Why Cyclic CIC? (Motivation, Calculi, Transformations)

(Scidonia/cyclic notes)

January 30, 2026

Abstract

This paper-style note motivates a cyclic variant of a Calculus of Inductive Constructions (CIC)-like calculus. The central claim is that *the induction/recursion principle need not be fixed syntactically*: instead, proofs can be finite graphs with cycles, validated by a global progress condition. This extra freedom makes it possible for program/proof transformations (commuting conversions, unfolding/refolding, etc.) to *identify* proofs that were previously distinct—in the strongest sense, they may become literally the same cyclic object after normalization. Such "post-hoc proof identity" is hard to express in systems where recursion is baked in as a fixed-point operator or as a rigid family of induction schemata. We then outline (i) the source term calculus, (ii) a fix-free cyclic proof/term language with explicit substitution evidence, (iii) the global condition on cycles, and (iv) representative transformations like CaseCase and head beta-reduction.

Contents

1 Introduction

The design goal of this project is not simply to add "cycles" for convenience. The goal is to change what we treat as *primitive* in a proof calculus.

In conventional systems, recursive reasoning is packaged as a fixed-point operator or as a family of induction/recursion schemata. Those choices are powerful, but they also hard-code a particular presentation of recursion into the syntax of proofs. As a result, proofs that are semantically equivalent can remain syntactically far apart, even after many semantics-preserving transformations.

Cyclic proof theory suggests a different decomposition:

> A proof is a finite graph of locally-correct steps; recursion is represented by cycles; and soundness is ensured by a global progress condition.

This decomposition has an attractive consequence for proof engineering and normalization: we can refactor recursive structure (fold/refold), commute conversions, and perform administrative normalization without being forced to preserve the exact boundaries of `fix` binders or induction schemata. In the best case, normalization can reveal that two proofs are not just equivalent, but *the same cyclic object* after refolding.

The rest of the paper outlines:

- the source CIC-like term calculus (Section 4);
- the cyclic proof/term graph architecture (Section 5);
- the global condition on cycles (Section 6);
- and representative transformations like CaseCase and head beta-reduction (Section 7).

2 Prior work and relation to Cyclic CIC

This section positions the present development relative to (i) cyclic proof theory, (ii) proof-theoretic treatments of induction via global discharge conditions, and (iii) CIC kernel engineering.

Cyclic proof theory: from concrete systems to abstract frameworks. Cyclic proof systems represent potentially infinite derivations using finite graphs with back-edges, validated by a global soundness condition. This approach originates in cyclic proofs for inductive definitions and infinite descent [3, 4]. Afshari and Wehr develop a modern, highly abstract account of cyclic proof checking based on trace conditions, activation algebras, and categorical structure [1]. Our work is complementary in two ways: (1) we instantiate cyclic proofs inside a typed setting (a CIC-like calculus with evaluation and CIU (*closed instantiations of uses*) observational equivalence); (2) we focus on using cyclic proofs as *normal forms stable under program/proof transformations* (commuting conversions, unfold/refold, etc.). In particular, many of our proofs of interest are about preservation of observational equivalence under transformations rather than about the abstract complexity of proof checking.

Local induction vs global discharge. Sprenger and Dam compare a proof system with a local well-founded induction rule to a cyclic system validated by a global induction discharge condition, and give effective translations between the two (in the setting of the first-order μ-calculus) [8]. We use this as direct prior evidence for the core design thesis of this project: *the induction principle need not be fixed syntactically.* Where Sprenger–Dam study the relationship between local and global presentations of induction in a fixed-point logic, we pursue the same decomposition in a dependent type-theoretic setting, and then exploit it operationally: cycles become the unit of fold/refold during normalization and compilation of proof objects.

Engineering CIC kernels. Asperti et al. give a detailed kernel-level account of CIC checking in the Matita prover, emphasizing compactness and clear engineering boundaries [2]. Our contribution is not a new kernel design; instead, we propose a proof-object language where recursive structure is externalized as graph structure plus a global condition. Nevertheless, the kernel perspective is relevant: if cyclic proofs are to support a strong notion of post-transformation proof identity, then proof checking and definitional equality need representations that survive aggressive normalization and refolding.

Supercompilation and termination control. Supercompilation is a semantics-driven program transformation methodology centered on symbolic evaluation ("driving"), generalization, and fold/refold steps [9, 7]. A key practical ingredient is a *control* (or "whistle") mechanism preventing infinite unfolding, often based on homeomorphic embedding and Kruskal-style well-quasi-ordering arguments [5, 6]. Our stance is that cyclic proof normalization is *precisely supercompilation*—but performed at the level of typed judgements/proof objects rather than untyped programs. The global cycle condition and the control relation play the same role they do in supercompilation: they justify refolding and prevent uncontrolled infinite unfolding.

3 Why go cyclic? Proof identity and non-fixed induction principles

A standard way to represent recursive reasoning in type theory is to bake recursion into the term language:

- either directly, via a fixed-point constructor (`fix`), or
- indirectly, via a pre-chosen set of induction/recursion schemata tied to the type formers.

This choice has a subtle but important consequence: it commits the system to a *particular syntactic presentation* of recursive reasoning. Two proofs may establish the same semantic property, but remain permanently different pieces of syntax because they encode recursion using different unfoldings, different induction schemata, or different administrative structure.

3.1 The core motivation: induction principles need not be fixed

Cyclic proof theory takes a different stance:

> The recursive structure of a proof is not a primitive binder or a fixed schema; it is a *cycle* in a finite proof graph.

A cycle does not, by itself, guarantee soundness. Instead we equip the finite graph with a *global progress condition* (Section 6) that rules out "bad" cycles and validates the intended infinite unfolding.

The key conceptual gain is *modularity of recursion*: we can change the cyclic structure (by folding, refolding, or commuting constructors through eliminators) without changing the local steps of the proof. In a fixed-point calculus, the recursion principle is pinned to the syntax of `fix`; in a schema-based setting it is pinned to the chosen eliminators. In a cyclic calculus, the recursion principle is an emergent property of the graph plus the global condition.

3.2 Post-transformation proof identity

This flexibility is not merely aesthetic. It enables a strong normalisation/canonicalisation story for proofs:

- Start from two proofs that are syntactically different (for example: one uses an unfolding step early, another delays it; one distributes a case earlier; one uses a different cycle boundary).
- Apply a sequence of semantics-preserving transformations (Section 7).
- After normalization/refolding, the resulting cyclic proof graphs may become *isomorphic* or even *literally identical* as finite objects.

In other words, transformations can expose that two proofs are "the same proof" once recursion is treated as *structure* (cycles) rather than as a fixed syntactic operator or schema. This is precisely the kind of identification that is difficult in traditional settings: the syntactic boundary of a `fix` binder (or the syntactic choice of induction schema) tends to be rigid, so normalization can preserve semantics while still leaving behind irreducible syntactic differences.

Cyclic proofs aim to make such equivalences more intensional: if two proofs normalize to the same cyclic representative (up to a chosen notion of graph equivalence such as bisimulation), we can treat them as equal in a strong, structurally meaningful sense.

4 Source calculus (terms, typing, semantics)

This section summarizes the *source* term language and its typing and operational semantics. It corresponds closely to:

- term syntax: `theories/Syntax/Term.v`
- typing: `theories/Judgement/Typing.v` (inductive judgement `has_type`)
- call-by-name semantics: `theories/Semantics/Cbn.v`

4.1 Term syntax

We write terms t, u and types A, B using the following constructors (de Bruijn indices for variables):

$$\begin{aligned} t \quad ::= \quad & \mathsf{Var}\, x \mid \mathsf{Sort}\, i \mid \Pi(A).\, B \mid \lambda(A).\, t \mid t\, u \mid \mathsf{fix}(A).\, t \\ & \mid \mathsf{Ind}\, I \mid \mathsf{roll}_c^I(\overline{p}; \overline{r}) \mid \mathsf{case}^I(t; C; \overline{br}) \end{aligned}$$

4.2 Typing

A typing environment Σ stores inductive signatures (strictly-positive definitions). A context Γ is a list of types. The main judgement is $\Sigma;\, \Gamma \vdash t : A$.

Below is the core typing system (informally mirroring `Typing.has_type`). We omit the full inductive signature side-conditions and arity bookkeeping, but include the essential shape.

$$\frac{\Gamma(x) = A}{\Sigma;\, \Gamma \vdash \mathsf{Var}\, x : A}\ (\textsc{TyVar})$$

$$\frac{}{\Sigma;\, \Gamma \vdash \mathsf{Sort}\, i : \mathsf{Sort}\, i + 1}\ (\textsc{TySort})$$

$$\frac{\Sigma;\, \Gamma \vdash A : \mathsf{Sort}\, i \qquad \Sigma;\, A :: \Gamma \vdash B : \mathsf{Sort}\, j}{\Sigma;\, \Gamma \vdash \Pi(A).\, B : \mathsf{Sort}\, \max(i, j)}\ (\textsc{TyPi})$$

$$\frac{\Sigma;\, \Gamma \vdash A : \mathsf{Sort}\, i \qquad \Sigma;\, A :: \Gamma \vdash t : B}{\Sigma;\, \Gamma \vdash \lambda(A).\, t : \Pi(A).\, B}\ (\textsc{TyLam})$$

$$\frac{\Sigma;\, \Gamma \vdash t : \Pi(A).\, B \qquad \Sigma;\, \Gamma \vdash u : A}{\Sigma;\, \Gamma \vdash t\, u : B[u/0]}\ (\textsc{TyApp})$$

$$\frac{\Sigma;\, \Gamma \vdash A : \mathsf{Sort}\, i \qquad \Sigma;\, A :: \Gamma \vdash t : A \uparrow}{\Sigma;\, \Gamma \vdash \mathsf{fix}(A).\, t : A}\ (\textsc{TyFix})$$

Inductives, constructors, and case. The development includes strictly-positive inductives $\mathsf{Ind}\, I$, constructor values $\mathsf{roll}_c^I(\overline{p}; \overline{r})$, and elimination by $\mathsf{case}^I(t; C; \overline{br})$. Typing is guided by an inductive signature lookup (see `StrictPos.v` and `Typing.v`). Branches are typed by building an iterated Pi-type over the constructor arguments.

4.3 Call-by-name operational semantics

We use a weak-head, call-by-name reduction relation $t \to u$ (see `Semantics/Cbn.v`). Key rules include:

$$\frac{}{(\lambda(A).\, t)\, u \to t[u/0]}\ (\textsc{Step-Beta})$$

$$\frac{t \to t'}{t\, u \to t'\, u}\ (\textsc{Step-App1})$$

$$\frac{}{\mathsf{fix}(A).\, t \to t[\mathsf{fix}(A).\, t/0]}\ (\textsc{Step-Fix})$$

$$\frac{t \to t'}{\mathsf{case}^I(t; C; \overline{br}) \to \mathsf{case}^I(t'; C; \overline{br})}\ (\textsc{Step-CaseScrut})$$

$$\frac{branch(\overline{br}, c) = br}{\mathsf{case}^I((\mathsf{roll}^I_c(\overline{p}; \overline{r})); C; \overline{br}) \to br\ (\overline{p} +\!\!+ \overline{r})}\ (\textsc{Step-CaseRoll})$$

We write $t \to^* u$ for the reflexive-transitive closure of $\to$. Values are λ-abstractions and constructor rolls.

5 Cyclic CIC objects (proof graphs and fix-free vertex language)

There are two related layers in the development:

(1) a generic notion of *preproof / cyclic proof* as a finite digraph whose nodes are labelled by judgements;

(2) a concrete *fix-free* vertex language (nodes like `nLam`, `nApp`, `nBack`) used by read-off/extraction.

5.1 Preproofs: locally-correct finite proof graphs

A preproof is a finite directed graph together with:

- a node-label function $\mathsf{label} : V \to \mathsf{Judgement}$
- a successor function $\mathsf{succ} : V \to \mathrm{list}\ V$
- a local correctness property stating that each node is justified by a rule instance: $\mathsf{Rule}(\mathsf{label}(v),\ \mathrm{map}\ \mathsf{label}\ (\mathsf{succ}($

This corresponds to `theories/Preproof/Preproof.v`. A *rooted* preproof additionally designates a root vertex.

5.2 Cyclic proofs: global condition + witness

A cyclic proof packages a preproof together with an additional witness W and a predicate $\mathsf{progress\_ok}$ establishing the global soundness condition. See `theories/CyclicProof/CyclicProof.v`.

5.3 Fix-free vertex language (used by read-off)

The read-off compiler produces a fix-free cyclic term graph whose node labels are:

$$\begin{array}{l}\texttt{nVar}\ x \mid \texttt{nSort}\ i \mid \texttt{nPi} \mid \texttt{nLam} \mid \texttt{nApp} \\ \texttt{nInd}\ I \mid \texttt{nRoll}\ (I, c, n_p, n_r) \mid \texttt{nCase}\ (I, n_{br}) \\ \texttt{nSubstNil}\ k \mid \texttt{nSubstCons}\ k \mid \texttt{nBack}\end{array}$$

Key design points (see `theories/Transform/ReadOff.v`):

- There is *no* `fix` node label.
- Recursion is represented by `nBack` nodes, which point to: (i) a cycle-target vertex, and (ii) an explicit substitution-evidence vertex.
- Substitutions are represented as linked lists of evidence nodes: `nSubstNil` and `nSubstCons`, mirroring list-backed substitutions.

5.4 Judgements over vertices

A first-cut judgement language over vertices (see `theories/Transform/CyclicRules.v`) is:

$$\mathsf{jTy}\ \Gamma\ v\ A \quad | \quad \mathsf{jEq}\ \Gamma\ v\ w\ A \quad | \quad \mathsf{jSub}\ \Delta\ sv\ \Gamma$$

Intuitively:

- $\mathsf{jTy}\ \Gamma\ v\ A$: vertex v has type-vertex A under context of vertices Γ.
- $\mathsf{jSub}\ \Delta\ sv\ \Gamma$: substitution vertex sv is well-typed as a substitution from Γ into Δ.
- jEq provides intensional equality structure (initially refl/symm/trans; computation rules are planned).

The rules reference abstract graph-level operations shiftV and substV, intended to be implemented by fix-free rewriting (or by derived operations on the graph).

Backlink rule (core idea). A backlink node `nBack` represents a recursive call as an instantiation of an earlier obligation:

$$\frac{\begin{array}{c}\mathsf{jTy}\ \Gamma_0\ \mathit{target}\ A_0 \\ \mathsf{jSub}\ \Gamma\ sv\ \Gamma_0\end{array}}{\mathsf{jTy}\ \Gamma\ \mathit{back}\ (\mathsf{substV}\ sv\ A_0)}\ (\textsc{TyBack})$$

This is where recursion lives in the cyclic object: the graph edge from *back* to *target* closes the cycle, while *sv* records how the recursive call is instantiated.

6 Global conditions on cycles

Local correctness is not enough: an arbitrary cyclic preproof may be unsound. Cyclic proof theory therefore adds a *global* condition to rule out "bad" cycles.

The repository currently develops a ranking-style template (see `theories/Progress/Ranking.v`).

6.1 Ranking condition (template)

Fix a finite digraph G with vertices V. Assume:

- a predicate $\mathsf{progress\_edge}(v, w)$ selecting edges that must make strict progress,
- a ranking domain M with well-founded strict order $<_M$,
- a rank function $\mathsf{rank} : V \to M$.

The ranking condition requires:

(a) **Well-foundedness**: $<_M$ is well-founded.

(b) **Monotonicity**: along every edge $v \to w$, ranks do not increase.

(c) **Strictness on progress edges**: if $v \to w$ is a progress edge, then $\mathsf{rank}(w) <_M \mathsf{rank}(v)$.

(d) **Every directed cycle has progress**: every directed cycle in G contains at least one progress edge.

6.2 Why this matters for transformations

Transformations that introduce, remove, or restructure cycles (fold/refold, read-off compilation, or higher-level normalisation steps) are only meaningful if the resulting cyclic object still has a progress witness.

Conceptually:

> Finite graph + local rule correctness + global cycle progress ⇒ sound infinite unfolding/meaning.

Thus, semantic preservation results for transformations typically factor into:

(1) a *local* preservation argument (the transformation preserves the intended judgement/evaluation meaning), and

(2) a *global* argument transporting the progress witness (or rebuilding one).

7 Transformations: CaseCase, head beta, unfold/refold

This section outlines the main transformations currently being developed at the term level, with the intended lift to cyclic proof objects via extraction. The motivating theme is not merely that these rewrites preserve observational equivalence, but that they support a notion of *canonical cyclic representatives*.

7.1 Head beta-reduction (call-by-name)

The file `theories/Transform/BetaReduce.v` defines a single-step head reduction:

$$\mathsf{beta\_reduce\_once}(t) \triangleq \begin{cases} (\lambda(A).\, body)\ u \mapsto body[u/0] & \text{if } t = (\lambda(A).\, body)\ u \\ t & \text{otherwise.} \end{cases}$$

It also proves a CIU-style correctness theorem for typed terms (see `CIUJudgement.ciu_jTy`):

$$\Sigma;\, \Gamma \vdash t : A \implies \mathsf{ciu\_jTy}\ \Sigma\ \Gamma\ t\ (\mathsf{beta\_reduce\_once}(t))\ A.$$

7.2 CaseCase: commuting nested case

The file `theories/Transform/CaseCase.v` implements a commuting conversion for nested cases. Informally:

$$\mathsf{case}^J((\mathsf{case}^I(x; C; \overline{br_I})); D; \overline{br_J}) \quad \rightsquigarrow \quad \mathsf{case}^I(x; C; \overline{br'_I})$$

where each inner branch is transformed to:

$br'_I \triangleq$ (rebuild a lambda prefix for the constructor arity) $\ldots$ then $\mathsf{case}^J((br_I\ \overline{a}); D; \overline{br_J})$.

In the development, the constructor arity/types are obtained from the inductive signature environment, and the branch transform is implemented by a function `commute_branch_typed_rec`.

7.3 Unfolding and refolding of cycles

Beyond term-level rewrites, cyclic proof objects admit transformations that restructure graph sharing and cycles:

- **Unfold**: expand a back-link or a shared subgraph, temporarily increasing size but exposing computation.
- **Fold/refold**: identify repeated sub-derivations and introduce a back-link, or rearrange cycle boundaries (e.g. SCC-normal forms).

These transformations are the point where cyclicity pays off. Because recursion is represented structurally (by cycles) rather than by a fixed syntactic operator, refolding can change *which* cyclic structure represents a proof without changing what it proves. The global progress condition is what makes this sound: it is the invariant that must be preserved or transported when cycles are restructured.

8 Key theorems (mechanized)

This section states the main semantic preservation theorems mechanized in the Coq development. Our semantic equivalence is CIU-style observational equivalence at a typing judgement (values as observables).

8.1 Typed CIU equivalence

Write $\mathsf{ciu\_jTy}(\Sigma, \Gamma, t, u, A)$ for typed CIU equivalence (see `theories/Equiv/CIUJudgement.v`). Intuitively, t and u are equivalent at type A if, for every well-typed closing substitution by values, they terminate to the same values.

8.2 Head beta reduction

Let β_1 be head call-by-name beta reduction (`theories/Transform/BetaReduce.v`).

Theorem 8.1 (Head beta preserves typed CIU)**.** *For all* Σ, Γ, t, A*,*

$$\Sigma;\ \Gamma \vdash t : A \implies \mathsf{ciu\_jTy}(\Sigma, \Gamma, t, \beta_1(t), A).$$

This corresponds to the proved Coq theorem `ciu_jTy_beta_reduce_once`.

8.3 Read-off and extraction round-trip

Let read_off_raw compile a source term to a fix-free cyclic graph, and let extract_read_off be the specialized extraction back to terms. The Coq development proves a strong round-trip equation (`theories/Transform/ReadOffExtractCorrectness.v`).

Theorem 8.2 (Round-trip identity)**.** *For every term t,*

$$\mathsf{extract\_read\_off}(t) = t.$$

An immediate corollary is CIU preservation:

Theorem 8.3 (Round-trip preserves typed CIU)**.** *For all Σ, Γ, t, A,*

$$\mathsf{ciu\_jTy}(\Sigma, \Gamma, t, \mathsf{extract\_read\_off}(t), A).$$

These correspond to the proved Coq theorems `extract_read_off_id` and `extract_read_off_ciu`.

8.4 Case–case commuting conversion

Let $\mathsf{CaseCase}(\Sigma, t)$ be the one-step commuting conversion for nested case (`theories/Transform/CaseCase.v`).

Theorem 8.4 (CaseCase preserves typed CIU)**.** *For all Σ, Γ, t, A,*

$$\Sigma;\, \Gamma \vdash t : A \implies \mathsf{ciu\_jTy}(\Sigma, \Gamma, t, \mathsf{CaseCase}(\Sigma, t), A).$$

This corresponds to the proved Coq theorem `ciu_jTy_commute_case_case_once`.

8.5 Lifting transformations

As a first "lifting" result, the repository contains a term-level wrapper that uses the above CIU theorem to justify the CaseCase transform on extracted cyclic objects (`theories/Transform/CaseCaseProof.v`).

Theorem 8.5 (CaseCase transform preserves typed CIU (by reduction to term theorem))**.** *For all Σ, Γ, p, A, if p is well-typed at A, then applying CaseCase to p preserves typed CIU.*

This corresponds to the proved Coq theorem `case_case_transform_preserves_equiv`.

8.6 Ranking-based global progress (proved)

The cyclic proof layer uses a global condition to validate cycles. One convenient instantiation uses a natural-number ranking that is monotone on all edges and strictly decreasing on designated progress edges.

Proposition 8.6 (Natural-number ranking implies progress condition)**.** *If a finite proof graph admits a rank $\rho : V \to \mathbb{N}$ such that ranks never increase along edges and strictly decrease along progress edges, and every directed cycle contains a progress edge, then the global progress condition holds.*

This corresponds to `progress_ok_nat` in `theories/Progress/Measures.v`.

9 Canonical forms via supercompilation-style control

The long-term goal is a theory of *canonical cyclic proofs*: intensional representatives stable under semantics-preserving transformations. In this project, the intended normalization procedure is explicitly inspired by supercompilation [9, 7].

9.1 Control relation (homeomorphic embedding)

Supercompilation requires a termination control (a "whistle") preventing infinite unfolding. The repository implements a homeomorphic embedding relation on terms and extends it to typed judgements (`theories/Progress/Embedding.v`). We write $j_1 \preceq j_2$ for judgement embedding.

The classical meta-theorem (by Kruskal-style arguments [5]) is that homeomorphic embedding is a well-quasi-order on finite terms, which implies that an infinite sequence must contain an embedding pair. Operationally, this justifies the control rule:

> If a newly produced configuration embeds a previous one, stop unfolding and refold (generalize/backlink) instead.

9.2 Reductions "modulo unfolding"

Let $\to$ be the operational step relation on terms (Section 4). Let $\rightsquigarrow$ denote a single *unfolding* step at a cyclic back-link (or, in the source calculus, unfolding a `fix`). We treat $\rightsquigarrow$ as admissible but controlled: it is only permitted when it does not violate the control relation.

9.3 Canonical form (intended)

We say a cyclic object is in *canonical form* if:

(1) (No further reductions) There is no reduction step $p \to p'$ available in the cyclic object.

(2) (Unfolding is controlled) Any unfolding step $p \rightsquigarrow p_1$ that would enable further reductions $p_1 \to^+ p_2$ is forbidden by the control relation: along the would-be unfolding/reduction trace, some configuration embeds a previous one (a "whistle" blows), so the procedure must refold rather than continue unfolding.

This definition mirrors supercompilation practice: normal forms are not merely "stuck"; they are *maximally reduced subject to a termination control*. The presence of cycles (back-links) turns this into a canonicalization problem for graphs rather than trees: refolding chooses cycle targets and substitutions, and the global progress condition validates that the chosen cycles are productive/sound.

9.4 Connection to transformations

Under this view, transformations like head beta and CaseCase are the "driving" steps, while read-off/extract and fold/refold manipulate the sharing and cycles. The control relation (embedding on typed judgements) is the mechanism that prevents infinite unfolding and makes the canonical form finite.

10 Conclusion

This paper proposes and mechanizes a cyclic variant of a CIC-like calculus. The main contributions are:

- **A cyclic CIC calculus.** We give (to our knowledge) the first development of a CIC-style setting where proof objects are finite graphs with cycles, rather than trees with a fixed recursion/induction discipline.
- **Cyclic normalization as supercompilation.** We make a direct connection between cyclic proof normalization and supercompilation: driving steps correspond to local computation/commuting conversions, while fold/refold and back-links correspond to supercompiler-style memoization and generalization, governed by a termination control (homeomorphic embedding).
- **Verified meta-theory and transformation correctness.** Substantial parts of the pipeline and its supporting theorems are verified in Coq, including key semantic preservation results and the read-off/extraction round-trip theorem.

Verified results and current status. The mechanized development currently includes:

- a proved CIU preservation theorem for head beta reduction (Theorem 8.1);
- a proved round-trip identity theorem for read-off and extraction (Theorem 8.2) and its CIU corollary (Theorem 8.3);
- a proved ranking-based sufficient condition for the global progress predicate (Proposition 8.6);
- a proved CIU preservation theorem for the CaseCase commuting conversion (Theorem 8.4).

Future work is to complete the remaining transformation theorems, strengthen the canonical-form theory, and refine the notion of canonical cyclic representatives (graph equivalence and refolding invariants).

References

[1] Bahareh Afshari and Dominik Wehr. Abstract cyclic proofs. *Mathematical Structures in Computer Science*, 34:552–577, 2024.

[2] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. A compact kernel for the calculus of inductive constructions. *Sādhanā*, 34(1):71–144, 2009.

[3] James Brotherston. Cyclic proofs for first-order logic with inductive definitions. In *TABLEAUX*, 2006.

[4] James Brotherston and Alex Simpson. Sequent calculi for induction and infinite descent. In *LICS*, 2011.

[5] Joseph B. Kruskal. Well-quasi-ordering, the tree theorem, and Vazsonyi's conjecture. *Transactions of the American Mathematical Society*, 1960.

[6] Michael Leuschel. Homeomorphic embedding for online termination of partial deduction. In *Logic-Based Program Synthesis and Transformation*, 2002.

[7] Morten Heine Sørensen and Robert Glück. An introduction to supercompilation. In *Partial Evaluation and Semantics-Based Program Manipulation*, 1995. Often cited introductory survey; venue details vary by edition.

[8] Christoph Sprenger and Mads Dam. On the structure of inductive reasoning: Circular and tree-shaped proofs in the μ-calculus. In *Foundations of Software Science and Computation Structures (FoSSaCS 2003)*, volume 2620 of *Lecture Notes in Computer Science*, pages 425–440. Springer, 2003.

[9] Valentin F. Turchin. *The Concept of a Supercompiler*. Springer, 1986. Classic reference introducing supercompilation.