs.2016.06.008

both a primigenial programming language and a prime test bed for formal reasoning on tree structures that feature (name) binding.

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 assumed to have bound names 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 crucial 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 [4] or its more up-to-date version *locally nameless* syntax [2,3], 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 formal procedures and what could be considered the natural features of syntax. The same has to be said of the map representation introduced in [9].

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 to define functions by recursion by direct use of the BVC, so as to ease the burden associated to the verification of the validity of the procedure.

A first attempt in this direction is [6], 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 work 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 [5,12,13], 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 step functions used in recursive definitions, on the bound names chosen for the representatives of the α-classes involved. Implementations of this approach have been tried in Isabelle/HOL [15] and Coq [1]. 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 [6] 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 (complete) 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, suppose you need to prove $\mathcal{P}(\lambda x.M)$; now, if what you have is a step from $\mathcal{P}(M^*)$ to $\mathcal{P}(\lambda x^*.M^*)$ for a convenient renaming of the term, then you will be able to use your strong size-induction hypothesis on M^*, since this is still of a size lesser than that of $\mathcal{P}(\lambda x.M)$. Hence you will arrive at $\mathcal{P}(\lambda x^*.M^*)$ and from there to the desired $\mathcal{P}(\lambda x.M)$ because of the α-compatibility of $\mathcal{P}$. This motivates trying to provide a mechanism of this kind to formalise the use of the BVC, and that is what we attempt in this paper. The result is that we are able to provide principles of alpha-structural induction and recursion, implementing the BVC in constructive type theory, using just the ordinary first-order, name-carrying syntax and actually *without* using the strong induction on the size of the terms —i.e. we are able to derive the principles in question from just simple structural induction on concrete terms. To such effect we define α-equivalence by using the basic concepts of nominal abstract syntax, namely freshness and swapping of names. Equality remains the simple definitional one and we do not either perform any kind of quotient construction. The whole development is implemented in the Agda system [10].

The rest of the paper goes as follows: in section 2 we present the infrastructure just mentioned. Section 3 presents the principles, starting from the simple structural induction on terms and ending up with the recursion principle on α-classes. In section 4 we show several applications that bring about certain feeling for the usefulness of the method. Finally, section 5 compares with related work and points out conclusions and further work.

The present is actually a literate Agda document, where we hide some code for reasons of conciseness. The entire code is available at:

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

and has been compiled with the last Agda version 2.4.2.2 and 0.9 standard library.

2 Infrastructure

2.1 Agda

Agda implements Constructive Type Theory [8] (*type theory* for short). It is actually a functional programming language in which:

(i) Inductive types can be introduced as usual, i.e. by enumeration of their constructors, but they can be parameterised in objects of other types. Because of the latter it is said that type theory features *families* of types (indexed by a base type) or *dependent* types.

(ii) Functions on families of types respect the dependence on the base object, which is to say that they are generally of the form $(x : \alpha) \to \beta_x$ where β_x is the type parameterised on x of type α. Therefore the type of the output of a function depends on the *value* of the input.

(iii) Functions on inductive types are defined by *pattern-maching* equations.

(iv) Every function of the language must be terminating. The standard form of recursion that forces such condition is *structural* recursion and is, of course, syntactically checked.

(v) Because of the preceding feature, type theory can be interpreted as a constructive logic. Specifically, this is achieved by representing propositions as inductive types whose constructors are the introduction rules, i.e. methods of direct proof, of the propositions in question.

Therefore we can say in summary that sets of data, predicates and relations are defined inductively, i.e. by enumeration of their constructors.

2.2 Syntax

The set Λ of terms is as usual. It is built up from a denumerable set of names, which we shall call *atoms*, borrowing terminology from nominal abstract syntax.

```
data Λ : Set where
  v    : Atom → Λ
  _·_  : Λ → Λ → Λ
  ƛ    : Atom → Λ → Λ
```

The following is called the *freshness* relation. It holds when a variable does not occur free in a term. Parameters to a function written between curly brackets can be omitted when invoking the function.

```
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
```

Next comes the fundamental operation of *swapping* of atoms. A finite sequence

(composition) of atom swaps constitutes a (finite) atom *permutation* which is the renaming mechanism to be used on terms. The action of atom swaps is first defined on atoms themselves:

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

Here it extends to terms:

```
(_•_)_ : 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)
```

And the same goes for permutations, which are *lists* of swaps:

```
_•_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
```

We now introduce α-conversion, denoted by $\sim\alpha$. We use a syntax-directed definition that uses co-finite quantification in the case of the lambda abstractions:

```
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
```

The idea is that for proving two abstractions α-equivalent you should be able to prove the respective bodies α-equivalent when you rename the bound names to any name not free in both abstractions. The condition on the new name can be generalised to "any name not in a given list", yielding an equivalent relation. The latter condition is harder to prove, but more convenient to use when you assume $\sim\alpha$ to hold, which is more often the case in the forthcoming proofs.

3 Alpha-Structural Induction and Recursion Principles

We start with the simple structural induction over the concrete Λ terms:

The next induction principle provides a strong hypothesis for the lambda ab-

```
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. Concrete Structural Induction Principle

straction case: it namely allows to assume the property for all renamings (given by finite permutations of names) of the body of the abstraction:

```
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. Strong Permutation Induction Principle

Notice that the hypothesis provided for the case of abstractions is akin to the corresponding one of the principle of strong or complete induction on the size of terms, only that expressed in terms of name permutations. This principle can be derived from the former, i.e. from simple structural induction, in very much the same way as complete induction on natural numbers is derived from ordinary mathematical induction. That is to say, we can use structural induction to prove $(\forall \pi) P(\pi \bullet M)$ given the hypotheses of the new principle, from which $P\,M$ follows. For the interesting case of abstractions, we have to prove $(\forall \pi) P(\pi \bullet \lambda\, a\, M)$, which is equal to $(\forall \pi) P(\lambda\, (\pi \bullet_{\mathsf{A}} a)\, (\pi \bullet M))$. The hypothesis of the new principle give us in this case $(\forall M', b)((\forall \pi') P(\pi' \bullet M') \to P(\lambda\, b\, M'))$. Now, instantiating M' as $\pi \bullet M$ and b as $\pi \bullet_{\mathsf{A}} a$, we obtain the desired result if we know that $(\forall \pi') P(\pi' \bullet \pi \bullet M)$, which holds by induction hypothesis of the structural principle.

We call a predicate α-compatible if it is preserved by α-conversion:

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

For α-compatible predicates we can use the preceding principle to derive the following:

```
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
```

This new principle enables us to carry out the proof of the abstraction case by

choosing a bound name different from the names in a given list vs. It gives a way to emulate the Barendregt Variable Convention (BVC) since, indeed, the names to be avoided will always be finitely many; in using the principle we must provide a list that includes them. This same principle is provided in [1], only that we here give it a proof in terms of the ones previously introduced, instead of just postulating it. Our aim is to employ this principle whenever possible, thereby hiding the use of the swap operation which is confined to the previous principles exposed. The interesting case in the implementation of the principle is of course that of the functional abstraction. We must put ourselves in the position in which we are using the former strong principle and are given an abstraction $\lambda\, b\, M$ for which we have to prove P. We have to employ to this effect the clause of our new principle corresponding to the functional abstractions, which forces us to employ a name b^* out of the given list vs. Therefore we can aspire at proving P for a renaming of the original term, say $\lambda\, b^*\, M^*$. The required result will then follow from the α-compatibility of the predicate P provided $\lambda\, b^*\, M^* \sim_\alpha \lambda\, b\, M$. This imposes the condition that the name b^* be chosen fresh in the original term $\lambda\, b\, M$ —and that $M^* = (b^* \bullet b)\, M$. We know PM^* and therefore $P(\lambda\, b^*\, M^*)$ because we know P for any renaming of M, by the hypothesis of the strong principle from which we start.

A very important point in this implementation is that, given the list of names to be avoided, we can and do choose b^* deterministically for each class of α-equivalent terms. Indeed, if we determine b^* as e.g. the first name out of the given list that is fresh (i.e. not free) in the originally given term, then the result will be one and the same for every term of each α-class, since α-equivalent terms have the same free variables. Hence the representative of each α-class chosen by this method will be fixed for each list of names to be avoided, which constitutes a basis for using the method for defining *functions* on the α-classes. This will work by associating to (each term of) the class the result of the corresponding computation on the canonically chosen representative.

More precisely, let us say that a function $f : \Lambda \to A$ is *strongly* α-compatible iff $M \sim_\alpha N \Rightarrow f\, M = f\, N$. We can now define an iteration principle over raw terms which always produces strongly α-compatible functions. For the abstraction case, this principle also allows us to give a list of variables from where the abstractions variables are not to be chosen. This iteration principle is derived from the BVC induction principle (TermαPrimInd) in a direct manner, just using a trivial constant predicate equivalent to the type A. We exhibit the type and code of the iterator:

$$
\begin{aligned}
&\Lambda\mathsf{It} \;:\; \{l : \mathsf{Level}\}(A : \mathsf{Set}\ l)\\
&\to (\mathsf{Atom} \to A)\\
&\to (A \to A \to A)\\
&\to \mathsf{List}\ \mathsf{Atom} \times (\mathsf{Atom} \to A \to A)\\
&\to \Lambda \to A
\end{aligned}
$$

To repeat the idea, the iterator works as a function on α-classes because for each given abstraction, it will yield the result obtained by working on a canonically chosen representative that is determined by the list of names to be avoided and the

(free names of the) α-class in question.

Strong compatibility would not obtain if we tried directly to formulate a recursion instead of an iteration principle, but we can recover the more general form by the standard procedure of computing pairs one of whose components is a term. Thereby we arrive at 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 Applications in Meta-Theory

We present several applications of the iteration/recursion principle defined in the preceding 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 [11]. This work presents a sequence of increasing complexity functions, with the purpose of testing the applicability of recursion principles over λ-calculus terms.

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 have that α-equivalent terms have the same free variables.

The relation _*_ holds when a variable occurs 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 our BVC-like induction principle to prove the following proposition:

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

In the case of lambda abstractions we are able to simplify the proof by choosing the bound name different from a. This flexibility comes at a cost, i.e. we need to prove that the predicate Pfv*a is α-compatible in order to use the chosen induction principle. This α-compatibility 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. This in turn could be directly obtained if we defined the relation establishing that a variable a is free in a term as a recursive function, as follows:

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

For the variable case we return the propositional equality of the searched variable to the term variable. The application case is the disjoint union of the types returned by the recursive calls. Finally, in the abstraction case we can choose the abstraction variable to be different from the searched one. In this way we can ignore the abstraction variable and return just the recursive call containing the evidence of any free ocurrence of the searched variable in the abstraction body. This implementation is strong compatible by construction because we have built it from our iterator principle, so it is also immediate from this definition that α-equivalent terms have the same free variables.

4.2 Substitution

We implement capture avoiding substitution in the following way:

```
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 , ϰ) M
```

It shows to be quite close to the simple pencil-and-paper version assuming the BVC. Notice that we explicitly indicate that the bound name of the canonical representative to be chosen must be different from the replaced variable and not occur free in the substituted term. 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 ]
lemmaSubst1 {M} {N} P a
  = lemmaΛItStrongαCompatible
          Λ (hvar a P) _·_ (a :: fv P) ϰ M N
```

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 results we directly obtain the α-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 ]
    □
```

In turn, with the preceding result we can derive that our substitution operation is α-equivalent with a naïve one for fresh enough bound names:

```
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 pencil-and-paper inductive proofs over α-equivalence classes of terms about substitution operation. As an example we show next the substitution composition lemma:

```
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 ] ]
```

We first 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 among the replaced variables or free in the subsituted terms. In this way the substitution operations

become α-compatible to naïve substitutions, and the induction hypothesis allows us to complete the inductive proof in a direct maner. The code fragment becomes:

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

5 Conclusions

The main contribution of this work is a full implementation in Constructive Type Theory of principles of induction and recursion allowing to work on α-classes of terms of the lambda calculus. The crucial component seems to be what we called a BVC-like induction principle allowing to choose the bound name in the case of the abstractions so that it does not belong to a given list of names. This principle is, on the one hand, derived (for α-compatible predicates) from ordinary structural induction on concrete terms, thus avoiding any form of induction on the size of terms or other more complex forms of induction. And, on the other hand, it gives rise to principles of recursion that allow to define functions on α-classes, especifically, functions giving identical results for α-equivalent terms. We have also shown by way of a number of examples that the principles provide a flexible framework quite able to pleasantly mimic pencil-and-paper practice.

Our work departs from e.g. [13] in that we do fix the choice of representatives for

implementing the alpha-structural recursion thereby forcing this principle to yield identical results for α-equivalent terms. This might be a little too concrete but, on the other hand, it gives us the possibility of completing a simple full implementation on an existing system, as different from other works which base themselves on postulates or more sophisticated systems of syntax or methods of implementation.

We wish to continue exploring the capabilities of this method of formalisation by studying its application to the meta-theory of type systems. We also wish to deepen its comparison to the method based on Stoughton's substitutions [14], which we started to investigate in [7] and which we believe can give rise to formulations similar to the one exposed here.

References

[1] Brian Aydemir, Aaron Bohannon, and Stephanie Weirich. Nominal reasoning techniques in Coq. *Electron. Notes Theor. Comput. Sci.*, 174(5):69–77, June 2007.

[2] Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 3–15, New York, NY, USA, 2008. ACM.

[3] Arthur Charguéraud. The locally nameless representation. *J. Autom. Reasoning*, 49(3):363–408, 2012.

[4] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with applications to the church-rosser theorem. *Indagationes Mathematicae (Koninglijke Nederlandse Akademie van Wetenschappen)*, 34(5):381–392, 1972. http://www.win.tue.nl/automath/archive/pdf/aut029.pdf Electronic Edition.

[5] Murdoch J. Gabbay and Andrew M. Pitts. A New Approach to Abstract Syntax with Variable Binding. *Formal Aspects of Computing*, 13(3—5):341—363, July 2001.

[6] Andrew D. Gordon and Thomas F. Melham. Five axioms of alpha-conversion. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics, 9th International Conference, TPHOLs'96, Turku, Finland, August 26-30, 1996, Proceedings*, volume 1125 of *Lecture Notes in Computer Science*, pages 173–190. Springer, 1996.

[7] Álvaro Tasistro, Ernesto Copello, and Nora Szasz. Formalisation in constructive type theory of stoughton's substitution for the lambda calculus. *Electronic Notes in Theoretical Computer Science*, 312(0):215 – 230, 2015. Ninth Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2014).

[8] P. Martin-Löf and G. Sambin. *Intuitionistic type theory*. Studies in Proof Theory. Bibliopolis, 1984.

[9] Helmut Schwichtenberg Masahiko Sato, Randy Pollack and Takafumi Sakurai, Viewing lambda-terms through maps, 2013, Available from http://homepages.inf.ed.ac.uk/rpollack/export/Maps_SatoPollackSchwichtenbergSakurai.pdf

[10] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[11] 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.

[12] Andrew M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186(2):165 – 193, 2003. Theoretical Aspects of Computer Software (TACS 2001).

[13] Andrew M. Pitts. Alpha-structural recursion and induction. *J. ACM*, 53(3):459–506, May 2006.

[14] A. Stoughton. Substitution revisited. *Theor. Comput. Sci.*, 59:317–325, 1988.

[15] Christian Urban and Christine Tasson. Nominal techniques in isabelle/hol. In Robert Nieuwenhuis, editor, *Automated Deduction – CADE-20*, volume 3632 of *Lecture Notes in Computer Science*, pages 38–53. Springer Berlin Heidelberg, 2005.

A Iteration/Recursion Applications

In the following sections we successfully apply our iteration/recursion principle to all the examples from [11]. This work presents a sequence of functions whose definitions are increasing in complexity 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*.

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

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

Next we present the corresponding encodings into 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.

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

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

A.3 Alpha Equality

This function 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 the last line to ask if the two bound names are the same.

A.4 Recursion Mentioning a Bound Variable

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

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

```
_⇒_ : 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. Assume *cons* insert an element in front of a list.

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

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

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

To implement this function with our iterator principle we must change the order of the parameters, so our iterator principle now returns a function that is waiting for 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 = Λlt  (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
```