

# The Z Property

Bertram Felgenhauer, Julian Nagele, Vincent van Oostrom, Christian Sternagel\*

December 14, 2021

## Abstract

We formalize the Z property introduced by Dehornoy and van Oostrom [1]. First we show that for any abstract rewrite system, Z implies confluence. Then we give two examples of proofs using Z: confluence of lambda-calculus with respect to beta-reduction and confluence of combinatory logic.

## Contents

<b>1</b>	<b>The Z property</b>	<b>1</b>
<b>2</b>	<b>Lambda Calculus has the Church-Rosser property</b>	<b>3</b>
2.1	Ad-hoc methods for nominal-functions over lambda terms . . .	3
2.2	Substitutions . . . . .	4
<b>3</b>	<b>Combinatory Logic has the Church-Rosser property</b>	<b>9</b>

## 1 The Z property

```
theory Z
imports Abstract-Rewriting.Abstract-Rewriting
begin

locale z-property =
  fixes bullet :: 'a ⇒ 'a (-• [1000] 1000)
  and R :: 'a rel
  assumes Z: (a, b) ∈ R ⇒ (b, a•) ∈ R* ∧ (a•, b•) ∈ R*
begin

lemma monotonicity:
  assumes (a, b) ∈ R*
  shows (a•, b•) ∈ R*
```

---

\*This work was partially supported by FWF (Austrian Science Fund) projects P27502 and P27528.

**using** *assms*  
**by** (*induct*) (*auto dest: Z*)

**lemma** *semi-confluence*:

**shows**  $(R^{-1} \circ R^*) \subseteq R^\downarrow$

**proof** (*intro subrelI, elim relcompEpair, drule converseD*)

**fix** *d a c*

**assume**  $(a, c) \in R^*$  **and**  $(a, d) \in R$

**then show**  $(d, c) \in R^\downarrow$

**proof** (*cases*)

**case** (*step b*)

**then have**  $(a^\bullet, b^\bullet) \in R^*$  **by** (*auto simp: monotonicity*)

**then have**  $(d, b^\bullet) \in R^*$  **using**  $\langle (a, d) \in R \rangle$  **by** (*auto dest: Z*)

**then show** *?thesis* **using**  $\langle (b, c) \in R \rangle$  **by** (*auto dest: Z*)

**qed** *auto*

**qed**

**lemma** *CR*: *CR R*

**by** (*intro semi-confluence-imp-CR semi-confluence*)

**definition**  $R_d = \{(a, b). (a, b) \in R^* \wedge (b, a^\bullet) \in R^*\}$

**end**

**locale** *angle-property* =

**fixes** *bullet* ::  $'a \Rightarrow 'a (-\bullet [1000] 1000)$

**and** *R* ::  $'a \text{ rel}$

**and**  $R_d$  ::  $'a \text{ rel}$

**assumes** *intermediate*:  $R \subseteq R_d \ R_d \subseteq R^*$

**and** *angle*:  $(a, b) \in R_d \implies (b, a^\bullet) \in R_d$

**sublocale** *angle-property*  $\subseteq$  *z-property*

**proof**

**fix** *a b*

**assume**  $(a, b) \in R$

**with** *angle intermediate* **have**  $(b, a^\bullet) \in R_d$  **and**  $(a^\bullet, b^\bullet) \in R_d$  **by** *auto*

**then show**  $(b, a^\bullet) \in R^* \wedge (a^\bullet, b^\bullet) \in R^*$  **using** *intermediate* **by** *auto*

**qed**

**sublocale** *z-property*  $\subseteq$  *angle-property bullet R z-property.R\_d bullet R*

**proof**

**show**  $R \subseteq R_d$  **and**  $R_d \subseteq R^*$  **unfolding** *R\_d-def* **using** *Z* **by** *auto*

**fix** *a b*

**assume**  $(a, b) \in R_d$

**then show**  $(b, a^\bullet) \in R_d$  **using** *monotonicity* **unfolding** *R\_d-def* **by** *auto*

**qed**

**end**

## 2 Lambda Calculus has the Church-Rosser property

```

theory Lambda-Z
imports
  Nominal2.Nominal2
  HOL-Eisbach.Eisbach
  Z
begin

atom-decl name

nominal-datatype term =
  Var name
| App term term
| Abs x::name t::term binds x in t

```

### 2.1 Ad-hoc methods for nominal-functions over lambda terms

```

ML <
fun graph-aux-tac ctxt =
  SUBGOAL (fn (subgoal, i) =>
    (case subgoal of
      Const (@{const-name Trueprop}, -) $ (Const (@{const-name eqvt}, -) $ Free
(f, -)) =>
        full-simp-tac (
          ctxt addsimps [@{thm eqvt-def}, Proof-Context.get-thm ctxt (f ^ -def)] i
          | - => no-tac))
    )
  >

method-setup eqvt-graph-aux =
  <Scan.succeed (fn ctxt : Proof.context => SIMPLE-METHOD' (graph-aux-tac
ctxt))>
  show equivariance of auxilliary graph construction for nominal functions

method without-alpha-lst methods m =
  (match termI in H [simproc del: alpha-lst]: - => <m>)

method Abs-lst =
  (match premises in
    atom ?x ‡ c and P [thin]: [[atom -]]lst. - = [[atom -]]lst. - for c :: 'a::fs =>
      <rule Abs-lst1-fcb2' [where c = c, OF P]>
    | P [thin]: [[atom -]]lst. - = [[atom -]]lst. - => <rule Abs-lst1-fcb2' [where c = (),
OF P]>
  )

method pat-comp-aux =
  (match premises in
    x = (- :: term) => - for x => <rule term.strong-exhaust [where y = x and c =
x]>
  )

```

$| x = (\text{Var } -, -) \implies - \text{ for } x :: - :: fs \Rightarrow$   
 $\langle \text{rule term.strong-exhaust [where } y = \text{fst } x \text{ and } c = x] \rangle$   
 $| x = (-, \text{Var } -) \implies - \text{ for } x :: - :: fs \Rightarrow$   
 $\langle \text{rule term.strong-exhaust [where } y = \text{snd } x \text{ and } c = x] \rangle$   
 $| x = (-, -, \text{Var } -) \implies - \text{ for } x :: - :: fs \Rightarrow$   
 $\langle \text{rule term.strong-exhaust [where } y = \text{snd } (\text{snd } x) \text{ and } c = x] \rangle$

**method** *pat-comp* = (*pat-comp-aux*; *force simp: fresh-star-def fresh-Pair-elim*)

**method** *freshness uses fresh* =  
 (**match conclusion in**  
 $- \# - \Rightarrow \langle \text{simp add: fresh-Unit fresh-Pair fresh} \rangle$   
 $| - \#* - \Rightarrow \langle \text{simp add: fresh-star-def fresh-Unit fresh-Pair fresh} \rangle$ )

**method** *solve-eqv-at* =  
 (*simp add: eqv-at-def*; *simp add: perm-supp-eq fresh-star-Pair*)+

**method** *nf uses fresh* = *without-alpha-1st*  $\langle$   
*eqv-graph-aux*, *rule TrueI*, *pat-comp*, *auto*, *Abs-1st*,  
*auto simp: Abs-fresh-iff pure-fresh perm-supp-eq*,  
*(freshness fresh: fresh)*+,  
*solve-eqv-at?* $\rangle$

## 2.2 Substitutions

**nominal-function** *subst*

**where**

$\text{subst } x \ s \ (\text{Var } y) = (\text{if } x = y \text{ then } s \text{ else } \text{Var } y)$   
 $| \text{subst } x \ s \ (\text{App } t \ u) = \text{App} \ (\text{subst } x \ s \ t) \ (\text{subst } x \ s \ u)$   
 $| \text{atom } y \ \# \ (x, s) \implies \text{subst } x \ s \ (\text{Abs } y \ t) = \text{Abs } y \ (\text{subst } x \ s \ t)$   
**by** *nf*

**nominal-termination** (*eqv*) **by** *lexicographic-order*

**lemma** *fresh-subst*:

$\text{atom } z \ \# \ s \implies z = y \vee \text{atom } z \ \# \ t \implies \text{atom } z \ \# \ \text{subst } y \ s \ t$   
**by** (*nominal-induct t avoiding: z y s rule: term.strong-induct*) *auto*

**lemma** *fresh-subst-id* [*simp*]:

$\text{atom } x \ \# \ t \implies \text{subst } x \ s \ t = t$   
**by** (*nominal-induct t avoiding: x s rule: term.strong-induct*) (*auto simp: fresh-at-base*)

The substitution lemma.

**lemma** *subst-subst*:

**assumes**  $x \neq y$  **and**  $\text{atom } x \ \# \ u$   
**shows**  $\text{subst } y \ u \ (\text{subst } x \ s \ t) = \text{subst } x \ (\text{subst } y \ u \ s) \ (\text{subst } y \ u \ t)$   
**using** *assms* **by** (*nominal-induct t avoiding: x y u s rule: term.strong-induct*) (*auto simp: fresh-subst*)

**inductive-set** *Beta* ( $\{\rightarrow_\beta\}$ )

**where**

*root*:  $atom\ x \# t \implies (App\ (Abs\ x\ s)\ t,\ subst\ x\ t\ s) \in \{\rightarrow_\beta\}$   
| *Appl*:  $(s,\ t) \in \{\rightarrow_\beta\} \implies (App\ s\ u,\ App\ t\ u) \in \{\rightarrow_\beta\}$   
| *Appr*:  $(s,\ t) \in \{\rightarrow_\beta\} \implies (App\ u\ s,\ App\ u\ t) \in \{\rightarrow_\beta\}$   
| *Abs*:  $(s,\ t) \in \{\rightarrow_\beta\} \implies (Abs\ x\ s,\ Abs\ x\ t) \in \{\rightarrow_\beta\}$

**abbreviation** *beta*  $((-/ \rightarrow_\beta -) [56, 56] 55)$

**where**

$s \rightarrow_\beta t \equiv (s,\ t) \in \{\rightarrow_\beta\}$

**equivariance** *Betap*

**lemmas** *Beta-eqvt* = *Betap.eqvt* [*to-set*]

**nominal-inductive** *Betap*

**avoids** *Abs*:  $x$

| *root*:  $x$

**by** (*simp-all add: fresh-star-def fresh-subst*)

**lemmas** *Beta-strong-induct* = *Betap.strong-induct* [*to-set*]

**abbreviation** *betas* (**infix**  $\rightarrow_{\beta^*}$  50)

**where**

$s \rightarrow_{\beta^*} t \equiv (s,\ t) \in \{\rightarrow_\beta\}^*$

**nominal-function** *app-beta* :: *term*  $\Rightarrow$  *term*  $\Rightarrow$  *term*

**where**

$atom\ x \# u \implies app\text{-}beta\ (Abs\ x\ s')\ u = subst\ x\ u\ s'$

|  $app\text{-}beta\ (Var\ x)\ u = App\ (Var\ x)\ u$

|  $app\text{-}beta\ (App\ s\ t)\ u = App\ (App\ s\ t)\ u$

**by** (*nf fresh: fresh-subst*)

**nominal-termination** (*eqvt*) **by** *lexicographic-order*

**nominal-function** *bullet* :: *term*  $\Rightarrow$  *term* ( $-^\bullet$  [1000] 1000)

**where**

$(Var\ x)^\bullet = Var\ x$

|  $(Abs\ x\ t)^\bullet = Abs\ x\ t^\bullet$

|  $(App\ s\ t)^\bullet = app\text{-}beta\ s^\bullet\ t^\bullet$

**by** *nf*

**nominal-termination** (*eqvt*) **by** *lexicographic-order*

**lemma** *app-beta-exhaust* [*case-names Redex no-Redex*]:

**fixes**  $c :: 'a :: fs$

**assumes**  $\bigwedge x\ s'. atom\ x \# c \implies s = Abs\ x\ s' \implies thesis$

**and**  $(\bigwedge t. app\text{-}beta\ s\ t = App\ s\ t) \implies thesis$

**shows** *thesis*

**by** (*cases s rule: term.strong-exhaust [of - - c]*) (*auto simp: fresh-star-def fresh-Pair intro: assms*)

**lemma** *App-Betas*:

```

assumes  $s \rightarrow_{\beta^*} t$  and  $u \rightarrow_{\beta^*} v$ 
shows  $App\ s\ u \rightarrow_{\beta^*} App\ t\ v$ 
using assms(1)
proof (induct)
  case base
  show ?case using assms(2) by (induct) (auto intro: Beta.intros rtrancl-into-rtrancl)
qed (auto intro: Beta.intros rtrancl-into-rtrancl)

```

```

lemma Abs-Betas:
  assumes  $s \rightarrow_{\beta^*} t$ 
  shows  $Abs\ x\ s \rightarrow_{\beta^*} Abs\ x\ t$ 
using assms by (induct) (auto intro: Beta.intros rtrancl-into-rtrancl)

```

```

lemma self:
   $t \rightarrow_{\beta^*} t^\bullet$ 
proof (nominal-induct t rule: term.strong-induct)
  case (App t u)
  then show ?case
    by (cases t• rule: app-beta-exhaust[of u•])
      (auto intro: App-Betas Beta.intros rtrancl-into-rtrancl)
qed (auto intro: Abs-Betas)

```

```

lemma fresh-atom-bullet:
   $atom\ (x::name) \# t \implies atom\ x \# t^\bullet$ 
proof (nominal-induct t avoiding: x rule: term.strong-induct)
  case (App t u)
  then show ?case by (cases t• rule: app-beta-exhaust[of u•]) (auto intro: fresh-subst)
qed auto

```

```

lemma subst-Beta:
  assumes  $t \rightarrow_{\beta} t'$ 
  shows  $subst\ x\ s\ t \rightarrow_{\beta} subst\ x\ s\ t'$ 
using assms
proof (nominal-induct avoiding: x s rule: Beta-strong-induct)
  case (root y t u)
  then show ?case by (auto simp: subst-subst fresh-subst intro: Beta.root)
qed (auto intro: Beta.intros)

```

```

lemma Beta-in-subst:
  assumes  $s \rightarrow_{\beta} s'$ 
  shows  $subst\ x\ s\ t \rightarrow_{\beta^*} subst\ x\ s'\ t$ 
using assms
by (nominal-induct t avoiding: x s s' rule: term.strong-induct)
  (auto intro: App-Betas Abs-Betas)

```

```

lemma subst-Betas:
  assumes  $s \rightarrow_{\beta^*} s'$  and  $t \rightarrow_{\beta^*} t'$ 
  shows  $subst\ x\ s\ t \rightarrow_{\beta^*} subst\ x\ s'\ t'$ 
using assms(1)

```

**proof** (*induct*)  
**case** *base*  
**from** *assms*(2) **show** ?*case* **by** (*induct*) (*auto simp: subst-Beta intro: rtrancl-into-rtrancl*)  
**next**  
**case** (*step u v*)  
**from** *Beta-in-subst* [*OF this*(2), *of x t*] **and** *this*(3) **show** ?*case* **by** *auto*  
**qed**

**lemma** *Beta-fresh*:  
**fixes** *x :: name*  
**assumes**  $s \rightarrow_{\beta} t$  **and** *atom x*  $\#$  *s*  
**shows** *atom x*  $\#$  *t*  
**using** *assms* **by** (*nominal-induct rule: Beta-strong-induct*) (*auto simp: fresh-subst*)

**lemma** *Abs-BetaD*:  
**assumes**  $Abs\ x\ s \rightarrow_{\beta} t$   
**shows**  $\exists u. t = Abs\ x\ u \wedge s \rightarrow_{\beta} u$   
**using** *assms*  
**proof** (*nominal-induct Abs x s t avoiding: s rule: Beta-strong-induct*)  
**case** (*Abs u v y*)  
**then show** ?*case*  
**by** (*auto simp: Abs1-eq-iff Beta-fresh fresh-permute-left intro!: exI [of - (y  $\leftrightarrow$  x)  $\cdot$  v]*)  
*(metis Abs1-eq-iff(3) Beta-eqvt flip-commute)*  
**qed**

**lemma** *Abs-BetaE*:  
**assumes**  $Abs\ x\ s \rightarrow_{\beta} t$   
**obtains** *u* **where**  $t = Abs\ x\ u$  **and**  $s \rightarrow_{\beta} u$   
**using** *assms* **by** (*blast dest: Abs-BetaD*)

**lemma** *Abs-BetasE*:  
**assumes**  $Abs\ x\ s \rightarrow_{\beta^*} t$   
**obtains** *u* **where**  $t = Abs\ x\ u$  **and**  $s \rightarrow_{\beta^*} u$   
**using** *assms* **by** (*induct Abs x s t*) (*auto elim: Abs-BetaE intro: rtrancl-into-rtrancl*)

**lemma** *bullet-App*:  
 $(App\ s^{\bullet}\ t^{\bullet}, (App\ s\ t)^{\bullet}) \in \{\rightarrow_{\beta}\}^=$   
**by** (*cases s<sup>•</sup> rule: term.strong-exhaust[of - - t<sup>•</sup>]*)  
*(auto simp: fresh-star-def intro: Beta.root)*

**lemma** *rhs*:  
 $subst\ x\ s^{\bullet}\ t^{\bullet} \rightarrow_{\beta^*} (subst\ x\ s\ t)^{\bullet}$   
**proof** (*nominal-induct t avoiding: x s rule: term.strong-induct*)  
**case** (*App t<sub>1</sub> t<sub>2</sub>*)  
**show** ?*case*  
**proof** (*cases t<sub>1</sub><sup>•</sup> rule: app-beta-exhaust[of (x, t<sub>2</sub>, s)]*)  
**case** (*Redex y u*)  
**then have**  $Abs\ y\ (subst\ x\ s^{\bullet}\ u) \rightarrow_{\beta^*} (subst\ x\ s\ t_1)^{\bullet}$

**using** *App(1)* [of *x s*] **by** (*simp add: fresh-star-def fresh-atom-bullet*)  
**with** *Abs-BetasE* **obtain** *v* **where**  $(\text{subst } x \ s \ t_1)^\bullet = \text{Abs } y \ v$  **and**  $\text{subst } x \ s^\bullet \ u \rightarrow_{\beta^*} v$  **by** *blast*  
**then show** *?thesis* **using** *subst-subst* **and** *subst-Betas* **and** *App(2)* **and** *Redex*  
**by** (*simp add: fresh-atom-bullet fresh-subst*)  
**next**  
**case** (*no-Redex*)  
**then have**  $\text{subst } x \ s^\bullet \ ((\text{App } t_1 \ t_2)^\bullet) \rightarrow_{\beta^*} \text{App } ((\text{subst } x \ s \ t_1)^\bullet) \ ((\text{subst } x \ s \ t_2)^\bullet)$   
**using** *App* **by** (*auto intro: App-Betas*)  
**then show** *?thesis* **using** *bullet-App* **by** (*force intro: rtrancl-into-rtrancl*)  
**qed**  
**qed** (*force dest: fresh-atom-bullet intro: Abs-Betas*)+

**lemma** *Betas-fresh*:  
**fixes** *x :: name*  
**assumes**  $s \rightarrow_{\beta^*} t$  **and** *atom x # s*  
**shows** *atom x # t*  
**using** *assms* **by** (*induct*) (*auto dest: Beta-fresh*)

**lemma** *Var-BetaD*:  
**assumes**  $\text{Var } x \rightarrow_{\beta} t$   
**shows** *False*  
**using** *assms* **by** (*induct Var x t*)

**lemma** *Var-BetasD*:  
**assumes**  $\text{Var } x \rightarrow_{\beta^*} t$   
**shows**  $t = \text{Var } x$   
**using** *assms* **by** (*induct*) (*auto dest: Var-BetaD*)

**lemma** *app-beta-Betas*:  
**assumes**  $s \rightarrow_{\beta^*} s'$  **and**  $t \rightarrow_{\beta^*} t'$   
**shows**  $\text{app-beta } s \ t \rightarrow_{\beta^*} \text{app-beta } s' \ t'$   
**using** *assms*  
**proof** (*cases s rule: term.strong-exhaust [of - - t]*)  
**case** (*App s<sub>1</sub> s<sub>2</sub>*)  
**with** *assms* **show** *?thesis*  
**by** (*cases s' rule: app-beta-exhaust [of t']*) (*auto intro: root rtrancl-into-rtrancl App-Betas*)  
**qed** (*auto simp: fresh-star-def Betas-fresh subst-Betas elim: Abs-BetasE intro: App-Betas dest!: Var-BetasD*)

**lemma** *lambda-Z*:  
**assumes**  $s \rightarrow_{\beta} t$   
**shows**  $t \rightarrow_{\beta^*} s^\bullet \wedge s^\bullet \rightarrow_{\beta^*} t^\bullet$   
**using** *assms*  
**proof** (*nominal-induct rule: Beta-strong-induct*)  
**case** (*Appl s t u*)  
**then have**  $\text{App } t \ u \rightarrow_{\beta^*} \text{App } s^\bullet \ u^\bullet$  **using** *self* **by** (*auto intro: App-Betas*)  
**also have**  $\text{App } s^\bullet \ u^\bullet \rightarrow_{\beta^*} (\text{App } s \ u)^\bullet$  **using** *bullet-App* **by** *force*



**finally show** *?case using Appl by (auto intro: App-Betas app-beta-Betas)*  
**next**  
**case** (*Appr s t u*)  
**then have** *App u t  $\rightarrow_{\beta^*}$  App u<sup>•</sup> s<sup>•</sup> using self by (auto intro: App-Betas)*  
**also have** *App u<sup>•</sup> s<sup>•</sup>  $\rightarrow_{\beta^*}$  (App u s)<sup>•</sup> using bullet-App by force*  
**finally show** *?case using Appr by (auto intro: App-Betas app-beta-Betas)*  
**qed** (*auto intro: Abs-Betas subst-Betas rhs simp: self fresh-atom-bullet*)

**interpretation** *lambda-z: z-property bullet Beta*  
**by** (*standard (fact lambda-Z)*)

**end**

### 3 Combinatory Logic has the Church-Rosser property

**theory** *CL-Z imports Z*  
**begin**

**datatype** *CL = S | K | I | App CL CL (' - - [999, 999] 999)*

**inductive-set** *red :: CL rel where*

*L: (t, t')  $\in$  red  $\implies$  (' t u, ' t' u)  $\in$  red*  
*| R: (u, u')  $\in$  red  $\implies$  (' t u, ' t u')  $\in$  red*  
*| S: (' ' S x y z, ' ' x z ' y z)  $\in$  red*  
*| K: (' ' K x y, x)  $\in$  red*  
*| I: (' I x, x)  $\in$  red*

**lemma** *App-mono:*

*(t, t')  $\in$  red\*  $\implies$  (u, u')  $\in$  red\*  $\implies$  (' t u, ' t' u')  $\in$  red\**

**proof** –

**assume** *(t, t')  $\in$  red\* hence (' t u, ' t' u)  $\in$  red\**

**by** (*induct t' rule: rtrancl-induct (auto intro: rtrancl-into-rtrancl red.intros)*)

**moreover assume** *(u, u')  $\in$  red\* hence (' t' u, ' t' u')  $\in$  red\**

**by** (*induct u' rule: rtrancl-induct (auto intro: rtrancl-into-rtrancl red.intros)*)

**ultimately show** *?thesis by auto*

**qed**

**fun** *bullet-app :: CL  $\Rightarrow$  CL  $\Rightarrow$  CL where*

*bullet-app (' ' S x y) z = ' ' x z ' y z*  
*| bullet-app (' K x) y = x*  
*| bullet-app I x = x*  
*| bullet-app t u = ' t u*

**lemma** *bullet-app-red:*

*(' t u, bullet-app t u)  $\in$  red=*

**by** (*induct t u rule: bullet-app.induct (auto intro: red.intros)*)

```

lemma bullet-app-redsI:
  (s, ' t u) ∈ red* ⇒ (s, bullet-app t u) ∈ red*
using bullet-app-red[of t u] by auto

lemma bullet-app-redL:
  (t, t') ∈ red ⇒ (bullet-app t u, bullet-app t' u) ∈ red*
by (induct t u rule: bullet-app.induct)
  (auto 0 6 intro: App-mono bullet-app-redsI elim: red.cases simp only: bullet-app.simps)

lemma bullet-app-redR:
  (u, u') ∈ red ⇒ (bullet-app t u, bullet-app t u') ∈ red*
by (induct t u rule: bullet-app.induct) (auto intro: App-mono)

lemma bullet-app-mono:
  assumes (t, t') ∈ red* (u, u') ∈ red* shows (bullet-app t u, bullet-app t' u') ∈
  red*
proof –
  have (bullet-app t u, bullet-app t' u) ∈ red* using assms(1)
  by (induct t' rule: rtrancl-induct) (auto intro: rtrancl-trans bullet-app-redL)
  moreover have (bullet-app t' u, bullet-app t' u') ∈ red* using assms(2)
  by (induct u' rule: rtrancl-induct) (auto intro: rtrancl-trans bullet-app-redR)
  ultimately show ?thesis by auto
qed

fun bullet :: CL ⇒ CL where
  bullet (' t u) = bullet-app (bullet t) (bullet u)
| bullet t = t

lemma bullet-incremental:
  (t, bullet t) ∈ red*
by (induct t rule: bullet.induct) (auto intro: App-mono bullet-app-redsI)

interpretation CL:z-property bullet red
proof (unfold-locales, intro conjI)
  fix t u assume (t, u) ∈ red thus (u, bullet t) ∈ red*
  proof (induct t arbitrary: u rule: bullet.induct)
  case (1 t1 t2) show ?case using 1(3)
  by (cases) (auto intro: 1 App-mono bullet-app-redsI bullet-incremental)
  qed (auto elim: red.cases)
next
  fix t u assume (t, u) ∈ red thus (bullet t, bullet u) ∈ red*
  by (induct t u rule: red.induct) (auto intro: App-mono bullet-app-mono bullet-app-redsI)
qed

lemmas CR-red = CL.CR

end

```

## References

- [1] P. Dehornoy and V. v. Oostrom. Z, proving confluence by monotonic single-step upperbound functions. In *Logical Models of Reasoning and Computation (LMRC'2008)*, 2008.