Nominal Compositional Z Property in Coq

José R. I. Soares and Flávio L. C. de Moura*

Departamento de Ciência da Computação
Universidade de Brasília, Brasília, Brazil
jose.soares@aluno.unb.br, flaviomoura@unb.br

Abstract. TBD

1 Introduction

This work is about the development of a framework for studying calculi with explicit substitutions in a nominal setting[11], *i.e.* an approach to abstract syntax where bound names and α-equivalence are invariant with respect to permuting names. It extends the previous work [8] and [15] as follows: we formalized the confluence proof of a calculus with explicit substitution via the compositional Z property following the steps of [18]. In our framework, variables are represented by atoms that are structureless entities with a decidable equality, where different names mean different atoms and different variables. Its syntax is close to the usual paper and pencil notation used in λ-calculus, whose grammar of terms is given by:

$$t ::= x \mid \lambda_x.t \mid t\ t \tag{1}$$

where x represents a variable which is taken from an enumerable set, $\lambda_x.t$ is an abstraction, and $t\ t$ is an application. The abstraction is the only binding operator: in the expression $\lambda_x.t$, x binds in t, called the scope of the abstraction. This means that all free occurrence of x in t is bound in $\lambda_x.t$. A variable that is not in the scope of an abstraction is free. A variable in a term is either bound or free, but note that a varible can occur both bound and free in a term, as in $(\lambda_y.y)\ y$. The main rule of the λ-calculus, named β-reduction, is given by:

$$(\lambda_x.t)\ u \to_\beta \{x := u\}t \tag{2}$$

where $\{x := u\}t$ represents the result of substituting all free occurrences of variable x in t with u in such a way that renaming of bound variable may be done in order to avoid the variable capture of free variables. The substitution $\{x := u\}t$ is called *metasubstitution*.

In a calculus with explicit substitution the grammar (1) is extended with a constructor that aims to simulate the metasubstitution.

---
* Corresponding author

$$t ::= x \mid \lambda_x.t \mid t\ t \mid [x := u]t \tag{3}$$

where $[x := u]t$ represents a term with an operator that will be evaluated with specific rules of a substitution calculus. The intended meaning of the explicit substitution is that it will simulate the metasubstitution. This formalization aims to be a generic framework applicable to any calculi with explicit substitutions using a named notation for variables.

The following inductive definition corresponds to the grammar (3), where the explicit substitution constructor, named `n_sub`, has a special notation. Accordingly, `n_sexp` denotes the set of nominal λ-expressions equipped with an explicit substitution operator, which, for simplicity, we will refer to as just *terms*.

```
Inductive n_sexp : Set :=
| n_var (x:atom)
| n_abs (x:atom) (t:n_sexp)
| n_app (t1:n_sexp) (t2:n_sexp)
| n_sub (t1:n_sexp) (x:atom) (t2:n_sexp).

Notation "[ x := u ] t" := (n_sub t x u) (at level 60).
```

The `Notation` statement allow us to write ($[x := u]\ t$) instead of (`n_sub` $t\ x\ u$), which is closer to paper and pencil notation, as well as to the syntax of the grammar (3).

The contributions of this work are as follows:

1.
2.

2 The Nominal Framework

As usual in the standard presentations of the λ-calculus, our formalization is done considering terms modulo α-equivalence. This means that terms that differ only by the names of bound variables are *equal*. Formally, the notion of α-equivalence is defined by the following rules of inference:

$$(aeq\_abs\_same)\ \frac{t_1 =_\alpha t_2}{\lambda_x.t_1 =_\alpha \lambda_x.t_2} \qquad \frac{x \neq y \qquad x \notin fv(t_2) \qquad t_1 =_\alpha (y\ x)t_2}{\lambda_x.t_1 =_\alpha \lambda_y.t_2}\ (aeq\_abs\_diff)$$

$$(aeq\_var)\ \frac{}{x =_\alpha x} \qquad \frac{t_1 =_\alpha t_1' \qquad t_2 =_\alpha t_2'}{t_1\ t_2 =_\alpha t_1'\ t_2'}\ (aeq\_app) \qquad \frac{t_1 =_\alpha t_1' \qquad t_2 =_\alpha t_2'}{[x := t_2]t_1 =_\alpha [x := t_2']t_1'}\ (aeq\_sub\_same)$$

$$\frac{t_2 =_\alpha t_2' \qquad x \neq y \qquad x \notin fv(t_1') \qquad t_1 =_\alpha (y\ x)t_1'}{[x := t_2]t_1 =_\alpha [y := t_2']t_1'}\ (aeq\_sub\_diff)$$

where $fv(t)$ denotes the set of free variables of t, and $(x\ y)t$ is defined as follows:

$$(x\ y)t := \begin{cases} (x\ y)z, & \text{if } t = z; \\ \lambda_{(x\ y)z}.(x\ y)t_1, & \text{if } t = \lambda_z.t_1; \\ (x\ y)t_1\ (x\ y)t_2, & \text{if } t = t_1\ t_2 \\ [(x\ y)z := (x\ y)t_2]((x\ y)t_1)\ , & \text{if } t = [z := t_2]t_1 \end{cases} \tag{4}$$

and $(x\ y)z := \begin{cases} y, & \text{if } z = x; \\ x, & \text{if } z = y; \\ z, & \text{otherwise.} \end{cases}$

The corresponding Coq code for α-equivalence is given by the inductive definition `aeq` below. Note that each rule corresponds to a constructor with its corresponding name.

```
Inductive aeq : Rel n_sexp :=
| aeq_var : ∀ x, aeq (n_var x) (n_var x)
| aeq_abs_same : ∀ x t1 t2, aeq t1 t2 → aeq (n_abs x t1)(n_abs x t2)
| aeq_abs_diff : ∀ x y t1 t2, x ≠ y → x `notin` fv_nom t2 → aeq t1 (swap y x t2) →
  aeq (n_abs x t1) (n_abs y t2)
| aeq_app : ∀ t1 t2 t1' t2', aeq t1 t1' → aeq t2 t2' → aeq (n_app t1 t2) (n_app t1' t2')
| aeq_sub_same : ∀ t1 t2 t1' t2' x, aeq t1 t1' → aeq t2 t2' → aeq ([x := t2] t1) ([x := t2'] t1')
| aeq_sub_diff : ∀ t1 t2 t1' t2' x y, aeq t2 t2' → x ≠ y → x `notin` fv_nom t1' → aeq t1 (swap y x t1') →
aeq ([x := t2] t1) ([y := t2'] t1').

Notation "t =a u" := (aeq t u) (at level 60).
```

The key point of the nominal approach is that the swap operation is stable under α-equivalence in the sense that, $t_1 =_\alpha t_2$ if, and only if $(x\ y)t_1 =_\alpha (x\ y)t_2, \forall t_1, t_2, x, y$, while the metasubstitution is not. The following corollary stablishes this result in Coq:

```
Corollary aeq_swap: ∀ t1 t2 x y, t1 =a t2 ↔ (swap x y t1) =a (swap x y t2).
```

In order to see that metasubstitution is not stable under α-equivalence, note that if $x \neq y$ then we have that $\{x := y\}x =_\alpha \{x := y\}y$ but $x \neq_\alpha y$.

As presented in introduction, the main operation of the λ-calculus is the β-reduction (2) that expresses how to evaluate a function applied to an argument. The β-contractum $\{x := u\}t$ represents a capture free in the sense that no free variable becomes bound by the application of the metasubstitution. This operation is in the meta level because it is outside the grammar of the λ-calculus. In textbooks [2], the metasubstition is usually defined as follows:

$$\{x := u\}t = \begin{cases} u, & \text{if } t = x; \\ y, & \text{if } t = y \text{ and } x \neq y; \\ \{x := u\}t_1\ \{x := u\}t_2, & \text{if } t = t_1\ t_2; \\ \lambda_y.(\{x := u\}t_1), & \text{if } t = \lambda_y.t_1. \end{cases}$$

where it is assumed the so called *Barendregt's variable convention*:

> If $t_1, t_2, \ldots, t_n$ occur in a certain mathematical context (e.g. definition, proof), then in these terms all bound variables are chosen to be different from the free variables.

This means that we are assumming that both $x \neq y$ and $y \notin fv(u)$ in the case $t = \lambda_y.t_1$. This approach is very convenient in informal proofs because it avoids having to rename bound variables. In order to formalize the capture free substitution, *i.e.* the metasubstitution, there are different possible approaches. In our case, we rename bound variables whenever the metasubstitution is propagated inside a binder, *i.e.* inside an abstraction or an explicit substitution.

In a formal framework, like a proof assistant, the implementation of Barendregt's variable is not trivial[23]. In our approach, we rename bound variables whenever the metasubstitution needs to be propagated inside an abstraction or an explicit substitution:

$$\{x := u\}t = \begin{cases} u, & \text{if } t = x; \\ y, & \text{if } t = y \ (x \neq y); \\ \{x := u\}t_1 \ \{x := u\}t_2, & \text{if } t = t_1 \ t_2; \\ \lambda_x.t_1, & \text{if } t = \lambda_x.t_1; \\ \lambda_z.(\{x := u\}((y \ z)t_1)), & \text{if } t = \lambda_y.t_1, x \neq y \text{ and } z \notin fv(t) \cup fv(u) \cup \{x\}; \\ [x := \{x := u\}t_2]t_1, & \text{if } t = [x := t_2]t_1; \\ [z := \{x := u\}t_2]\{x := u\}((y \ z)t_1), & \text{if } t = [y := t_2]t_1, x \neq y \text{ and } z \notin fv(t) \cup fv(u) \cup \{x\}. \end{cases} \tag{5}$$

and the corresponding Coq code is as follows:

```
Function subst_rec_fun (t:n_sexp) (u :n_sexp) (x:atom) {measure size t} : n_sexp :=
  match t with
  | n_var y ⇒ if (x == y) then u else t
  | n_abs y t1 ⇒ if (x == y) then t else let (z,_) :=
     atom_fresh (fv_nom u `union` fv_nom t `union` {{x}}) in n_abs z (subst_rec_fun (swap y z t1) u x)
  | n_app t1 t2 ⇒ n_app (subst_rec_fun t1 u x) (subst_rec_fun t2 u x)
  | n_sub t1 y t2 ⇒ if (x == y) then n_sub t1 y (subst_rec_fun t2 u x) else let (z,_) :=
     atom_fresh (fv_nom u `union` fv_nom t `union` {{x}}) in
     n_sub (subst_rec_fun (swap y z t1) u x) z (subst_rec_fun t2 u x) end.

Definition m_subst (u : n_sexp) (x:atom) (t:n_sexp) := subst_rec_fun t u x.
Notation "{ x := u } t" := (m_subst u x t) (at level 60).
```

We list several important properties of the metasubstitution include its compatibility with α-equivalence when the variables of the metasubstitutions are the same.

```
Lemma aeq_m_subst_in: ∀ t u u' x, u =a u' → ({x := u}t) =a ({x := u'}t).
Lemma aeq_m_subst_out: ∀ t t' u x, t =a t' → ({x := u}t) =a ({x := u}t').
```

Also the propagation of the metasubstitution inside an abstraction with an adequate renaming to avoid capture of variables:

```
Lemma m_subst_abs_neq: ∀ t u x y z, x ≠ y → z `notin` fv_nom u `union` fv_nom (n_abs y t) `union` {{x}}
→ {x := u}(n_abs y t) =a n_abs z ({x := u}(swap y z t)).
```

The next corollary stablishes the compatibility of the metasubstitution operation with α-equivalence when the variables of the metasubstitutions are different:

```
Corollary aeq_m_subst_neq: ∀ t1 t1' t2 t2' x y, t2 =a t2' → x ≠ y → x `notin` fv_nom t1' → t1 =a (swap
x y t1') → ({x := t2}t1) =a ({y := t2'}t1').
```

and the Substitution Lemma whose formalization was the main achievement of [15] together with several lemmas that formed the infrastructure concerning α-equivalence and metasubstitution.

```
Lemma m_subst_lemma: ∀ t1 t2 t3 x y, x ≠ y → x `notin` (fv_nom t3) →
            ({y := t3}({x := t2}t1)) =a ({x := ({y := t3}t2)}({y := t3}t1)).
```

The reflexive-transitive closure of a binary relation R is defined as

$$\frac{}{t \twoheadrightarrow t}\ (\mathit{refl}) \qquad\qquad \frac{t_1 \to t_2 \qquad t_2 \twoheadrightarrow t_3}{t_1 \twoheadrightarrow t_3}\ (\mathit{rtrans})$$

Since we are working modulo α-equivalence, an application of the axiom (refl) must account for the fact that a term reduces to itself in zero steps, as well as to any other term within its α-equivalence class. To address this, we define the reflexive-transitive closure of a binary relation modulo α-equivalence as follows:

$$\frac{t_1 =_\alpha t_2}{t_1 \twoheadrightarrow t_2}\ (\mathit{refl}) \qquad \frac{t_1 \to t_2 \qquad t_2 \twoheadrightarrow t_3}{t_1 \twoheadrightarrow t_3}\ (\mathit{rtrans}) \qquad \frac{t_1 =_\alpha t_2 \qquad t_2 \twoheadrightarrow t_3}{t_1 \twoheadrightarrow t_3}\ (\mathit{rtrans\_aeq})$$

and the corresponding Coq definition is as follows:

```
Inductive refltrans (R: Rel n_sexp) : Rel n_sexp :=
| refl: ∀ t1 t2, t1 =a t2 → refltrans R t1 t2
| rtrans: ∀ t1 t2 t3, R t1 t2 → refltrans R t2 t3 → refltrans R t1 t3
| rtrans_aeq: ∀ t1 t2 t3, t1 =a t2 → refltrans R t2 t3 → refltrans R t1 t3.
```

In the next section, we present the λ_x-calculus and its confluence proof. To do

3 The λ_x calculus with explicit substitutions

The λ_x calculus[4,16,21] is the simplest extension of the λ-calculus with explicit substitutions. In this section, we will present its confluence proof via the compositional Z property[18].

Calculi with explicit substitutions are formalisms that deconstruct the metasubstitution operation into finer-grained steps, thereby functioning as an intermediary between the λ-calculus and its practical implementations. In other words, these calculi shed light on the execution models of higher-order languages. In fact, the development of a calculus with explicit substitutions faithful to the λ-calculus, in the sense of the preservation of some desired properties were the main motivation for such a long list of calculi with explicit substitutions invented in the last decades [1,22,3,6,17,13,5,7,14]. The core idea is that β-reduction is divided into two parts, one that initiates the simulation of a β-step, and another that completes the simulation as suggested by the following figure:

$$(\lambda_x.t_1)\ t_2 \xrightarrow[\beta]{\hspace{6em}} \{x := t_2\}t_1$$

$$(\lambda_x.t_1)\ t_2 \xrightarrow[\texttt{beta}]{} [x := t_2]t_1 \twoheadrightarrow_{\texttt{subst}} \{x := t_2\}t_1$$

In this figure, the `beta` step initiates the simulation of the β-reduction while the `subst` steps, forming a set of rules known as the *substitution calculus*, completes the simulation. In the case of the λ_x-calculus, the formalization of the `beta` step is done as follows: firstly, one reduces an application when its left hand side is an abstraction:

```
Inductive betax : Rel n_sexp :=
 | step_betax : ∀ (t1 t2: n_sexp) (x: atom),
     betax (n_app (n_abs x t1) t2) (n_sub t1 x t2).
```

then this reduction is done modulo α-equivalence:

```
Inductive betax_aeq: Rel n_sexp :=
| betax_aeq_step: ∀ t t' u u', t =a t' → betax t' u' → u' =a u → betax_aeq t u.
```

and finally, the `beta` step in the case of the λ_x-calculus is the contextual closure of the *beta_aeq* reduction as given by the following notation:

```
Definition betax_ctx t u := ctx betax_aeq t u.
Notation "t –>Bx u" := (betax_ctx t u) (at level 60).
Notation "t –»Bx u" := (refltrans betax_ctx t u) (at level 60).
```

$$(\lambda_x.t_1)\ t_2 \to_{Bx} [x := t_2]t_1 \tag{6}$$

The substitution calculus of the λ_x-calculus, named x-calculus, is formed by the following rules, where $x \neq y$:

$$\begin{array}{llll}
[y := t]y & \to_{var} & t & \\
[y := t]x & \to_{gc} & x & \\
[y := t_2](\lambda_y.t_1) & \to_{abs1} & \lambda_y.t_1 & \\
[y := t_2](\lambda_x.t_1) & \to_{abs2} & \lambda_x.([y := t_2]t_1) & \text{, if } x \notin fv(t_2) \\
[y := t_2](\lambda_x.t_1) & \to_{abs3} & \lambda_z.([y := t_2](x\ z)t_1) & \text{, where } z \text{ is a fresh variable, and } x \in fv(t_2) \\
[y := t_3](t_1\ t_2) & \to_{app} & ([y := t_3]t_1)\ ([y := t_3]t_2) &
\end{array} \tag{7}$$

In rule *abs*2, the condition that z is a fresh variable means that z is a new variable not present in the set $(fv(t_1) \cup fv(t_2) \cup \{x\} \cup \{y\}$. The notation used in [18] inherently handles α-equivalence, requiring only one rule for abstraction: $[y := t_2](\lambda_x.t1) \to \lambda_x.([y := t_2]t_1)$. In this rule assumes that bound variables are renamed as needed to avoid capturing free variables. In our formalization, this rule was divided into *abs*1, *abs*2 and *abs*3 to explicitly prevent variable capture. The corresponding Coq code is as follows:

```
Inductive pix : n_sexp → n_sexp → Prop :=
| step_var : ∀ (t: n_sexp) (y: atom),
    pix (n_sub (n_var y) y t) t
| step_gc : ∀ (t: n_sexp) (x y: atom),
    x ≠ y → pix (n_sub (n_var x) y t) (n_var x)
| step_abs1 : ∀ (t1 t2: n_sexp) (y : atom),
    pix (n_sub (n_abs y t1) y t2) (n_abs y t1)
| step_abs2 : ∀ (t1 t2: n_sexp) (x y: atom),
    x ≠ y → x `notin` fv_nom t2 →
    pix (n_sub (n_abs x t1) y t2) (n_abs x (n_sub t1 y t2))
| step_abs3 : ∀ (t1 t2: n_sexp) (x y z: atom),
    x ≠ y → z ≠ x → z ≠ y → x `in` fv_nom t2 → z `notin` fv_nom t1 → z `notin` fv_nom t2 →
                   pix (n_sub (n_abs x t1) y t2) (n_abs z (n_sub (swap x z t1) y t2))
| step_app : ∀ (t1 t2 t3: n_sexp) (y: atom),
    pix (n_sub (n_app t1 t2) y t3) (n_app (n_sub t1 y t3) (n_sub t2 y t3)).
```

In a similar way to the rule $->Bx$, we define $->x$ as the contextual closure of the rules in the inductive definition *pix* modulo α-equivalence, and $->lx$ as the union of $->Bx$ and $->x$.

The next lemma show that the explicit substitution implements the metasubstitution when *t1* is pure.

```
Lemma pure_pix: ∀ t1 x t2, pure t1 → ([x := t2]t1) −»x ({x := t2}t1).
```

The Z property is a promising techique used to prove confluence of reduction systems [24,10,9]. Shortly, a function f: *n_sexp* → *n_sexp* has the Z property for the binary relation R if the following diagram holds:

$$\begin{tikzcd} t_1 \arrow[r, "R"'] & t_2 \arrow[dl, dotted, two heads, "R"] \\ ft_1 \arrow[r, dotted, two heads, "R"'] & ft_2 \end{tikzcd}$$

An extension of the Z property, known as *Compositional Z* property gives a sufficient condition for that a compositional function satisfies the Z property [18]. For the λ_x-calculus, we will prove that the following diagram holds:

$$\begin{tikzcd} t_1 \arrow[r, "lx"'] & t_2 \arrow[dl, dotted, two heads, "lx"] \\ B(P\ t_1) \arrow[r, dotted, two heads, "lx"'] & B(P\ t_2) \end{tikzcd} \tag{8}$$

where P (resp. B) is the complete permutation (resp. complete development) recursively defined as:

```
Fixpoint P (t : n_sexp) := match t with
                           | n_var x ⇒ n_var x
                           | n_abs x t1 ⇒ n_abs x (P t1)
                           | n_app t1 t2 ⇒ n_app (P t1) (P t2)
                           | n_sub t1 x t2 ⇒ {x := (P t2)}(P t1)
                           end.
```

and

```
Fixpoint B (t : n_sexp) := match t with
```

```
| n_var x ⇒ n_var x
| n_abs x t1 ⇒ n_abs x (B t1)
| n_app t1 t2 ⇒ match t1 with
                  | n_abs x t3 ⇒ {x := (B t2)}(B t3)
                  | _ ⇒ n_app (B t1) (B t2)
                  end
| n_sub t1 x t2 ⇒ n_sub (B t1) x (B t2)
end.
```

The complete permutation function P and the complete development B have several interesting properties. In what follows, we will list the most relevant ones to show how to get the confluence proof for the λ_x-calculus. The first point to be noticed is that $P\ t$ removes all explicit substitution of t, therefore $P\ t$ is a pure term:

```
Lemma pure_P: ∀ t, pure (P t).

Lemma aeq_swap_P: ∀ t x y, (P (swap x y t)) =a (swap x y (P t)).

Lemma aeq_P: ∀ t1 t2, t1 =a t2 → (P t1) =a (P t2).

Lemma pure_B: ∀ t, pure t → pure (B t).
```

Pure terms are standard λ-terms, that is, expressions constructed from variables, applications and abstractions. In the following subsection, we define a reduction rule similar to the β-reduction of the λ-calculus, but over *n_sexp* expressions.

3.1 The β-reduction

In this subsection, we define a reduction rule analogous to β-reduction in the λ-calculus modulo α-equivalence. While it shares the same redex as the −>*Bx* rule, its contractum is a term with a metasubstitution. Like the original, this rule is also defined modulo α-equivalence, and we refer to it as the β-rule.

```
Inductive beta_redex : Rel n_sexp :=
| step_beta : ∀ (t1 t2: n_sexp) (x: atom),
    beta_redex (n_app (n_abs x t1) t2) ({x := t2}t1).

Inductive beta_aeq: Rel n_sexp :=
| beta_aeq_step: ∀ t t' u u', t =a t' → beta_redex t' u' → u' =a u → beta_aeq t u.

Definition beta_ctx t u := ctx beta_aeq t u.
Notation "t ->B u" := (beta_ctx t u) (at level 60).
Notation "t -»B u" := (refltrans beta_ctx t u) (at level 60).
```

The next lemma shows that the β-rule does not introduce explicit substitutions, and several other properties can be found in the source code of the formalization.

```
Lemma pure_beta_trans: ∀ t1 t2, pure t1 → t1 ->B t2 → pure t2.
```

As expected, one step β-reduction can be simulated by the reflexive-transitive closure of the −>*lx* rule.

```
Lemma refltrans_pure_beta: ∀ t1 t2, pure t1 → t1 ->B t2 → t1 -»lx t2.

Corollary refltrans_pure_beta_refltrans: ∀ t1 t2, pure t1 → t1 -»B t2 → t1 -»lx t2.
```

Since we are working modulo α-equivalence, a straightforward instantiation of A with *n_sexp* is insuficient. This is because the reflexive closure of −>*x* must encompass not only syntactic equality but also α-equivalence. In fact, the proof that −>*lx* has the Z property (diagram (8)) is proved by the following two diagrams, since −>*lx* = −>*x* ∪ −>*Bx*:

$$
\begin{tikzcd}
t_1 \arrow[r, "x"'] & t_2 \arrow[dl, dotted, two heads, "x"] \\
P\ t_1 \arrow[r, dotted, two heads, "x"'] \arrow[d, dotted, two heads, "lx"'] & P\ t_2 \\
B(P\ t_1) \arrow[r, dotted, two heads, "lx"'] & B(P\ t_2)
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
t_1 \arrow[r, "Bx"'] & t_2 \arrow[dl, dotted, two heads, "lx"] \\
B(P\ t_1) \arrow[r, dotted, two heads, "lx"'] & B(P\ t_2)
\end{tikzcd}
\tag{9}
$$

Note that the complete permutation P replaces every explicit substitution in the input term t with the corresponding metasubstitution in the output $P\ t$. Furthermore, the following lemma (denoted *pi_P*) demonstrates that applying the complete permutation to a term before and after an $->x$ step results in the same term, up to the renaming of bound variables:

`Lemma` *pi_P*: $\forall$ *t1 t2*, *t1* $->x$ *t2* $\to$ (*P t1*) $=a$ (*P t2*).

Proof. The proof is by induction on the reduction $t_1 \to_x t_2$. The non trivial case is when $t_1 \to_{abs3} t_2$. In this case, $t_1 =_\alpha [y := t_2'](\lambda_x.t_1') \to_{abs3} \lambda_z.[y := t_2']((x\ z)t_1') =_\alpha t_2$, where $x \neq y$ and z is a fresh variable. In this case, the proof is as follows:

$$
\dfrac{
(\texttt{aeq\_P})\ \dfrac{t_1 =_\alpha [y := t_2'](\lambda_x.t_1')}{P\ t_1 =_\alpha P\ ([y := t_2'](\lambda_x.t_1'))}
\qquad
\dfrac{(\star) \qquad \dfrac{\lambda_z.[y := t_2']((x\ z)t_1') =_\alpha t_2}{P\ (\lambda_z.[y := t_2']((x\ z)t_1')) =_\alpha P\ t_2}\ (\texttt{aeq\_P})}{P\ ([y := t_2'](\lambda_x.t_1')) =_\alpha P\ t_2}\ (\alpha\text{-trans})
}{P\ t_1 =_\alpha P\ t_2}\ (\alpha\text{-trans})
$$

where $(\star)$ is given by

$$
\dfrac{\dfrac{\dfrac{}{\{y := P\ t_2'\}\lambda_x.(P\ t_1') =_\alpha \lambda_z.\{y := P\ t_2'\}((x\ z)P\ t_1')}\ (\text{m\_subst\_abs\_neq})}{\{y := P\ t_2'\}\lambda_x.(P\ t_1') =_\alpha \lambda_z.\{y := P\ t_2'\}P\ ((x\ z)t_1')}\ (\text{aeq\_swap\_P})}{P\ ([y := t_2'](\lambda_x.t_1')) =_\alpha P\ (\lambda_z.[y := t_2']((x\ z)t_1'))}\ (\text{def-P})
$$

$\square$

This simplifies the left diagram in (9) as follows:

$$
\begin{tikzcd}
t_1 \arrow[r, "x"'] & t_2 \arrow[dl, dotted, two heads, "x"] \\
P\ t_1 =_\alpha P\ t_2 \arrow[d, dotted, two heads, "lx"'] & \\
B(P\ t_1) \arrow[r, dotted, two heads, "lx"'] & B(P\ t_2)
\end{tikzcd}
\tag{10}
$$

Which in turn simplifies to

$$
\begin{tikzcd}
t_1 \arrow[r, "x"'] & t_2 \arrow[dl, dotted, two heads, "x"] \\
P\ t_1 =_\alpha P\ t_2 \arrow[d, dotted, two heads, "lx"'] & \\
B(P\ t_1) =_\alpha B(P\ t_2) &
\end{tikzcd}
\tag{11}
$$

due to the lemma *aeq_B*:

Lemma *aeq_B*: $\forall$ *t1 t2*, *t1* $=a$ *t2* $\rightarrow$ (*B t1*) $=a$ (*B t2*).

Proof. In this case, the proof is done by induction on the size of the term t_1. We developed a customized induction principle for this kind of proof:

```
Lemma n_sexp_induction: forall P : n_sexp -> Prop,           (forall x, P (n_var x)) ->
              (forall t1 z, (forall t2, size t2 = size t1 -> P t2) -> P (n_abs z t1)) ->
                                        (forall t1 t2, P t1 -> P t2 -> P (n_app t1 t2)) ->
(forall t1 t3 z, P t3 -> (forall t2, size t2 = size t1 -> P t2) -> P (n_sub t1 z t3)) ->
                                                                     (forall t, P t).
```

The non trivial case is when $t_1 = (\lambda_x.t_{11})\ t_{12}$ and $t_2 = (\lambda_y.t_{21})\ t_{22}$, for some terms t_{11}, t_{12}, t_{21} and t_{22} and variables x and y. We need to prove that $B((\lambda_x.t_{11})\ t_{12}) =_\alpha B((\lambda_y.t_{21})\ t_{22})$. If $x = y$ then we are done by the induction hypothesis, and if $x \neq y$ then

$$\dfrac{\dfrac{\dfrac{}{B\ t_{12} =_\alpha B\ t_{22}}\,\text{(i.h.)} \qquad \dfrac{}{B\ t_{11} =_\alpha (x\ y)(B\ t_{21})}\,\text{(i.h.)}}{\{x := B\ t_{12}\}(B\ t_{11}) =_\alpha \{y := B\ t_{22}\}(B\ t_{21})}\,\text{(aeq\_m\_subst\_neq)}}{B((\lambda_x.t_{11})\ t_{12}) =_\alpha B((\lambda_y.t_{21})\ t_{22})}\,\text{(def-B)}$$

Note that the induction hypothesis can be applied to the right branch, as the swap does not affect the size of terms, *i.e.* $|(x\ y)(B\ t_{21})| = |B\ t_{21}|$. $\square$

One challenging task in this formalization was the proof of the next lemma stating that the metasubstitution of complete development of its components reduces (via β-reduction) to the complete development of the metasubstitution.

Lemma *refltrans_m_subst_B_beta*: $\forall$ *t1 t2 x*, *pure t1* $\rightarrow$ *pure t2* $\rightarrow$ ($\{x$:= *B t2*$\}$ *B t1*) $\twoheadrightarrow$*B* (*B* ($\{x$:= *t2*$\}$ *t1*)).

Proof. The proof is by induction on the size of the term t_1. This proof also uses a customized induction principle given by

```
n_sexp_size_induction : forall P: n_sexp -> Prop,
                                  (forall x : atom, P (n_var x)) ->
(forall (t1: n_sexp) (z: atom), (forall t1': n_sexp, size t1' <
                 size (n_abs z t1) -> P t1') -> P (n_abs z t1)) ->
(forall t1 t2: n_sexp, (forall t1': n_sexp, size t1' <
               size (n_app t1 t2) -> P t1') -> P (n_app t1 t2)) ->
(forall (t1 t2: n_sexp) (z: atom), (forall t1': n_sexp, size t1' <
             size ([z := t2] t1) -> P t1') -> P ([z := t2] t1)) ->
                                           forall t : n_sexp, P t
```

The interesting case is the application case. If $t_1 = t_{11}\ t_{12}$ then we need to prove that

$$\{x := B\ t_2\}(B(t_{11}\ t_{12})) \twoheadrightarrow_\beta B(\{x := t_2\}(t_{11}\ t_{12}))$$

We proceed by case analysis on the structure of t_{11}. If t_{11} is the variable x then our goal is

$$(B\ t_2)\ (\{x := B\ t_2\}(Bt_{12}) \twoheadrightarrow_\beta B\ (t_2\ (\{x := t_2\}t_{12}))$$

and in turn, we proceed by case analysis on the structure of t_2. The non-trivial case, again is when $t_2 = \lambda_y.t_2'$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{x := B\ (\lambda_y.t_2')\}(Bt_{12}) \twoheadrightarrow_\beta B(\{x := \lambda_y.t_2'\}t_{12})}\,\text{(i.h.)}}{\{y := (\{x := B\ (\lambda_y.t_2')\}(Bt_{12}))\}(B\ t_2') \twoheadrightarrow_\beta \{y := (B(\{x := (\lambda_y.t_2')\}t_{12}))\}(B\ t_2')}\,\text{(compat)}}{(\lambda_y.B\ t_2')\ (\{x := B\ (\lambda_y.t_2')\}(Bt_{12})) \twoheadrightarrow_\beta \{y := (B(\{x := (\lambda_y.t_2')\}t_{12}))\}(B\ t_2')}\,(\beta)}{(B\ (\lambda_y.t_2'))\ (\{x := B\ (\lambda_y.t_2')\}(Bt_{12}) \twoheadrightarrow_\beta B\ ((\lambda_y.t_2')\ (\{x := (\lambda_y.t_2')\}t_{12})))}\,\text{(def-B)}$$

where the (compat) rule is applied in a general sense, as it serves as a structural compatibility rule in various proofs. Note that all compatibility rules can be found in the source files of the formalization.

Another non-trivial case occurs when $t_{11} = \lambda_y.t'_{11}$, leading to the goal

$$\{x := B\ t_2\}(B((\lambda_y.t'_{11})\ t_{12})) \twoheadrightarrow_\beta B(\{x := t_2\}((\lambda_y.t'_{11})\ t_{12}))$$

whose derivation is given by

$$\dfrac{\{x := B\ t_2\}(\{y := B\ t_{12}\}(B\ t'_{11})) \twoheadrightarrow_\beta B(\{x := t_2\}((\lambda_y.t'_{11})\ t_{12}))}{\{x := B\ t_2\}(B((\lambda_y.t'_{11})\ t_{12})) \twoheadrightarrow_\beta B(\{x := t_2\}((\lambda_y.t'_{11})\ t_{12}))}\ (\textsf{def-B})$$

Then we have two cases, either $x = y$ or $x \neq y$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{x := B\ t_2\}(B\ t_{12}) \twoheadrightarrow_\beta B\{x := t_2\}t_{12}}\ (\textsf{i.h.})}{\{y := (\{x := B\ t_2\}(B\ t_{12}))\}(B\ t'_{11}) \twoheadrightarrow_\beta \{y := B(\{x := t_2\}t_{12})\}(B\ t'_{11})}\ (\textsf{compat})}{\{y := (\{x := B\ t_2\}(B\ t_{12}))\}(B\ t'_{11}) \twoheadrightarrow_\beta B((\lambda_y.t'_{11})\ \{x := t_2\}t_{12})}\ (\textsf{def-B})}{\{x := B\ t_2\}(\{y := B\ t_{12}\}(B\ t'_{11})) \twoheadrightarrow_\beta B(\{x := t_2\}((\lambda_y.t'_{11})\ t_{12}))}\ (x = y)}{\{x := B\ t_2\}(B((\lambda_y.t'_{11})\ t_{12})) \twoheadrightarrow_\beta B(\{x := t_2\}((\lambda_y.t'_{11})\ t_{12}))}\ (\textsf{def-B})$$

$$\dfrac{\dfrac{\dfrac{\dfrac{(\star)}{\{x := B\ t_2\}(\{y := B\ t_{12}\}(B\ t'_{11})) \twoheadrightarrow_\beta \{z := B(\{x := t_2\}t_{12})\}(B(\{x := t_2\}(z\ y)t'_{11}))}}{\{x := B\ t_2\}(\{y := B\ t_{12}\}(B\ t'_{11})) \twoheadrightarrow_\beta B((\lambda_z.(\{x := t_2\}(z\ y)t'_{11}))\ (\{x := t_2\}t_{12}))}\ (\textsf{def-B})}{\{x := B\ t_2\}(\{y := B\ t_{12}\}(B\ t'_{11})) \twoheadrightarrow_\beta B(\{x := t_2\}((\lambda_y.t'_{11})\ t_{12}))}\ (x \neq y)}{\{x := B\ t_2\}(B((\lambda_y.t'_{11})\ t_{12})) \twoheadrightarrow_\beta B(\{x := t_2\}((\lambda_y.t'_{11})\ t_{12}))}\ (\textsf{def-B})$$

where $(\star)$ is obtained by decomposing the reduction

$$\{x := B\ t_2\}(\{y := B\ t_{12}\}(B\ t'_{11})) \twoheadrightarrow_\beta \{z := B(\{x := t_2\}t_{12})\}(B(\{x := t_2\}(z\ y)t'_{11}))$$

using $\{z := \{x := B\ t_2\}(B\ t_{12})\}(\{x := B\ t_2\}((z\ y)(B\ t'_{11})))$ as the intermediare term, and each subreduction is solved as follows

$$\dfrac{\dfrac{\dfrac{}{\{x := B\ t_2\}(\{z := B\ t_{12}\}((z\ y)B\ t'_{11})) =_\alpha \{z := \{x := B\ t_2\}(B\ t_{12})\}(\{x := B\ t_2\}((z\ y)(B\ t'_{11})))}\ (\textsf{SL})}{\{x := B\ t_2\}(\{z := B\ t_{12}\}((z\ y)B\ t'_{11})) \twoheadrightarrow_\beta \{z := \{x := B\ t_2\}(B\ t_{12})\}(\{x := B\ t_2\}((z\ y)(B\ t'_{11})))}\ (\textsf{refl})}{\{x := B\ t_2\}(\{y := B\ t_{12}\}(B\ t'_{11})) \twoheadrightarrow_\beta \{z := \{x := B\ t_2\}(B\ t_{12})\}(\{x := B\ t_2\}((z\ y)(B\ t'_{11})))}\ (\textsf{ren})$$

where the rule (ren) denotes a renaming of bound variables, the rule (SL) denotes the Substitution Lemma (*m_subst_lemma*) above (see Section 2) and

$$\dfrac{(\textsf{i.h.})\ \dfrac{}{\{x := B\ t_2\}((z\ y)(B\ t'_{11})) \twoheadrightarrow_\beta B(\{x := t_2\}(z\ y)t'_{11})} \qquad \dfrac{}{\{x := B\ t_2\}(B\ t_{12}) \twoheadrightarrow_\beta B(\{x := t_2\}t_{12})}\ (\textsf{i.h.})}{\{z := \{x := B\ t_2\}(B\ t_{12})\}(\{x := B\ t_2\}((z\ y)(B\ t'_{11}))) \twoheadrightarrow_\beta \{z := B(\{x := t_2\}t_{12})\}(B(\{x := t_2\}(z\ y)t'_{11}))}$$

In general, proofs involving the complete development B are challenging. The following lemma presents another example of a complex proof. Once again, the difficult case arises when B receives an application as an argument, though we will leave this proof withour further commentary.

Lemma *beta_implies_refltrans_B*: ∀ *t1 t2*, *pure t1* → *t1* –>*B t2* → (*B t1*) –»*B* (*B t2*).

Corollary *refltrans_beta_B*: ∀ *t1 t2*, *pure t1* → *t1* –»*B t2* → *B t1* –»*B B t2*.

The proof of (11) is concluded by applying lemma *refltrans_P* and lemma *pure_refltrans_B*.

Lemma *refltrans_P*: ∀ *t*, *t* –»*x* (*P t*).

Lemma *pure_refltrans_B*: ∀ *t*, *pure t* → *t* –»*lx* (*B t*).

The second diagram in (9) is proved by the following two lemmas:

Lemma *refltrans_lx_P2*: ∀ *t1 t2*, *t1* –>*Bx t2* → *t2* –»*lx* (*B*(*P t1*)).

Proof. The proof is by induction on $t_1 \to_{Bx} t_2$. The interesting cases are when

$$t_1 = [x := t_{12}]t_{11} \to_{Bx} [x := t_{12}]t'_{11} = t_2, \text{ with } t_{11} \to_{Bx} t'_{11}$$

and

$$t_1 = [x := t_{12}]t_{11} \to_{Bx} [x := t'_{12}]t_{11} = t_2, \text{ with } t_{12} \to_{Bx} t'_{12}.$$

Both cases have similar proofs, therefore we consider only the first reduction. We proceed as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{t'_{11} \twoheadrightarrow_{lx} B(P\ t_{11})}\,\text{(i.h.)} \quad \dfrac{(\star)}{t_{12} \twoheadrightarrow_{lx} B(P\ t_{12})}}{[x := t_{12}]t'_{11} \twoheadrightarrow_{lx} [x := B(P\ t_{12})]B(P\ t_{11})}\,\text{(compat)}}{[x := t_{12}]t'_{11} \twoheadrightarrow_{lx} B([x := P\ t_{12}](P\ t_{11}))}\,\text{(def-B)} \quad \dfrac{(\star\star)}{B([x := P\ t_{12}](P\ t_{11})) \twoheadrightarrow_{lx} B(\{x := P\ t_{12}\}(P\ t_{11}))}}{\dfrac{[x := t_{12}]t'_{11} \twoheadrightarrow_{lx} B(\{x := P\ t_{12}\}(P\ t_{11}))}{[x := t_{12}]t'_{11} \twoheadrightarrow_{lx} B(P([x := t_{12}]t_{11}))}\,\text{(def-P)}}\,\text{(trans)}$$

where $(\star)$ is easily proved by lemmas *refltrans_P*, *pure_P* and *pure_refltrans_B*, and $(\star\star)$ is proved as follows:

$$\dfrac{\dfrac{\dfrac{(\star\star\star)}{[x := B(P\ t_{12})](B(P\ t_{11})) \twoheadrightarrow_{lx} \{x := B(P\ t_{12})\}(B(P\ t_{11}))} \quad (\star\star\star\star)}{[x := B(P\ t_{12})](B((P\ t_{11}))) \twoheadrightarrow_{lx} B(\{x := P\ t_{12}\}(P\ t_{11}))}\,\text{(trans)}}{B([x := P\ t_{12}](P\ t_{11})) \twoheadrightarrow_{lx} B(\{x := P\ t_{12}\}(P\ t_{11}))}\,\text{(def-B)}$$

where $(\star\star\star)$ is proved by lemma *pure_pix* since $\to_x \subseteq \to_{lx}$, and $(\star\star\star\star)$ is given by the reduction

$$\{x := B(P\ t_{12})\}(B((P\ t_{11}))) \twoheadrightarrow_{lx} B(\{x := P\ t_{12}\}(P\ t_{11})) \tag{12}$$

Since the terms $(P\ t_{11})$ and $(P\ t_{12})$ are pure, the reduction (12) can be done with $\to_\beta$, that is, it can be translated to

$$\{x := B(P\ t_{12})\}(B((P\ t_{11}))) \twoheadrightarrow_{lx} B(\{x := P\ t_{12}\}(P\ t_{11}))$$

and we are done by lemma *refltrans_m_subst_B_beta*. □

Lemma *refltrans_lx_B_P*: ∀ *t1 t2*, *t1* –>*Bx t2* → (*B* (*P t1*)) –»*lx* (*B* (*P t2*)).

Proof. Similar to the reasoning in the previous lemma, the reduction $(B(Pt_1)) \twoheadrightarrow_{lx} (B(Pt_2))$ can be translated to $(B(Pt_1)) \twoheadrightarrow_\beta (B(Pt_2))$, since both $(P\ t_1)$ and $(P\ t_2)$ are pure terms. We leave the details to the interested reader explore in the source code of the formalization. □

TODO

- citar Metalib
- criar repo e inserir link no documento
- proofs close to paper and pencil approach
- Adaptações em ZtoConfl

4 Conclusion

We presented the general structure of a framework that extends the work of [8] and [15] for studying generic calculi with explicit substitutions using the nominal approach. All proofs are constructive, with no reliance on the classical axioms. Specifically, we applies this framework to prove the confluence of the λ_x-calculus through the compositional Z property, following the method in [18].

For future work, we plan to use this framework to study additional calculi with explicit substitutions, such as those in [20,14,19,12].

References

1. M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit Substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.
2. H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* Number v. 103 in Studies in Logic and the Foundations of Mathematics. North-Holland ; Sole distributors for the U.S.A. and Canada, Elsevier Science Pub. Co, Amsterdam ; New York : New York, N.Y, rev. ed edition, 1984.
3. Zine-El-Abidine Benaissa, Daniel Briaud, Pierre Lescanne, and Jocelyne Rouyer-Degli. $\Lambda\nu$, a calculus of explicit substitutions which preserves strong normalisation. *Journal of Functional Programming*, 6(5):699–722, September 1996.
4. R. Bloo and K. Rose. Preservation of Strong Normalisation in Named Lambda Calculi with Explicit Substitution and Garbage Collection. In *CSN-95: COMPUTER SCIENCE IN THE NETHERLANDS*, pages 62–72, 1995.
5. Roel Bloo and Herman Geuvers. Explicit Substitution: On the Edge of Strong Normalization. *Theoretical Computer Science*, 211(1-2):375–395, 1999.
6. Pierre-Louis Curien, Thérèse Hardin, and Jean-Jacques Lévy. Confluence Properties of Weak and Strong Calculi of Explicit Substitutions. *Journal of the ACM*, 43(2):362–397, 1996.
7. R. David and B. Guillaume. A lambda-calculus with explicit weakening and explicit substitution. *Mathematical Structures in Computer Science*, 11(1):169–206, 2001.
8. Flávio L. C. de Moura and Leandro O. Rezende. A formalization of the (compositional) z property. In *Fifth Workshop on Formal Mathematics for Mathematicians*, 2021.
9. P Dehornoy and V van Oostrom. Z, proving confluence by monotonic single-step upperbound functions. *Logical Models of Reasoning and Computation (LMRC-08)*, page 85, 2008.
10. B. Felgenhauer, J. Nagele, V. van Oostrom, and C. Sternagel. The Z Property. *Archive of Formal Proofs*, 2016, 2016.
11. 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 2002.
12. Yuki Honda, Koji Nakazawa, and Ken-etsu Fujita. Confluence Proofs of Lambda-Mu-Calculi by Z Theorem. *Studia Logica*, January 2021.
13. Fairouz Kamareddine and Alejandro Ríos. Extending a λ-calculus with explicit substitution which preserves strong normalisation into a confluent calculus on open terms. *Journal of Functional Programming*, 7(4):395–420, July 1997.
14. Delia Kesner. A Theory of Explicit Substitutions with Safe and Full Composition. *Logical Methods in Computer Science*, Volume 5, Issue 3:816, July 2009.
15. Maria J. D. Lima and Flávio L. C. de Moura. A Formalized Extension of the Substitution Lemma in Coq. *EPTCS*, 389:80–95, 2023.
16. R. Lins. A new formula for the execution of categorical combinators. *8th Conference on Automated Deduction (CADE)*, volume 230 of LNCS:89–98, 1986.
17. C. A. Muñoz. Confluence and Preservation of Strong Normalisation in an Explicit Substitutions Calculus. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996*, pages 440–447, 1996.
18. Koji Nakazawa and Ken-etsu Fujita. Compositional Z: Confluence Proofs for Permutative Conversion. *Studia Logica*, 104(6):1205–1224, 2016.
19. Koji Nakazawa and Ken-etsu Fujita. Z for call-by-value. In *6th International Workshop on Cofluence (IWC 2017)*, pages 57–61, 2017.
20. Koji Nakazawa, Ken-etsu Fujita, and Yuta Imagawa. Z property for the shuffling calculus. *Mathematical Structures in Computer Science*, pages 1–13, January 2023.

21. K. Rose. Explicit cyclic substitutions. In Michaël Rusinowitch and Jean-Luc Rémy, editors, *3rd International Workshop on Conditional Term Rewriting Systems (CTRS)*, volume 656, pages 36–50. Springer-Verlag, 1992.
22. K. H. Rose, R. Bloo, and F. Lang. On Explicit Substitution With Names. *J Autom Reasoning*, 49(2):275–300, 2011.
23. Christian Urban, Stefan Berghofer, and Michael Norrish. Barendregt's Variable Convention in Rule Inductions. In Frank Pfenning, editor, *Automated Deduction – CADE-21*, pages 35–50, Berlin, Heidelberg, 2007. Springer.
24. Vincent van Oostrom. Z; syntax-free developments. In Naoki Kobayashi, editor, *6th International Conference on Formal Structures for Computation and Deduction (FSCD 2021)*, volume 195 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:22, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.