Detecting and Refining Ambiguous Specifications, Automatically (DRAFT)

A proof-by-cases soundness gate for differentiate-after-optimize. Floats are not rings.

June Kim
Independent Researcher
june@june.kim
ORCID 0009-0005-3153-9396

June 22, 2026

Abstract

An optimizing autodiff compiler applies hundreds of algebraic rewrites and then differentiates the result, and checking that a rewrite preserves values and a derivative rule preserves gradients means an expensive build, so per-pattern tests stay syntactic and edge-case soundness goes unchecked. We show that for a bounded fragment of elementwise and transcendental rules this soundness is a finite case analysis, not a search: many of these rewrites are real-algebra identities applied to floats, floats are not a ring, and the unsound ones fail in the real field itself, before floating point enters. A static finite-class-cover analyzer predicts the witnessing input from rule structure with no execution (mechanical proof by cases); a dynamic gate confirms it against the real implementation. From structure alone the analyzer reproduces two confirmed Enzyme-JAX soundness bugs, including a subtle constant-sign dependence. On this fragment its only blind spot is the class the contract already sanctions (fast-math precision); past the fragment's stated edge it is silent by construction, not wrong. As coding agents increasingly generate and apply such rewrites, a cheap proof-by-cases proto-test is a verification layer they can run without the build: a separation oracle for the correctness contract the optimizer otherwise cannot afford to check, which generates the missing guard or certifies the rule sound. It gives a formal analogue of one dimension, the numerical-contract ambiguity, of the open problem of detecting and refining ambiguous specifications, recently posed for N-version programming with coding agents.

The open problem this addresses. Ron, Baudry and Monperrus (N-Version Programming with Coding Agents, arXiv:2606.20158, 2026) find that correlated faults across agent-generated implementations trace to specification ambiguity, and pose as future work: "to study how to use the correlated faults to detect and refine ambiguous specifications in an automated manner." We give a formal analogue of one dimension of that loop, the numerical-contract ambiguity (rounding and overflow, left free by the IEEE/fast-math contract; and, separately, the subgradient at a kink, an AD-convention freedom), distinct from the requirements ambiguity N-version surfaced (§4). On this dimension the detect-and-refine loop becomes a decision procedure, with Enzyme-JAX as the case study.

Working draft. The claims are scoped and the boundary is stated explicitly; the sweep (§9) now closes the unary elementwise derivative table. This links out casually to its artifacts: code, the gate, and the issues live at github.com/kimjune01/enzyme-soundness-gate.

1 Introduction

Coding agents now generate and apply program transformations, including the algebraic rewrites and AD derivative rules at the heart of autodiff compilers like Enzyme (Moses & Churavy 2020, arXiv:2010.01709) and the Reactant stack, whose premise, differentiate after optimization for asymptotically faster gradients, comes from that line of work. The field building these agents is moving up a level, from the model to the harness, reframing the goal as programming with trust (Roychoudhury et al. 2025, arXiv:2502.13767): deployment turns on verification and testing built into the agent, not on raw generation. Surveys of software-engineering agents organize the field around those same missing pieces (Liu et al. 2024, arXiv:2409.02977), and trajectory-level evaluation argues that scoring final outputs misses the reasoning and failure causes inside a run (Yehudai et al. 2025, arXiv:2503.16416).

The bottleneck is not producing a rewrite but trusting it: a rewrite can preserve every value and still poison the gradient, and confirming either against the real compiler means an expensive Bazel/XLA build. So the cheap thing wins, and coverage stays syntactic: roughly five hundred lit tests run `enzymexlamlir-opt --enzyme-hlo-opt | FileCheck`, verifying that a rewrite fires and emits the expected IR text. They never run the program and never differentiate. The semantic and AD oracle exists in the harness but is driven by seven end-to-end models (llama, maxtext, neuralgcm) that rarely reach edge cases. Writing the per-pattern tests is guess-and-check: the author picks a few inputs that look right, eyeballs the emitted IR, and moves on. The guesses come from the rule the author already believes in, so the loop settles into a self-consistent, narrow fixed point and never reaches the counterexample at the edge of the rule, the confirmation bias Wason (1960) named. Proof by cases replaces the guessing.

Problem	Solution
Soundness needs an expensive build	A static proof-by-cases analyzer predicts the witnessing edge input from rule structure, with no execution.
Per-pattern tests are syntactic	The maintainer's contract (same values, same gradient) is the oracle; a divergence self-adjudicates.
Bug or sanctioned slack?	The residue rubric is a decision procedure: a divergence invalid in the real field is a bug, valid-in-the-reals-but-not-on-floats is sanctioned.
Where is the analysis valid?	An explicitly bounded fragment; the boundary is named, audited, and lands where the question is ill-posed, not where a bug hides.

The thesis is that on the elementwise and transcendental fragment, soundness is a finite case analysis. We hand the rule's structure to an analyzer that predicts the witnessing edge input by exhaustive cases, and confirm that one prediction cheaply. From structure alone the analyzer reproduces two soundness bugs we reported to Enzyme-JAX (#2570, #2571, §8), including a constant-sign subtlety that the gate then confirmed.

2 The gap between two test oracles

Two lines of prior work bracket the problem. Differential compiler testing (CSmith; Equivalence Modulo Inputs) checks that an optimization preserves value. Autodiff fuzzing (NablaFuzz) differentially tests AD behavior at the API and function level across libraries. Neither compares the gradient before and after a value-preserving optimization, which is the precise contract of differentiate-after-optimize.

	unit = primitive	unit = optimization / composition
checks value	type checkers	DL-compiler fuzzers (NNSmith, Tzer, MT-DLComp)
checks gradient	AD fuzzers (NablaFuzz)	this work (the isolated cell)

The cell this work isolates is gradient preservation of the composition optimize-then-AD. Our flagship finding, a derivative rule whose primal is correct on negatives but whose gradient is NaN there, lives in it, invisible to a value-only oracle or to an API-level AD test that never sees the optimization.

3 The contract as the test oracle

The maintainer's stated contract turns triage from a matter of taste into a matter of conformance. Every divergence between an optimized program and its reference falls into one of:

- value divergence on an ordinary input → a bug, full stop;
- gradient divergence at a smooth point (the gradient is unique there) → a bug, full stop;
- divergence only at a non-smooth tie (both subgradients valid), or only under inf / nan / fast-math (tacit precision relaxations the contract permits) → sanctioned residue.

We hunt the first two and filter the third. A finding either violates the stated contract or it does not; nothing rests on our judgment of whether it should be a bug.

4 Which ambiguity: the numerical dimension

Monperrus et al.'s observation is at the level of requirements: independently written implementations diverge where the problem specification fails to pin the intended behavior, so the ambiguity is accidental underspecification, and refining it means completing the spec. That dimension is open-ended; there is no algebra for "what did the author intend."

The ambiguity here is a different dimension, and it reads cleanly as an objective subject to a constraint. Enzyme's goal predicate is maximize performance subject to correctness (same value, same gradient): the rewrites push the objective, the contract is the feasible region. A soundness bug is the optimizer reaching a faster point outside that region because a correctness constraint was missing or slack, and the witness `domain(L) \ domain(R)` is the separating point that exhibits it. (The frame is the objective/constraint split and the cutting-plane loop, not literal LP: the decision space is discrete, and the analyzer supplies the separation oracle rather than the solver.)

This is the object we share with the N-version loop: an under-constrained program. An ambiguous spec is a feasible region with constraints missing, which is why independent implementations land at different points and fail in correlated ways. N-version discovers a missing constraint through those correlated faults, its separation oracle; ours is a different oracle, exact and algebraic, on the fragment where one exists. Same loop, constraint generation, found by a different oracle.

- Detect. The separation step. The residue rubric finds a violated correctness constraint and separates it from a granted freedom: a fast-math relaxation (numerical), or a non-smooth tie (AD convention). A divergence is a freedom exactly when the rewrite is valid in the real field but invalid for floats-as-non-ring (§5), so the split is a decision procedure, not a label.
- Refine. Constraint generation. A bug witness generates the missing guard (fire $\log(a^*a) \to 2^*\log(a)$ only on `domain(R)`), which picks the intended behavior and narrows the admitted rewrite set: the strong, requirements-style refinement. A residue witness instead certifies that the contract admits the divergence under its fast-math slack. Promotion to a regression test (§6) is that refinement made executable.

Two honest limits on "refine": the guard states the constraint a rule must satisfy (no NaN gradient on finite negatives), and producing the derivative that satisfies it is a separate step; and the residue branch certifies that a freedom exists while declining to pick which behavior, because the contract declines too. The numerical dimension mechanizes precisely because it is algebraically characterizable; the requirements dimension N-version surfaced is the harder, open one. The claim is narrow and exact: on one formalizable ambiguity dimension, the detect-and-refine loop is a decision procedure.

5 Ring identities on a non-ring

Many of these rewrites are real-algebra identities: `log(a*a) = 2 log(a)`, `(a+b)+c = a+(b+c)`, `(a/b)/c = a/(b*c)` are theorems in the real field, applied as if floats obeyed those laws. Floats are not a ring: no associativity, no exact inverses, a finite range, NaN and inf, rounding. The bugs in this class split along whether the identity is even true in the reals.

A rewrite `L = R` on this fragment is a ring identity applied to floats, and it can fail in exactly two ways:

	FALSE even in $\mathbb{R}$	TRUE in $\mathbb{R}$, false on floats
why	domain(L) wider than domain(R); e.g. `log(a*a) = 2*log(a)` at `a<0`	floats are not a ring; e.g. `(a+b)+c` vs `a+(b+c)`
verdict	self-adjudicating bug; the analyzer decides this	sanctioned residue; fast-math forgives, analyzer blind by design

The relevant fast-math flag licenses the very law being applied, so this blind spot is the sanctioned class by design.

A self-adjudicating bug is false even in the reals. `log(a*a) = 2 log(a)` fails in the partial real field: the left side is defined for `a != 0` (since `a*a > 0`), the right only for `a > 0`. As partial functions they have different domains, so the identity is false before floats enter, and a real-valued model with domain tracking decides it. Fast-math does not excuse it; the fast-math flags license applying real-algebra laws to floats, and this identity fails as a real-algebra law.

Sanctioned residue is true in the reals, false because floats are not a ring: reassociation, overflow, rounding, denormals. A real-field model is blind to these, because in the ring they hold. And the relevant fast-math flag (reassoc, nnan, ninf, nsz) licenses exactly the real-algebra law the rewrite applied, so on the fragment, this blind spot is exactly the sanctioned class, by design not luck: what fast-math forgives is what a real-field model cannot see. That turns the residue filter from a heuristic into a decision procedure for this fragment:

> within the fragment (no extra flag assumptions), a divergence invalid in the partial real field is a self-adjudicating bug, and one valid in the reals but invalid for floats-as-non-ring is sanctioned residue.

(The gradient-at-a-tie residue is a second, analytic source, handled by treating ties as a special class.)

6 A finite-cover analyzer

A proto-test is a pair (witnessing input class, predicted verdict) derived from a rule's structure, not yet a real test because its oracle is model-predicted rather than ground-truth-pinned. It is promoted to a regression test by confirming the witness against the real implementation and pinning the expected value.

The flow: rule structure feeds the static analyzer, which emits the proto-test (witness and predicted verdict); confirming the witness against the real oracle promotes it to a minimal regression test. The static side is cheap, about thirty ops modeled once; the confirming run is expensive and spent only on the flagged.

The static generator is a finite-class-cover analyzer (`domain_analysis.py`). A rewrite $L \to R$ is domain-unsound when `R` is undefined (non-finite) somewhere `L` is defined; the witness is any input in `domain(L) \ domain(R)`. The choice of which inputs to test is not hand-authored per rule: a fixed cover for the tier-1 operator family applies to every rule, and only the rule's structure varies, so authoring cost is amortized over the operator semantics (about thirty ops, modeled once) rather than paid per rule.

The cover must be refined by the rule's own definedness break-points. A single sign representative per variable is sound only when the rule's break-points coincide with the class boundaries, that is, when the sole break-point is zero, as in both findings (`log(a*a)`, `cbrt'`): every negative behaves alike, so `a=-2` witnesses the whole class. A rule with an interior break-point breaks this: $\log((a-3)(a+3)) \to \log(a-3)+\log(a+3)$ is real-valued and unsound for `a<-3` (both factors negative, product positive, but `log` of each factor is NaN), yet a point-sampled `a=-2` agrees on both sides and the rule is wrongly certified sound. Sign is not a uniform partition for it. The sound form, then, is not point-sampling a fixed cover but tracking each subterm's sign/definedness set and refining the partition at the constants the rule introduces, a sound over-approximation (this is what the sweep, §9, builds; the shipped point-sampler is its exact instance only on break-point-zero rules). Tier-1 decidability also assumes the exponent in any `pow` is a literal constant: a non-integer exponent narrows the domain on negatives where an integer one does not, and a variable exponent leaves the fragment.

The dynamic gate confirms by reimplementing the rule's two forms, enumerating an edge lattice, and comparing value and gradient against the reference, auto-filtering the residue.

7 The result: proof by cases on a bounded fragment

The method is mechanical proof by cases (deduction over a finite partition) replacing the author's induction by sampling. The non-trivial content is exhaustiveness: proof by cases is a proof only when the cases cover the domain, when the finite partition lifts to the infinite input space with uniform behavior per class.

Postcondition of enumerating the combinatorial cover. A total verdict over the abstraction that lifts to a soundness decision over the whole concrete domain, with a constructive witness per violation, if and only if the cover is a proven exhaustive and uniform partition and the operator model is faithful. Otherwise it degrades to "agreement on the sampled representatives." Enumeration converts an infinite test obligation into a finite partition-soundness obligation; it pays only when that obligation is dischargeable.

Bound away the ambiguous zones, and point at the boundary. Legitimate precisely because each excluded zone is independently characterized as a place where there is no well-posed bug:

excluded zone	why it is not a well-posed bug
ties	no unique subgradient (no fact of the matter)
precision / non-ring	fast-math sanctioned by the contract
fractal / float-boundary	ground truth itself aliases (the question dissolves)
reachability	discharged by confirmation, separately

The complement is the region where the question is well-posed, and on it the case analysis is unconditional (modulo faithfulness and the model-to-system step). The integrity condition: the boundary is fixed by the structure of ill-posedness and is not allowed to grow to absorb an inconvenient but well-posed bug. A boundary that expands for convenience is gerrymandering, not scoping. The result is stronger for naming its boundary loudly than a broader claim that hides one.

The partition exists in three tiers:

tier	operators	status
tier 1: finite	log sqrt cbrt pow div exp	decidable by enumeration; in scope, exhaustive
tier 2: parametric	tan, gamma (poles)	needs symbolic reasoning
tier 3: fractal	sin(1/x) near 0	no partition, float-aliased; out of scope, ill-posed

On the fragment, the blind spots are exactly the sanctioned and ill-posed zones; the fragment's edge is where we stop, and §11 names what lies past it. A tensor compiler's elementwise core is tier 1 but for a few transcendentals like tan and gamma, which is what makes it a clean target.

8 Case study: Enzyme-JAX

The two findings below were reported to Enzyme-JAX, and the static analyzer recovers both from rule structure alone.

issue	rule	the bug
#2570	`LogSimplify`	$\log(a^* a) \to 2^* \log(a)$, $\log(\text{pow}(x, y)) \to y^* \log(x)$, and the constant `log(a*b)/log(a/b)` cases narrow the domain (real for $a \neq 0$, real only for `a>0`) and return NaN on finite negatives. The analyzer recovers all four from structure, including the constant-sign dependence: a negative constant breaks the identity, a positive one is sound.

issue	rule	the bug
#2571	`CbrtOp` derivative	the rule routes the gradient through `pow(x,-2/3)`, NaN for negative `x`, while `cbrt` is real and smooth there. Primal correct, gradient poisoned. The repro is a one-line mutation of the maintainer's own test: flip the inputs from positive to negative, expected gradients unchanged, the rule yields NaN.

Both are self-adjudicating divergences on ordinary finite inputs, and both fall out of `domain(L) \ domain(R)` over the sign cover.

9 The sweep

A parser over Enzyme-JAX's declarative derivative table (`HLODerivatives.td`) lifts each rule's forward-derivative expression to the analyzer's language and decides domain-narrowing soundness with no execution. The analyzer is not the point-sampler of §6 but its sound form: a sign + definedness abstract interpreter that propagates, per subexpression and per input sign class, whether the value is guaranteed finite-real, guaranteed non-finite, or undetermined. A class it cannot certify is reported undetermined and flagged, never silently passed; that is what earns the no-false-negative guarantee. A rule narrows the domain exactly when its derivative is not guaranteed defined on a class where the true derivative is.

Run over the real table, the sweep is a completeness statement. Of the 34 derivative rules, 19 are unary; 12 of those are real-elementwise (the fragment), and the analyzer decides all twelve: it flags one, `CbrtOp`, as domain-narrowing on negatives, which is exactly the gate-confirmed bug #2571, and it proves the other eleven (`log`, `log1p`, `sqrt`, `rsqrt`, `exp`, `expm1`, `sin`, `cos`, `tanh`, `logistic`, `neg`) definedness-preserving in the real field. The remaining seven unary rules (`abs`, `real`, `imag`, `reverse`, `convert`, `transpose`, `fft`) are complex or structural and fall outside the elementwise-real fragment by construction. The static verdicts agree with the dynamic `jax.grad` gate everywhere the two overlap (`cbrt` flagged, `sqrt` and `tanh` clean), and the analyzer passes the correct #2571 fix (the integer-exponent form `pow(cbrt(x), -2)`) as sound while flagging the buggy non-integer `pow(x, -2/3)`, the same integer-versus-non-integer distinction §6 names.

So on the unary elementwise fragment the soundness question is not merely cheap to test, it is closed: every rule is either a flagged, confirmable domain-narrowing bug or a proof of real-field definedness. That is the separation oracle run to completion over the fragment: cut or certify on every rule, none left undetermined. The C++ imperative rewrites and the binary/structural rules need a heavier front end and the break-point refinement of §6, and are left to future work; the unary derivative table is the clean target, and it is done. Code: `td_sweep.py`.

10 Related work

The lineage is shared with `abductor` (Kim 2026, doi:10.5281/zenodo.20738161), the gate and hypothesis-graph substrate this work runs on, and findings are recorded as a hypothesis graph (Kim 2026); what follows is a loose copy of abductor's lineage, narrowed to the testing dimension.

10.1 Why sampling plateaus

A rule author who "writes their own tests" samples them from the hypothesis they already hold, so the suite reaches a self-consistent narrow fixed point and never meets the counterexample that would force revision, confirmation-biased hypothesis testing in the sense of Wason (1960). Optimizing against a metric you also author is a Goodhart trap (Goodhart 1975; Strathern 1997). A handed, structure-derived case split fixes the objective before the hypothesis.

10.2 Partition testing and its uniformity gap

Testing one representative per input class is equivalence partitioning (Myers 1979) and the category-partition method (Ostrand & Balcer 1988, doi:10.1145/62959.62964), with domain testing (White & Cohen 1980) its geometric form; checking a finite or bounded domain exhaustively as a correctness argument is bounded-exhaustive testing (Sullivan et al. 2004) under the small-scope hypothesis (Andoni et al. 2003), and for single-argument float functions it is folklore: a `float32` unary function has only four billion inputs, so

testing them all is a proof (Dawson 2014), and correctly-rounded math libraries do exactly this (RLIBM, Lim & Nagarakatte 2021, arXiv:2104.04043). The technique here is that old, and we claim no novelty in it. Partition testing's long-known weakness is that per-class uniformity is assumed, not proven, so a misplaced representative misses the bug, the reason it "does not inspire confidence" (Hamlet & Taylor 1990, doi:10.1109/32.62448). Their own qualification is that it regains value when the partition is narrowly based on where failures occur. That is the gap this fills on a stated fragment: the ring/non-ring residue characterization (§5) proves definedness is uniform per sign class instead of assuming it, and the classes are derived from the operator's algebra rather than drawn by a tester. Equivalence partitioning with a discharged uniformity obligation is a decision procedure, not a heuristic, and that obligation, not the partitioning, is the contribution.

10.3 Differential and random compiler testing

CSmith (Yang et al. 2011) and Equivalence Modulo Inputs (Le, Afshari & Su 2014) find miscompilations by differential testing; metamorphic testing of DL compilers and the DL-compiler fuzzers NNSmith (Liu et al. 2023) and Tzer test optimization soundness, but on forward value equivalence. The gradient of a value-preserving optimization is the cell they leave uncovered.

10.4 Autodiff testing

NablaFuzz (Yang et al. 2023) differentially tests gradients across AD modes and filters numerical noise, but at the API and function level across libraries; it does not compare the gradient before and after a compiler optimization. Finite-difference gradient checking is the classical reference oracle.

10.5 Rewrite verification

Alive2 (Lopes, Lee, Hur, Liu & Regehr 2021) and Alive-FP (Menendez, Nagarakatte & Gupta 2016) prove peephole rewrites correct by SMT, including floating-point. They are the verification pole; this is the testing complement for transforms outside the decidable fragment (interprocedural differentiate-after-optimize).

10.6 Test generation and spec inference

Test amplification (DSpot; Danglot, Vera-Perez, Baudry & Monperrus 2019) generalizes an existing test by mutating its inputs. Daikon (Ernst et al. 2007) infers likely invariants from observed runs. Oracle-guided component-based synthesis (Jha, Gulwani, Seshia & Tiwari 2010) drives a candidate to a fix with distinguishing inputs. Our generator sits among these: it lifts a rule's happy-path structure into the edge case the author missed.

10.7 The disagreement structure

The gate computes a symmetric difference between what the optimizer believes and what is true; `abductor` recovers it from an O(d) sketch via set reconciliation (Eppstein, Goodrich, Uyeda & Varghese 2011) and reads a sequence of gate runs as anytime-valid evidence (e-values: Vovk & Wang 2021).

10.8 Named open problems

A survey of compiler testing (Chen, Patra, Pradel et al. 2020) lists "test oracles beyond equivalence relations" as open; a gradient-preservation oracle is exactly that. N-version programming with coding agents (Ron, Baudry & Monperrus 2026) poses automated detection and refinement of ambiguous specifications as future work, one dimension of which this mechanizes (§4).

The contribution is not a new proof technique, the case analysis is elementary. It is the reduction of an open soundness question to a decidable finite case analysis on a characterized fragment, with confirmed bugs.

11 Limitations

The verdict is about a model, not the system: the analyzer's operator semantics must be faithful, and a flagged witness must still be reachable in the real compiler (discharged by confirmation, not by the case analysis). The fragment is elementwise and finitely partitionable; structural soundness (shape, broadcast, aliasing) and the tier-2 parametric and tier-3 fractal regimes are out of scope. The completeness statement (§9) covers the unary elementwise derivative table; the binary, structural, and imperative C++ rewrites need a heavier front end and the break-point refinement of §6.

12 Conclusion

On a precisely stated fragment, the soundness of an autodiff compiler's elementwise rewrites is a finite case analysis rather than a search: the rules are ring identities, the bugs are the identities that fail in the ring itself,

and a fixed class cover decides them from structure alone. The method's limits are stated as a boundary one can audit, and that boundary lands exactly where the soundness question is sanctioned or ill-posed. A static pass predicts cheaply and exhaustively; a dynamic gate confirms; and the surviving witnesses are one-line mutations of the maintainer's own tests. For coding agents that increasingly write and apply these rules, that is a verification layer they can run without the build.

13 Availability

Code, the gate, the static analyzer, and reproduction: github.com/kimjune01/enzyme-soundness-gate. Dual-licensed AGPL-3.0 and CC BY-SA-NS. The findings are EnzymeAD/Enzyme-JAX issues #2570 and #2571.