A Formalization of the (Compositional) Z Property

Flávio L. C. de Moura

Departamento de Ciência da Computação, Universidade de Brasília, Brazil
`flaviomoura@unb.br`

Abstract. Rewriting theory is a well established model of computation equivalent to the Turing machines, and the most well known rewriting system is the λ-calculus. Confluence is an important and undecidable property related to the determinism of the computational process. Direct proofs of confluence are, in general, difficult to be done. Therefore, alternative characterizations of confluence can circumvent this difficulty for different contexts. This is the case of the so called Z property, which has been successfully used to prove confluence in several situations such as the λ-calculus with $\beta\eta$-reduction, extensions of the λ-calculus with explicit substitutions, the $\lambda\mu$-calculus, etc. In this work we present a direct and constructive proof that the Z property implies confluence, which is formalized in the Coq proof assistant. In addition, we formalized an extension of the Z property, known as the Compositional Z, that has a modular approach for compositional functions.

1 Introduction

Confluence is an important and undecidable property concerning the determinism of the computational process. In this sense, one says that a program is confluent if every two ways of evaluating it, result in the very same answer. In the particular case of Abstract Rewriting Systems (ARS), which are the focus of this work, confluence can be beautifully expressed by diagrams as we will see in the next section.

The contributions of this work are as follows:

- We present a new proof that the Z property implies confluence, which is direct and constructive.
- The proof that the Z property implies confluence is formalized in the Coq proof assistant, and the presentation is made interleaving Coq code followed by an explanation in English of the code. In this way, the annotations are done directly in the Coq files using the coqdoc annotation style. We believe that this approach is interesting for those that are not familiar with the Coq proof assistant because the Coq code followed by English explanations gives a good idea on how they relate to each other. This discipline also forces a better organization of the formalization and of the proofs so that the explanation in English is comprehensible.
- We formalize an extension of the Z property, known as compositional Z property, as presented in [3].

2 A Formalization of the Z property

In this section, we present a formalization of the Z property in the context of ARS, which are sets with a binary relation. A binary relation is a predicate over a type A:

`Definition` Rel (A:`Type`) := $A \to A \to$ `Prop`.

If (A, R), is an ARS and $a, b \in A$ then we write $a \to_R b$ (or $R\ a\ b$ in the Coq syntax below) to denote that $(a, b) \in R$, and in this case, we say that a R-reduces to b in one step. The transitive closure of $\to_R$, written $\to_R^+$, is defined as usual by the following inference rules:

$$\frac{a \to_R b}{a \to_R^+ b}\ (\mathit{singl}) \qquad\qquad \frac{a \to_R b \qquad b \to_R^+ c}{a \to_R^+ c}\ (\mathit{transit})$$

This definition corresponds to the following Coq code, where $\to_R$ (resp. $\to_R^+$) corresponds to R (resp. $\mathit{trans}\ R$):

`Inductive` *trans* $\{A\}$ (R: *Rel* A) : *Rel* A :=
| *singl*: $\forall$ a b, R a b $\to$ *trans* R a b
| *transit*: $\forall$ b a c, R a b $\to$ *trans* R b c $\to$ *trans* R a c.

The reflexive transitive closure of $\to_R$, written $\twoheadrightarrow_R$, is defined by:

$$\frac{a \to_R b}{a \twoheadrightarrow_R b}\ (refl) \qquad\qquad \frac{a \to_R b \qquad b \twoheadrightarrow_R c}{a \twoheadrightarrow_R c}\ (rtrans)$$

This definition corresponds to the following Coq code, where $\twoheadrightarrow_R$ is written as *refltrans R*:

`Inductive` *refltrans* $\{A$:`Type`$\}$ (R: *Rel* A) : $A \to A \to$ `Prop` :=
| *refl*: $\forall$ a, *refltrans* R a a
| *rtrans*: $\forall$ a b c, R a b $\to$ *refltrans* R b c $\to$ *refltrans* R a c.

The reflexive transitive closure of a relation is used to define the notion of confluence: no matter how the reduction is done, the result will always be the same. In other words, every divergence is joinable as stated by the following diagram:

$$\begin{tikzcd}
 & a \arrow[dl, twoheadrightarrow] \arrow[dr, twoheadrightarrow] & \\
b \arrow[dr, dotted, twoheadrightarrow] & & c \arrow[dl, dotted, twoheadrightarrow] \\
 & d &
\end{tikzcd}$$

Formally, this means that if an expression a can be reduced in two different ways to b and c, then there exists an expression d such that both b and c reduce to d. The existential quantification is expressed by the dotted lines in the diagram. This notion is defined in the Coq system as follows:

`Definition` *Confl* $\{A$:`Type`$\}$ (R: *Rel* A) := $\forall$ a b c, (*refltrans* R) a b $\to$ (*refltrans* R) a c $\to$ ($\exists$ d, (*refltrans* R) b d $\land$ (*refltrans* R) c d).

In [?], V. van Oostrom gives a sufficient condition for an ARS to be confluent. This condition is based on the *Z Property* that is defined as follows:

Definition 1. *Let* $(A,\to_R)$ *be an ARS. A mapping* $f : A \to A$ *satisfies the Z property for* $\to_R$, *if* $a \to_R b$ *implies* $b \twoheadrightarrow_R fa \twoheadrightarrow_R fb$, *for any* $a, b \in A$.

The name of the property comes from the following diagrammatic representation of this definition:

$$\begin{tikzcd}
a \arrow[r, "R"'] & b \arrow[dl, dotted, twoheadrightarrow, "R"] \\
fa \arrow[r, dotted, twoheadrightarrow, "R"'] & fb
\end{tikzcd}$$

If a function f satisfies the Z property for $\to_R$ *then we say that f is Z for* $\to_R$, *and the corresponding Coq definition is given by the following predicate:*

Definition f_is_Z {A:Type} (R: Rel A) (f: A → A) := $\forall$ *a b, R a b* $\to$ *((refltrans R) b (f a)* $\land$ *(refltrans R) (f a) (f b)).*

Alternatively, an ARS $(A,\to_R)$ *satisfies the Z property if there exists a mapping* $f : A \to A$ *such that f is Z for* $\to_R$:

Definition Z_prop {A:Type} (R: Rel A) := $\exists$ *f:A* $\to$ *A,* $\forall$ *a b, R a b* $\to$ *((refltrans R) b (f a)* $\land$ *(refltrans R) (f a) (f b)).*

The first contribution of this work is a constructive proof of the fact that the Z property implies confluence. Our proof uses nested induction, and hence it differs from the one in [?] *(that follows* [?]*) and the one in* [?] *in the sense that it does not rely on the analyses of whether a term is in normal form or not, avoiding the* *necessity of the law of the excluded middle. As a result, we have an elegant inductive proof of the fact that if an ARS satisfies the Z property then it is confluent.*

```
Theorem Z_prop_implies_Confl {A:Type}: ∀ R: Rel A, Z_prop R → Confl R.
Proof.
  intros R HZ_prop.
  unfold Z_prop, Confl in *.
  intros a b c Hrefl1 Hrefl2.
  destruct HZ_prop as [g HZ_prop].
  generalize dependent c.
  induction Hrefl1.
  - intros c Hrefl2.
    ∃ c; split.
    + assumption.
    + apply refl.
  - intros c0 Hrefl2.
    generalize dependent c.
    induction Hrefl2.
    + admit.
    + intros c' H2 IH1.
    generalize dependent b.

    assert (Hbga: refltrans R b (g a)).
    { apply HZ_prop; assumption. }
    assert (Haga: refltrans R a (g a)).
    { apply rtrans with b; assumption. }
    clear H. generalize dependent b.
    induction Hrefl2.
    + intros b Hrefl1 IHHrefl1 Hbga.
      assert (IHHrefl1_ga := IHHrefl1 (g a));
        apply IHHrefl1_ga in Hbga.
      destruct Hbga.
      ∃ x; split.
      × apply H.
      × apply refltrans_composition with (g a); [assumption | apply H].
    + intros b0 Hrefl1 IHHrefl1 Hb0ga.
      apply IHHrefl2 with b0.
      × apply refltrans_composition with (g a); apply HZ_prop; assumption.
      × assumption.
      × assumption.
      × apply refltrans_composition with (g a); [ assumption | apply HZ_prop; assumption].
Qed.
```

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

```
Theorem Z_prop_implies_SemiConfl {A:Type}: ∀ R: Rel A, Z_prop R → SemiConfl R.
Proof.
  intros R HZ_prop.
  unfold Z_prop in HZ_prop.
  unfold SemiConfl.
  destruct HZ_prop.
  intros a b c Hrefl Hrefl'.
  assert (Haxa: refltrans R a (x a)).
  { apply rtrans with b. - assumption. - apply H. assumption. }
  apply H in Hrefl.
  destruct Hrefl.
  clear H1.
  generalize dependent b.
  induction Hrefl'.
  - intros.
    ∃ (x a).
    split; assumption.
  - intros.
    destruct IHHrefl' with b0.
    + apply refltrans_composition with (x a); apply H; assumption.
    + apply refltrans_composition with (x b).
      × apply refltrans_composition with (x a).
        ** assumption.
        ** apply H.
           assumption.
      × apply refl.
    + ∃ x0.
      assumption.
Qed.

Theorem Semi_equiv_Confl {A: Type}: ∀ R: Rel A, Confl R ↔ SemiConfl R.
Proof.
  unfold Confl.
  unfold SemiConfl.
  intro R.
  split.
  - intros.
    apply H with a.
    + apply rtrans with b.
      × assumption.
      × apply refl.
    + assumption.
  - intros.
    generalize dependent c.
    induction H0.
    + intros.
      ∃ c.
      split.
      × assumption.
      × apply refl.
    + intros.
      specialize (H a).
      specialize (H b).
      specialize (H c0).
      apply H in H0.
      × destruct H0.
        destruct H0.
        apply IHrefltrans in H0.
        destruct H0.
        destruct H0.
```

```
          ∃ x0.
          split.
          ** assumption.
          ** apply refltrans_composition with x; assumption.
        × assumption.
Qed.
```

```
Corollary Zprop_implies_Confl_via_SemiConfl {A:Type}: ∀ R: Rel A, Z_prop R → Confl R.
Proof. intros R HZ_prop. apply Semi_equiv_Confl. generalize dependent HZ_prop.
       apply Z_prop_implies_SemiConfl. Qed.
```

3 An extension of the Z property: Compositional Z

```
Definition f_is_weak_Z {A} (R R': Rel A) (f: A → A) := ∀ a b, R a b → ((refltrans R') b (f a) ∧ (refltrans
R') (f a) (f b)).

Definition comp {A} (f1 f2: A → A) := fun x:A ⇒ f1 (f2 x).
Notation "f1 # f2" := (comp f1 f2) (at level 40).

Inductive union {A} (red1 red2: Rel A) : Rel A :=
| union_left: ∀ a b, red1 a b → union red1 red2 a b
| union_right: ∀ a b, red2 a b → union red1 red2 a b.
Notation "R1 !_! R2" := (union R1 R2) (at level 40).

Lemma union_or {A}: ∀ (r1 r2: Rel A) (a b: A), (r1 !_! r2) a b ↔ (r1 a b) ∨ (r2 a b).
Proof.
  intros r1 r2 a b; split.
  - intro Hunion.
    inversion Hunion; subst.
    + left; assumption.
    + right; assumption.
  - intro Hunion.
    inversion Hunion.
    + apply union_left; assumption.
    + apply union_right; assumption.
Qed.

Require Import Setoid.
Require Import ZArith.

Lemma equiv_refltrans {A}: ∀ (R R1 R2: Rel A), (∀ x y, R x y ↔ (R1 !_! R2) x y) → ∀ x y, refltrans (R1
!_! R2) x y → refltrans R x y.
Proof.
  intros.
  induction H0.
  - apply refl.
  - apply rtrans with b.
    + apply H. assumption.
    + assumption.
  Qed.

Definition Z_comp {A:Type} (R :Rel A) := ∃ (R1 R2: Rel A) (f1 f2: A → A), (∀ x y, R x y ↔ (R1 !_!
R2) x y) ∧ f_is_Z R1 f1 ∧ (∀ a b, R1 a b → (refltrans R) ((f2 # f1) a) ((f2 # f1) b)) ∧ (∀ a b, b = f1 a →
(refltrans R) b (f2 b)) ∧ (f_is_weak_Z R2 R (f2 # f1)).

Lemma refltrans_union {A:Type}: ∀ (R R' :Rel A) (a b: A), refltrans R a b → refltrans (R !_! R') a b.
```

```
Proof.
  intros R R' a b Hrefl.
  induction Hrefl.
  - apply refl.
  - apply rtrans with b.
    + apply union_left. assumption.
    + assumption.
Qed.
```

```
Require Import Setoid.
Lemma refltrans_union_equiv {A}: ∀ (R R1 R2 : Rel A), (∀ (x y : A), (R x y ↔ (R1 !_! R2) x y)) → ∀ (x
y: A), refltrans (R1 !_! R2) x y → refltrans R x y.
Proof.
  intros.
  induction H0.
  + apply refl.
  + apply rtrans with b.
    - apply H. assumption.
    - assumption.
Qed.
```

```
Theorem Z_comp_implies_Z_prop {A:Type}: ∀ (R :Rel A), Z_comp R → Z_prop R.
Proof.
  intros R H.
  unfold Z_prop. unfold Z_comp in H. destruct H as
  [ R1 [ R2 [f1 [f2 [Hunion [H1 [H2 [H3 H4]]]]]]]].
  ∃ (f2 # f1).
  intros a b HR.
  apply Hunion in HR. inversion HR; subst. clear HR.
  - split.
    + apply refltrans_composition with (f1 a).
      × apply H1 in H.
        destruct H as [Hb Hf].
        apply (refltrans_union R1 R2) in Hb.
        apply refltrans_union_equiv with R1 R2.
        ** apply Hunion.
        ** apply Hb.
      × apply H3 with a; reflexivity.
    + apply H2; assumption.
  - apply H4; assumption.
Qed.
```

Now we can use the proofs of the theorems Z_comp_implies_Z_prop and Z_prop_implies_Confl to conclude that compositional Z is a sufficient condition for confluence.

```
Corollary Z_comp_is_Confl {A}: ∀ (R: Rel A), Z_comp R → Confl R.
Proof.
  intros R H.
  apply Z_comp_implies_Z_prop in H.
  apply Z_prop_implies_Confl; assumption.
Qed.
```

```
Theorem Z_comp_thm {A:Type}: ∀ (R :Rel A) (R1 R2: Rel A) (f1 f2: A → A), (∀ x y, R x y ↔ (R1 !_!
R2) x y) ∧ f_is_Z R1 f1 ∧ (∀ a b, R1 a b → (refltrans R) ((f2 # f1) a) ((f2 # f1) b)) ∧ (∀ a b, b = f1 a →
(refltrans R) b (f2 b)) ∧ (f_is_weak_Z R2 R (f2 # f1)) → f_is_Z R (f2 # f1).
```

```
Proof.
  intros R R1 R2 f1 f2 H.
  destruct H as [Hunion [H1 [H2 [H3 H4]]]].
  unfold f_is_Z.
  intros a b Hab.
  apply Hunion in Hab.
  inversion Hab; subst. clear Hab; split.
  - apply refltrans_composition with (f1 a).
    assert (Hbf1a: refltrans (R1 !_! R2) b (f1 a)).
    { apply refltrans_union. apply H1; assumption. }
    apply equiv_refltrans with R1 R2.
    + assumption.
    + assumption.
    + apply H3 with a; reflexivity.
  - unfold comp.
    assert (H' := H).
    apply H1 in H.
    destruct H as [H Hf1].
    clear H.
    apply H2; assumption.
  - apply H4; assumption.
Qed.
```

```
Corollary Z_comp_eq_corol {A:Type}: ∀ (R :Rel A) (R1 R2: Rel A) (f1 f2: A → A), (∀ x y, R x y ↔ (R1
!_! R2) x y) ∧ (∀ a b, R1 a b → (f1 a) = (f1 b)) ∧ (∀ a, (refltrans R1) a (f1 a)) ∧ (∀ b a, a = f1 b →
(refltrans R) a (f2 a)) ∧ (f_is_weak_Z R2 R (f2 # f1)) → f_is_Z R (f2 # f1).
Proof.
  intros R R1 R2 f1 f2 H.
  destruct H as [Hunion [H1 [H2 [H3 H4]]]].
  pose proof (Z_comp_thm := Z_comp_thm R R1 R2 f1 f2).
  apply Z_comp_thm. split.
  - assumption.
  - split.
    + unfold f_is_Z.
      intros a b Hab. split.
      × apply H1 in Hab.
        rewrite Hab.
        apply H2.
      × apply H1 in Hab.
        rewrite Hab.
        apply refl.
    + split.
      × intros a b Hab.
        unfold comp.
        apply H1 in Hab.
        rewrite Hab.
        apply refl.
      × split; assumption.
Qed.
```

```
Definition Z_comp_eq {A:Type} (R :Rel A) := ∃ (R1 R2: Rel A) (f1 f2: A → A), (∀ x y, R x y ↔ (R1 !_!
R2) x y) ∧ (∀ a b, R1 a b → (f1 a) = (f1 b)) ∧ (∀ a, (refltrans R1) a (f1 a)) ∧ (∀ b a, a = f1 b → (refltrans
R) a (f2 a)) ∧ (f_is_weak_Z R2 R (f2 # f1)).
```

```
Lemma Z_comp_eq_implies_Z_comp {A:Type}: ∀ (R : Rel A), Z_comp_eq R → Z_comp R.
Proof.
  intros R Heq. unfold Z_comp_eq in Heq.
  destruct Heq as [R1 [R2 [f1 [f2 [Hunion [H1 [H2 [H3 H4]]]]]]]].
  unfold Z_comp.
  ∃ R1, R2, f1, f2.
  split.
  - assumption.
  - split.
    + unfold f_is_Z.
      intros a b H; split.
      × apply H1 in H. rewrite H. apply H2.
      × apply H1 in H. rewrite H. apply refl.
    + split.
      × intros a b H.
        unfold comp.
        apply H1 in H.
        rewrite H.
        apply refl.
      × split; assumption.
Qed.
```

```
Lemma Z_comp_eq_implies_Z_prop {A:Type}: ∀ (R : Rel A), Z_comp_eq R → Z_prop R.
Proof.
  intros R Heq.
  unfold Z_comp_eq in Heq.
  destruct Heq as [R1 [R2 [f1 [f2 [Hunion [H1 [H2 [H3 H4]]]]]]]].
  unfold Z_prop. ∃ (f2 # f1).
  intros a b Hab.
  split.
  - apply Hunion in Hab.
    inversion Hab; subst.
    + unfold comp.
      apply H1 in H. rewrite H.
      apply refltrans_composition with (f1 b).
      × assert (H5: refltrans R1 b (f1 b)).
        {
          apply H2.
        }
        apply refltrans_union_equiv with R1 R2.
        ** assumption.
        ** apply refltrans_union. assumption.
      × apply H3 with b. reflexivity.
    + apply H4. assumption.
  - apply Hunion in Hab.
    inversion Hab; subst.
    + unfold comp. apply H1 in H. rewrite H. apply refl.
    + apply H4. assumption.
Qed.
```

4 Conclusion

In this work we presented a constructive proof that the Z property implies confluence, an important property for rewriting systems. In addition, we formally proved this result in the Coq proof assistant. The corresponding files are available in our GitHub repository: `https://github.com/flaviodemoura/Zproperty`.

The Z property was presented by V. van Oostrom as a sufficient condition for an ARS to be confluent [1], and since then has been used to prove confluence in different contexts such as the λ-calculus with $\beta\eta$-reduction, extensions of the λ-calculus with explicit substitutions and the $\lambda\mu$-calculus. The Coq proofs of the main results are commented line by line which serve both as an informal presentation of the proofs (i.e. proofs explained in natural language) and as its formal counterpart. Moreover, we formalize an extension of the Z property, known as compositional Z property, as presented in [3]

As future work, this formalization will be used to prove the confluence property of a calculus with explicit substitution based on the λ_{ex}-calculus (cf. [2]). In addition, we hope that our formalization can be used as a framework for proving confluence of others rewriting systems.

References

1. *B. Felgenhauer, J. Nagele, V. van Oostrom, and C. Sternagel. The Z property.* Archive of Formal Proofs*, 2016.*
2. *D. Kesner. A Theory of Explicit Substitutions with Safe and Full Composition.* Logical Methods in Computer Science*, 5(3:1):1–29, 2009.*
3. *Koji Nakazawa and Ken-etsu Fujita. Compositional Z: confluence proofs for permutative conversion.* Studia Logica*, 104(6):1205–1224, 2016.*