

Verifier Closure for Fixed-Core Interval Programs: A Certified Replay Architecture in Lean 4

Hidemitsu Maeki^{1*}

^{1*}GhostDrift Mathematical Institute, Inc., Tokyo, Japan.

Corresponding author(s). E-mail(s): hide.pollini.kenkou@gmail.com;

1 Introduction

Numerical components of autonomous and cyber-physical systems—control loops, neural inference layers, signal-processing blocks—are expected to satisfy hard safety envelopes, and those envelopes must sometimes be verified by a party that did not build the component. Two obstacles stand in the way. Floating-point behaviour is not uniformly reproducible across platforms, compiler settings, and arithmetic libraries unless the execution semantics are fixed explicitly. Moreover, the nonlinear real arithmetic needed to reason about typical safety constraints is computationally costly.

The classical response, originating in proof-carrying code [3] and developed in certified-checker architectures, is to relocate the hard work: let the certificate producer construct the evidence and keep the checker simple, deterministic, and independently re-runnable. This paper asks how far that response can be pushed for a deliberately narrow setting. We call the resulting property *verifier closure*: for the chosen primitive core and certificate language, every semantic obligation needed for verifier acceptance is reduced to deterministic checks of quantifier-free ground integer conditions and explicit Euclidean witnesses, with the soundness of that reduction mechanically verified in Lean 4. We fix the primitive core

$$\Sigma_{\text{prim}} = \{+, -, \times, \text{inv}, \text{sqrt}, \text{relu}\}$$

and ask: for straight-line programs whose operations are drawn from Σ_{prim} and whose specification constraints are polynomial, can certificate checking be reduced to quantifier-free ground integer formulas?

The verifier-closure question. An interval-arithmetic verifier must decide, for each program row, whether a claimed output interval is a sound enclosure of the

corresponding real output. For primitives such as \times , inv , and sqrt the real-arithmetic content of such a decision is non-trivial: it involves bounding products over rectangles, reciprocals over nonzero intervals, and square roots over non-negative intervals. Closure asks whether these decisions can be reduced, without loss of soundness, to integer inequalities that an untrusted certificate producer supplies and a simple checker verifies. For the fixed core Σ_{prim} , the answer is yes, and the reduction is witnessed by explicit Euclidean data: corner products and floor/ceiling quotients for \times ; floor/ceiling quotients of D^2 for inv ; integer square root witnesses for sqrt .

What this paper is and is not. This paper establishes the verifier-closure property for Σ_{prim} ; it does not claim:

- a general decision procedure for non-linear real arithmetic;
- a certificate generator or completeness theorem: the verifier only checks submitted certificates, and the existence or automatic generation of certificates for all true programs is left to external producers;
- that any deployed floating-point implementation is verified (this requires a separate implementation-inclusion contract, Section 8);
- that specification constraints can involve inv or sqrt (the mechanised specification grammar covers $\{+, -, \times\}$ only).
- native branching or loops: the certificate language covers straight-line programs only; control-flow extensions are discussed in Section 10.

Existing approaches and where we diverge.

Interval arithmetic (see Section 2) defines sound transfer functions for each primitive but does not, on its own, reduce the checker’s obligation set to integer formulas; real-arithmetic reasoning remains inside the verification kernel.

Proof-carrying code [3] reduces trust in the checker by shifting proof construction to the producer, but the expressive power of the proof object depends on the target logic: re-checking may still require non-trivial reasoning. Here the proof object is narrowed to a finite ledger of integer inequalities plus Euclidean witnesses; the checker is deterministic replay with no search.

SMT-based non-linear arithmetic tools such as Z3 [6] and CVC5 [7] support non-linear real arithmetic but perform search inside the solver, which remains inside the trusted computing base. By contrast, our verifier performs no search at check time; the search burden is delegated entirely to the certificate producer.

1.1 Contributions

The central contribution is this verifier-closure architecture. For the chosen primitive core Σ_{prim} , it is supported by the following results.

1. **Strict Galois insertion** between real intervals and an encoded fixed-point integer domain \mathcal{I}_D , derived from the well-ordering of \mathbb{Z} (Theorem 5, Section 3). This is the foundation that lets every subsequent definition stay in \mathbb{Z} : concretisation and

outward abstraction form an exact adjunction, and an already-encoded interval is recovered exactly after a concretisation–re-abstraction round trip.

2. **Total truth-preserving normalisation map** τ from certificate-side check schemata to Σ_{int} , with truth preservation under standard integer semantics (Lemma 11, Section 4). This lets the verifier evaluate only ground integer formulas while the certificate is written in a richer source language including $>$, \geq , \neq and implication.
3. **Closed-form, witness-bearing rules** for each $\sigma \in \Sigma_{\text{prim}}$, with every non-trivial soundness obligation reduced to explicit Euclidean inequalities carried in the certificate (Section 5).
4. **Certificate language and replay verifier** built from program DAGs, ledgers, and witness tuples, together with a deterministic replay algorithm over Σ_{int} (Section 6).
5. **End-to-end soundness and unique-trajectory theorem**: verifier acceptance implies the existence of a unique concrete real trajectory and enclosure of every certified specification constraint, with structural replay cost linear in $n + s + r$ (Sections 6.3–7).

Transfer to a deployed floating-point implementation is handled separately as an explicit *implementation-inclusion contract* (Section 8), not formalised in Lean by design.

Project identifier. The project identifier ADIC, used in the Lean namespace and in the repository name, stands for Advanced Data Integrity by Ledger of Computation. It is unrelated to p -adic numbers, adic spaces, or any algebraic-geometric use of “adic”.

1.2 Running example

Example 1 (Running example \mathcal{P}_0) Fix $D = 2$. Take two input nodes v_1, v_2 with encoded envelopes $(2, 4)$ and $(6, 10)$, decoding to real ranges $[1, 2]$ and $[3, 5]$. The program DAG contains a single \times -node v_3 with parents (v_1, v_2) . The specification is the single inequality $v_3 - 10 \leq 0$.

The certificate supplies multiplication witness $(p_{\min}, p_{\max}, Q_f, Q_c) = (12, 40, 6, 20)$, where $p_{\min} = 2 \cdot 6 = 12$ and $p_{\max} = 4 \cdot 10 = 40$, and a specification ledger node for $v_3 - 10$ referencing `const(20)` at $D = 2$, with root index 0 and upper-bound check $0 \leq 0$. The formal definitions of encoded intervals are given in Section 3, and the definitions of program rows, specification rows, and replay ledgers are given in Section 6; this example is only a guide to the notation.

1.3 Paper organisation

Sections 2–4 set up the related work, interval domain, and integer normalisation layer. Sections 5–7 define the witness-bearing primitive rules, certificate language, replay verifier, and main soundness theorems. Sections 8–11 isolate the deployment contract, describe the supporting artefacts, discuss scope, and conclude.

2 Related Work

Interval arithmetic and formally verified interval reasoning. Moore’s foundational analysis [5], the IEEE 1788 standard [10], and recent Isabelle/HOL formalisation

of interval arithmetic for program verification [9] provide the semantic basis for sound interval transfer functions. Coq-based interval reasoning and computation-driven proofs of real bounds, as developed by Melquiond [11], show how large numerical proof obligations can be discharged by certified computation inside a proof assistant. These works establish the correctness of interval operations and interval-based reasoning principles inside proof-assistant environments. The present paper draws a narrower replay boundary: the replay verifier is not a real-arithmetic proof procedure and does not invoke interval reasoning at check time. After certificate production, the trusted replay kernel checks only finite ground-integer obligations: Euclidean quotient witnesses, integer square-root witnesses, endpoint inequalities, and root non-positivity checks. The tradeoff is reduced expressiveness: the primitive core is fixed, specification-side operations are restricted, and certificate production may require external search. The gain is deterministic, search-free replay with no non-linear real-arithmetic reasoning in the trusted checker.

Certificate-based numerical verification. Certificate-based approaches to rigorous numerical approximation have been developed in proof assistants. Bréhard, Mahboubi, and Pous [12] validate approximations a posteriori and explicitly treat operations such as division and square root. Verified certificate checkers for numerical error bounds, such as FloVer [18], likewise separate certificate generation from independently checked validation. These works are close in spirit to ours because they reduce trust in the producer by replaying compact evidence in a theorem-prover-backed environment. The target, however, is different. FloVer checks finite-precision roundoff-error certificates in Coq and HOL4, while Bréhard, Mahboubi, and Pous validate a posteriori approximation certificates in Coq, including division and square root. The present paper deliberately narrows the object being checked: accepted interval-program rows are compiled to a bounded ground-integer replay ledger, the checker performs no solver or real-arithmetic search at check time, non-trivial primitive steps are discharged by explicit Euclidean witnesses, and the resulting verifier-closure theorem is mechanised in Lean 4. The value of the restriction is that each replay step becomes a small integer-certificate check rather than an analytic proof obligation inside the checker.

Proof-carrying code and certified proof checkers. Necula’s proof-carrying code [3] introduced the producer–consumer separation in which code is shipped with evidence that can be checked by a trusted consumer. Certified proof checkers continue this line by reducing trust in external proof producers. SMTCoq [13, 14], for example, integrates external SAT/SMT solvers into Coq through proof witnesses and certified checking. More recent certified checker architectures, such as the Imandra proof checker for neural-network verification [8], pursue a similar trust-reduction goal. The present paper adopts the same trust-shifting principle, but specialises it to deterministic replay of interval-program certificates. The verifier does not reconstruct a large logical derivation; it checks a ledger of local integer obligations whose soundness is proved once and for all in Lean.

SMT proof certificates and proof reconstruction. Modern SMT solvers such as Z3 [6] and cvc5 [7] support powerful automated reasoning over arithmetic fragments. A separate line of work studies how solver results can be accompanied by detailed

proof objects and independently checked or reconstructed, including fine-grained proof production [15], proof-producing automated reasoning [16], and Alethe/cvc5 proof reconstruction in Isabelle/HOL [17]. These systems aim to make solver results independently checkable. Our verifier takes a more restrictive route: certificate production may use external search, but the replay kernel contains no SMT search and no non-linear real-arithmetic decision procedure. Its trusted computation is restricted to deterministic checks of ground integer constraints.

Abstract interpretation and Galois connections. Abstract interpretation [4] provides the conceptual frame: a concrete semantics is related to an abstract domain by a sound abstraction/concretisation discipline, and each transfer function is proved sound with respect to that relation. The present paper specialises this frame to an encoded integer interval domain where the Galois insertion is mechanised and where transfer soundness for each primitive operation is discharged by finite integer certificates. The result is not a new abstract interpretation framework; rather, it is a certified replay architecture showing that, for this primitive core, the semantic obligation of an interval-program run can be reduced to a deterministic sequence of locally checkable integer conditions.

Taken together, these lines of work provide verified interval operators, certificate-based numerical validation, certified proof reconstruction, and SMT-proof replay. The present paper closes a different boundary: for a deliberately fixed primitive core, it proves that verifier acceptance can be reduced to deterministic replay of a finite ledger of ground integer obligations, with soundness established once and for all in Lean.

3 Fixed-Point Encoding and the Galois Insertion

The verifier works entirely in the integer domain. This section constructs the encoded interval domain \mathcal{I}_D , the maps γ_D and α_D that connect it to the real interval lattice, and the Galois insertion relating them. All definitions correspond directly to Lean structures in `ADIC_RSound_Replay.lean`.

We write

$$\mathbf{RInt} = \{ [l, u] \mid l, u \in \mathbb{R}, l \leq u \}$$

for the set of non-empty closed real intervals, including degenerate one-point intervals.

Definition 2 (Scaling denominator) A *scaling denominator* is a positive natural number $D \in \mathbb{N}^+$.

Definition 3 (Encoded interval domain \mathcal{I}_D) Fix $D \in \mathbb{N}^+$. The *encoded interval domain* is

$$\mathcal{I}_D = \{ (a, b) \in \mathbb{Z}^2 \mid a \leq b \}.$$

Elements of \mathcal{I}_D are written $(a, b)_D$ when the denominator needs to be emphasised. For $I = (a, b) \in \mathcal{I}_D$, write $\underline{I} = a$ and $\bar{I} = b$ for its lower and upper integer endpoints. The *subset order* on \mathcal{I}_D is

$$(a, b) \sqsubseteq (c, d) \iff c \leq a \text{ and } b \leq d.$$

Under this order, $(a, b) \sqsubseteq (c, d)$ if and only if the real interval $[a/D, b/D]$ is contained in $[c/D, d/D]$. (Lean: `IEnc D`, order `IEnc.instLE`.)

Definition 4 (Concretisation γ_D and outward abstraction α_D)

- (a) The *concretisation* $\gamma_D : \mathcal{I}_D \rightarrow \mathbf{RInt}$ decodes an encoded interval into a real closed interval:

$$\gamma_D(a, b) = \left[\frac{a}{D}, \frac{b}{D} \right].$$

- (b) The *outward abstraction* $\alpha_D : \mathbf{RInt} \rightarrow \mathcal{I}_D$ encodes a real closed interval by rounding outward to the nearest integer grid:

$$\alpha_D[l, u] = (\lfloor D \cdot l \rfloor, \lceil D \cdot u \rceil).$$

Here $\lfloor \cdot \rfloor$ is the floor and $\lceil \cdot \rceil$ is the ceiling function on \mathbb{R} .

(Lean: `IEnc.gamma`, `IEnc.alpha`.)

What these maps do. γ_D decodes an integer pair into a real interval; α_D encodes a real interval by rounding its endpoints outward to ensure that no real value inside the original interval is excluded from the encoded result. When the real interval $[l, u]$ does not sit exactly on the integer grid of scale D (i.e., Dl or Du is not an integer), $\alpha_D[l, u]$ may be strictly wider than $[l, u]$; this conservative rounding is intentional and is the source of the enclosure guarantee.

The crucial additional property—established in Theorem 5 and mechanised in Lean—is that on *already-encoded* intervals, this widening does not occur: concretising and then re-abstracting an encoded interval recovers it exactly.

Theorem 5 (Strict Galois insertion) *Let $D \in \mathbb{N}^+$.*

- (a) (Galois connection.) *For all $J \in \mathbf{RInt}$ and all $(a, b) \in \mathcal{I}_D$,*

$$\alpha_D(J) \sqsubseteq (a, b) \iff J \subseteq \gamma_D(a, b).$$

- (b) (Strictness.) *For all $(a, b) \in \mathcal{I}_D$,*

$$\alpha_D(\gamma_D(a, b)) = (a, b).$$

(Lean: `gc_alpha_gamma (a)`; `alpha_gamma_strict (b)`.)

Proof Part (a) follows by unfolding the definitions: $\alpha_D(J) \sqsubseteq (a, b)$ means $a \leq \lfloor Dl \rfloor$ and $\lceil Du \rceil \leq b$, which by the floor/ceiling characterisation is equivalent to $a/D \leq l$ and $u \leq b/D$, i.e. $J = [l, u] \subseteq [a/D, b/D] = \gamma_D(a, b)$. Part (b): $\alpha_D(\gamma_D(a, b)) = (\lfloor D \cdot (a/D) \rfloor, \lceil D \cdot (b/D) \rceil) = (a, b)$ since $D \cdot (a/D) = a \in \mathbb{Z}$. \square

Part (b) says: an already-encoded interval, when first decoded to a real interval by γ_D and then re-encoded by α_D , is recovered exactly. Equivalently, γ_D is injective: distinct encoded intervals decode to distinct real intervals. As a consequence, every decision the verifier makes in \mathcal{I}_D faithfully reflects a real-valued distinction; no representational information is lost when working with encoded intervals in \mathcal{I}_D .

Example 6 (\mathcal{P}_0 : encoding and round-trip) For $D = 2$, the encoded envelope $(2, 4)$ decodes via γ_D to $[1, 2]$. Applying α_D to $[1, 2]$ gives $(\lfloor 2 \rfloor, \lceil 4 \rceil) = (2, 4)$, confirming $\alpha_D(\gamma_D(2, 4)) = (2, 4)$. An arbitrary real interval $[1.1, 1.9]$ would encode as $\alpha_D[1.1, 1.9] = (\lfloor 2.2 \rfloor, \lceil 3.8 \rceil) = (2, 4)$ —a strictly wider result, illustrating the outward rounding behaviour of α_D on non-grid-aligned inputs.

4 Ground Integer Signature and Normalisation τ

The verifier evaluates only quantifier-free ground integer formulas. This section defines the target signature Σ_{int} and the source signature Σ_{src} , and constructs a total normalisation map τ from source formulas to target formulas, together with its truth-preservation theorem.

4.1 Integer term language

Definition 7 (Integer terms and Σ_{int}) The set IntTerm of *integer terms* is the smallest set such that: $\text{const}(z) \in \text{IntTerm}$ for every $z \in \mathbb{Z}$; and $\text{add}(t_1, t_2)$, $\text{sub}(t_1, t_2)$, $\text{mul}(t_1, t_2) \in \text{IntTerm}$ for all $t_1, t_2 \in \text{IntTerm}$. The evaluation map $\llbracket \cdot \rrbracket : \text{IntTerm} \rightarrow \mathbb{Z}$ is the expected one: $\llbracket \text{const}(z) \rrbracket = z$, $\llbracket \text{add}(t_1, t_2) \rrbracket = \llbracket t_1 \rrbracket + \llbracket t_2 \rrbracket$, etc. (Lean: *IntTerm*, *IntTerm.eval*.)

Definition 8 (Core atoms and quantifier-free formulas) *Core atoms* are expressions of the form $t_1 = t_2$, $t_1 < t_2$, or $t_1 \leq t_2$ where $t_1, t_2 \in \text{IntTerm}$ (Lean: *Atom*). *Quantifier-free formulas* QFForm are built from core atoms by conjunction, disjunction, and negation (Lean: *QFForm*). The evaluation $\text{eval} : \text{QFForm} \rightarrow \text{Bool}$ is standard; a formula *holds* if $\text{eval}(p) = \text{true}$. These formulas constitute the *target signature* Σ_{int} .

In the Lean formalisation, some primitive obligations are represented by named decidable predicates such as *CheckMul*, *CheckInv*, and *CheckSqrt*, rather than only by a single syntactic *QFForm* object. This is a presentation choice: each such predicate is a finite conjunction of ground integer equalities and inequalities, and therefore corresponds to a quantifier-free ground integer obligation in the paper-level signature.

4.2 Source-level certificate language

Certificate checks are written in a richer source language that includes $>$, \geq , \neq , implication, and a literal \top . This richer language is the *source signature* Σ_{src} .

Definition 9 (Source atoms and formulas) *Source atoms* extend core atoms with: $t_1 > t_2$ (strict greater-than), $t_1 \geq t_2$ (greater-or-equal), $t_1 \neq t_2$ (disequality) (Lean: *SrcAtom*). *Source formulas* SrcForm extend QFForm with: $P \Rightarrow Q$ (implication), \top (truth literal) (Lean: *SrcForm*). Evaluation $\text{eval}_{\text{src}} : \text{SrcForm} \rightarrow \text{Bool}$ is the expected extension.

4.3 Normalisation τ

Definition 10 (Truth-preserving normalisation map τ) The map $\tau : \text{SrcForm} \rightarrow \text{QFForm}$ is defined structurally as follows. On source atoms:

$$\begin{aligned} \tau(t_1 = t_2) &= (t_1 = t_2), & \tau(t_1 < t_2) &= (t_1 < t_2), & \tau(t_1 \leq t_2) &= (t_1 \leq t_2), \\ \tau(t_1 > t_2) &= (t_2 < t_1), & \tau(t_1 \geq t_2) &= (t_2 \leq t_1), & \tau(t_1 \neq t_2) &= (t_1 < t_2) \vee (t_2 < t_1). \end{aligned}$$

On compound forms:

$$\begin{aligned} \tau(P \wedge Q) &= \tau(P) \wedge \tau(Q), & \tau(P \vee Q) &= \tau(P) \vee \tau(Q), & \tau(\neg P) &= \neg \tau(P), \\ \tau(P \Rightarrow Q) &= \neg \tau(P) \vee \tau(Q), & \tau(\top) &= (0 = 0). \end{aligned}$$

(Lean: *tau*, *tauAtom*.)

Lemma 11 (Truth preservation of τ) *For every $p \in \text{SrcForm}$,*

$$\text{eval}_{\text{src}}(p) = \text{eval}_{\text{qt}}(\tau(p)).$$

In particular, p holds if and only if $\tau(p)$ holds. (Lean: `tau_preserves` (evaluation equality), `tau_truth` (Prop-level iff).)

Proof By structural induction on p . The only non-trivial base case is \neq : the formula $t_1 \neq t_2$ holds iff $\llbracket t_1 \rrbracket \neq \llbracket t_2 \rrbracket$, which by integer trichotomy is equivalent to $\llbracket t_1 \rrbracket < \llbracket t_2 \rrbracket \vee \llbracket t_2 \rrbracket < \llbracket t_1 \rrbracket$, i.e. to the evaluation of $\tau(t_1 \neq t_2)$. All other cases are immediate from the definitions. \square

4.4 Division witnesses: FDIV and CDIV

Two integer-witnessed predicates play a central role in the multiplication and inversion rules.

Definition 12 (FDIV and CDIV witnesses) Let $z, d, Q \in \mathbb{Z}$ with $d \neq 0$.

- FDIV(z, d, Q) holds iff $Q = \lfloor z/d \rfloor$, equivalently:

$$d > 0 : Q \cdot d \leq z < (Q + 1) \cdot d; \quad d < 0 : Q \cdot d \geq z > (Q + 1) \cdot d.$$

- CDIV(z, d, Q) holds iff $Q = \lceil z/d \rceil$, equivalently:

$$d > 0 : (Q - 1) \cdot d < z \leq Q \cdot d; \quad d < 0 : (Q - 1) \cdot d > z \geq Q \cdot d.$$

(Lean: `CheckFDIV`, `CheckCDIV`.)

Lemma 13 (Division witness correctness) *If FDIV(z, d, Q) holds then $(Q : \mathbb{R}) \leq (z : \mathbb{R}) / (d : \mathbb{R})$; if CDIV(z, d, Q) holds then $(z : \mathbb{R}) / (d : \mathbb{R}) \leq (Q : \mathbb{R})$. (Lean: `fdiv_bound`, `cdiv_bound`.)*

Proof We prove the sharper real inequalities associated with the two witnesses. For FDIV(z, d, Q) there are two cases. If $d > 0$, the defining inequalities give $Qd \leq z < (Q + 1)d$. After casting to \mathbb{R} and dividing by the positive number d , we obtain $Q \leq z/d < Q + 1$. If $d < 0$, the defining inequalities are $Qd \geq z > (Q + 1)d$. Dividing by the negative number d reverses the inequalities and again yields $Q \leq z/d < Q + 1$. In both cases the desired lower bound follows.

For CDIV(z, d, Q) the argument is dual. If $d > 0$, the defining inequalities $(Q - 1)d < z \leq Qd$ give $Q - 1 < z/d \leq Q$ after division by the positive denominator. If $d < 0$, the defining inequalities $(Q - 1)d > z \geq Qd$ give the same real inequalities after division by the negative denominator and reversal of order. Hence in both cases $z/d \leq Q$, as required. \square

The division lemmas are the sole bridge between integer checks and real-valued division bounds; they are used in the multiplication and inversion soundness proofs.

5 Primitive Rules and R-SOUND

Definition 14 (R-SOUND package) Let $\sigma \in \Sigma_{\text{prim}}$ be a primitive of arity k . An R-SOUND package for σ consists of

$$(\text{Dom}_{\sigma}, \text{Check}_{\sigma}, T_{\sigma}, \text{sound}_{\sigma}).$$

Here Dom_σ is a domain predicate on the input encoded intervals, Check_σ is a quantifier-free ground integer predicate over the denominator, input intervals, and certificate witness, and T_σ returns the certified output interval. The proof component sound_σ is the obligation

$$\text{Dom}_\sigma(\vec{I}) \wedge \text{Check}_\sigma(D, \vec{I}, W) \wedge \bigwedge_{i=1}^k x_i \in \gamma_D(I_i) \implies \sigma(\vec{x}) \in \gamma_D(T_\sigma(D, \vec{I}, W)).$$

For $+$, $-$, and relu , the witness is trivial and the check predicate is true. For \times , the domain predicate is trivial. For inv and sqrt , the domain predicate is the corresponding non-zero or non-negative input condition, included in CheckInv and CheckSqrt , respectively. At replay time the verifier evaluates only the integer check predicate; the soundness component is discharged once in the metatheory and by the corresponding Lean theorem.

For $+$, $-$, and relu , no witness is required; the transfer function is computed directly from endpoints. For \times , inv , and sqrt , the certificate carries explicit Euclidean witnesses.

5.1 Addition and subtraction

Definition 15 (Transfer functions T_+ and T_-) For $I = (a, b), J = (c, d) \in \mathcal{I}_D$:

$$T_+(I, J) = (a + c, b + d), \quad T_-(I, J) = (a - d, b - c).$$

Both are well-formed: $a + c \leq b + d$ from $a \leq b$ and $c \leq d$; similarly for T_- . (Lean: `T.add`, `T.sub`.)

Theorem 16 (R-SOUND for $+$ and $-$) For all $I, J \in \mathcal{I}_D$ and all $x, y \in \mathbb{R}$:

- (a) if $x \in \gamma_D(I)$ and $y \in \gamma_D(J)$ then $(x + y) \in \gamma_D(T_+(I, J))$;
- (b) if $x \in \gamma_D(I)$ and $y \in \gamma_D(J)$ then $(x - y) \in \gamma_D(T_-(I, J))$.

(Lean: `rsound.add`, `rsound.sub`.)

Proof For (a): if $a/D \leq x \leq b/D$ and $c/D \leq y \leq d/D$, then $(a + c)/D \leq x + y \leq (b + d)/D$, so $x + y \in \gamma_D(T_+(I, J))$. Part (b) is analogous. \square

Example 17 (\mathcal{P}_0 : subtraction in the specification) The specification expression $v_3 - 10$ is compiled into a subtraction node with $I = (6, 20)$ (the interval for v_3 at $D = 2$) and $J = (20, 20)$ (the constant 10 encoded as $(10D, 10D) = (20, 20)$). Then $T_-(I, J) = (6 - 20, 20 - 20) = (-14, 0)$, confirming that the root-check bound $0/D = 0$ is non-positive.

5.2 Multiplication

Multiplication is the first primitive requiring explicit Euclidean witnesses. The real product xy with $x \in [a/D, b/D]$ and $y \in [c/D, d/D]$ lies in the convex hull of the four corner products ac, ad, bc, bd (all divided by D^2). The certificate supplies the min and max of these corners, plus floor/ceiling witnesses for the division by D .

The corner products $p_i = ac, ad, bc, bd$ live at D^2 -scale: they bound $XY = D^2xy$. The predicates $\text{FDIV}(p_{\min}, D, Q_f)$ and $\text{CDIV}(p_{\max}, D, Q_c)$ rescale these corner bounds from D^2 -scale to D -scale, producing encoded endpoints Q_f, Q_c . Dividing $Q_f \leq Dxy \leq Q_c$ by D then recovers the real enclosure $Q_f/D \leq xy \leq Q_c/D$.

Definition 18 (Multiplication witness and check) A *multiplication witness* is a tuple $W = (p_{\min}, p_{\max}, Q_f, Q_c) \in \mathbb{Z}^4$. For $I = (a, b), J = (c, d) \in \mathcal{I}_D$, let $p_1 = ac, p_2 = ad, p_3 = bc, p_4 = bd$. The check $\text{CheckMul}(D, I, J, W)$ holds iff:

- $p_{\min} \in \{p_1, p_2, p_3, p_4\}$ and $p_{\min} \leq p_i$ for all $i \in \{1, 2, 3, 4\}$;
- $p_{\max} \in \{p_1, p_2, p_3, p_4\}$ and $p_{\max} \geq p_i$ for all $i \in \{1, 2, 3, 4\}$;
- $\text{FDIV}(p_{\min}, D, Q_f)$ and $\text{CDIV}(p_{\max}, D, Q_c)$.

Here and below, $\langle u, v \rangle$ is only display notation for the encoded interval $(u, v) \in \mathcal{I}_D$.

The output encoded interval is $\langle Q_f, Q_c \rangle$. (*Lean: `WMul, CheckMul`.*)

What the witness provides. The order clauses alone are sufficient for the containment argument in Theorem 19: they provide lower and upper bounds on all four corner products. The membership clauses serve a separate certificate-integrity role. They let the verifier check that the producer has selected extrema from the actual computed corner products, rather than submitting unrelated loose bounds that merely happen to be sound. This keeps the witness canonical for replay and for possible future completeness statements. Thus the soundness proof uses the order inequalities, while the full CheckMul predicate also records exact corner selection. The FDIV/CDIV conditions witness the integer floor/ceiling of those extremes divided by D . Together they guarantee $Q_f/D \leq xy \leq Q_c/D$, i.e. $xy \in \gamma_D(Q_f, Q_c)$, and all of this is decided by integer arithmetic alone.

Theorem 19 (R-SOUND for \times) *If $\text{CheckMul}(D, I, J, W)$ holds and $x \in \gamma_D(I), y \in \gamma_D(J)$, then $xy \in \gamma_D(Q_f, Q_c)$. (Lean: `rsound_mul`; closedness $Q_f \leq Q_c$: `rsound_mul_closed`.)*

Proof Write $I = (a, b), J = (c, d)$, and $W = (p_{\min}, p_{\max}, Q_f, Q_c)$. We work with scaled integers $X = Dx$ and $Y = Dy$ to eliminate the denominator D from the input intervals and compare the corner products ac, ad, bc, bd directly at integer scale. Since $x \in \gamma_D(I)$ and $y \in \gamma_D(J)$, we have $a \leq X \leq b$ and $c \leq Y \leq d$.

First, XY is bounded by the four corner products. Indeed, for fixed Y , the affine function $t \mapsto tY$ on $[a, b]$ has its minimum and maximum at the endpoints, so XY lies between aY and bY . For fixed a and b , the quantities aY and bY are likewise bounded by their endpoint values over $Y \in [c, d]$. Therefore

$$\min\{ac, ad, bc, bd\} \leq XY \leq \max\{ac, ad, bc, bd\}.$$

The order clauses of CheckMul assert, by integer comparisons, that p_{\min} is no larger than each of the four products and that p_{\max} is no smaller than each of them. Hence

$$p_{\min} \leq XY \leq p_{\max}.$$

The last two clauses of CheckMul give $\text{FDIV}(p_{\min}, D, Q_f)$ and $\text{CDIV}(p_{\max}, D, Q_c)$. By Lemma 13,

$$Q_f \leq \frac{p_{\min}}{D}, \quad \frac{p_{\max}}{D} \leq Q_c.$$

Since $D > 0$, the bound $p_{\min} \leq XY \leq p_{\max}$ gives

$$\frac{p_{\min}}{D} \leq \frac{XY}{D} \leq \frac{p_{\max}}{D}.$$

Combining this with the FDIV/CDIV bounds above yields

$$Q_f \leq \frac{XY}{D} \leq Q_c.$$

But $X = Dx$ and $Y = Dy$, so

$$\frac{XY}{D} = \frac{D^2xy}{D} = Dxy.$$

Thus the last displayed inequality is

$$Q_f \leq Dxy \leq Q_c.$$

Dividing this chain by the positive number D gives

$$\frac{Q_f}{D} \leq xy \leq \frac{Q_c}{D}.$$

This is exactly $xy \in \gamma_D(Q_f, Q_c)$. The same chain also gives $Q_f \leq Q_c$, so the output interval is well formed. \square

Example 20 (\mathcal{P}_0 : multiplication witness) With $I = (2, 4)$, $J = (6, 10)$, $D = 2$, the corner products are $p_1 = 12$, $p_2 = 20$, $p_3 = 24$, $p_4 = 40$. The witness $(12, 40, 6, 20)$ satisfies $\text{FDIV}(12, 2, 6)$ (since $6 \cdot 2 = 12$) and $\text{CDIV}(40, 2, 20)$ (since $20 \cdot 2 = 40$). Thus the verifier confirms $v_3 = v_1v_2 \in [3, 10]$.

5.3 Inversion

Inversion $x \mapsto 1/x$ requires that x is bounded away from zero. The certificate supplies witnesses for the reciprocal bounds.

Definition 21 (Inversion domain and check) For $I = (a, b) \in \mathcal{I}_D$:

- $\text{DomInv}(I)$ holds iff $a > 0$ or $b < 0$ (the interval is bounded away from zero).
- An *inversion witness* is a pair $(Q_l, Q_u) \in \mathbb{Z}^2$.
- $\text{CheckInv}(D, I, W)$ holds iff $\text{DomInv}(I)$, $\text{FDIV}(D^2, b, Q_l)$, and $\text{CDIV}(D^2, a, Q_u)$.

The output encoded interval is $\langle Q_l, Q_u \rangle$. (*Lean*: *DomInv*, *CheckInv*, *WInv*.)

Why D^2 ? The encoded interval for $1/x$ at scale D is $\langle Q_l, Q_u \rangle$ where $Q_l/D \leq 1/x \leq Q_u/D$. For positive $I = (a, b)$, the real endpoints are a/D and b/D , so the reciprocal ranges over

$$\left[\frac{1}{b/D}, \frac{1}{a/D} \right] = \left[\frac{D}{b}, \frac{D}{a} \right].$$

Thus the encoded lower endpoint satisfies $Q_l = \lfloor D \cdot (D/b) \rfloor = \lfloor D^2/b \rfloor$, witnessed by $\text{FDIV}(D^2, b, Q_l)$.

For example, if $D = 2$ and $I = (2, 4)$, then $x \in [1, 2]$, $D^2/b = 4/4 = 1$, and $D^2/a = 4/2 = 2$, giving the encoded reciprocal interval $(1, 2)$, i.e. $[1/2, 1]$. If $D = 2$ and $I = (-4, -2)$, then $x \in [-2, -1]$, $D^2/b = 4/(-2) = -2$, and $D^2/a = 4/(-4) = -1$, giving the encoded reciprocal interval $(-2, -1)$, i.e. $[-1, -1/2]$. Thus the same D^2 -based witness form handles both positive and negative domains; the sign of the denominator is handled by the corresponding branch of FDIV or CDIV.

Theorem 22 (R-SOUND for inv) *If $\text{CheckInv}(D, I, W)$ holds and $x \in \gamma_D(I)$, then $1/x \in \gamma_D(Q_l, Q_u)$. (Lean: `rsound_inv`; closedness: `rsound_inv_closed`.)*

Proof Write $I = (a, b)$ and $W = (Q_l, Q_u)$. The domain part of CheckInv says $a > 0$ or $b < 0$. Thus every $x \in \gamma_D(I) = [a/D, b/D]$ is non-zero, and the reciprocal is defined throughout the certified interval.

There are two domain cases. If $0 < a \leq b$, then $x \in [a/D, b/D] \subset (0, \infty)$, and $t \mapsto 1/t$ is decreasing on the positive reals. If $a \leq b < 0$, then $x \in [a/D, b/D] \subset (-\infty, 0)$, and the same map is decreasing on the negative reals. Hence in both cases

$$\frac{D}{b} = \frac{1}{b/D} \leq \frac{1}{x} \leq \frac{1}{a/D} = \frac{D}{a}.$$

The witness clauses are

$$\text{FDIV}(D^2, b, Q_l), \quad \text{CDIV}(D^2, a, Q_u).$$

In the positive-domain case $0 < a \leq b$, these use the $d > 0$ branches of Lemma 13. In the negative-domain case $a \leq b < 0$, both denominators a and b are negative, so they use the $d < 0$ branches. In either case Lemma 13 gives

$$Q_l \leq \frac{D^2}{b}, \quad \frac{D^2}{a} \leq Q_u.$$

Dividing by the positive number D gives

$$\frac{Q_l}{D} \leq \frac{D}{b} = \frac{1}{b/D}, \quad \frac{1}{a/D} = \frac{D}{a} \leq \frac{Q_u}{D}.$$

Combining these inequalities with the reciprocal endpoint bound yields

$$\frac{Q_l}{D} \leq \frac{1}{x} \leq \frac{Q_u}{D}.$$

Therefore $1/x \in \gamma_D(Q_l, Q_u)$. The same endpoint chain yields $Q_l \leq Q_u$, so the encoded output interval is well formed. \square

5.4 Square root

Square root $x \mapsto \sqrt{x}$ requires $x \geq 0$. The certificate supplies integer witnesses for the lower and upper square root bounds.

Definition 23 (Square root domain and check) For $I = (a, b) \in \mathcal{I}_D$:

- $\text{DomSqrt}(I)$ holds iff $a \geq 0$.
- A *square root witness* is a pair $(p, q) \in \mathbb{Z}^2$.
- $\text{CheckSqrt}(D, I, W)$ holds iff $\text{DomSqrt}(I)$, $p \geq 0$, $q \geq 0$, $p^2 \leq a \cdot D$, and $b \cdot D \leq q^2$.

The output encoded interval is $\langle p, q \rangle$. (Lean: `DomSqrt`, `CheckSqrt`, `WSqrt`.)

Why integer squares? The encoded lower bound $p/D \leq \sqrt{a/D}$ is equivalent to $(p/D)^2 \leq a/D$, i.e. $p^2 \leq aD$. Similarly $\sqrt{b/D} \leq q/D$ is equivalent to $b/D \leq (q/D)^2$, i.e. $bD \leq q^2$. Both conditions involve only integer arithmetic (squaring and multiplication by D), so the check is decidable in Σ_{int} .

Theorem 24 (R-SOUND for sqrt) *If $\text{CheckSqrt}(D, I, W)$ holds and $x \in \gamma_D(I)$, then $\sqrt{x} \in \gamma_D(p, q)$. (Lean: `rsound_sqrt`; closedness: `rsound_sqrt_closed`.)*

Proof Write $I = (a, b)$ and $W = (p, q)$. From $\text{CheckSqrt}(D, I, W)$ we have $a \geq 0, p \geq 0, q \geq 0, p^2 \leq aD$, and $bD \leq q^2$. Since $D > 0$, the two square inequalities are equivalent to

$$\left(\frac{p}{D}\right)^2 \leq \frac{a}{D}, \quad \frac{b}{D} \leq \left(\frac{q}{D}\right)^2.$$

If $x \in \gamma_D(I)$, then $a/D \leq x \leq b/D$. Hence

$$\left(\frac{p}{D}\right)^2 \leq x \leq \left(\frac{q}{D}\right)^2.$$

Also $x \geq 0$ because $a \geq 0$ and $D > 0$.

The lower bound follows from square-root monotonicity: since $p/D \geq 0$ and $(p/D)^2 \leq x$, we have $p/D \leq \sqrt{x}$. The upper bound follows from $q/D \geq 0$ and $x \leq (q/D)^2$, which imply $\sqrt{x} \leq q/D$. Therefore

$$\frac{p}{D} \leq \sqrt{x} \leq \frac{q}{D},$$

i.e. $\sqrt{x} \in \gamma_D(p, q)$. Moreover $p^2 \leq aD \leq bD \leq q^2$ and $p, q \geq 0$ imply $p \leq q$, so the output interval is well formed. \square

5.5 Rectified linear unit

Definition 25 (ReLU transfer function) $\text{relu}(x) = \max(x, 0)$. For $I = (a, b) \in \mathcal{I}_D$:

$$T_{\text{relu}}(I) = (\max(a, 0), \max(b, 0)).$$

No witness is required. (Lean: `T_relu`.)

Theorem 26 (R-SOUND for relu) *For all $I \in \mathcal{I}_D$ and $x \in \gamma_D(I)$, $\text{relu}(x) \in \gamma_D(T_{\text{relu}}(I))$. (Lean: `rsound_relu`.)*

Proof Write $I = (a, b)$. Since $D > 0$,

$$\frac{\max(a, 0)}{D} = \max\left(\frac{a}{D}, 0\right), \quad \frac{\max(b, 0)}{D} = \max\left(\frac{b}{D}, 0\right).$$

If $x \in \gamma_D(I)$, then $a/D \leq x \leq b/D$. The map $t \mapsto \max(t, 0)$ is monotone, so

$$\max\left(\frac{a}{D}, 0\right) \leq \max(x, 0) \leq \max\left(\frac{b}{D}, 0\right).$$

By definition $\text{relu}(x) = \max(x, 0)$ and $T_{\text{relu}}(I) = (\max(a, 0), \max(b, 0))$. Hence $\text{relu}(x) \in \gamma_D(T_{\text{relu}}(I))$. \square

Remark 27 (Constant-size primitive checks) For each primitive in Σ_{prim} , the number of atomic integer checks performed by a single row checker is bounded by a constant depending only on the primitive symbol, not on the bit-length of the integers or on the ledger size. This follows by inspection of the fixed row schemata: addition, subtraction, and ReLU require only fixed endpoint computations; multiplication uses four corner products, two extremum-selection clauses, and two Euclidean division witnesses; inversion uses one fixed domain clause and two Euclidean division witnesses; and square root uses one fixed domain clause and fixed square-root witness inequalities. Thus each row expands to a bounded conjunction of integer atoms, with a bound depending only on the primitive symbol and not on the program size.

6 Certificate Language, DAGs, and the Replay Verifier

Design principle. A certificate bundles the program description and all Euclidean witnesses into a single *ledger*. The replay verifier processes the ledger row by row in topological order, performing only integer comparisons at each step. If every row-level check passes and every specified root constraint is satisfied, the verifier returns **true**. There is no backtracking, no constraint solving, and no floating-point evaluation inside the verifier core.

6.1 Program rows and DAGs

Definition 28 (Program rows) A *program row* over denominator D is one of:

<code>input(q, I_c)</code>	(external input oracle port q , claimed envelope I_c)
<code>const(k)</code>	(integer constant k , decoded as k/D)
<code>add(i, j, I_c)</code>	(sum of rows i and j , claimed bound I_c)
<code>sub(i, j, I_c)</code>	(difference of rows i and j)
<code>mul(i, j, W, I_c)</code>	(product, multiplication witness W)
<code>inv(i, W, I_c)</code>	(reciprocal, inversion witness W)
<code>sqrt(i, W, I_c)</code>	(square root, sqrt witness W)
<code>relu(i, I_c)</code>	(rectified linear unit)

A *program DAG* is a topologically ordered list of program rows with a well-formedness condition: every index i or j referenced in a row must be strictly less than the position of that row in the list. (*Lean: ProgRow, ProgDAG, ProgRowsTopological.*)

Definition 29 (Admissible input oracle) An *input oracle* $u : \mathbb{N} \rightarrow \mathbb{R}$ is admissible for a program ledger \mathcal{L} if for every `input(q, I_c)`-row in \mathcal{L} , $u(q) \in \gamma_D(I_c)$. (*Lean: RowsInputAdmissible.*)

6.2 Certificates and the replay verifier

Definition 30 (Certificate) A *certificate* $\mathcal{C} = (\mathcal{L}_{\text{prog}}, \mathcal{L}_{\text{spec}}, \text{roots})$ consists of: a program ledger $\mathcal{L}_{\text{prog}}$ (a topologically ordered list of program rows); a specification ledger $\mathcal{L}_{\text{spec}}$ (a topologically ordered list of specification rows, defined in Section 6.3); and a list $\text{roots} \subseteq \mathbb{N}$ of indices into the specification interval environment. A *DAG certificate* $\mathcal{C}_{\text{DAG}} = (G, S, \text{roots})$ wraps a program DAG G and a specification DAG S ; it is lowered to a certificate by compiling each DAG to its topological row list. (*Lean: Cert, DAGCert, DAGCert.toCert.*)

Definition 31 (Replay verifier \mathcal{V}) The *replay verifier* $\mathcal{V}(\mathcal{C})$ executes three phases:

- (I) Prog-Replay: process each row of $\mathcal{L}_{\text{prog}}$ in order, building an interval environment ι_{prog} by checking each row's integer condition; return **false** if any check fails.
- (II) Spec-Replay: process each row of $\mathcal{L}_{\text{spec}}$ in order, building ι_{spec} ; return **false** if any check fails.
- (III) Root-Check: for each index $r \in \text{roots}$, check that $\iota_{\text{spec}}[r].\text{hi} \leq 0$; return **false** if any index is out of range or the bound fails.

Return `true` if all three phases succeed. (*Lean*: `verifierBool`, `verifierDAGBool`, `replayRowBool`, `replaySpecRowBool`.)

6.3 Specification rows and compilation

Definition 32 (Specification rows) A *specification row* over denominator D and program-environment length n_{prog} is one of:

<code>var</code> (v, I_c)	(reference to program value $v < n_{\text{prog}}$, claimed bound I_c)
<code>const</code> (k)	(constant k , decoded as k/D)
<code>add</code> (i, j, I_c)	(sum of spec values i and j)
<code>sub</code> (i, j, I_c)	(difference)
<code>mul</code> (i, j, W, I_c)	(product, with multiplication witness W)

The specification grammar covers $\{+, -, \times, \text{var}, \text{const}\}$. Primitives `inv` and `sqrt` are not available as specification constructors in the current formalisation; extending the specification language to further operations would require corresponding R-SOUND results on the specification side and is left to future work. (*Lean*: `SpecRow`.)

Source-level specification expressions. In practice, a specification e is written as a polynomial expression over program outputs. The Lean formalisation includes an inductive type `SpecExpr` (constructors: `var`, `const`, `add`, `sub`, `mul`) with direct real semantics, and a compiler `compileSpecExprTree` that maps a `SpecExpr` to a tree-shaped sequence of `SpecRows`.

Theorem 33 (Specification compilation and DAG row-list lowering correspondence)

- (a) (Tree step.) *For every specification expression e and program value environment x , the compiled tree-shaped row sequence evaluates under x to the same value as e directly.* (*Lean*: `compileSpecExprTree_correct`.)
- (b) (DAG row-list lowering step.) *For every program DAG G , input oracle u , and program environment ρ , program DAG semantics coincide with the replay row-list semantics obtained by lowering DAG nodes to replay rows:*

$$\text{ProgDAGSem}(D, u, G, \rho) \Leftrightarrow \text{EvalProgFrom}(D, u, [], \text{compileProgDAG}(D, G), \rho).$$

The analogous statement holds for specification DAGs, with specification output environments in place of ρ . (*Lean*: `compileProgDAG_semantics_correct`, `compileSpecDAG_semantics_correct`.)

Proof For the tree step, proceed by structural induction on e . The cases `var` and `const` are immediate because compilation preserves the constructor and the real semantics of the compiled tree and of the source expression are definitionally identical. For `add`(e_1, e_2), the induction hypotheses give equality of the compiled and direct semantics for e_1 and for e_2 ; substituting these two equalities into the defining equation for addition gives equality for the parent expression. The subtraction and multiplication cases are the same, using the corresponding defining equations. Thus every compiled tree expression has exactly the same real value as the source expression from which it was compiled.

For the DAG row-list lowering step, proceed by induction over the topological order of the DAG nodes. At each step, the lowering map forgets the node identifier and appends exactly the corresponding replay row payload. The induction invariant states that the environment obtained after evaluating the first k DAG nodes is the same as the environment obtained after replaying the first k lowered rows. The base case is the empty prefix, and the step case follows because the k -th lowered row has the same parents, primitive symbol, witness, and output interval as the k -th DAG node. Hence

$$\text{ProgDAGSem}(D, u, G, \rho) \iff \text{EvalProgFrom}(D, u, [], \text{compileProgDAG}(D, G), \rho).$$

The specification-DAG case is identical, with specification rows and EvalSpecFrom . \square

Remark 34 Theorem 33(a) covers tree-structured compilation (expressions compiled without sub-expression sharing). Claims about common-subexpression elimination or arbitrary DAG sharing are not within the scope of this theorem or its Lean counterpart.

7 Main Soundness and Complexity Results

Two component soundness theorems and one collected closure theorem close the loop. Theorem 35 proves program replay soundness: Boolean acceptance yields a total and unique real program trajectory enclosed by the certified program intervals. Theorem 36 lifts this to the specification ledger and the certified root conditions. These two results are collected in Theorem 37, the named verifier-closure theorem stated as the main result of the paper. Theorem 38 bounds the structural cost of replay. The corresponding formal statements are mechanically verified in Lean 4; the precise paper–Lean correspondence is given in Table 1.

The first theorem isolates program replay soundness; the second theorem is the end-to-end acceptance result after specification replay and root checking.

Theorem 35 (Trajectory totality, enclosure, and uniqueness) *If $\mathcal{V}(\mathcal{C}) = \text{true}$, then for every admissible input oracle u there exists a unique program environment ρ_{prog} such that:*

- (a) $\text{EvalProgFrom}(D, u, [], \mathcal{L}_{\text{prog}}, \rho_{\text{prog}})$ holds;
- (b) every value $\rho_{\text{prog}}[i]$ is enclosed by the corresponding certified interval in ι_{prog} ;
- (c) all primitive domain conditions required by the accepted program rows hold.

Lean correspondence: Table 1.

Proof Unfold the Boolean verifier. Since $\mathcal{V}(\mathcal{C}) = \text{true}$, the program replay phase must return some certified program interval environment ι_{prog} , the specification replay phase must return some certified specification interval environment ι_{spec} , and the root-check phase must succeed. Soundness of the Boolean replay functions converts these three successful Boolean computations into the propositional facts

$$\text{ReplayProgFrom}(D, [], \mathcal{L}_{\text{prog}}, \iota_{\text{prog}}), \quad \text{ReplaySpecFrom}(D, \iota_{\text{prog}}, [], \mathcal{L}_{\text{spec}}, \iota_{\text{spec}}),$$

together with the root upper-bound condition on ι_{spec} .

Here u assigns concrete real values to input ports; in the running example, if the two input rows are ordered as v_1, v_2 , then $u(0) = 1.5$ corresponds to v_1 and $u(1) = 4$ corresponds to v_2 .

Fix an admissible input oracle u . We first prove existence and enclosure for the program trajectory by induction on the program replay derivation. The empty ledger produces the empty real environment, and the enclosure relation is immediate. In the step case, suppose the already replayed prefix has produced a real environment enclosed by the certified interval environment. The next row is handled by cases. Input rows are enclosed by admissibility of u ; constant rows are enclosed by the singleton interval; addition and subtraction rows use Theorem 16; multiplication, inversion, square root, and ReLU rows use Theorems 19, 22, 24, and 26, respectively. In the inversion and square-root cases, the row checks include the corresponding domain clauses, so the real operation is defined. Appending the newly computed real value preserves the environment-enclosure invariant. Thus there exists a real program environment ρ_{prog} such that $\text{EvalProgFrom}(D, u, [], \mathcal{L}_{\text{prog}}, \rho_{\text{prog}})$ holds and every entry of ρ_{prog} is contained in the corresponding certified interval of ι_{prog} .

It remains to prove uniqueness. Each program-row semantics is functional. For an input or constant row the output value is fixed by definition. For an arithmetic row, any two evaluations must read the same previously produced operands, and by induction those operands are equal; the resulting output therefore agrees. For inversion and square root, the same argument applies because each is a single-valued real function on its verified domain; for ReLU it applies because $\max(x, 0)$ is single-valued on all real inputs. \square

What this buys. Totality certifies that the program is well-defined on all admissible inputs under the certified intervals; enclosure says the certified intervals are genuine over-approximations; uniqueness says the real trajectory is deterministic given u . For \mathcal{P}_0 : after ACCEPT, every product $z = u_{v_1} u_{v_2}$ with $(u_{v_1}, u_{v_2}) \in [1, 2] \times [3, 5]$ satisfies $z \in [3, 10]$.

Theorem 36 (Specification enclosure) *If $\mathcal{V}(C) = \text{true}$ and u is admissible, let ρ_{prog} be the unique program environment given by Theorem 35. Then there exists a unique specification environment ρ_{spec} such that $\text{EvalSpecFrom}(D, \rho_{\text{prog}}, [], \mathcal{L}_{\text{spec}}, \rho_{\text{spec}})$ holds, and for every root index $r \in \text{roots}$ there exists a certified root interval $I_r = \iota_{\text{spec}}[r]$ such that $\rho_{\text{spec}}[r] \in \gamma_D(I_r)$ and $\overline{I_r}/D \leq 0$. Lean correspondence: Table 1.*

Proof As in Theorem 35, Boolean acceptance yields certified interval environments ι_{prog} and ι_{spec} such that program replay and specification replay both hold propositionally, and such that every root index passes the upper-endpoint check.

Fix an admissible input oracle u . Program replay soundness first gives a real program environment ρ_{prog} with $\text{EvalProgFrom}(D, u, [], \mathcal{L}_{\text{prog}}, \rho_{\text{prog}})$ and an enclosure invariant between ρ_{prog} and ι_{prog} . Specification replay soundness then gives a real specification environment ρ_{spec} with $\text{EvalSpecFrom}(D, \rho_{\text{prog}}, [], \mathcal{L}_{\text{spec}}, \rho_{\text{spec}})$ and an enclosure invariant between ρ_{spec} and ι_{spec} .

Let $r \in \text{roots}$. The successful root check provides an interval I_r with $\iota_{\text{spec}}[r] = I_r$ and $\overline{I_r} \leq 0$. By the specification-enclosure invariant, the real root value $\rho_{\text{spec}}[r]$ exists and belongs to $\gamma_D(I_r)$. Therefore

$$\rho_{\text{spec}}[r] \leq \frac{\overline{I_r}}{D} \leq 0,$$

because $D > 0$. This proves the specification constraint at every listed root.

Uniqueness follows from the row-functional argument used in Theorem 35: the program trajectory is unique, and once it is fixed, the specification row semantics is functional by

induction over the specification ledger. Therefore the specification trajectory is unique. For DAG certificates, `DAGCert.toCert` lowers the program and specification DAGs to the corresponding replay ledgers; the DAG part of Theorem 33 is used only as a row-list lowering correspondence between DAG semantics and replay-row semantics. Hence the same enclosure and root non-positivity conclusion holds at the DAG level. \square

What this buys. Specification enclosure connects the abstract replay output to a concrete claim about real behaviour: every specification constraint identified in roots holds for all admissible inputs. For \mathcal{P}_0 : the root interval for $v_3 - 10$ has $\bar{I}_r/D = 0/2 = 0$, confirming $v_3 - 10 \leq 0$ for all admissible inputs.

Theorem 37 (Verifier Closure for Σ_{prim}) *For the primitive core*

$$\Sigma_{\text{prim}} = \{+, -, \times, \text{inv}, \text{sqrt}, \text{relu}\},$$

if the replay verifier accepts a certificate $\mathcal{V}(\mathcal{C}) = \text{true}$, then for every admissible input u , program replay and specification replay determine unique real trajectories enclosed by the certified intervals, and every certified root constraint satisfies

$$\rho_{\text{spec}}[r] \leq 0 \quad (r \in \text{roots}).$$

All replay obligations are discharged by deterministic checks of quantifier-free ground integer conditions and explicit Euclidean witnesses, with no real-arithmetic search at replay time.

Proof This is the combination of Theorems 35 and 36, together with the integer-witness reductions of Sections 4–5. \square

7.1 Replay complexity

Theorem 38 (Structural linear replay cost) *Let $n = |\mathcal{L}_{\text{prog}}|$, $s = |\mathcal{L}_{\text{spec}}|$, and $r = |\text{roots}|$. The structural replay-cost model assigns exactly $n + s + r$ local check positions to a certificate: n program-row checks, s specification-row checks, and r root checks. In particular, the structural replay cost is $O(n + s + r)$. (Lean: `verifierBoolStructuralCost_eq_size`, `replayVerifierStructuralCost_eq_size`, `replayVerifierStructuralCost_linear`, `verifierBoolStructuralCost_linear`.)*

Proof By definition of the structural replay-cost model, each row of $\mathcal{L}_{\text{prog}}$ contributes one local program-row check position, each row of $\mathcal{L}_{\text{spec}}$ contributes one local specification-row check position, and each element of roots contributes one root-check position. Therefore the total number of structural check positions is

$$|\mathcal{L}_{\text{prog}}| + |\mathcal{L}_{\text{spec}}| + |\text{roots}| = n + s + r.$$

Hence the structural replay cost is $O(n + s + r)$. The Lean formalisation verifies the equality between the implemented structural cost counter and this paper-level size measure; the value is that mechanised correspondence, not the asymptotic estimate alone. If roots are represented as distinct specification output indices, then $r \leq s$, and the bound simplifies to $O(n + s)$. \square

Remark 39 (Certificate size) At the same structural level, the certificate size is linear in $n + s + r$. Each program row carries either no witness or one witness tuple from a primitive schema of constant arity: multiplication carries four integers, inversion and square root carry two integers, and addition, subtraction, and ReLU carry none. Specification rows and root checks also carry a bounded amount of integer data per row. Hence the number of witness integers and local certificate fields is $O(n + s + r)$. As in Remark 40, this is a structural size statement, not a bit-length bound.

Remark 40 (Bit-complexity) By Remark 27, each row-processing step involves at most a constant number of integer comparisons. If the cost of each integer comparison on b -bit integers is $C(b)$, the total check time is $O((n + s + r)C(b))$. This bound is stated informally: the Lean formalisation models the structural step count only and does not model arbitrary-precision arithmetic cost. The factor $C(b)$ is *not* Lean-verified.

8 Deployment Binding: The Implementation-Inclusion Contract

Separation of concerns. Every theorem in Section 7 concerns the mathematical semantics of each primitive as formalised in the Lean verifier core. The verifier treats each primitive $\sigma \in \Sigma_{\text{prim}}$ as a mathematical function $f_\sigma : \mathbb{R}^k \rightarrow \mathbb{R}$ (or partial function, for `inv` and `sqrt`). A deployed device evaluates σ through floating-point, fixed-point, or other implementation-specific routines whose outputs may differ from $f_\sigma(x)$ due to rounding, NaN/Inf propagation, or undefined behaviour.

The verifier deliberately excludes all such implementation detail from its trusted computing base. Transfer to a deployed implementation is factored out as an explicit *implementation-inclusion contract*. This is a separate obligation on the implementer, *not* part of the Lean formalisation, and *not* discharged by the replay verifier.

Assumption 41 (Implementation-inclusion contract) Let $D \in \mathbb{N}^+$ and let $\mathcal{C} = (\mathcal{L}_{\text{prog}}, \mathcal{L}_{\text{spec}}, \text{roots})$ be a certificate accepted by \mathcal{V} . For an admissible input u , let

$$\hat{\rho}_u^{\text{prog}} : \{0, \dots, |\mathcal{L}_{\text{prog}}| - 1\} \rightarrow \mathbb{R}$$

denote the deployed values produced for the program rows, and let

$$\hat{\rho}_u^{\text{spec}} : \{0, \dots, |\mathcal{L}_{\text{spec}}| - 1\} \rightarrow \mathbb{R}$$

denote the deployed values produced for the specification rows.

The *implementation-inclusion contract* has two parts.

Program side. For each program row index p with $\mathcal{L}_{\text{prog}}[p] = \sigma(i_1, \dots, i_k, W_p, I_c)$:

- (a) the deployed operand values $\hat{\rho}_u^{\text{prog}}(i_1), \dots, \hat{\rho}_u^{\text{prog}}(i_k)$ satisfy the domain condition of σ ;
- (b) the deployed output value satisfies $\hat{\rho}_u^{\text{prog}}(p) \in \gamma_D(I_c)$.

Specification side. For each specification row index q whose certified interval is I_c :

- (c) every referenced program row or specification row value is defined;
- (d) the deployed specification value satisfies $\hat{\rho}_u^{\text{spec}}(q) \in \gamma_D(I_c)$.

For a `var`-row in $\mathcal{L}_{\text{spec}}$ referencing program row p , the deployed specification value is taken from the corresponding program row:

$$\hat{\rho}_u^{\text{spec}}(q) = \hat{\rho}_u^{\text{prog}}(p).$$

Contract violations include floating-point rounding outside the certified interval, NaN or Inf outputs within the certified domain, domain mismatches, and undefined branch behaviour.

Remark 42 Assumption 41 is stated mathematically and is *not* formalised in Lean. This is a deliberate design choice: the Lean proof obligation is focused on the verifier core, and the implementation binding is an obligation on the system integrator, not on the verifier.

Corollary 43 (Transfer to deployed implementation) *If $\mathcal{V}(\mathcal{C}) = \text{true}$ and Assumption 41 holds, then for every admissible input u and every root index $r \in \text{roots}$, the deployed specification root value $\hat{\rho}_u^{\text{spec}}(r)$ satisfies the corresponding specification constraint.*

Proof The verifier does not reason directly about the deployed floating-point execution; it reasons about certified interval enclosures. By verifier acceptance (Theorem 36), for each root index $r \in \text{roots}$, the replay verifier has established a certified root interval $I_r = \iota_{\text{spec}}[r]$ whose upper endpoint satisfies $I_r \leq 0$. By Assumption 41 (specification side, clause (d)), the deployed specification value at the root node satisfies $\hat{\rho}_u^{\text{spec}}(r) \in \gamma_D(I_r) = [\underline{I}_r/D, \overline{I}_r/D]$. Therefore

$$\hat{\rho}_u^{\text{spec}}(r) \leq \frac{\overline{I}_r}{D} \leq 0.$$

Thus every deployed specification root value satisfies the corresponding specification constraint. \square

The separation serves two purposes. First, it keeps the trusted kernel small: only the replay verifier and its primitive rules need be trusted, not any floating-point library. Second, it makes the scope of the theorems precise: if the contract is violated—by rounding drift, domain mismatch, or implementation divergence—the deployment guarantee does not hold, but the verifier’s mathematical theorems of Section 7 are untouched.

9 Supporting Artefacts

The theorems of the preceding sections stand independently. The following external artefact accompanies the paper; it is not part of the verifier’s trusted computing base.

Lean 4 formalisation. The accompanying Lean 4 formalisation is provided as a reproducible Lake project. Its main proof file is `ADIC.RSound.Replay.lean`, which imports `Mathlib` and mechanically verifies the theorem-level content listed in Table 1. The repository also contains `lean-toolchain`, `lakefile.lean`, `lake-manifest.json`, and a GitHub Actions workflow for independent replay.

All theorem statements listed as mechanically verified in Table 1 are proved without `sorry`.

The formalisation does *not* cover:

- Assumption 41 (implementation-inclusion contract) or Corollary 43—deliberate design;
- the $C(b)$ factor of Remark 40—only structural step count is modelled;
- `inv` or `sqrt` as specification-side operators—`SpecRow` covers $\{\text{var}, \text{const}, +, -, \times\}$ only.

Lean \leftrightarrow paper correspondence.

Table 1 Lean 4 coverage of paper statements. \checkmark = mechanically proved in `ADIC_RSound_Replay.lean`; \times = not formalised (stated mathematically only).

Paper statement	Lean identifier(s)	
Def 3: \mathcal{I}_D	<code>IEnc D</code>	\checkmark
Def 4: γ_D, α_D	<code>IEnc.gamma, IEnc.alpha</code>	\checkmark
Thm 5: Galois insertion	<code>gc.alpha_gamma, alpha_gamma_strict</code>	\checkmark
Def 7: <code>IntTerm</code>	<code>IntTerm, IntTerm.eval</code>	\checkmark
Def 8: <code>Atom, QFForm</code>	<code>Atom, QFForm</code>	\checkmark
Def 9: <code>SrcAtom, SrcForm</code>	<code>SrcAtom, SrcForm</code>	\checkmark
Def 10: τ	<code>tau, tauAtom</code>	\checkmark
Lem 11: truth preservation	<code>tau.preserves, tau.truth</code>	\checkmark
Def 12: <code>FDIV, CDIV</code>	<code>CheckFDIV, CheckCDIV</code>	\checkmark
Lem 13: division witnesses	<code>fdiv.bound, cdiv.bound</code>	\checkmark
<code>DomInv, DomSqrt</code>	<code>DomInv, DomSqrt</code>	\checkmark
Def 18: <code>CheckMul</code>	<code>CheckMul, WMul</code>	\checkmark
Def 21: <code>CheckInv</code>	<code>CheckInv, WInv</code>	\checkmark
Def 23: <code>CheckSqrt</code>	<code>CheckSqrt, WSqrt</code>	\checkmark
Def 25: T_{relu}	<code>T.relu</code>	\checkmark
Thm 16: R-SOUND $+$, $-$	<code>rsound.add, rsound.sub</code>	\checkmark
Thm 19: R-SOUND \times	<code>rsound.mul, rsound.mul_closed</code>	\checkmark
Thm 22: R-SOUND <code>inv</code>	<code>rsound.inv, rsound.inv_closed</code>	\checkmark
Thm 24: R-SOUND <code>sqrt</code>	<code>rsound.sqrt, rsound.sqrt_closed</code>	\checkmark
Thm 26: R-SOUND <code>relu</code>	<code>rsound.relu</code>	\checkmark
Def 28: <code>ProgRow</code>	<code>ProgRow, ProgDAG</code>	\checkmark
Def 30: <code>Cert, DAGCert</code>	<code>Cert, DAGCert</code>	\checkmark
Def 31: \mathcal{V}	<code>verifierBool, verifierDAGBool</code>	\checkmark
Def 32: <code>SpecRow</code>	<code>SpecRow</code>	\checkmark
Thm 33: compilation	<code>compileSpecExprTree.correct,</code> <code>compileProgDAG.semantics.correct,</code> <code>compileSpecDAG.semantics.correct</code>	\checkmark
Thm 35: totality + enclosure (via chain <code>acceptProp \rightarrow replayProg.total_sound \rightarrow verifierBool.sound</code>)	<code>verifierBool.sound,</code> <code>verifierBool.sound.unique,</code> <code>replayProg.total_sound</code>	\checkmark
Row eval uniqueness	<code>evalRow.unique</code>	\checkmark
Thm 36: spec enclosure (via <code>replaySpec.total_sound, roots.nonpos.sound</code>)	<code>replaySpec.total_sound,</code> <code>verifierDAGBool.direct.sound,</code> <code>dagCert.end.to_end.sound</code>	\checkmark
Unique trajectory (end-to-end)	<code>acceptProp.sound.unique</code>	\checkmark
Thm 38: structural cost	<code>verifierBoolStructuralCost.eq_size,</code> <code>replayVerifierStructuralCost.linear</code>	\checkmark
Remark 40: $O((n + s + r)C(b))$	—	\times
Assm 41: impl. contract	—	\times
Cor 43: deployment transfer	—	\times
Spec ops: <code>inv/sqrt</code> in spec DAGs	—	\times
Remark 27: per-primitive atom count	—	\times

10 Discussion

How to read the verifier-closure result. Theorems 35 and 36 are not results about interval arithmetic in general, and they are not complexity results for non-linear real arithmetic. They are results about a specific verifier architecture: given a certificate whose Euclidean witnesses are consistent with the integer checks, the verifier can confirm safety without any real-arithmetic reasoning at check time. The certificate producer must still solve a non-trivial problem to find those witnesses; the contribution is that the solution can be *checked* cheaply, search-free, and inside a mechanically verified kernel.

This does not mean that the local primitive witnesses are non-constructive. For the primitives in Σ_{prim} , the witnesses are obtained by explicit integer operations: endpoint arithmetic for $+$, $-$, and relu ; corner products and Euclidean floor/ceiling division for \times ; Euclidean division of D^2 for inv ; and integer square-root bounds for sqrt . What is not claimed is a global completeness theorem for finding full replay ledgers for all true specifications.

Why straight-line programs? The straight-line restriction is part of the closure boundary. The replay ledger is processed once, in a known acyclic dependency order, and each row has a single certified semantic role. Conditional branches would require evidence identifying the taken branch and certifying the branch condition. Loops would require either materialisation by bounded unrolling, or separate loop invariants and termination evidence. In the bounded-unrolling case, the theorem applies only after the unrolled program has been materialised as an ordinary straight-line ledger. Native control-flow constructs are not part of the present certificate language. These extensions are therefore outside the closure theorem rather than being covered implicitly by it.

Limitations. The formalisation has four explicit limitations. First, the specification grammar covers only $\{+, -, \times, \text{var}, \text{const}\}$; specification constraints involving inv or sqrt are not supported in the current Lean model. Second, the complexity result is structural: it counts row-processing steps, not individual bit-operations; the $C(b)$ factor is argued informally. Third, the implementation-inclusion contract is a mathematical assumption, not a mechanised result; it is an obligation on the system integrator.

Fourth, the scale denominator D is chosen on the producer side. The soundness theorem is parametric in any positive D : a smaller D gives coarser encoded intervals, while a larger D reduces discretisation error at the cost of larger integers and higher bit-level arithmetic cost. The verifier checks the submitted D -scaled certificate; it does not optimise D or prove that the chosen scale is complete for a given application.

Relation to prior work. The architecture builds on three existing lines. Classical interval arithmetic provides sound transfer functions stated in real terms; the present contribution shows that, for Σ_{prim} , those functions reduce to integer checks witnessed by Euclidean data, so the checker is simplified and mechanically verified—at the cost of requiring a certificate producer. Proof-carrying code provides the architectural principle of off-loading proof construction to the producer and keeping the checker lightweight; here the proof object is narrowed to a finite integer ledger, which further restricts the checker’s trusted base. SMT solvers for non-linear arithmetic perform search inside

their kernel; the replay verifier contains no such search, and its soundness is established in Lean.

Why a fixed primitive core matters. Restricting to Σ_{prim} is not merely a simplification; it is what makes closure possible. For primitives outside Σ_{prim} (for example \exp , \tanh , or sigmoid), the graph of the operation is not captured by the polynomial and Euclidean witness patterns used here. The theorem therefore does not assert that Σ_{prim} is necessary or maximal; it asserts that this enumerated core is closed under constant-size certificate rows whose checks reduce to finite conjunctions of ground integer inequalities.

Why implementation binding is separated. Folding the implementation-inclusion contract into the Lean formalisation would require modelling a specific floating-point or fixed-point arithmetic system, binding the proof to a particular hardware target. By isolating it as a separate assumption, the verifier core remains hardware-agnostic, and the same Lean file can be used as the trusted kernel for different deployment platforms. The cost is that the end-to-end guarantee (Corollary 43) is not fully mechanised; this is a known and deliberate boundary.

11 Conclusion

We have established a verifier-closure theorem for straight-line interval programs over the fixed primitive core $\Sigma_{\text{prim}} = \{+, -, \times, \text{inv}, \text{sqrt}, \text{relu}\}$: every obligation the replay verifier must discharge is a quantifier-free ground integer formula, every non-trivial primitive rule is witnessed by explicit Euclidean data, and the verifier is solver-search-free at check time. The result is mechanically verified in Lean 4 using Mathlib.

The core mechanised argument runs from the strict Galois insertion $(\alpha_D(\gamma_D(I)) = I)$ and truth-preserving normalisation τ through per-primitive R-SOUND theorems for all six elements of Σ_{prim} , to the replay verifier and its end-to-end soundness. The Lean proof follows the same structure: the primitive R-SOUND theorems are composed with the replay semantics, the specification replay, and the root non-positivity check to obtain Boolean verifier soundness, unique trajectory, and the structural cost bound. The complete Lean 4–paper correspondence, including the theorem names used in the mechanised development, is given in Table 1.

Three items remain outside the Lean boundary by design: the bit-complexity factor $C(b)$, the implementation-inclusion contract (Assumption 41), and the extension of specification rows to inv or sqrt . The first is argued informally; the second is an obligation on the system integrator, not the verifier kernel; the third is left as future work.

Extending the specification grammar to inv and sqrt would require corresponding R-SOUND results on the specification side and non-trivial changes to the certificate format. Mechanising the implementation-inclusion contract for a specific floating-point target remains a separate verification problem. Both directions would strengthen the end-to-end deployment guarantee currently captured by Corollary 43.

A Paper-Only Claims

Table 2 enumerates all claims in the paper that have no Lean counterpart.

Table 2 Claims stated mathematically in the paper with no Lean counterpart.

Location	Content	Treatment
Remark 40	$O((n + s + r)C(b))$ with bit-cost $C(b)$	Informal; not Lean-verified
Remark 27	Per-primitive atom count bounded by a constant	Informal arithmetic argument
Assm 41	Implementation-inclusion contract	Mathematical assumption; design boundary
Cor 43	Transfer to deployed implementation	Follows from Assm 41; not mechanised
Thm 33 note	Optimised DAG sharing / CSE	Out of scope; tree-to-rows and DAG-to-rows only

Statements and Declarations

Use of AI tools. During the preparation of this manuscript, the author used large-language-model tools for language editing, structural revision support, and consistency checking. The author reviewed all outputs and takes full responsibility for the final content of the manuscript.

Author contributions. The author is solely responsible for the conception, formalisation, proof development, manuscript preparation, and final approval of the work.

Funding. No funding was received to assist with the preparation of this manuscript.

Competing interests. The author is affiliated with GhostDrift Mathematical Institute, Inc. The company studies broader audit and verification architectures that may have commercial relevance, and the author is involved in related patent applications concerning applied audit-system architecture. The present paper is a separate mathematical and mechanised formalisation of an interval replay verifier over Σ_{prim} . The definitions, theorems, proofs, and Lean artefact reported here are fully stated in the manuscript and repository and are independently checkable.

Ethics approval. Not applicable.

Consent to participate. Not applicable.

Consent for publication. Not applicable.

Code availability. The Lean 4 formalisation accompanying this manuscript is available at <https://github.com/GhostDriftTheory/adic-lean-proof-replay>. The repository is a reproducible Lake project containing `ADIC_RSound_Replay.lean`, `lean-toolchain`,

`lakefile.lean`, `lake-manifest.json`, `README.md`, and a GitHub Actions workflow. A clean clone followed by `lake exe cache get` and `lake build` verifies the artifact.

Data availability. An archival record of the Lean 4 formalisation accompanying this manuscript is deposited on Zenodo and is available at <https://doi.org/10.5281/zenodo.19808324>.

References

- [1] L. de Moura and S. Ullrich. The Lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28*, Lecture Notes in Computer Science, volume 12699, pages 625–635. Springer, 2021. https://doi.org/10.1007/978-3-030-79876-5_37.
- [2] The mathlib Community. The Lean Mathematical Library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2020)*, pages 367–381. ACM, 2020. <https://doi.org/10.1145/3372885.3373824>.
- [3] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1997)*, pages 106–119. ACM, 1997. <https://doi.org/10.1145/263699.263712>.
- [4] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1977)*, pages 238–252. ACM, 1977. <https://doi.org/10.1145/512950.512973>.
- [5] R. E. Moore. *Interval Analysis*. Prentice-Hall, Englewood Cliffs, NJ, 1966.
- [6] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, Lecture Notes in Computer Science, volume 4963, pages 337–340. Springer, 2008. https://doi.org/10.1007/978-3-540-78800-3_24.
- [7] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar. cvc5: A versatile and industrial-strength SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2022)*, Lecture Notes in Computer Science, volume 13243, pages 415–442. Springer, 2022. https://doi.org/10.1007/978-3-030-99524-9_24.
- [8] R. Desmartin, O. Isac, G. O. Passmore, E. Komendantskaya, K. Stark, and G. Katz. A certified proof checker for deep neural network verification in Imandra. In *16th International Conference on Interactive Theorem Proving (ITP 2025)*, Leibniz International Proceedings in Informatics (LIPIcs), volume 352, article 1, pages 1:1–1:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. <https://doi.org/10.4230/LIPIcs.ITP.2025.1>.
- [9] A. D. Brucker, T. Cameron-Burke, and A. Stell. Formally verified interval arithmetic and its application to program verification. In *FormaliSE 2024: 12th International Conference on Formal Methods in Software Engineering*, pages 111–121. IEEE/ACM, 2024. <https://doi.org/10.1145/3644033.3644370>.

- [10] IEEE Computer Society. IEEE Standard for Interval Arithmetic. *IEEE Std 1788-2015*, 2015. <https://doi.org/10.1109/IEEESTD.2015.7140721>.
- [11] G. Melquiond. Proving bounds on real-valued functions with computations. In *Automated Reasoning: 4th International Joint Conference, IJCAR 2008*, Lecture Notes in Computer Science, volume 5195, pages 2–17. Springer, 2008. https://doi.org/10.1007/978-3-540-71070-7_2.
- [12] F. Bréhard, A. Mahboubi, and D. Pous. A certificate-based approach to formally verified approximations. In *10th International Conference on Interactive Theorem Proving (ITP 2019)*, Leibniz International Proceedings in Informatics (LIPIcs), volume 141, article 8, pages 8:1–8:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. <https://doi.org/10.4230/LIPIcs.ITP.2019.8>.
- [13] M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Werner. A modular integration of SAT/SMT solvers to Coq through proof witnesses. In *Certified Programs and Proofs (CPP 2011)*, Lecture Notes in Computer Science, volume 7086, pages 135–150. Springer, 2011. https://doi.org/10.1007/978-3-642-25379-9_12.
- [14] B. Ekici, A. Mebsout, C. Tinelli, C. Keller, G. Katz, A. Reynolds, and C. Barrett. SMTCoq: A plug-in for integrating SMT solvers into Coq. In *Computer Aided Verification (CAV 2017)*, Lecture Notes in Computer Science, volume 10426, pages 126–133. Springer, 2017. https://doi.org/10.1007/978-3-319-63390-9_7.
- [15] H. Barbosa, J. C. Blanchette, M. Fleury, and P. Fontaine. Scalable fine-grained proofs for formula processing. *Journal of Automated Reasoning*, 64(3):485–510, 2020. <https://doi.org/10.1007/s10817-018-09502-y>.
- [16] H. Barbosa, C. W. Barrett, B. Cook, B. Dutertre, G. Kremer, H. Lachnitt, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, C. Tinelli, and Y. Zohar. Generating and exploiting automated reasoning proof certificates. *Communications of the ACM*, 66(10):86–95, 2023. <https://doi.org/10.1145/3587692>.
- [17] H. Lachnitt, M. Fleury, H. Barbosa, J. Jakpor, B. Andreotti, A. Reynolds, H.-J. Schurr, C. Barrett, and C. Tinelli. Improving the SMT proof reconstruction pipeline in Isabelle/HOL. In *16th International Conference on Interactive Theorem Proving (ITP 2025)*, Leibniz International Proceedings in Informatics (LIPIcs), volume 352, article 26, pages 26:1–26:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. <https://doi.org/10.4230/LIPIcs.ITP.2025.26>.
- [18] H. Becker, N. Zyuzin, R. Monat, E. Darulova, M. O. Myreen, and A. C. J. Fox. A verified certificate checker for finite-precision error bounds in Coq and HOL4. In *Formal Methods in Computer Aided Design (FMCAD 2018)*, pages 1–10. IEEE, 2018. <https://doi.org/10.23919/FMCAD.2018.8603019>.