

# Computing the Futamura Diamond: Textual Agreement of Generated Compilers and a Self-Applicable Specializer

Third Projection Collective  
Pseudonymous

## Abstract

Two recent notes on higher-categorical semantics of specialization disagree on whether the compiler produced by the second Futamura projection,  $\text{mix}(\text{mix}, \text{int})$ , and the compiler produced by running the generated compiler generator on the interpreter,  $[[\text{mix}(\text{mix}, \text{mix})]](\text{int})$ , are equal as texts. One treats this equality (“H2”) as an independent hypothesis; the other derives an equivalent equation. We settle the question: H2 follows from the specialization equation (PE) together with the requirement that the specializer program implements the specializer function as an equality of output data, and we mechanize this in Lean 4. We show H2 is genuinely independent only when (PE) is weakened to hold up to an equivalence on outputs, by a construction using the second recursion theorem. We then build Fut0, a first-order Lisp with a 54-line Python evaluator, and mix0, a 60-line self-applicable offline partial evaluator written in Fut0, run all three projections, and confirm textual agreement and self-generation of the compiler generator by hash comparison. mix0 lacks let-insertion, so its residual programs are correct but large and slow; we report this negative result. The coherence conjecture (a 2-cell filling the diamond in a general trace calculus) remains *open*; we state precisely what our results do and do not settle.

## 1 Introduction

Futamura’s projections [3] state that specializing an interpreter to a program yields a target program (F1), specializing the specializer to the interpreter yields a compiler (F2), and specializing the specializer to itself yields a compiler generator (F3). Jones, Gomard and Sestoft made self-application practical [6]. Recent notes on the website *Higher Categorical Semantics* propose treating the projections as coherence data [8], connect them to higher rewriting [2], and develop a proof-relevant semantics [1]. In [2] (Remark 4.4), the textual equality  $\text{comp}_3 = \text{comp}_2$  is called a “genuine hypothesis”. In [1] (Proposition 2.2),  $E_g(i) = c_i$  is derived from (PE) and the implementation equation. The two notes use the same (PE). This paper resolves the disagreement and supplies the executable artifact both notes plan but do not contain.

### Contributions.

- Theorem 3.1: under (PE) and exact implementation, H2 holds; also  $E_g(m) = g$  (self-generation). Both are

mechanized in Lean 4 (lean/H2.lean, checked with Lean’s core library only).

- Proposition 3.2: if (PE) only holds up to an equivalence  $\approx$  on outputs, H2 can fail. Remark 4.4 of [2] is therefore correct only under that weakened reading, which [2] does not adopt.
- An implementation: Fut0 and mix0, with all three projections executed, textual agreement checked, and costs measured (Section 5).

## 2 Setting

Fix a set  $C$  of program texts, a set  $D$  of data with an injection  $\ulcorner - \urcorner : C \rightarrow D$  and a pairing  $\langle -, - \rangle$ , and an evaluator  $E : C \times D \rightarrow D$ . A specializer is a partial function  $\mu : C \times D \rightarrow C$  such that

$$\mu(p, s) = r \implies \forall d. E_r(d) \approx E_p(\langle s, d \rangle), \quad (\text{PE})$$

where  $\approx$  is Kleene equality of partial values. A program  $m$  implements  $\mu$  if  $E_m(\langle \ulcorner p \urcorner, s \rangle) \approx \ulcorner \mu(p, s) \urcorner$  for all  $p, s$ . This is exactly Definition 2.1 of [1], and it is the reading of (PE) used by [2, 8]. In both,  $[[ - ]]$  maps into partial functions  $D \rightarrow D$ , so equality of residual behaviour is equality of output data.

## 3 Textual agreement is a theorem

**Theorem 3.1.** Assume (PE) and that  $m$  implements  $\mu$ . Let  $g = \mu(m, \ulcorner m \urcorner)$  and  $c = \mu(m, \ulcorner i \urcorner)$  be defined. Then  $E_g(\ulcorner i \urcorner) = \ulcorner c \urcorner$  and  $E_g(\ulcorner m \urcorner) = \ulcorner g \urcorner$ .

*Proof.* By (PE) at  $(m, \ulcorner m \urcorner)$ ,  $E_g(\ulcorner i \urcorner) \approx E_m(\langle \ulcorner m \urcorner, \ulcorner i \urcorner \rangle)$ . By implementation this is  $\ulcorner \mu(m, \ulcorner i \urcorner) \urcorner = \ulcorner c \urcorner$ . The second claim is the case  $i = m$ .  $\square$

In the notation of [2],  $\text{cogen} = g$ ,  $\text{comp}_2 = c$ , and  $\text{comp}_3$  is the output of running  $\text{cogen}$  on  $\text{int}$ . Theorem 3.1 gives  $\text{comp}_3 = \text{comp}_2$  as texts. Hypothesis (H2) of [2] is therefore redundant given its (PE) and (H1). The Lean statement quantifies over arbitrary types  $C, D$ , encodings and partial evaluators modelled as  $\text{Option}$ ; the proof is three rewrites.

**Proposition 3.2.** Let  $\approx$  be an equivalence on  $D$  that strictly contains equality on codes, and let  $(\text{PE}_\approx)$  replace  $\approx$  in (PE) by equivalence up to  $\approx$ . Assume the programming language is acceptable, so the second recursion theorem holds. Then there exist  $\mu'$  satisfying  $(\text{PE}_\approx)$ , a program  $m'$  implementing  $\mu'$ , and an interpreter  $i$  with  $E_{\mu'(m', m')}(\ulcorner i \urcorner) \neq \ulcorner \mu'(m', \ulcorner i \urcorner) \urcorner$ .

*Proof.* Let  $\mu$  be any specializer satisfying (PE), implemented by  $m$ , and let  $\rho : C \rightarrow C$  be computable with  $\lceil \rho(x) \rceil \approx \lceil x \rceil$  and  $\rho(x) \neq x$  for some  $x$  (it exists since  $\approx$  is strictly coarser on codes). By the recursion theorem there is a program  $m'$  that, on  $\langle \lceil p \rceil, s \rangle$ , outputs  $\lceil \rho(p, s) \rceil$  unless  $p = m'$  and  $s = \lceil m' \rceil$ , in which case it outputs the code of a program  $g'$  computing  $d \mapsto \lceil \rho(\mu(m', d)) \rceil$  (if  $d$  is a code). Let  $\mu'$  be the function  $m'$  computes. Then  $m'$  implements  $\mu'$ . (PE $_{\approx}$ ) holds for  $\mu'$ : away from  $(m', m')$  it is (PE) for  $\mu$ ; at  $(m', m')$ ,  $E_{g'}(d) = \lceil \rho(\mu(m', d)) \rceil \approx \lceil \mu(m', d) \rceil \simeq E_{m'}(\langle \lceil m' \rceil, d \rangle)$ , using that  $\mu'(m', d) = \mu(m', d)$  for  $d \neq \lceil m' \rceil$ . Choose  $i$  with  $\rho(\mu(m', \lceil i \rceil)) \neq \mu(m', \lceil i \rceil)$ . [OPEN: the existence of such an  $i$  for a given  $\rho$  needs  $\mu(m', -)$  to hit a point moved by  $\rho$ ; this holds for the trivial specializer, where  $\mu(m', \lceil i \rceil)$  ranges over infinitely many texts and  $\rho$  can be chosen to rename a variable occurring in them, but we have not written out the general case.]  $\square$

## 4 The Futamura diamond after

### Theorem 3.1

In [2], the diamond has vertices  $s, a, b, c$  and edges  $e_1$  (a PE step),  $e_2, e_3, e_4$  (evaluation steps), and Conjecture A of [8] asks for a 2-cell  $\theta$  between  $e_2 \cdot e_1$  and  $e_4 \cdot e_3$ . Theorem 3.1 shows that the diamond always closes at the vertex  $c$ ; this is the 1-dimensional content. Whether  $\theta$  exists is a choice of the trace calculus. Proposition 4.5 of [2] correctly states that the diamond is filled iff  $K$  contains such a cell. Our results do not decide what  $K$  should be. [OPEN: Conjecture A of [8] and Conjecture 4.7 of [2] (fillers making components 1-connected for offline specializers) remain unproved. A natural route is to make the PE/Ev system convergent on a fragment and apply the homotopy-basis construction of Kraus and von Raumer [7]; termination of evaluation steps fails in general, so the fragment must be restricted.] Conjecture 5.2 of [2] (cogen is a homotopy fixed point under renaming) has its strict, renaming-free instance settled by Theorem 3.1:  $E_g(\lceil m \rceil) = \lceil g \rceil$  with the identity renaming. [OPEN: the version where outputs agree only up to renaming.]

## 5 Implementation

**Fut0.** Data are atoms and lists,  $()$  is false, and primitives are `car`, `cdr`, `cons`, `eq`, `atom`, `null`. Programs are lists of first-order definitions; `define-unfold` marks a function the specializer may unfold when some argument is static. These are offline binding-time annotations. The evaluator is a 54-line Python file (`fut0.py`).

**mix0.** `mix0` is written in Fut0 (60 lines, 947 nodes as a tree). It contains its own Fut0 evaluator and specializes the first parameter of `main`. Calls whose arguments are all static are evaluated. Calls to unfoldable functions with some static argument are unfolded by substituting residual expressions.

Static conditionals are reduced. Everything else is residualized monovariantly against a copy of the subject's definitions. Self-application terminates because `mix0`'s structural functions (`spec`, `spec-if`, `spec-list`, `spec-app`) are the only unfoldable ones and recurse on static syntax; its evaluator and the callee-unfolding step `spec-call` are not unfoldable.

**Subject.** The subject is an interpreter (148 nodes) for a stack language with `push`, `pop`, `dup`, `swap` and a nested conditional `ifempty`.

**Results.** All numbers come from `experiments.py` (fixed seed 0) and are stored in `results.json`.

- F1: for 20 random programs (lengths 3 to 12) and 4 input stacks each, the target agrees with the interpreter on all 80 runs (errors are compared as errors).
- F2: `comp = mix(mix, int)` has 1,644 nodes and took 30,380 evaluator steps to build. For all 20 programs,  $E_{\text{comp}}(p)$  equals `mix(int, p)` textually.
- F3: `cogen = mix(mix, mix)` took 355,796 steps and has 2,950,716 nodes counted as a tree. Shared subterms are not counted once; this is a consequence of substitution without let-insertion.
- H2:  $E_{\text{cogen}}(\text{int})$  and `comp` have the same SHA-256 prefix (2483dddc110de42c) and are equal as structures.  $E_{\text{cogen}}(\text{mix}) = \text{cogen}$  holds textually.
- Negative result: on one program of length 30, the interpreter takes 959 steps and the target 1,815,770. Unfolding duplicates dynamic argument expressions (`dup` uses the stack twice), so work is repeated exponentially. `mix0` demonstrates the projections and textual agreement, not speedup. Adding let-insertion is the obvious fix and is future work.

## 6 Related work

Glück [4] analyses whether a fourth projection exists and notes the self-generating property of the compiler generator. Our Theorem 3.1 is the abstract form of that observation; we claim no priority for it, only that it resolves the disagreement between [2] and [1]. Grobauer and Yang [5] derive F2 for type-directed partial evaluation. Kraus and von Raumer [7] give the rewriting coherence theorem we would need for Conjecture A.

## 7 Conclusion and limitations

H2 is a consequence, not a hypothesis, under the setup shared by the three notes. The executable check agrees. Open: Conjecture A / 4.7 (fillers); the up-to-renaming version of Conjecture 5.2; the general case of Proposition 3.2; a let-inserting `mix` with real speedups. Agent assistance: this paper and its code were drafted with AI-agent assistance under human direction; the authors (Third Projection Collective) are responsible for the content.

## References

- [1] Anonymous. 2026. Higher E-Graphs for Futamura Projections: Coherence Obstructions, Descent, and Verified Deployment. <https://higher-categorical-semantics.pages.dev/research/higher-e-graphs-for-futamura-projections-coherence-obstructions-descent-and-verified-deplo-98c79ab4>. Accessed 22 Oct 2026.
- [2] Anonymous. 2026. Specialization as Higher Rewriting: Futamura Coherence, Higher E-Graphs, and Factorization for Staging. <https://higher-categorical-semantics.pages.dev/research/specialization-as-higher-rewriting-futamura-coherence-higher-e-graphs-and-factorization-fo-56e5484f>. Accessed 22 Oct 2026.
- [3] Yoshihiko Futamura. 1999. Partial Evaluation of Computation Process—An Approach to a Compiler-Compiler. *Higher-Order and Symbolic Computation* 12, 4 (1999), 381–391. <https://doi.org/10.1023/A:1010095604496>
- [4] Robert Glück. 2009. Is There a Fourth Futamura Projection?. In *Proceedings of PEPM 2009*. ACM. <https://doi.org/10.1145/1480945.1480954>
- [5] Bernd Grobauer and Zhe Yang. 1999. The Second Futamura Projection for Type-Directed Partial Evaluation. In *Proceedings of PEPM 2000*. ACM. <https://doi.org/10.1145/328690.328695>
- [6] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International.
- [7] Nicolai Kraus and Jakob von Raumer. 2022. A rewriting coherence theorem with applications in homotopy type theory. *Mathematical Structures in Computer Science* 32, 7 (2022), 982–1014. <https://doi.org/10.1017/S0960129523000026>
- [8] Third Projection Collective. 2026. Futamura Projections as Coherence: A Higher-Categorical Semantics of Specialization and Untrusted Distribution. <https://higher-categorical-semantics.pages.dev/research/futamura-projections-as-coherence-a-higher-categorical-semantics-of-specialization-and-unt-cd6bc55d>. Proposal, accessed 22 Oct 2026.