Proving Confluence of a Calculus with Explicit Substitutions in Coq

Leandro O. Rezende
Departamento de Matemática
Universidade de Brasília
Brasília, Brasil
L-ordo.ab.chao@hotmail.com

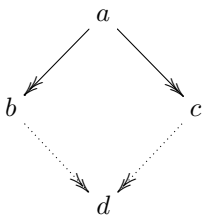
Flávio L. C. de Moura
Departamento de Ciência da Computação
Universidade de Brasília
Brasília, Brasil
flaviomoura@unb.br

***Abstract*—Rewriting theory is a well established model of computation equivalent to the Turing machines. The most well known rewriting system is the λ-calculus, the theoretical foundation of the functional paradigm of programming. Confluence is an important property related to the determinism of the results given by a rewriting system. In this work, which is still in progress, we formalize the confluence of an extension of the λ-calculus with explicit substitutions following the steps in [1]. The formalization is done in the Coq proof assistant, and the whole proof is constructive, i.e. it does not rely on the law of excluded middle or on the proof by contradiction principles. This is important because the last step of this project aims the generate certified code via the extraction mechanism of Coq.**

***Index Terms*—Confluence, λ-calculus, Formal Methods, Proof Assistants, Coq**

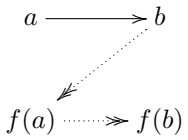
I. Introduction

This work is divided into two parts: The first part is about a characterization of the confluence property for ARSs (Abstract Rewriting Systems). An ARS is simply a pair composed of a set and binary operation over this set. Given an ARS $(A, \rightarrow)$, where A is a set and $\rightarrow$: $A \times A$ is a binary relation over A, an expression $a \in A$ usually can be reduced in different ways in order to produce a result. Informally, the confluence property states that, no matter how the reduction is done, the result will always be the same. The following diagram is used to express this idea:



The diagram states that if the expression a can be reduced in two different ways to the expressions b and c, then there exists an expression d such that both b and c reduces to d. In [2], a different characterization of confluence is given by the so called Z-property:

Definition 1: Let $(A, \rightarrow)$ be an abstract rewriting system (ARS). The system $(A, \rightarrow)$ has the Z property, if there exists a map $f : A \rightarrow A$ such that:



So the first part of this work was to formally prove that if an ARS $(A, \rightarrow)$ satisfies the Z-property then $(A, \rightarrow)$ is confluent. This part is complete.

The second part of this work is to use the first one to get confluence of an extension of the λ-calculus with explicit substitutions known as λex-calculus [1].

A. The λex-calculus

Calculi with explicit substitutions are extensions of the λ-calculus in which the substitution operation is a primitive operation, making them a formalism that is closer to implementations based on the λ-calculus. During the last three decades, this area of research attracted the attention of the scientific community [1], [3]–[9]. The main reason for the development of several different calculi with explicit substitutions was that none of them were faithful to the system they were supposed to model: the λ-calculus. This means that none of these calculi satisfy simultaneously a set of important properties: Confluence on open and closed terms, Simulation of one step β-reduction, Termination of the associated substitution calculus and Preservation of Strong Normalisation. In 2009, Delia Kesner published a paper in which she presents a calculus that satisfies all these properties: the λex-calculus. The terms of the λex-calculus is given by the following grammar, equation and rules:

$$\texttt{(terms)} \qquad t ::= x \mid t\ t \mid \lambda x.t \mid t[x/t]$$

$t[x/u][y/v]$	$=_C$	$t[y/v][x/u]$	if $y \notin \texttt{fv}(u)$ and $x \notin \texttt{fv}(v)$
$(\lambda x.t)u$	$\rightarrow_{\texttt{B}}$	$t[x/u]$	
$x[x/u]$	$\rightarrow_{\texttt{Var}}$	u	
$t[x/u]$	$\rightarrow_{\texttt{Gc}}$	t,	if $x \notin \texttt{fv}(t)$
$(t\ v)[x/u]$	$\rightarrow_{\texttt{App}}$	$t[x/u]\ v[x/u]$	
$(\lambda y.t)[x/u]$	$\rightarrow_{\texttt{Lamb}}$	$\lambda y.t[x/u]$	
$t[x/u][y/v]$	$\rightarrow_{\texttt{Comp}}$	$t[y/v][x/u[y/v]]$,	if $y \in \texttt{fv}(u)$

B. The Framework

A direct implementation or formalization of the λex-calculus is not straightforward because terms are considered modulo renaming of bound variables, i.e. modulo α-conversion. A interesting way to avoid to deal with α-conversion is to use the so called it De Bruijn notation [10]. Nevertheless, De Briuijn notation also has it disadvantages: the manipulation of free variables is tricky. The chosen framework for our formalization takes the benefits of named and De Bruijn notation. It is known as Locally Nameless Representation [11]: it uses names for free variables and De Bruijn indexes for bound variables.

II. The Formalization

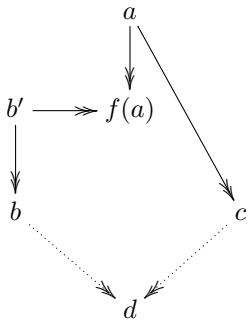
Our formalization is being done in Coq [12], a constructive proof assistant written in the OCaml programming language. The Z and confluence properties are written in the Coq language as follows:

```
Definition Zprop {A:Type} (R: Rel A) := ∃ f:A → A, ∀ a b, R a b → ((refltrans R) b (f a) ∧ (refltrans R) (f a) (f b)).
Definition Confl {A:Type} (R: Rel A) := ∀ a b c, (refltrans R) a b → (refltrans R) a c → (∃ d, (refltrans R) b d ∧ (refltrans R) c d).
```

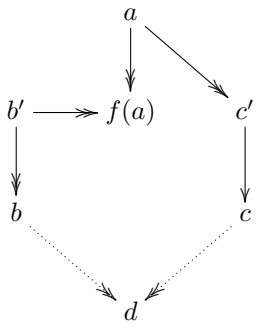
The first part of this work aimed to prove the following theorem:

```
Theorem Zprop_implies_Confl {A:Type}: ∀ R: Rel A, Zprop R → Confl R.
```

The challenging point of this proof was the structure of the induction on the number of steps of *refltrans R a b* and *refltrans R a c*. Since there is no information of the number of steps of either *refltrans R a b* or *refltrans R a c*, just doing induction on one followed by the other is not enough, as it would require us to prove that *refltrans R b c* or *refltrans R c b*. Luckily, we can use induction on *refltrans R a b*, then discard the information of the reduction from *a* to *b'* (where *R a b'* and *refltrans R b' b*) after we use the Z-property to assure that *refltrans R a* (*f a*) $\wedge$ *refltrans R b'* (*f a*), where *f a* comes from applying the Z-property. Then, our proof can be shown as the following diagram:



Now, we use induction on *refltrans R a c*, use the fact that *refltrans R a* (*f a*), coupled with the induction hypothesis from the induction on *refltrans R a b*, to close the induction basis.



Now, to close the proof, we can use the two induction hypotheses to conclude the proof that the Z-property implies confluence.

According to the locally nameless representation variables must be split in two classes because the bound ones are coded as indexes, while the free ones use names:

```
Inductive pterm : Set :=
  | pterm_bvar : nat → pterm
  | pterm_fvar : var → pterm
  | pterm_app : pterm → pterm → pterm
  | pterm_abs : pterm → pterm
  | pterm_sub : pterm → pterm → pterm.
```

This new grammar requires some caution because there are some expressions generated by it that do not correspond to a term. In fact, a expression formed by a sole De Bruijn index is not a term because indexes are intended to represent bound variables only. Therefore, the expressions that correspond to the usual notion of term is a proper subset of the above grammar and is characterized by the following predicate:

```
Inductive term : pterm → Prop :=
  | term_var : ∀ x,
      term (pterm_fvar x)
  | term_app : ∀ t1 t2,
      term t1 →
      term t2 →
      term (pterm_app t1 t2)
  | term_abs : ∀ L t1,
      (∀ x, x "notin L → term (t1 ^ x)) →
      term (pterm_abs t1)
  | term_sub : ∀ L t1 t2,
     (∀ x, x "notin L → term (t1 ^ x)) →
      term t2 →
      term (pterm_sub t1 t2).
```

In order to prove that the λex-calculus has the Z-property, one needs to find a function f from terms to terms that satisfies the definition. Following the steps of [1] we take f as the superdevelopment function:

```
Fixpoint sd (t : pterm) : pterm :=
  match t with
  | pterm_bvar i ⇒ t
  | pterm_fvar x ⇒ t
  | pterm_abs t1 ⇒ pterm_abs (sd t1)
  | pterm_app t1 t2 ⇒ let t0 := (sd t1) in
```

```
                    match t0 with
                    | pterm_abs t' ⇒ t' ^^ (sd t2)
                    | _ ⇒ pterm_app (sd t1) (sd t2)
                    end
  | pterm_sub t1 t2 ⇒ (sd t1) ^^ (sd t2)
  end.
```

where t ^^ t' corresponds to the term obtained from t after replacing all occurrences of its first dangling De Bruijn index by t', i.e. the operation ^^ simulates the metasubstitution of the λ-calculus. The following challenging step is to prove that this function allows us to prove that the λex-calculus satisfies the Z-property, and then conclude that it is confluent:

```
Lemma BxZlex: ∀ a b, a →_lex b → b →_lex× (sd a) ∧ (sd a) →_lex× (sd b).

Theorem Zlex: Zprop lex.
Proof.
  unfold Zprop.
  ∃ sd.
  apply BxZlex.
Qed.

Corollary lex_is_confluent: Confl lex.
Proof.
  apply Zprop_implies_Confl.
  apply Zlex.
Qed.
```

III. Conclusion

By concluding this work, we will have formalized the confluence property of an extension of the λ-calculus. A parallel project is developing the formalization of the termination property of the λex-calculus. By merging both formalizations we aim to automatically extract certified Ocaml code of an explicit substitutions calculus that satisfies all the desired properties: confluence, simulation of one-step β-reduction, termination of the associated calculus and preservation of strong normalization.

References

[1] D. Kesner, "A Theory of Explicit Substitutions with Safe and Full Composition," *Logical Methods in Computer Science*, vol. 5, no. 3:1, pp. 1–29, 2009.

[2] V. van Oostrom, "Z - draft: For your mind only," 2007, unpublished.

[3] R. Lins, "A new formula for the execution of categorical combinators," *8th Conference on Automated Deduction (CADE)*, vol. volume 230 of LNCS, pp. 89–98, 1986.

[4] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy, "Explicit Substitutions," *Journal of Functional Programming*, vol. 1, no. 4, pp. 375–416, 1991.

[5] 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*, 1995, pp. 62–72.

[6] Z. el A. Benaissa, D. Briaud, P. Lescanne, and J. Rouyer-Degli, "$\lambda\upsilon$, a Calculus of Explicit Substitutions which Preserves Strong Normalization," *JFP*, vol. 6, no. 5, pp. 699–722, 1996.

[7] F. Kamareddine and A. Ríos, "Extending a λ-calculus with Explicit Substitution which Preserves Strong Normalisation into a Confluent Calculus on Open Terms," *Journal of Functional Programming*, vol. 7, pp. 395–420, 1997.

[8] 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*. IEEE Computer Society, 1996, pp. 440–447. [Online]. Available: https://doi.org/10.1109/LICS.1996.561460

[9] G. Nadathur and D. S. Wilson, "A Notation for Lambda Terms: A Generalization of Environments," vol. 198, pp. 49–98, 1998.

[10] N. G. de Bruijn, "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem," *Indagationes Mathematicae (Proceedings)*, vol. 75, no. 5, pp. 381–392, 1972. [Online]. Available: http://dx.doi.org/10.1016/1385-7258(72)90034-0

[11] A. Charguéraud, "The Locally Nameless Representation," *Journal of Automated Reasoning*, pp. 1–46, 2011. [Online]. Available: http://dx.doi.org/10.1007/s10817-011-9225-2

[12] T. C. D. Team, "The coq proof assistant, version 8.8.2," 2018. [Online]. Available: https://doi.org/10.5281/zenodo.1174360