Strong Normalization for CoC and CIC

Namdak Tonpa

May 2025

Abstract

This article develops a specialized framework for proving strong normalization in the Calculus of Constructions (CoC) and the Calculus of Inductive Constructions (CIC). Building on Girard's normalization framework, we adapt neutral terms, elimination contexts, and reducibility candidates to handle dependent types, universes, inductive types, and general induction. The framework is formalized with definitions, lemmas, and a proof of strong normalization, explicitly addressing the complexities of general induction. Applications to Coq's type theory are discussed, emphasizing the framework's modularity and robustness.

1 Strong Normalization for CoC and CIC

The Calculus of Constructions (CoC) [15] is a dependently typed lambda calculus with impredicative universes, forming the core of many proof assistants. The Calculus of Inductive Constructions (CIC) [5] extends CoC with inductive types and general induction principles, enabling expressive data structures and proofs, as seen in Coq. Strong normalization, ensuring that every well-typed term reduces to a normal form in finitely many steps, is essential for the consistency of these systems.

The Riba's work *Toward a General Rewriting-Based Framework for Reducibility* [1], provides a unified approach to reducibility proofs using rewriting relations and elimination contexts. This article presents a specialized framework for CoC and CIC, adapting Girard concepts to their dependent types, universes, inductive types, and the general induction principle of CIC. We formalize neutral terms, elimination contexts, and reducibility candidates, proving strong normalization and addressing the complexities of general induction. The framework's modularity makes it suitable for Coq and extensible to other dependently typed systems.

Syntax

We define the syntax for CoC and CIC, including the general induction principle. The set of terms $\mathcal{T}$ in CoC and CIC is defined as:

$$t ::= x \mid \text{Sort}\, s \mid \Pi(.x : A).B \mid \lambda x : A.b \mid f\, a$$

$$\mid \text{Ind}(I : A)\{c_1 : C_1, \ldots, c_n : C_n\} \mid \text{Constr}(i, I, t_1, \ldots, t_m) \mid \text{case}(t, I, P, b_1, \ldots, b_n)$$

where: - x is a variable, - $\text{Sort}\, s$ represents universes ($s = \text{Prop}, \text{Type}_i$), - $\Pi(.x : A).B$ is a dependent function type, - $\lambda x : A.b$ is a lambda abstraction, - $f\, a$ is an application, - $\text{Ind}(I : A)\{c_1 : C_1, \ldots, c_n : C_n\}$ defines an inductive type I with constructors $c_i : C_i$, - $\text{Constr}(i, I, t_1, \ldots, t_m)$ is the i-th constructor of I, - $\text{case}(t, I, P, b_1, \ldots, b_n)$ is a dependent case expression for general induction on I.

CoC includes only the first five constructs ($x, \text{Sort}, \Pi, .\lambda, .\ \ $), while CIC adds inductive types, constructors, and case expressions.

Semantics

Here we define typing rules, and rewriting relations for CoC and CIC, including the general induction principle.

1.1 Rewriting Relation

The rewriting relation $\rightarrow \subseteq \mathcal{T} \times \mathcal{T}$ includes: - Beta-reduction: $(\lambda x : A.b)\, a \rightarrow [x \mapsto a]b$. - Inductive reduction (iota-reduction): For an inductive type $\text{Ind}(I : A)\{c_1 : C_1, \ldots, c_n : C_n\}$, if $t = \text{Constr}(i, I, t_1, \ldots, t_m)$, then:

$$\text{case}(t, I, P, b_1, \ldots, b_n) \rightarrow b_i\, t_1 \ldots t_m$$

where b_i is the branch corresponding to constructor c_i.

A term t is *strongly normalizing* if every reduction sequence starting from t is finite. Typing judgments are of the form $\Gamma \vdash t : A$, where $\Gamma = [x_1 : A_1, \ldots, x_n : A_n]$ is a context.

1.2 General Induction in CIC

The general induction principle (dependent elimination) allows reasoning about inductive types with dependent predicates. For an inductive type $\text{Ind}(I : A)\{c_1 : C_1, \ldots, c_n : C_n\}$, the case expression $\text{case}(t, I, P, b_1, \ldots, b_n)$ has type $P\, t$, where: - $P : \Pi(.x : I).\text{Sort}\, s$ is a dependent predicate, - Each branch $b_i : \Pi(.y_1 : T_1). \ldots \Pi(.y_m : T_m).P\, \text{Constr}(i, I, y_1, \ldots, y_m)$ corresponds to constructor c_i.

This principle generalizes simple pattern matching by allowing the result type to depend on the scrutinized term t.

2 Neutral Terms

Neutral terms are defined to exclude terms that trigger immediate reductions, accommodating both beta- and iota-reductions in CIC.

Definition 2.1 (Neutral Terms)**.** A term $t \in \mathcal{T}$ is *neutral*, denoted $t \in \mathcal{N}$, if it is not a lambda abstraction ($\lambda x : A.b$) or a constructor term ($\text{Constr}(i, I, t_1, \ldots, t_m)$). Formally:

$$\mathcal{N} = \{t \in \mathcal{T} \mid t = x \text{ or } t = \text{Sort}\, s \text{ or } t = \Pi(.x : A).B \text{ or } t = f\, a \text{ or}$$

$$t = \text{case}(t', I, P, b_1, \ldots, b_n) \text{ where } t' \notin \text{Constr}\}$$

Case expressions are neutral unless their scrutinee is a constructor, reflecting the iota-reduction rule [1].

3 Elimination Contexts

Elimination contexts are extended to handle general induction, capturing the reduction behavior of case expressions.

Definition 3.1 (Elimination Contexts)**.** An *elimination context* $E \in \mathcal{E}$ is defined inductively:

$$E ::= [] \mid E\, t \mid \text{case}(E, I, P, b_1, \ldots, b_n), \quad t, b_i \in \mathcal{T}$$

The application $E[t]$ is: - $[][t] = t$, - $E\, u[t] = E[t]\, u$, - $\text{case}(E, I, P, b_1, \ldots, b_n)[t] = \text{case}(E[t], I, P, b_1, \ldots, b_n)$.

A set $\mathcal{E}$ is *adequate* if: 1. **Closure under composition**: If $E_1, E_2 \in \mathcal{E}$, then $E_1[E_2] \in \mathcal{E}$. 2. **Stability under reduction**: If $E[t] \to t'$, then either $t' = E'[t]$ for some $E' \in \mathcal{E}$, or $t' \in \mathcal{N}$, or $t' = \text{Constr}(i, I, t_1, \ldots, t_m)$.

The inclusion of dependent case expressions ensures that general induction is modeled correctly [2].

4 Reducibility Candidates

Reducibility candidates are defined to ensure strong normalization, accommodating dependent types, universes, and inductive types with general induction.

Definition 4.1 (Reducibility Candidates)**.** For a type $A \in \mathcal{A}$, a set $\mathcal{R}_A \subseteq \mathcal{T}$ is a *reducibility candidate* if:

1. **Strong normalization**: If $t \in \mathcal{R}_A$, then t is strongly normalizing.
2. **Closure under reduction**: If $t \in \mathcal{R}_A$ and $t \to t'$, then $t' \in \mathcal{R}_A$.
3. **Neutral terms**: If $t \in \mathcal{N}$ and for all $t \to t'$, $t' \in \mathcal{R}_A$, then $t \in \mathcal{R}_A$.
4. **Dependent types**: If $A = \Pi(.x : B).C$, then $t \in \mathcal{R}_A$ if for all $u \in \mathcal{R}_B$, $t\,u \in \mathcal{R}_{[x \mapsto u]C}$.
5. **Universes**: If $A = \operatorname{Sort} s$, then $\mathcal{R}_A$ contains all strongly normalizing terms of type s.
6. **Inductive types**: If $A = \operatorname{Ind}(I : A')$, then $\mathcal{R}_A$ contains all terms t such that for any $\operatorname{case}(t, I, P, b_1, \ldots, b_n)$, the result is in $\mathcal{R}_{P\,t}$.

For inductive types, the reducibility candidate ensures that terms behave correctly under general induction, reflecting the dependent nature of case expressions [2].

5 Strong Normalization for CoC and CIC

We prove strong normalization using the adapted reducibility framework, explicitly handling general induction.

Theorem 5.1 (Strong Normalization)**.** *For any context* Γ *and term* t*, if* $\Gamma \vdash t : A$*, then* $t \in \mathcal{R}_A$*, and thus* t *is strongly normalizing.*

Proof. The proof proceeds by induction on the typing derivation $\Gamma \vdash t : A$.

1. Case: $t = x$ If $\Gamma \vdash x : A$, then $(x : A) \in \Gamma$. Since $x \in \mathcal{N}$ and has no reductions, x *Radiolabelled*$\mathcal{R}_A$.

2. Case: $t = \mathrm{Sort}\, s$ If $\Gamma \vdash \mathrm{Sort}\, s : \mathrm{Sort}\, s'$, then $\mathrm{Sort}\, s \in \mathcal{N}$ and is irreducible, so $\mathrm{Sort}\, s \in \mathcal{R}_{\mathrm{Sort}\, s'}$.

3. Case: $t = \Pi(.x : A).B$ If $\Gamma \vdash A : \mathrm{Sort}\, s_1$, $\Gamma, x : A \vdash B : \mathrm{Sort}\, s_2$, then $\Gamma \vdash \Pi(.x : A).B : \mathrm{Sort}\, s$. By induction, $A \in \mathcal{R}_{\mathrm{Sort}\, s_1}$, $B \in \mathcal{R}_{\mathrm{Sort}\, s_2}$. Since $\Pi(.x : A).B \in \mathcal{N}$, it is in $\mathcal{R}_{\mathrm{Sort}\, s}$ if all reducts are, which holds trivially.

4. Case: $t = \lambda x : A.b$ If $\Gamma \vdash A : \mathrm{Sort}\, s$, $\Gamma, x : A \vdash b : B$, then $\Gamma \vdash \lambda x : A.b : \Pi(.x : A).B$. By induction, for all $u \in \mathcal{R}_A$, $[x \mapsto u]b \in \mathcal{R}_{[x \mapsto u]B}$. Thus, $(\lambda x : A.b)\, u \to [x \mapsto u]b \in \mathcal{R}_{[x \mapsto u]B}$, so $\lambda x : A.b \in \mathcal{R}_{\Pi(.x:A).B}$.

5. Case: $t = f\, a$ If $\Gamma \vdash f : \Pi(.x : A).B$, $\Gamma \vdash a : A$, then $\Gamma \vdash f\, a : [x \mapsto a]B$. By induction, $f \in \mathcal{R}_{\Pi(.x:A).B}$, $a \in \mathcal{R}_A$. Thus, $f\, a \in \mathcal{R}_{[x \mapsto a]B}$.

6. Case: $t = \mathrm{Ind}(I : A)\{c_1 : C_1, \ldots, c_n : C_n\}$ If $\Gamma \vdash A : \mathrm{Sort}\, s$, and each constructor $c_i : C_i$ is well-typed, then $\Gamma \vdash I : A$. By induction, $A \in \mathcal{R}_{\mathrm{Sort}\, s}$, and each $C_i \in \mathcal{R}_{\mathrm{Sort}\, s_i}$. Thus, $I \in \mathcal{R}_A$.

7. Case: $t = \mathrm{Constr}(i, I, t_1, \ldots, t_m)$ If $\Gamma \vdash \mathrm{Constr}(i, I, t_1, \ldots, t_m) : I\, u_1 \ldots u_k$, then each $t_j \in \mathcal{R}_{T_j}$ by induction. Although Constr is not neutral, its arguments are reducible, and any case on Constr reduces to a branch in $\mathcal{R}$, ensuring $t \in \mathcal{R}_{I\, u_1 \ldots u_k}$.

8. Case: $t = \mathrm{case}(t', I, P, b_1, \ldots, b_n)$ If $\Gamma \vdash t' : I\, u_1 \ldots u_k$, $\Gamma \vdash P : \Pi(.x : I).\mathrm{Sort}\, s$, and each branch $b_i : \Pi(.y_1 : T_1). \ldots \Pi(.y_m : T_m).P\, \mathrm{Constr}(i, I, y_1, \ldots, y_m)$, then $\Gamma \vdash \mathrm{case}(t', I, P, b_1, \ldots, b_n) : P\, t'$. By induction: - $t' \in \mathcal{R}_{I\, u_1 \ldots u_k}$, - $P \in \mathcal{R}_{\Pi(.x:I).\mathrm{Sort}\, s}$, - Each $b_i \in \mathcal{R}_{\Pi(.y_1:T_1). \ldots \Pi(.y_m:T_m).P\, \mathrm{Constr}(i,I,y_1,\ldots,y_m)}$. If $t' = \mathrm{Constr}(i, I, t_1, \ldots, t_m)$, then:

$$\mathrm{case}(t', I, P, b_1, \ldots, b_n) \to b_i\, t_1 \ldots t_m$$

Since b_i is reducible and each $t_j \in \mathcal{R}_{T_j}$, the result is in $\mathcal{R}_{P\, \mathrm{Constr}(i,I,t_1,\ldots,t_m)}$. If $t' \in \mathcal{N}$, the case expression is neutral, and all its reducts are in $\mathcal{R}_{P\, t'}$ by induction. Thus, $t \in \mathcal{R}_{P\, t'}$.

Since $\mathcal{R}_A$ contains only strongly normalizing terms, $t \in \mathcal{R}_A$ implies t is strongly normalizing [1, 2]. □

6 Comparison with Other Frameworks

Compared to other normalization proofs: - Girard's Candidates: Effective for CoC but less modular for CIC's inductive types and general induction [14]. - Werner's Proof: Specific to CIC, addressing general induction but less general for rewriting [2]. - Normalization by Evaluation (NbE): Semantic and efficient but complex for general induction [3].

7 Conclusion

This specialized framework extends Riba's rewriting-based reducibility approach to prove strong normalization for CoC and CIC, explicitly incorporating the general induction principle of CIC. By formalizing neutral terms, elimination contexts, and reducibility candidates tailored to dependent types, universes, inductive types, and dependent case expressions, it provides a robust tool for Coq's type theory. The framework's modularity supports extensions like universe polymorphism and guarded recursion, making it a versatile foundation for future research in dependently typed systems.

References

Metatheory

[1] Riba, C. (2008). Toward a General Rewriting-Based Framework for Reducibility. Submitted, available from the author's homepage.

[2] Werner, B. (1994). *Une Théorie des Constructions Inductives*. PhD thesis, Université Paris 7.

[3] Abel, A., & Sattler, C. (2012). Normalization by Evaluation for Call-by-Push-Value and Polarized Lambda Calculus. `https://hal.science/hal-00779623/document`.

[4] Harper, R., & Licata, D. (2007). Mechanizing Metatheory in a Logical Framework. *Journal of Functional Programming*, 17(4-5), 613–673.

CIC

[5] Coquand, T., & Paulin-Mohring, C. (1990). Inductively Defined Types. *Proceedings of the International Conference on Computer Logic (COLOG-88)*, Lecture Notes in Computer Science, 417, 50–66.

[6] Paulin-Mohring, C. (1992). Inductive Definitions in the System Coq: Rules and Properties. *Proceedings of the First International Conference on Typed Lambda Calculi and Applications (TLCA)*, Lecture Notes in Computer Science, 664, 328–345.

[7] Paulin-Mohring, C. (2014). Introduction to the Calculus of Inductive Constructions. *All about Proofs, Proofs for All*, HAL Archives, `https://inria.hal.science/hal-01094195/document`.

[8] Pfenning, F., & Paulin-Mohring, C. (1989). Inductively Defined Types in the Calculus of Constructions. *Proceedings of Mathematical Foundations of Programming Semantics (MFPS)*, Lecture Notes in Computer Science, 442, 209–228. `https://www.cs.cmu.edu/~fp/papers/mfps89.pdf`.

[9] Asperti, A., Ricciotti, W., Sacerdoti Coen, C., & Tassi, E. (2009). A Compact Kernel for the Calculus of Inductive Constructions. *Sadhana*, 34(1), 71–90. `https://www.cs.unibo.it/~sacerdot/PAPERS/sadhana.pdf`.

[10] Dybjer, P. (1997). Inductive Families. *Formal Aspects of Computing*, 9(4), 329–354.

[11] Bezem, M., Coquand, T., Dybjer, P., & Escardó, M. (2024). Type Theory with Explicit Universe Polymorphism. *arXiv preprint*, `https://arxiv.org/pdf/2212.03284`.

PTS

[12] Coquand, T. (1996). An Algorithm for Type-Checking Dependent Types. *Science of Computer Programming*, 26(1-3), 167–177.

[13] de Bruijn, N. G. (1972). Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem. *Indagationes Mathematicae*, 34(5), 381–392.

[14] Girard, J.-Y. (1972). *Interprétation Fonctionnelle et Élimination des Coupures de l'Arithmétique d'Ordre Supérieur*. PhD thesis, Université Paris 7.

[15] Coquand, T., & Huet, G. (1988). The Calculus of Constructions. *Information and Computation*, 76(2-3), 95–120. `https://core.ac.uk/download/pdf/82038778.pdf`.