Confluence by the Z-property for De Bruijn's λ-calculus with nameless dummies, based on PLFA[*]

Vincent van Oostrom

University of Sussex, School of Engineering and Informatics, Brighton, UK
`Vincent.van-Oostrom@sussex.ac.uk`

Abstract

We present Agda code of a proof of confluence via the Z-property, of the untyped $\lambda\beta$-calculus in De Bruijn's λ-calculus notation with nameless dummies, based on the formalisation of that calculus in the book Programming Language Foundations in Agda.

1 Introduction

Preparing the FSCD 2021 presentation of [13] on establishing confluence of rewrite systems by means of the *Z-property* for some given *bullet* function $\bullet$ (originating with Loader [12, Section 4.1] for the $\lambda\beta$-calculus with the *Gross–Knuth* bullet map, and with Dehornoy [7] for self-distributivity with the *full distribution* bullet map), I was looking for further applications of the method, and in particular for further ways[1] to test the hypothesis [13, Remark 52][2] that a proof of confluence via the Z-property should be shorter than via the *angle-property*.[3]

Recalling that the book Programming Language Foundations in Agda [18] (PLFA) had a section on confluence of the (untyped) λ-calculus, I had a closer look at it. As it turned out, the λ-calculus considered in PLFA is De Bruijn's λ-calculus with nameless dummies [5]. Moreover, the confluence proof formalised there is indeed based on the angle property with the *full development* bullet map.[4]

The above, combined with that I had wanted to learn some Agda for a long time already, that one of the PLFA authors[5] challenged me to do so, and that I had some time on my hand (in between jobs), made me decide in October 2021 to try to formalise a proof of confluence via the Z-property in Agda, of De Bruijn's $\lambda\beta$-calculus with nameless dummies, on the basis of the formalisation of that calculus in PLFA. This resulted in the Agda code in Section A below.[6] The code is only lightly commented. More comments are found in Section 2.

[*]This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by/4.0/. The Agda 2.6.1.1 code is based on the book Programming Language Foundations in Agda (version 20.07) [18], in particular on its chapters Untyped and Substitution. We developed our code in the beginning of October 2021 on a MacBook Pro 2019 (2.3 GHz 8-Core Intel Core i9, 32 GB), with Agda 2.6.1.1, Emacs 26.3 (9.0), and the snapshot of PLFA based on PLFA v20.07.

[1]Having corroborated it before for CL and the $\lambda\beta$-calculus with the *full superdevelopment* bullet map [9].

[2]Cf. the conclusions of [8].

[3]For the λ-calculus the latter property goes back to Gross, cf. [3, Section 3], for the function mapping a λ-term to the target of, what is nowadays known as, a Gross–Knuth step contracting all redexes in it, and its inductive pendant to Takahashi [16]. We isolated the property for abstract rewrite systems [17, Definition 1.1.8], dubbing it the *triangle* property at the time. Later, cf. [8] and [13, Definition 4] we rebaptised it into the *angle* property (already suggested to us by Marc Bezem in 2002) on the occasion of showing it equivalent to the Z-property [13, Lemma 8], and adopted the *bullet* name and notation $\bullet$ for the (Skolem) function involved.

[4]The full development function is the obvious generalisation of the Gross-Knuth function to the class of all orthogonal (first- and higher-order) term rewriting systems. For TRSs it is known as *full substitution*. See [13]

[5]Jeremy Siek, upon me contacting the authors of [18] in the summer of 2021.

[6]HTML source at http://www.javakade.nl/research/agda/plfa.part2.ConfluenceZ4.html (external hyperlinks do not work; see PLFA) and pure Agda code at http://www.javakade.nl/research/agda/ConfluenceZ4.agda.

2 On the formalisation

The first two paragraphs below describe the key ingredients of the Agda code from [18] we took as the basis of our formalisation, the *objects* and *steps* of the term rewrite system, and motivates why we did so. The last three paragraphs comment on what we formalised ourselves, split into: (i) supplementary code (50 loc) showing the system to be a *term rewrite system*, (ii) code (65 loc) for *the Z-property*, and (iii) code (25 loc) for *inferring confluence* from Z.

Objects of the rewrite system, from [18] The objects of the rewrite system are the λ-terms with nameless dummies of [5]. Instead of formalising these ourselves, we reused their formalisation as given in the module Untyped of [18].

That formalisation employs a completely inductive definition of *terms with bound variables*. The standard way to do so is to define them as *pairs* having a second component (the term) whose free variables are in the first component (its environment). In De Bruin's nameless setting [5] variables are natural number-indices, which allows to replace the environment by a single natural number that can be thought of as an *upper bound* on the free indices in the term.[7] This is the approach taken in the module Untyped of [18], and we reuse it.

We decided to do so as we recognised PLFA employing a particularly simple[8] case of the set-up of our [10]. In [10] environments are modelled as *stacks* of variables (allowing arbitrary repetitions). From that perspective, De Bruijn's λ-terms are not name*less* but rather have a *single* name. As a consequence, for De Bruijn's λ-terms environments are single-name stacks, allowing to reduce them to just their height, making stacks coincide with the natural-number-upper-bound-environments of PLFA. Moreover, the inference system $\vdash$ for defining terms in the module Untyped is a particularly simple[9] case of the inference system for *closed* terms of [14], Our intimate familiarity with these formal details of the formalisation of [18] made us decide to adopt it, apart from that it is of course practical and good engineering practice to reuse.

Steps of the rewrite system, from [18] The steps of the rewrite system are generated from the β-rule of [5]. Also here we reused their formalisation as given in the module Untyped of [18], which in turn is based on the module Substitution of [18], since the definition of β-steps is based on that of substitution (at meta-level). The main result of that module is `subst-commute`, known in the literature as the *Substitution Lemma* [2].[10] The proof of the Substitution Lemma given in the module Substitution is quite long *because* it is niftily indirect.[11]

Although such a long proof was at variance with our goal, we decided to reuse it anyway, because our development is modular in it, and because the Substitution Lemma is unavoidable

[7]Cf. the exercise in the module Untyped asking to prove that their formalisation of environments, called `Context`, is isomorphic to the natural numbers $\mathbb{N}$.

[8]In having no *jumps*, using the terminology of that paper.

[9]In having no *end-of-scope* operator, using the terminology of that paper.

[10]More precisely, `subst-commute` is the version for *parallel* substitutions corresponding to the *single* substitution version of [2].

[11]To allow for an algebraic proof of `subst-commute`, the authors first define the basic substitution *operations* corresponding to the substitution *operators* (known as *explicit* substitutions) of [1], or rather to those of [15], having laws that are *complete* w.r.t. so-called De Bruijn algebras, in the sense that explicit substitution expressions are equivalent iff they are provably equal by means of the laws. Then these operations are linked to substitutions, the algebraic laws are proven to hold for substitutions, and finally `subst-commute` is proven using the algebraic laws. However, that is overkill. E.g. [11] employs only 2 operations (in our terminology: *maximal* scope extrusion and substitution) and 6 laws governing their interaction to obtain the Substitution Lemma for De Bruijn's $\lambda\beta$-calculus. We recently adapted Huet's result, in Coq, to the generalised local De Bruin $\lambda\beta$-calculus with respect to *minimal* scope extrusion and substitution.

in *any* proof of confluence, also in proofs via the Z-property, since the lemma captures for *nested* redex-patterns $(\lambda y.(\lambda x.M)\,N))\,L$, the resolution of the *critical* peak of the β-rule:

$$(\lambda y.M[x:=N])\,L \leftarrow (\lambda y.(\lambda x.M)\,N))\,L \rightarrow ((\lambda x.M)\,N)[y:=L] = (\lambda x.M[y:=L])(N[y:=L])$$

in a canonical way by means of the valley, having the Substitution Lemma in its middle:

$$(\lambda y.M[x:=N])\,L \rightarrow M[x:=N][y:=L] =_{SL} M[y:=L][x:=N[y:=L]] \leftarrow (\lambda x.M[y:=L])(N[y:=L])$$

Proof that the rewrite system is a term rewrite system As a term rewrite system, the (untyped) $\lambda\beta$-calculus is a higher-order term rewrite system over a signature having two symbols, a unary (and binding) abstraction and a binary application, and a single rule β. In a term rewrite system the rewrite steps are obtained from the term rewrite rules by closing the latter under (unary term) contexts and substitutions simultaneously (in principle).

Since in the $\lambda\beta$-calculus contexts are built just from abstractions and applications, one usually does not bother with formalising neither the concept of a *signature* nor that of a *context*, instead giving the corresponding ad hoc *compatibility* clauses directly [2]. That is manageable since they are only three, traditionally [2] called ζ for abstraction and ξ_1 and ξ_2 for (the arguments of) application. The formalisation in the module Untyped follows suit.

Since in the $\lambda\beta$-calculus there is only a single rule β, one usually does not bother with formalising the concept of a *rule*, giving the corresponding ad hoc *rule scheme* directly instead. Since there is only one such rule scheme, traditionally [2] called $\boldsymbol{\beta}$, this is manageable. Here, the rule scheme $\boldsymbol{\beta}$ is obtained from the rule β by taking all its substitution instance, i.e. quantifying over the latter (at meta-level). The formalisation in the module Untyped follows suit.

Combining both above design decisions taken in [18] leads naturally to the formalisation of *rewrite steps* in Untyped as a data type $\longrightarrow$ inductively defined from the rule scheme $\boldsymbol{\beta}$ and the compatibility clauses: steps are represented as *proof terms* [17, Chapter 8] but instead of them being based on a *single* β-rule-symbol (with arity the number of free variables in the rule) as in [4], they are based on the *infinitely many* (nullary) rule-symbols of the rule scheme $\boldsymbol{\beta}$.

Since such a set-up does not automatically guarantee steps being closed under substitutions[12], i.e. does not guarantee we in fact have a *term* rewrite system, we proved it as `stp-subst` (after proving it for renamings `stp-rename`). We then proceeded with pointwise extending the definition of many-step reduction $\longrightarrow\!\!\!\!\rightarrow$ to substitutions $\longrightarrow\!\!\!\!\rightarrow$`s`, and showing the latter to be closed under term-contexts `trm-subst`, and reductions to be closed under them in `rew-rew`.

These basic and simple results constitute Part I of our formalisation (50 loc). That part (c/sh)ould go to the module Untyped of [18] as it exclusively concerns showing that the rewrite system is as intended, that it indeed is a *term* rewrite system closed under both contexts (already covered in [18]) and substitutions (absent from it as far as we know). It belongs there, since its results pertain neither to Z nor to confluence nor to any other rewrite property.

Proof of the Z-property We establish the Z-property for De Bruijn's λ-calculus with nameless dummies, based on the latter's formalisation as a term rewrite system as outlined above, where a rewrite system has the Z-property for a map $\bullet$ on its objects if for every step $a \rightarrow b$ we have $b \twoheadrightarrow a^\bullet \twoheadrightarrow b^\bullet$ [13].

As the bullet map $\bullet$ we take the *full development* map [13, Definition 42], mapping a term M to the target of the multistep contracting all redex-patterns in M. For the history of this bullet map for the λ-calculus, going back to Gross, Knuth, and Takahashi among others, see [3, 13].

[12]Closing under substitutions *first* and *then* contexts might differ from closing under both simultaneously.

For this bullet map $\bullet$ we follow the approach for proving the Z-property for orthogonal term rewriting as outlined in [13, Sections 3.3.1 and 3.4]: We successively establish the following properties,[13] with the Z-property being the conjunction of (upperbound) and (monotonic):

(extensive) $M \twoheadrightarrow_\beta M^\bullet$;

(upperbound) if $M \to_\beta N$ then $N \twoheadrightarrow_\beta M^\bullet$;

(rhss) $(M^\bullet)^{\sigma^\bullet} \twoheadrightarrow_\beta (M^\sigma)^\bullet$; and

(monotonic) if $M \to_\beta N$ then $M^\bullet \twoheadrightarrow_\beta N^\bullet$.

Typically, these properties may be shown either by induction on the term M (and then distinguishing cases on the steps $M \to N$ possible from M) or by induction on the proof term $M \to N$ (and then distinguishing cases on the possible shapes of M). This choice is largely immaterial. Here we found it convenient to prove the properties (extensive) and (rhhs) by induction on the term M, and the properties (upperbound) and (monotonic) by induction on the step $M \to N$.

This constitutes Part II (65 loc), the core of our formalisation.

Proof of confluence Confluence can be obtained from the Z-property in various ways [13, 6]. We randomly picked showing confluence via the so-called Strip Lemma [2], i.e. we picked the first proof of [13, Lemma 51], with the proof of strip being an easy recombination of (extensive), (upperbound) and (rew-monotonic), the lifting of (monotonic) from steps to reductions.

The formalisation of this final part is trivial (also in Agda), short (25 loc), and abstract in that it depends on the rewrite system having the Z-property, but on nothing else, in particular not on the concrete structure of the terms and steps of the $\lambda\beta$-calculus. Reusing the definition of confluence of [18] here, allows our formalisation to replace the module Confluence of [18].

3 Conclusions and related work

The development confirmed our expectations.

Whether our formalisation is short*est* and simpl*est* we do not know,[14] but its core (on the Z-property) being only 65 loc, the development is definitely short and simple.[15] It closely follows the formal pen-and-paper proof.

Basing ourselves on [18] was a good choice, as the book was easily accessible and the code was well-explained and modularised, allowing for easy reuse.

Despite lacking any prior knowledge of Agda, we found coding in it intuitive and its (emacs) interface based on the idea of refining *holes*[16] to be easy to get acquainted with.

[13]The (rhss) property here is the one which was intended in [13, Lemma 44(Rhs)]. The property (Rhs) given there, that $M^{(\sigma^\bullet)}$ reduces to $(M^\sigma)^\bullet$, is correct in it being a trivial consequence of (extensive) and (rhss). However, (Rhs) did not capture the idea that the result $((M^\bullet)^{(\sigma^\bullet)})$ of applying the bullet map to the *components* (M and σ) of the right-hand side of the term rewrite rule (M^σ) reduces to the result $((M^\sigma)^\bullet)$ of applying the bullet map to right-hand side. Consequently, the property (Rhs) given there is too weak to be useful for establishing (monotonic).

[14]Even if one could come up with objective measures for these, the formalised results are hard to compare as they are almost never exactly the same. For instance, the proof of Z here is for De Bruijn's $\lambda\beta$-calculus with *nameless* dummies, whereas that of [9] is for a *nominal* $\lambda\beta$-calculus. Also, the proof of confluence in [18] involves a formalisation of *multisteps*, which could be reused for obtaining *standardisation* results, whereas the Z property shown here could be reused for obtaining *hyper-normalisation* results.

[15]It seems not longer than the extant PLFA code on confluence, even when including the 50 loc for showing to have a (higher-order) term rewrite system.

[16]Instead of on a *tactics* language as I was used to from Coq.

Our initial estimate was that the required effort would be fairly limited, with our lack of Agda knowledge being compensated for by the reuse of the PLFA book. Still, the development was surprisingly smooth, taking us a full week only.[17] Both using the book [18] to base ourselves on and prior experience with the theory and with formalisation, certainly helped a lot.[18]

Since our formalisation in October 2021, we have tried (in intervals of 6 months) to initiate a discussion on this work and its relationship to their book with the authors of [18]. Unfortunately, and despite multiple promises to do so on their behalf, this never materialised. We find that surprising given that the direct incentive to work on this was a challenge by one of them.

Last year at IWC 2023, we learned from Riccardo Treglia that he had supervised a project in 2019/2020 of Andrea Laretto on formalising the Church–Rosser theorem for the $\lambda\beta$-calculus in Agda. Also there we have tried to initiate a discussion with the author on the relationship between both, but also that hasn't materialised yet.

The properties involved being shown by direct inductions on terms and steps (inductively defined as proof terms), corroborates our earlier experience [9] and expectation [13] that proving confluence via the Z-property is well-suited to formalisation in proof assistants.

We initially tried formalising the Z-property for the *full superdevelopment* [13, Definition 42] bullet function as in [9], since we expected that to result in even (a bit) shorter code. However, that requires reasoning by cases on the *result* of the bullet map, which, although supported in Agda, eluded our skills.[19] It should be of interest to see whether one can mimic the presentation on paper and have *single* generic proof that can then be instantiated both to the *full development* and to the *full superdevelopment* bullet map.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *J. Funct. Program.*, 1(4):375–416, 1991. doi:10.1017/S0956796800000186.

[2] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103. North-Holland, Amsterdam, 2nd revised edition, 1984.

[3] H.P. Barendregt, J. Bergstra, J.W. Klop, and H. Volken. Degrees, reductions and representability in the lambda calculus. Preprint 22, Utrecht University, Department of Mathematics, 1976.

[4] H.J.S. Bruggink. *Equivalence of Reductions in Higher-Order Rewriting.* PhD thesis, Utrecht University, 2008.

[5] N.G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indagationes Mathematicae*, 34:381–392, 1972.

[6] F.L.C. de Moura and L.O. Rezende. A formalization of the (compositional) z property. Presented at the 14th Conference on Intelligent Computer Mathematics, 2021.

[7] P. Dehornoy. *Braids and Self-Distributivity*, volume 192 of *Progress in Mathematics.* Birkhäuser, 2000.

[8] P. Dehornoy and V. van Oostrom. Z; proving confluence by monotonic single-step upperbound functions. In *Logical Models of Reasoning and Computation (LMRC-08), Moscow*, 2008. http://www.javakade.nl/research/talk/lmrc060508.pdf.

[17] In fact, the development took less time than writing this note now (more than two years after).

[18] In particular, knowledge of the topic through [13], of the Nominal Isabelle formalisation of Z through [9] (though I am not familiar with Isabelle itself), of explicit substitution calculi through [17], of the formalisation of confluence of the λ-calculus in Coq through [10], and of type theory, Coq, and Haskell through lecturing on them, all helped in flattening the learning curve.

[19] More precisely, the definition of the full superdevelopment function itself was unproblematic, but I then failed to use it in proofs. That may be due to a lack of Agda-proficiency, a lack of features in Agda 2.6.1.1, or a combination of both. We noticed later versions of Agda offer more support for such case distinctions, so it might be interesting to have another go at it.

[9] B. Felgenhauer, J. Nagele, V. van Oostrom, and C. Sternagel. The Z property. *Arch. Formal Proofs*, 2016, 2016. https://www.isa-afp.org/entries/Rewriting_Z.shtml.

[10] D. Hendriks and V. van Oostrom. ʎ. In F. Baader, editor, *Proceedings of CADE 19*, volume 2741 of *Lecture Notes in Artificial Intelligence*, pages 136–150. Springer, 2003. doi:10.1007/978-3-540-45085-6_11.

[11] G. Huet. Residual theory in λ-calculus: a formal development. *Journal of Functional Programming*, 4(3):371–394, July 1994.

[12] R. Loader. Notes on simply typed lambda calculus. ECS-LFCS- 98-381, Laboratory for Foundations of Computer Science, The University of Edinburgh, February 1998.

[13] V. van Oostrom. Z; syntax-free developments. In N. Kobayashi, editor, *6th International Conference on Formal Structures for Computation and Deduction, FSCD 2021, July 17-24, 2021, Buenos Aires, Argentina (Virtual Conference)*, volume 195 of *LIPIcs*, pages 24:1–24:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.FSCD.2021.24.

[14] V. van Oostrom, K.J. van de Looij, and M. Zwitserlood. Lambdascope. In *ALPS 2004*, page 9, 2004.

[15] S. Schäfer, G. Smolka, and T. Tebbi. Completeness and decidability of de Bruijn substitution algebra in Coq. In X. Leroy and A. Tiu, editors, *Proceedings of the 2015 Conference on Certified Programs and Proofs, CPP 2015, Mumbai, India, January 15-17, 2015*, pages 67–73. ACM, 2015. doi:10.1145/2676724.2693163.

[16] M. Takahashi. Parallel reductions in λ-calculus. *Information and Computation*, 118:120–127, April 1995. doi:10.1006/inco.1995.1057.

[17] Terese. *Term Rewriting Systems*. Cambridge University Press, 2003.

[18] P. Wadler, W. Kokke, and J.G. Siek. *Programming Language Foundations in Agda*. July 2020. https://plfa.inf.ed.ac.uk/20.07/.

A Formalisation

The following module `ConfluenceZ4.agda` may replace plfa/src/plfa/part2/Confluence.lagda.md.

```
module plfa.part2.ConfluenceZ4 where

open import Relation.Binary.PropositionalEquality using (_≡_; refl; sym; cong; cong₂)
open import Function using (_∘_)
open import Data.Product using (_×_; Σ; Σ-syntax; ∃; ∃-syntax; proj₁; proj₂)
  renaming (_,_ to ⟨_,_⟩)
open import plfa.part2.Substitution using (Rename; Subst; rename-subst-commute; subst-commute;
  ren; cong-sub; extensionality)
open import plfa.part2.Untyped
  using (_—→_; β; ξ₁; ξ₂; ζ; _—↠_; begin_; _—→⟨_⟩_; _—↠⟨_⟩_; _∎;
  abs-cong; appL-cong; appR-cong; —↠-trans;
  _⊢_; _∋_; `_; #_; _,_; ★; ƛ_; _·_; _[_];
  rename; ext; exts; Z; S_; subst; subst-zero)

{- Part I -}

{- basic facts on rewriting and substitution, in particular -}
{- we show that beta is apart from being ‘closed under contexts’ -}
{- as established in plfa, also closed under substitutions, -}
{- i.e.  that we are in a (higher-order) term rewriting setting.  -}
{- (these results only depend on Untyped section of plfa) -}

{- ϕ ranges over steps; Ψ,X over many-steps; R over substitution many-steps -}

{- rewriting is a quasi-congruence for application -}
app-cong : ∀{Γ} {K L M N : Γ ⊢ ★} → K —↠ L → M —↠ N → K · M —↠ L · N
app-cong Ψ X = —↠-trans (appL-cong Ψ) (appR-cong X)

{- steps are closed under renaming -}
stp-rename : ∀{Γ Δ A} {ρ : Rename Γ Δ} (M : Γ ⊢ A) {N : Γ ⊢ A}
  → M —→ N
    ------------------------
  → rename ρ M —→ rename ρ N
stp-rename (ƛ M) (ζ ϕ) = ζ (stp-rename M ϕ)
stp-rename (M · _) (ξ₁ ϕ) = ξ₁ (stp-rename M ϕ)
stp-rename (_ · M) (ξ₂ ϕ) = ξ₂ (stp-rename M ϕ)
stp-rename {ρ = ρ} ((ƛ L) · M) β with β {_} {rename (ext ρ) L} {rename ρ M}
... | G rewrite rename-subst-commute {_}{_}{L}{M}{ρ} = G

{- steps are closed under substitution -}
stp-subst : ∀{Γ Δ A} {σ : Subst Γ Δ} {M M′ : Γ ⊢ A}
  → M —→ M′
    -----------------------
  → subst σ M —→ subst σ M′
stp-subst (ξ₁ ϕ) = ξ₁ (stp-subst ϕ)
stp-subst (ξ₂ ϕ) = ξ₂ (stp-subst ϕ)
stp-subst {σ = σ} (β {_} {L} {M}) with β {_} {subst (exts σ) L} {subst σ M}
... | G rewrite subst-commute {_}{_}{L}{M}{σ} = G
stp-subst {σ = σ} (ζ ϕ) = ζ (stp-subst {σ = exts σ} ϕ)

{- parallel substitution many-step -}
_—↠s_ : ∀{Γ Δ} → Subst Γ Δ → Subst Γ Δ → Set
_—↠s_ {Γ}{Δ} σ τ = ∀{A} → (x : Γ ∋ A) → σ x —↠ τ x

{- applying parallel substitution many-step to term yields many-step -}
trm-subst : ∀{Γ Δ A} {σ τ : Subst Γ Δ}
  → σ —↠s τ
  → (M : Γ ⊢ A)
    -----------------------
  → subst σ M —↠ subst τ M
trm-subst R (` x) = R x
trm-subst {σ = σ} {τ = τ} R (ƛ M) = abs-cong (trm-subst exts-rews M) where
  {- many-step closed under renaming -}
  rew-rename : ∀{Γ Δ A} {ρ : Rename Γ Δ} {M N : Γ ⊢ A}
     → M —↠ N
     ------------------------
     → rename ρ M —↠ rename ρ N
  rew-rename {ρ = ρ} (M ∎) = (rename ρ M) ∎
  rew-rename {ρ = ρ} (M —→⟨ ϕ ⟩ Ψ) = rename ρ M —→⟨ stp-rename _ ϕ ⟩ rew-rename Ψ
  {- extension lemma for parallel substitution many-steps -}
  exts-rews : ∀{B} → exts σ {B = B} —↠s exts τ
  exts-rews Z = (‘ Z) ∎
  exts-rews (S x) = rew-rename (R x)
trm-subst R (L · M) = app-cong (trm-subst R L) (trm-subst R M)

{- applying singleton substitution many-step to many-step yields many-step -}
rew-rew : ∀{Γ} {M N : Γ , ⋆ ⊢ ⋆ } {K L : Γ ⊢ ⋆ }
  → M —↠ N
  → K —↠ L
    --------------------
  → M [ K ] —↠ N [ L ]
rew-rew {Γ} {K = K} {L} Ψ X = rews-subst Ψ where
  {- lifting many-step to a singleton substitution many-step -}
  rews-zero : subst-zero K —↠s subst-zero L
  rews-zero Z = X
  rews-zero (S x) = (‘ x) ∎
  {- applying parallel substitution many-step to many-step yields many-step -}
  rews-subst : ∀{A} {M N : Γ , ⋆ ⊢ A} → M —↠ N
    → subst (subst-zero K) M —↠ subst (subst-zero L) N
  rews-subst (M ∎) = trm-subst rews-zero M
  rews-subst (_ —→⟨ φ ⟩ Ψ) = _ —→⟨ stp-subst φ ⟩ (rews-subst Ψ)

{- Part II -}

{- beta is shown to have the Z property, taking the function -}
{- mapping a term to the result of its maximal multi-step, -}
{- i.e.  full development, as bullet and showing that -}
{- satisfies upperbound (condition 1) and monotonic (condition 2) -}
{- for beta steps.  based on Loader 98 and van Oostrom 21.  -}
{- (these results may replace the Confluence section of plfa) -}

{- take full developments for bullet map; gross/takahashi/loader -}
_• : ∀ {Γ A} → Γ ⊢ A → Γ ⊢ A
(‘ x)• = ‘ x
(ƛ M)• = ƛ (M •)
((ƛ M) · N)• =    M • [ N • ]
(M · N)• = (M •) ·(N •)

{- bullet is extensive -}
extensive : ∀ {Γ A} → (M : Γ ⊢ A) → M —↠ M •
extensive (‘ _) = _ ∎
extensive (ƛ M) = abs-cong (extensive M)
extensive ((ƛ M) · N) = _ —→⟨ β ⟩ rew-rew (extensive M) (extensive N)
extensive (‘ _ · N) = appR-cong (extensive N)
extensive (L · M · N) = app-cong (extensive (L · M)) (extensive N)

{- bullet gives upper bound to co-initial steps (first condition of Z) -}
upperbound : ∀ {Γ} → {M N : Γ ⊢ ⋆}
  → M —→ N
    --------
  → N —↠ M •
upperbound {_} {ƛ _} (ζ ϕ) = abs-cong (upperbound ϕ)
upperbound {_} {(` _) · _} {_} (ξ₂ ϕ) = appR-cong (upperbound ϕ)
upperbound {_} {(ƛ _) · M} {((ƛ _) · M)} (ξ₁ (ζ ϕ)) = _ —→⟨ β ⟩ rew-rew (upperbound ϕ) (extensive M)
upperbound {_} {(ƛ L) · _} {.((ƛ L) · _)} (ξ₂ ϕ) = _ —→⟨ β ⟩ rew-rew (extensive L) (upperbound ϕ)
upperbound {_} {(ƛ L) · M} {.(subst (subst-zero M) L)} β = rew-rew (extensive L) (extensive M)
upperbound {_} {_ · _ · M} {.(_ · M)} (ξ₁ ϕ) = app-cong (upperbound ϕ) (extensive M)
upperbound {_} {K · L · _} {.(_ · _ · _)} (ξ₂ ϕ) = app-cong (extensive (K · L)) (upperbound ϕ)

{- bullet commutes with renaming -}
rename-bullet : ∀{Γ Δ} (ρ : Rename Γ Δ) {A} (M : Γ ⊢ A) → rename ρ (M •) ≡ rename ρ M •
rename-bullet _ (` x) = refl
rename-bullet ρ (ƛ M) = cong ƛ_ (rename-bullet (ext ρ) M)
rename-bullet ρ ((` x) · M) rewrite rename-bullet ρ M = refl
rename-bullet ρ ((ƛ L) · M) rewrite sym (rename-subst-commute {_} {_} {L •} {M •} {ρ})
  rewrite rename-bullet ρ M rewrite rename-bullet (ext ρ) L = refl
rename-bullet ρ (K · L · M) = cong₂ _·_ (rename-bullet ρ (K · L)) (rename-bullet ρ M)

{- bullet commutes with extension -}
exts-bullet : ∀{Γ Δ} {σ τ : Subst Γ Δ}
  → ((x : Γ ∋ ⋆) → τ x ≡ σ x •)
    --------------------------------------
  → (x : Γ , ⋆ ∋ ⋆) → exts τ x ≡ exts σ x •
exts-bullet eq Z = refl
exts-bullet {σ = σ} eq (S x) rewrite (eq x) = rename-bullet S_ (σ x)

{- rhs lemma for parallel substitution -}
rhss : ∀{Γ Δ} (M : Γ ⊢ ⋆) {σ τ : Subst Γ Δ} → ((x : Γ ∋ ⋆) → τ x ≡ σ x •)
    ----------------------------
  → subst τ (M •) —↠ (subst σ M)•
rhss (` x) eq rewrite (eq x) = _ ∎
rhss (ƛ M) eq = abs-cong (rhss M (exts-bullet eq))
rhss ((` x) · M) {σ} eq rewrite (eq x) = —↠-trans
  (appR-cong (rhss M eq)) (app-bullet (σ x) (subst σ M)) where
  {- auxiliary rhs/monotonicity lemma for application -}
    app-bullet : ∀{Γ} (L M : Γ ⊢ ⋆) → L • · M • —↠ (L · M)•
    app-bullet (` _) _ = _ ∎
    app-bullet (ƛ _) _ = (_ —→⟨ β ⟩ _ ∎)
    app-bullet (_ · _) _ = _ ∎
rhss ((ƛ L) · M) {τ = τ} eq rewrite (sym (subst-commute {N = L •} {M •} {τ})) =
  rew-rew (rhss L (exts-bullet eq)) (rhss M eq)
rhss (K · L · M) eq = app-cong (rhss (K · L) eq) (rhss M eq)

{- bullet monotonic for steps (second condition of Z) -}
monotonic : ∀{Γ} → {M N : Γ ⊢ ⋆}
  → M —→ N
    ----------
  → M • —↠ N •
monotonic (ζ ϕ) = abs-cong (monotonic ϕ)
monotonic {_} {(` _) · _} {(` _) · _} (ξ₂ ϕ) = appR-cong (monotonic ϕ)
monotonic {Γ} {(ƛ M) · N} {.(subst (subst-zero N) M)} β = rhss M bullet-zero where
  {- bullet commutes with lifting terms to substitutions -}
  bullet-zero : (x : Γ , ★ ∋ ★) → subst-zero (N •) x ≡ subst-zero N x •
  bullet-zero Z = refl
  bullet-zero (S x) = refl
monotonic {_} {(ƛ _) · _} {(ƛ _) · _} (ξ₁ (ζ ϕ)) = rew-rew (monotonic ϕ) (_ ■)
monotonic {_} {(ƛ M) · _} {.((ƛ M) · _)} (ξ₂ ϕ) = rew-rew (M • ■) (monotonic ϕ)
monotonic {_} {_ · _ · _} {(ƛ _) · _} (ξ₁ ϕ) = —↠-trans (appL-cong (monotonic ϕ)) (_ —→⟨ β ⟩ _ ■)
monotonic {_} {_ · _ · _} {(‘ _) · _} (ξ₁ ϕ) = appL-cong (monotonic ϕ)
monotonic {_} {_ · _ · _} {_ · _ · _} (ξ₁ ϕ) = appL-cong (monotonic ϕ)
monotonic {_} {_ · _ · _} {_ · _ · _} (ξ₂ ϕ) = appR-cong (monotonic ϕ)

{- bullet is monotonic for rewrites (abstract from Z) -}
rew-monotonic : ∀{Γ} → {M N : Γ ⊢ ★}
  → M —↠ N
    ----------
  → M • —↠ N •
rew-monotonic (_ ■) = _ ■
rew-monotonic (_ —→⟨ ϕ ⟩ Ψ) = —↠-trans (monotonic ϕ) (rew-monotonic Ψ)

{- Barendregt’s Strip Lemma for beta (abstract from Z) -}
strip : ∀{Γ} {L M N : Γ ⊢ ★}
  → L —→ M
  → L —↠ N
    -----------------------------------
  → ∑[ K ∈ Γ ⊢ ★ ] (M —↠ K) × (N —↠ K)
strip {L = L} {N = N} ϕ Ψ = ⟨ N • ,
  ⟨ —↠-trans (upperbound ϕ) (rew-monotonic Ψ) , extensive N ⟩ ⟩

{- confluence of beta (abstract from Z) -}
confluence : ∀{Γ} {L M N : Γ ⊢ ★}
  → L —↠ M
  → L —↠ N
    ------------------------------------
  → ∑[ K ∈ Γ ⊢ ★ ] (M —↠ K) × (N —↠ K)
confluence (_ ■) X = ⟨ _ , ⟨ X , _ ■ ⟩ ⟩
confluence (_ —→⟨ ϕ ⟩ Ψ) X =
  ⟨ _ , ⟨ proj₁ d2 , —↠-trans (proj₂ d1) (proj₂ d2) ⟩ ⟩ where
  d1 = proj₂ (strip ϕ X)
  d2 = proj₂ (confluence Ψ (proj₁ d1))
```