

# Confluence by the Z-property for De Bruijn’s $\lambda$ -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 discuss the Agda formalisation of a proof of confluence of the untyped  $\lambda\beta$ -calculus, more precisely, of the Z-property for De Bruijn’s  $\lambda$ -calculus notation with nameless dummies, based on the latter’s formalisation in [Programming Language Foundations in Agda](#).

**Introduction** We consider confluence of the rewrite system  $\rightarrow_\beta$  of the  $\lambda\beta$ -calculus [3].

A sufficient condition for a rewrite system  $\rightarrow$  to be confluent is the *Z-property*, the existence of a map  $\bullet$  on its objects such that if  $a \rightarrow b$  then  $b \rightarrow a^\bullet \rightarrow b^\bullet$ . The Z-property originates with Loader [14, Section 4.1] for the  $\lambda\beta$ -calculus with the *Gross-Knuth* bullet map, and with Dehornoy [9] for self-distributivity with the *full distribution* bullet map. See [15] for more on it and on its theory, e.g. that it is not *necessary* for confluence.


In [15, Remark 52][10, Conclusions] we claimed that the proof of confluence of  $\rightarrow_\beta$  by means of the Z-property given there was (a bit) shorter than the proof of the same due to Takahashi [18] via the *angle* property, the existence of a map  $\bullet$  on the objects of  $\rightarrow$  together with a rewrite system  $\rightarrow$  with  $\rightarrow \subseteq \rightarrow \subseteq \rightarrow$  and such that if  $a \rightarrow b$  then  $b \rightarrow a^\bullet$  [19, 15].

**Remark.** In the same paper we showed [15, Lemma 8] that the Z-property and the angle property are *equivalent*. That the proof using the former is (a bit) shorter nonetheless is due to that it dispenses with introducing the auxiliary rewrite system  $\rightarrow$ ; the  $\bullet$ -map suffices.

Here we investigate that claim specialised to a *formalisation* of the  $\lambda$ -calculus, in particular to the formalisation of it in Agda in [Programming Language Foundations in Agda](#) [20]. There, in the module [Confluence](#), a proof of confluence of De Bruijn’s  $\lambda$ -calculus with nameless dummies [7] was formalised via the angle property. We present an alternative formalisation<sup>1</sup> of confluence via the Z-property; it again is (a bit) shorter than the one via the angle property. Our alternative formalisation is based on the October 2021 snapshot<sup>2</sup> of [20]; it may replace its [Confluence](#) module but is still based on its modules [Untyped](#) (formalising De Bruijn’s  $\lambda$ -calculus with nameless dummies) and [Substitution](#) (formalising the so-called Substitution Lemma).

We first describe and comment on the key ingredients of the Agda code from [20] we took as the basis of our formalisation, the *objects* and *steps* of the rewrite system, and motivate why we did so. We then do the same for our supplementary code, the alternative formalisation of [Confluence](#), 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, supplemented with comments on **design** decisions. We assume knowledge of rewriting [19] and the  $\lambda$ -calculus [3] with nameless dummies [7].

---

\*This work is licensed under the Creative Commons Attribution 4.0 International License . The Agda code was developed in the 1st week of October 2021 based on the then-current version of [Programming Language Foundations in Agda](#) [20] on a MacBook Pro 2019 with Agda 2.6.1.1, Emacs 26.3.

<sup>1</sup>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>.

<sup>2</sup>That snapshot already had the proof via the angle property of the 2022 version of [20], instead of the Tait–Martin-Löf proof via the *diamond* property found in its 2020 version, i.e. based on the existence of a rewrite system  $\rightarrow$  with  $\rightarrow \subseteq \rightarrow \subseteq \rightarrow$ , such that for  $a \rightarrow b$  and  $a \rightarrow c$  there is a  $d$  with  $b \rightarrow d$  and  $c \rightarrow d$ .

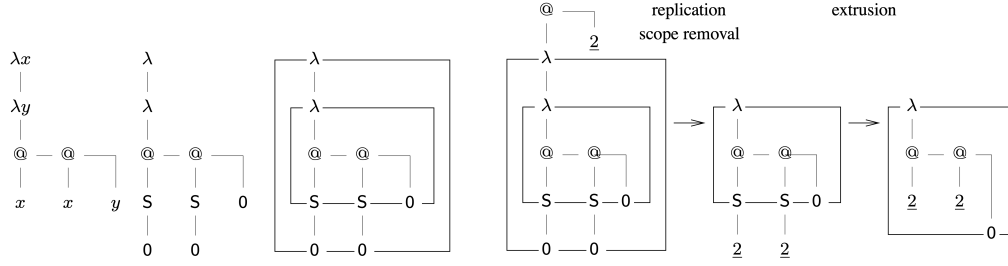


Figure 1: Church numeral  $\underline{2}$  with names, nameless and scopes (left), and factoring  $\underline{2} \underline{2} \rightarrow_{\beta} \underline{4}$  (right)

**Objects of the rewrite system** The objects of the rewrite system are the  $\lambda$ -terms with nameless dummies of [7], obtained by restricting *generalised* such terms [5, 12]  $t ::= 0 \mid St \mid \lambda t \mid tt$  by requiring  $t$  in  $St$  to be an *index*, a unary natural number generated from  $0$  and  $S$  only. Figure 1 (left) exemplifies the correspondence between named [3] and nameless [7] (better: *uni-named*)  $\lambda$ -terms for the Church numeral  $\underline{2} := \lambda \lambda (S0) ((S0) 0)$  (we employ the notational conventions of [3]).

It is a matter of hygiene to factor a notion for *open*  $\lambda$ -terms through the same for *closed*  $\lambda$ -terms, e.g. it is hygienic to define an *open* term  $M$  to be *solvable* [3] if its *closure*  $\hat{M}$  is, i.e. if  $\hat{M} \vec{P} =_{\beta} \mid$  for some  $\vec{P}$ . The formalisation in `Untyped` follows suit and carves out from the above  $\lambda$ -terms the *closed* ones [2, 16] by means of the inference rules (to be read bottom-up):

$$\frac{Si \vdash 0}{i \vdash t} 0 \quad \frac{Si \vdash St}{i \vdash t} S \quad \frac{i \vdash \lambda t}{Si \vdash t} \lambda \quad \frac{i \vdash t_1 t_2}{i \vdash t_1 \quad i \vdash t_2} @$$

Here the *scoping* judgment  $i \vdash t$  asserts that  $t$  is a  $\lambda$ -term having  $i$  as *upper bound* on its free indices, in particular  $0 \vdash t$  asserts  $t$  is closed. See [16, example 3] for a derivation showing  $\underline{2}$  is indeed closed. Scoping judgments are implemented in the module `Untyped` of [20], there split into separate judgments for *indices*  $i$  and  $\lambda$ -terms  $t$  built from them by means of abstraction ( $\lambda$ ) and application ( $@$ ). As is good engineering practice, we *reuse* it for our formalisation.

**Remark.** The reason for calling  $i \vdash t$  a *scoping* judgment in [20] is that making the scopes of  $\lambda$ -abstractions explicit [12, 16] as boxes, as done in Figure 1 for the Church numeral  $\underline{2}$ , makes it clear that each  $S$  represents an *end-of-scope* of, and each  $0$  a *reference* to, its  $\lambda$ -abstraction. The notion of *binding* for named  $\lambda$ -terms is recovered for nameless  $\lambda$ -terms by the context-free notion of *matching parentheses* along paths of the abstract syntax tree of the  $\lambda$ -term, viewing each  $\lambda$  as an opening parenthesis ( and each end-of-scope  $S$  or reference  $0$  as a closing parenthesis ) [12]. A box is formed by an abstraction together with its matching end-of-scopes and references.

**Design.** Since in the  $\lambda\beta$ -calculus terms are constructed just from abstractions and applications, in the literature [3] one usually abstains from formalising the concept of a *symbol / signature* [19] to construct terms from. The formalisation in the module `Untyped` follows suit. For applied  $\lambda$ -calculi this design choice leads to *redundancy*, e.g. an inference rule for each construct.

**Steps of the rewrite system** The  $\rightarrow_{\beta}$ -steps are generated by closing the  $\beta$ -rule scheme [7]:

$$\beta : (\lambda t) s \rightarrow t[s]$$

under contexts, where  $t[s]$  denotes substitution of  $s$  for the free occurrences of  $0$  in  $t$ ; it must be hygienic in that it should *preserve* scoping: if  $i \vdash (\lambda t) s$ , i.e. if  $Si \vdash t$  and  $i \vdash s$ , then  $i \vdash t[s]$ .

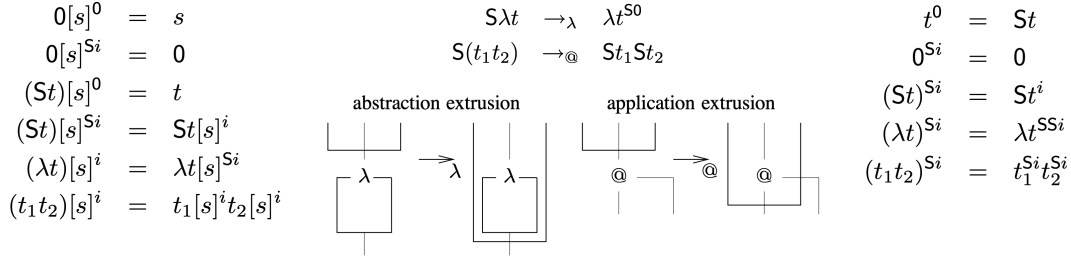


Figure 2: Alternative substitution (left), scope extrusion (middle) and minimal lifting (right)

Contracting  $(\lambda t) s$  can be split into two phases. In the first phase the  $\lambda$  together with its box are removed and each  $0$  on it is replaced by  $s$ , see Figure 1 (right). This yields a *generalised*  $\lambda$ -term, upon which the second phase *extrudes* any offending scopes, to yield the  $\lambda$ -term  $t[s]$ .

Contracting  $(\lambda t) s$  to  $t[s]$  in the module `Untyped` of [20] factors through `subst-zero` yielding a *parallel* substitution  $\sigma$  mapping  $0 \mapsto s$  and  $S^i \mapsto i$  implementing the first phase, with the second *lifting* phase brought about during the parallel substitution process  $t[s] = t^\sigma$  via `subst`.<sup>3</sup> That `subst` satisfies the so-called *Substitution Lemma* [3] is shown in the module `Substitution` of [20]. Since our code is modular in it, we *reuse* also that module.

**Remark.** The Substitution Lemma is unavoidable in *any* proof of confluence as it captures the resolution of the *critical peak* of the  $\beta$ -rule arising from *nested* redex-patterns  $(\lambda y. (\lambda x. M) N) L$ :

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

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

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

**Design.** To allow for an algebraic proof of the Substitution Lemma, the module `Substitution` of [20] first defines the basic substitution *operations* corresponding to the explicit substitution *operators* of [17], based on [1] but having laws that are *complete* w.r.t. *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 the Substitution Lemma is proven using the algebraic laws.

The result of taking that indirect approach is that the module `Substitution` is largish. As it essentially only provides (both to [20] and to our code) the Substitution Lemma `subst-commute`, it should be feasible and interesting to replace it by a smaller module on a different basis.

E.g., [13] employs only 2 operations, substitution (`subst_rec`) and lifting (`lift_rec`), and 6 laws governing their interaction to formalise the Substitution Lemma in Coq. That development can be seen as implementing  $t[s]$  as  $t[s]^0$  followed by *maximal scope extrusion* as sketched above, defined in [16], and repeated here in Figure 2. We *formalised*<sup>4</sup> the Substitution Lemma for *generalised*  $\lambda$ -terms in Coq, based on *minimal* scope extrusion, only extruding scopes as far as needed to make  $\beta$ -redexes visible [12]. Somewhat surprisingly Huet's 6 laws still hold.

**Design.** In the module `Untyped` of [20] steps are generated as in [3] via the *compatible* closure, via clauses conventionally called  $\zeta$  for abstraction and  $\xi_1$  and  $\xi_2$  for (the arguments of) application. Already here the absence of a signature leads to redundancy (between  $\xi_1$  and  $\xi_2$ ). The redundancy will be worse for applied  $\lambda$ -calculi having larger signatures.

<sup>3</sup>Now lifting the substitution into a box top-down via its  $\lambda$ , instead of bottom-up via an  $S$  as earlier.

<sup>4</sup>HTML at <http://www.javakade.nl/research/coq/ConfluencebyZofGeneralisedLocalDeBruijnInCoq.html>.

$$\begin{array}{l}
\text{app-cong} : \forall \{ \Gamma \} \{ K L M N : \Gamma \vdash * \} \rightarrow K \longrightarrow L \rightarrow M \longrightarrow N \rightarrow K \cdot M \longrightarrow L \cdot N \\
\text{rew-rew} : \forall \{ \Gamma \} \{ M N : \Gamma, * \vdash * \} \{ K L : \Gamma \vdash * \} \\
\rightarrow M \longrightarrow N \\
\rightarrow K \longrightarrow L \\
\text{-----} \\
\rightarrow M [ K ] \longrightarrow N [ L ]
\end{array}$$
Figure 3: Closure of reduction under application (`app-cong`) and substitution (`rew-rew`)

**Term rewrite system** To say that we have a *term* rewrite system [19] is to say that steps (and reductions) are closed under *contexts* and *substitutions* [19]. We supplemented the results in [20] with the remaining ones needed to show that, with the main ones being closure of reduction under application (`app-cong`) and under substitution (`rew-rew`), displayed in Figure 3. The 50 loc needed for it constitute Part I of our formalisation and confirm sanity of the module `Untyped`.

**Remark.** For `rew-rew`, we first showed closure of *steps* under substitution (`stp-subst`) after which we pointwise extended the definition of many-step reduction  $\longrightarrow$  to many-step reduction of substitutions  $\longrightarrow\text{s}$  and showed the latter to be closed under term-contexts `trm-subst`, finally allowing us to show reductions are closed under substitution for `rew-rew`, all missing from [20].

**Design.** Since in the  $\lambda$ -calculus there is only a single rule  $\beta$ , one usually abstains from formalising the concept of a *rule*, instead giving the corresponding ad hoc *rule scheme* directly (one can think of the scheme as obtained from the rule by taking all its substitution instances). Since in the module `Untyped` there is only one such rule scheme, conventionally [3] called  $\beta$ , this is manageable. However, for applied  $\lambda$ -calculi this would again lead to redundancy.

It should be feasible and interesting to give an alternative formalisation of the untyped  $\lambda$ -calculus in [20] as a higher-order term rewrite systems [19, Chapter 9], proceeding in the spirit of [19, Chapter 8] and [6] via a signature of both *function* and *rule* symbols, with (*multi*)steps simply being terms over that signature, so-called *proofterms*.

**Z** To establish the Z-property for  $\rightarrow_\beta$  we follow the approach pioneered by Loader [14] with key properties outlined in [15, Sections 3.3.1, 3.4] for orthogonal term rewrite systems. We take as bullet map  $\bullet$  the *full development* map [15, Definition 42], recursively mapping a term  $M$  to the target obtained by contracting all  $\beta$ -redexes in  $M$ ; its Agda code is presented in Figure 4.

**Remark.** For the history of this bullet map for the  $\lambda$ -calculus, going back to Gross, Knuth, and Takahashi among others, see [4, 15].

**Design.** I initially tried formalising the Z-property for the *full superdevelopment* [15, Definition 42] bullet map going back to Aczel, van Raamsdonk and others, cf. [11], inside-out contracting  $\beta$ -redexes starting from  $M$ , but failed. (*Defining* the full superdevelopment map was unproblematic but I failed to *use* it.) It should be of interest to have a *single* generic proof that can be instantiated to both the full *development* and *superdevelopment* bullet maps.

$$\begin{array}{l}
\bullet : \forall \{ \Gamma A \} \rightarrow \Gamma \vdash A \rightarrow \Gamma \vdash A \\
(' x) \bullet = ' x \\
(\mathbb{N} M) \bullet = \mathbb{N} (M \bullet) \\
((\mathbb{N} M) \cdot N) \bullet = M \bullet [ N \bullet ] \\
(M \cdot N) \bullet = (M \bullet) \cdot (N \bullet)
\end{array}$$
Figure 4: Full-development bullet map  $\bullet$

The Agda code for the four key properties below, taken from [15, Sections 3.3.1, 3.4]<sup>5</sup>, with the Z-property the conjunction of ([upperbound](#)) and ([monotonic](#)), fitting on one page (Figure 5) bears witness to the simplicity of the proof of confluence via the Z-property. These 65 loc constitute Part II of the formalisation, its core, the rest being generic / boilerplate code.

- ([extensive](#))  $M \rightarrow_{\beta} M^{\bullet}$  ( $\bullet$  gives an upperbound);
- ([upperbound](#)) if  $M \rightarrow_{\beta} N$  then  $N \rightarrow_{\beta} M^{\bullet}$  ( $\bullet$  gives an upperbound on all 1-step reducts);  $\bullet$
- ([rhss](#))  $(M^{\bullet})^{\sigma^{\bullet}} \rightarrow_{\beta} (M^{\sigma})^{\bullet}$  (applying  $\bullet$  to whole exceeds applying it to parts (of the rhs)); and
- ([monotonic](#)) if  $M \rightarrow_{\beta} N$  then  $M^{\bullet} \rightarrow_{\beta} N^{\bullet}$  ( $\bullet$  is monotonic with respect to reduction).

**Remark.** Each property may be shown either by induction on the term  $M$  (and then distinguishing cases on steps (proofterms)  $M \rightarrow_{\beta} N$  possible from  $M$ ) or by induction on  $M \rightarrow_{\beta} N$  (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 ([rhss](#)) by induction on the term  $M$ , and the properties ([upperbound](#)) and ([monotonic](#)) by induction on  $M \rightarrow_{\beta} N$ .

**Confluence** That the Z-property entails confluence, holds abstractly [15, 8]. We arbitrarily chose to formalise the first proof of it in [15, Lemma 51], factoring [confluence](#) through the Strip Lemma [3, 20] [strip](#), and that through an easy recombination of ([extensive](#)), ([upperbound](#)) and ([rew-monotonic](#)), the lifting of ([monotonic](#)) from steps to reductions. Combining the trivial formalisation of this final part with that of Parts I,II yields some 140 loc<sup>6</sup> confirming our claim.

**Conclusion** Upon sending comments on the module [Confluence](#) to the authors of [20] in the summer of 2021, Jeremy Siek challenged me to formalise the comments myself. Since for a long time I had wanted to learn some Agda and in October of that year I had some time on my hands, I then took up the challenge resulting in the current alternative formalisation of that module. It closely followed the pen-and-paper proof [15] and confirmed my comments (that the module [Confluence](#) could be simplified by basing it on the Z-property; its core is only 65 loc). Formalising was surprisingly smooth (taking a full week only; not knowing Agda), helped by that the code in [20] is well-explained and modularised, allowing for easy reuse and adaptation. Accordingly, the proofs of the key properties in Figure 5 are by direct inductions on terms and steps (proofterms), mostly making use of properties in the modules [Untyped](#) and [Substitution](#).

**Design.** Having renaming *and* substitution *and* parallel substitution causes redundancy in [20].

Whether one can easily formalise a proof of confluence of the  $\lambda$ -calculus is often used as a litmus test for proof assistants so formalisations abound [11]. When starting the endeavour I was not aware of other formalisations of confluence via the Z-property in Agda. At IWC 2023, I learned from Riccardo Treglia that he had supervised the student project in 2019/2020 of Andrea Laretto on formalising the Church–Rosser theorem for the  $\lambda$ -calculus in Agda (not relying on [20]). I have tried to initiate a discussion, but that has not materialised yet. The authors of [20] still have to react to both the original comments and the subsequent formalisation, and the improvements suggested to the untyped  $\lambda$ -calculus and its confluence mostly still apply.

<sup>5</sup>The ([rhss](#)) property here is the one which was intended in [15, 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 *parts* ( $M$  and  $\sigma$ ) of the right-hand side of the term rewrite rule  $(M^{\sigma})$  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.

<sup>6</sup>Of which 50 loc show the  $\lambda$ -calculus in the module [Untyped](#) is a term rewrite system so belong in that module, leaving 90 loc for confluence itself in the module [Untyped](#).

## References

- [1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991. doi:10.1017/S095679680000186.
- [2] T. Altenkirch and B. Reus. Monadic presentations of lambda terms using generalized inductive types. In *Computer Science Logic (CSL '99), Madrid, 1999*, volume 1683 of *Lecture Notes in Computer Science*, pages 453–468. Springer, 1999. doi:10.1007/3-540-48168-0\\_32.
- [3] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103. North-Holland, Amsterdam, 2nd revised edition, 1984.
- [4] 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. URL: <https://dspace.library.uu.nl/handle/1874/15119>.
- [5] R.S. Bird and R.A. Paterson. de Bruijn notation as a nested datatype. *Journal of Functional Programming*, 9(1):77–91, 1999. doi:10.1017/S0956796899003366.
- [6] H.J.S. Bruggink. *Equivalence of Reductions in Higher-Order Rewriting*. PhD thesis, Utrecht University, 2008. URL: <https://dspace.library.uu.nl/handle/1874/27575>.
- [7] 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)*, 75(5):381–392, 1972. doi:10.1016/1385-7258(72)90034-0.
- [8] F.L.C. de Moura and L.O. Rezende. A formalization of the (compositional) Z property. Conference on Intelligent Computer Mathematics, 2021. URL: <http://flaviomoura.info/files/fmm21.pdf>.
- [9] P. Dehornoy. *Braids and Self-Distributivity*, volume 192 of *Progress in Mathematics*. Birkhäuser, 2000.
- [10] 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*. URL: <http://www.javakade.nl/research/talk/lmrc060508.pdf>.
- [11] B. Felgenhauer, J. Nagele, V. van Oostrom, and C. Sternagel. The Z property. *Archive of Formal Proofs*, June 2016. URL: [https://www.isa-afp.org/entries/Rewriting\\_Z.shtml](https://www.isa-afp.org/entries/Rewriting_Z.shtml).
- [12] D. Hendriks and V. van Oostrom.  $\lambda$ . In *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.
- [13] G. Huet. Residual theory in  $\lambda$ -calculus: a formal development. *Journal of Functional Programming*, 4(3):371–394, July 1994. doi:10.1017/S0956796800001106.
- [14] R. Loader. Notes on simply typed lambda calculus. ECS-LFCS- 98-381, Laboratory for Foundations of Computer Science, The University of Edinburgh, February 1998. URL: <http://www.lfcs.inf.ed.ac.uk/reports/98/ECS-LFCS-98-381/>.
- [15] V. van Oostrom. Z; syntax-free developments. In *6th Conference on Formal Structures for Computation and Deduction (FSCD 2021), Buenos Aires*, volume 195 of *LIPICs*, pages 24:1–24:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICs.FSCD.2021.24.
- [16] V. van Oostrom, K.J. van de Looij, and M. Zwitserlood. Lambdascope. In *Workshop on Algebra and Logic on Programming Systems, Kyoto*, page 9 pp., April 2004. URL: <http://www.javakade.nl/research/pdf/lambdascope.pdf>.
- [17] S. Schäfer, G. Smolka, and T. Tebbi. Completeness and decidability of de Bruijn substitution algebra in Coq. In *Proceedings of the 2015 Conference on Certified Programs and Proofs (CPP 2015), India*, pages 67–73. ACM, 2015. doi:10.1145/2676724.2693163.
- [18] M. Takahashi. Parallel reductions in  $\lambda$ -calculus. *Information and Computation*, 118:120–127, April 1995. doi:10.1006/inco.1995.1057.
- [19] Terese. *Term Rewriting Systems*. Cambridge University Press, 2003.
- [20] P. Wadler, W. Kokke, and J.G. Siek. *Programming Language Foundations in Agda*. 2024. <http://plfa.inf.ed.ac.uk/20.07/>, <http://plfa.inf.ed.ac.uk/22.08/>, <http://plfa.inf.ed.ac.uk/>.

```

extensive : ∀ {Γ A} → (M : Γ ⊢ A) → M →→ M •
extensive (' _) = - ■
extensive (λ M) = abs-cong (extensive M)
extensive ((λ 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)

upperbound : ∀ {Γ} → {M N : Γ ⊢ ★}
  → M →→ N
  -----
  → N →→ M •
upperbound { _ } { λ _ } (ζ φ) = abs-cong (upperbound φ)
upperbound { _ } { (' _) · _ } { _ } (ξ2 φ) = appR-cong (upperbound φ)
upperbound { _ } { (λ _) · M } { ((λ _) · M) } (ξ1 (ζ φ)) = - →→⟨ β ⟩ rew-rew (upperbound φ) (extensive M)
upperbound { _ } { (λ L) · _ } { ((λ L) · _) } (ξ2 φ) = - →→⟨ β ⟩ rew-rew (extensive L) (upperbound φ)
upperbound { _ } { (λ 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)

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 (λ 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 (λ _) _ = (- →→⟨ β ⟩) - ■
  app-bullet (_ · _) _ = - ■
rhss ((λ 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)

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 } { (λ 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 { _ } { (λ _) · _ } { (λ _) · _ } (ξ1 (ζ φ)) = rew-rew (monotonic φ) (- ■)
monotonic { _ } { (λ M) · _ } { ((λ M) · _) } (ξ2 φ) = rew-rew (M • ■) (monotonic φ)
monotonic { _ } { _ · _ · _ } { (λ _) · _ } (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)

```

Figure 5: Key properties (extensive,upperbound,rhss,monotonic) for confluence of λ-calculus by Z