

# The Replay Lattice: A Type System for Floating-Point Reproducibility

The foundation of a determinism-safe language

M. A. J. Haddon

michael@politepixels.io

PolitePixels

Saint Helier, Jersey, Channel Islands

## Abstract

Add the same floating-point numbers in a different order, and the last bits of the sum can change. A parallel reduction groups its input by worker count, and one program can then return different bits on different worker counts. Hence, a promise of bit-for-bit reproducibility depends on an unannounced schedule.

A floating-point value’s determinism is a *demand carried on the value*, one that holds only if every operation that ever touched the value obeyed it. Almost every mechanism the field has built to govern reordering is instead *attached to a site*, either a *grant*, a licence to reorder at one place in the program text, or a demand checked there alone. A grant is true where it is written. A demand’s truth is settled upstream, by the value’s history, which no single site can see. We present the Replay Lattice: three schedule-class contracts, exact below replay below approx, carried on every value of a dataflow array language, joined over the dependency path, checked at each declaration, and discharged into backend lowering obligations.

Two theorems come with the lattice. Soundness is a two-run induction over the dataflow graph, mechanised in Coq. The poisoning law says one free reordering taints everything downstream, and its proof is one line on top of the join. The result is a language in which *these bits must not depend on the schedule* is a sentence the compiler checks.

## CCS Concepts

• **Theory of computation** → **Type structures**; • **Software and its engineering** → *Functional languages*; *Parallel programming languages*; *Domain specific languages*; *Compilers*; *Formal software verification*; • **Mathematics of computing** → Numerical analysis.

## Keywords

determinism types, type qualifiers, floating-point determinism, floating-point reproducibility, schedule nondeterminism, reassociation, 2-safety hyperproperties, dataflow array languages

## 1 Introduction

Add 0.1, 0.2 and 0.3 as doubles. Grouped left to right you get 0.6000000000000001; grouped as  $0.1 + (0.2 + 0.3)$ , you get 0.6. However, the programmer does not choose which grouping a parallel program performs. A parallel reduction chunks its input by worker count, which is whatever the deployment provides.

This is more than a rounding curiosity. When a phylogenetics group reran 10,179 published datasets with only the parallelism changed, 31% of them came back with different answers [39]. \$7.5 replays that failure inside the artefact, with the repair beside it. The same effect is why a bit-exact regression test breaks under a new thread count, and why, in machine-learning inference, one prompt can yield different tokens depending on what else the server was batching [19].

We maintain a data-parallel array language whose purpose is reproducibility. While making it production-ready we hit an optimiser bug in its compiler, and that incident forced the discipline of carrying the demand through lowering.

The program it broke, in four lines:

```
x := indices(count: 100000.0)
a := sum(x)
replay b := sum(x)
return a - b
```

Our optimiser merged the two folds, because they were identical in every property it modelled, and silently pinned a value nobody had asked to pin. We expand upon the fix in §6.3. The language is our own and unreleased, which is why it is not named; no claim below rests on a measurement of it. We implement the checker for lanelang, a demonstration language built for this paper and released in full as a public artefact (Appendix A). The lattice itself is tied to neither language, because all it reads is a dependence graph and a table of operation classes.

Almost every mechanism the field offers is *attached to a site*, and half are *grants*, licences saying *you may reorder here*. Table 1 is a census of these mechanisms. Seven of its rows state a demand, and six of the seven attach to something other than a value, among them the per-call determinism level of CCCL, NVIDIA’s CUDA Core Compute Libraries [3]. The exception is GLSL’s precise, which is a demand on a value, and the closest row to what this paper builds.

**Reproducibility is not a grant.** “These bits must not depend on the schedule” is a *demand attached to a value*, and the site that writes it cannot check it, because the folds that decide those bits sit upstream, in code the site cannot see, and which schedule they ran under was the deployment’s choice. The join is what makes the demand checkable. This subject admits no escape downstream of a

free fold, because the rounding has already happened, and no later operation recovers the discarded low bits.

Our evidence for the gap is where the existing mechanisms stop, and CCCL’s stopping point is visible in its source. In particular, level resolution consults only the current call so that a `not_guaranteed` result may feed a `run_to_run` call, the weakest level feeding a stronger one, with no diagnostic anywhere [33]. The scheme answers the local question well and cannot ask the composition question at all, because no determinism level outlives the call.

**What this is.** This work is a type system for one property of one value, not a compiler, not a numerical method, and not a claim that any one schedule is the right one. Every number is asserted or printed by a numbered check a reader can run and listed against its claim in the artefact (Appendix A). A *control* is a fault staged on purpose for a check to catch, and one that does not fire fails the run. The exceptions are flagged where they occur: a few measurements come from one machine with no check behind them, and claims about the production compiler are observations a reader cannot rerun. §2 weighs the prior art on ordering, and §9 covers the rest.

#### Contributions.

1. The reframing of a floating-point schedule class as a demand carried on a value, and the type system it forces: three classes, a join over the dependency path, an inference pass that strengthens a producer only where nothing outside the demand reads it, and per-class backend lowering obligations (§3, §4, §6).
2. A reading of the class at lowering time that allows fusion, tiling, reordering and reduction splitting wherever they keep the grouping a demand fixes, and forbids merging twin reductions under value numbering (§6.3).
3. A mechanised soundness theorem for a schedule-indexed semantics, proved by two-run induction and checked in Coq with the poisoning law beside it, conditional on two stated assumptions, Assumptions 1 and 2, the first made concrete as a table (§3.5, §5).
4. An evaluation that reports its positive evidence with figures, then audits itself, finding and repairing ten occasions on which a passing check of ours had proved nothing (§7).

## 2 Grants and demands

Try to write the requirement *these bits do not depend on the schedule* in any system in Table 1. The first thing to look for is a promise that sits on the value itself, not on some line of code the value once passed through. Second, it has to be a requirement, not a permission to reorder. Third, somebody has to actually check it, over everything that ever touched the value. Fourth, it has to become an obligation on whoever picks the schedule. Go and look.

### 2.1 Determinism-safe languages

Every language that computes in floating point has a schedule: an order in which its reductions combine, set by the width at which they block or by a worker count that may vary between runs. However, almost none enforces anything about it. Reproducibility is lost in the gap between knowing and enforcing.

This paper borrows type safety as its framing, the property Milner proved, that well-typed expressions cannot “go wrong” [29]. Every JavaScript value knows its own type at runtime, and yet `'1' + 1` becomes `'11'`: the language coerces instead of refusing. Knowing a type and enforcing one are different things. Determinism non-safety can be defined in the same way, just on the topic of the schedule. If you add a schedule-pinned value to a value that is not, and the language coerces them together, then we can call it determinism-unsafe. Therefore, the remedy is the same as the type-safe one, which is refusal. We also place it at the declaration. What differs is that we must check everything that ever touched the value, not just the expression at that line.

With that in mind, we say a language is *determinism-safe* when five criteria hold:

- (i) Every value carries a schedule-determinism class.
- (ii) The class is a function of the value’s whole history, whereas a per-site knob is a function of the site where it is written.
- (iii) A declaration states a required class and is checked against the computed one.
- (iv) Violation is *rejection*, not advice, not a warning, not an inhibited optimisation.
- (v) Acceptance discharges into a stated obligation on whoever picks the schedule.

**Table 1: Mechanisms that govern reorder freedom: what each attaches to, and where its truth is settled.**

| System                | The knob                           | Attached to                 | Direction     | Enforced?              | Discharged into            |
|-----------------------|------------------------------------|-----------------------------|---------------|------------------------|----------------------------|
| LLVM/MLIR [49, 50]    | reassoc, contract                  | an instruction or operation | grant         | no                     | the optimiser              |
| SPIR-V [48]           | the FPFastMathMode decoration      | an instruction              | grant         | no                     | the driver                 |
| Icing opt: [8]        | a scope in the value grammar       | an expression               | grant         | verified               | a verified compiler        |
| Futhark reduce [46]   | a promise                          | an operator                 | grant         | no                     | the compiler               |
| Halide rfactor [40]   | a proved associativity             | a schedule directive        | grant         | in the reals           | the schedule               |
| OpenMP [34]           | the reduction clause               | a construct                 | grant         | no                     | the implementation         |
| MPI_REDUCE [28]       | an implicit reorder licence        | a collective call           | grant         | no                     | the implementation         |
| CCCL require [3, 33]  | a level                            | one call                    | <b>demand</b> | at that call only      | a kernel choice            |
| oneTBB reduce [52]    | a split/join fixed per partitioner | one call                    | <b>demand</b> | no                     | a library kernel           |
| Batch invariance [19] | an order fixed by extent           | one kernel                  | <b>demand</b> | no                     | the implementation         |
| Kernel contracts [53] | a three-level class                | one kernel                  | <b>demand</b> | by repeated invocation | a conformance verdict      |
| IEEE 754 [20]         | a reproducibility attribute        | a lexical block             | <b>demand</b> | by the language        | the implementation         |
| LS-DYNA [25]          | the consistency flag               | a run                       | <b>demand</b> | no                     | the solver                 |
| GLSL precise [47]     | a qualifier                        | a value, and its producers  | <b>demand</b> | no computed class      | the shader compiler        |
| <b>This work</b>      | a lane (§3)                        | a value                     | <b>demand</b> | <b>whole path</b>      | <b>backend obligations</b> |

**Only the subject of these five criteria is new.** Substitute a secrecy label for the class and the emitted instruction sequence for the schedule, and they describe a checked information-flow system. Volpano, Irvine and Smith first proved one sound in 1996, and its soundness theorem is already the two-run induction this paper’s proof uses [54]. Under that substitution, FaCT goes further and satisfies all five for constant-time execution [11].

Run Table 1 against these five criteria, and every row above ours fails. A grant states no class at all. Leaving a grant out does not state one either. A strict default or a sequential fold may fix the order at one site, but the code does not say that anything depends on that order. The requirement is invisible to the next person who changes the code, and to the next compiler pass. A demand attached to a site states one, and the site cannot see the value’s history. GLSL’s precise names one on the value and never computes it. A *determinism-safe language* is one where all five criteria hold at once.

Determinism-safe does not mean *deterministic*, since a language can produce the same bits every run on one machine and be determinism-unsafe, because no rule prevents a future schedule from changing them.

## 2.2 Why a demand is not settled locally

Two kitchen labels make the point. *May contain nuts* is settled at the dish that carries it. *Contains no nuts* is a claim about every ingredient that ever entered the pot, one nut anywhere falsifies it, and no later stirring takes it back out.

A grant is the first kind of sentence, existential over one site: *this operation may have been reordered*. Because its truth is settled locally, a second grant elsewhere cannot falsify it. In kitchen terms, you are “granting” the dish permission to contain nuts, rather than requiring their absence.

A demand is the second kind, universal over a history: *no operation that contributed to this value can be reordered in a way that depends on the schedule*. Its truth quantifies over the value’s ingredients, every input that fed it, directly or through others. Adding one operation anywhere in that history can falsify it, and no later operation can repair it, because the rounding it performed is already in the bits.

This one-way loss is the poisoning law, and it is why a per-call API is not simply inconvenient but *inexpressive*. When CCCL’s run\_to\_run reduction receives a pointer, the pointer is an unlabeled jar: it carries no history.

By contrast, GLSL has carried a demand on a *value*, where precise forbids reassociation and contraction, and propagates backward to “operations contributing to a variable’s value”, although only within one function [47]. The demand is on the value, and the reasoning is still per-function.

## 2.3 A property of two runs

The kitchen analogy explains why the whole history matters. Reproducibility adds another requirement, which is that the value must agree across two runs under different schedules. Calling a value schedule-invariant compares one run under schedule *s* with another under *s'*, bit for bit, and the literature calls such pair-of-runs

properties 2-safety [13]. A check attached to one call sits inside a single run and cannot see the other.

The join is what makes a 2-safety demand checkable here. It summarises the entire dependency path into one lattice element that travels with the value, which means a local comparison at the declaration site suffices. One fold left free upstream joins in as approx, and every expression its result touches inherits that. A declaration demanding replay then fails locally, no matter how far away the fold was. This approach draws on information-flow control [54], made for thread scheduling by observational-determinism type systems [41, 55], whose subject is interleaving on a shared location. A reduction lowered here has no interleaving to observe, because each worker owns its accumulator and the combine reads them in a fixed order.

## 2.4 The standard already drew this line

Clause 5 of IEEE 754 requires each basic operation to be correctly rounded, and both groupings in the opening example obey it. The bits differ anyway, because nothing varies between them except the bracketing, which is why per-operation correctness cannot carry reproducibility. The standard draws the same line itself: reductions sit in a subclause of their own, 9.4, which says implementations “may associate in any order or evaluate in any wider format” [20]. And the clause titled “Reproducible floating-point results” has never, since it arrived in 2008, covered 9.4, which means a reordered 9.4 sum can change its bits and break no clause.

## 2.5 Where checked guarantees stop

Several checked systems come close to the five criteria. This section asks one question of each: where does the checked guarantee stop? The answer is always a deliberate escape, a marked place where the programmer steps outside the check. DPJ, Deterministic Parallel Java [10], checks memory effects and takes the programmer’s word for commutativity, as its authors state plainly: commutativity “is not checked by the compiler”. EnerJ, a type system for approximate computing [37], keeps such values out of precise code, except where the programmer *endorses* one back in.

**The schedule is the subject on which no escape can be built.** Both escapes above hand the programmer something to certify, EnerJ a tolerated loss in a domain built to absorb one, a pixel or two wrong, and DPJ an unchecked annotation in the source, where a reader can grep for it and weigh the assumption. A reordered sum leaves neither. The low bits each rounding discarded are gone; the bits that remain are whatever the schedule made them, and the poisoning law proves the direction: a value that reached approx cannot be brought back. So where EnerJ and DPJ may summarise a history, certify it and drop it, this design has to carry the history on the value. The class therefore rides the value and joins along the dependency path, and the five criteria are forced, being what checking looks like when the history cannot be dropped.

The Replay Lattice’s one escape is where bytes enter from outside the program. An example is an input file. Knowing whether it will hold the same bytes run to run is a fact about a deployment instead of arithmetic, which means no checker can decide it. A blessing (§4.6) lets a programmer take that obligation on in writing, such as a checksum the run verifies.

### 3 The Replay Lattice as a type system

Lanelang, the demonstration language, represents programs as dataflow graphs of bulk array operations over double. It is pure, first-order and single-assignment, and its graph is acyclic apart from one carried edge per recurrence. Schedules parameterise reduction grouping rather than shared-memory interleaving.

#### 3.1 The three classes

The lattice has three points, the classes, and each is defined by the set of schedule pairs across which a value's bits must agree. The class a value carries is its *lane* (not the vector sense). A lane is this system's analogue of an information-flow label. All of this lives at compile time, and no class is carried or checked at runtime.

- **exact**: the value equals the serial reference oracle's result under the fixed build (one toolchain and ABI). For example, a strict left fold used as the designated reference.
- **replay**: the bits agree across every schedule in  $S$  (§5.1), whatever the worker count or traversal order; the host compiler, vector width and ABI sit outside  $S$ . It is parallel, and the name is the point, because you can rerun the program under a different worker count and the bits replay exactly, which is what an audit needs. For example, a fixed reduction tree whose shape depends only on the input shape.
- **approx**: no constraint. Reorder freely, maximum parallelism, schedule-dependent bits. Its kernel is the *free kernel*, which may reassociate.

**exact does not require correct rounding.** It is a schedule claim, equality with a named reference, and an oracle-order float fold accumulates rounding error like another; it simply accumulates the same error every time. The class asks for agreement across every schedule and equality with the reference, and an error-reducing kernel (Kahan [21], Neumaier [32], a long accumulator [22]) is neither required nor sufficient.

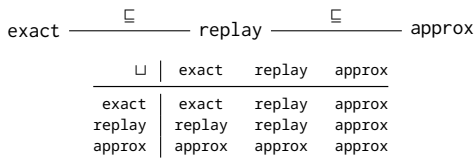


Figure 1: The lattice and its join.

**Why both replay and exact.** Across  $S$  the two classes are equally reproducible, and they differ in who fixes the order. Under replay the backend fixes it, and may fit it to its target provided the schedule never moves it. Under exact the program fixes it as the serial reference, and every backend must reproduce that answer. In a language standard's terms, approx leaves the reduction order unspecified, replay makes it implementation-defined, and exact specifies it. replay is what ships, and exact is the answer a test, a plain loop or another implementation can be held to.

**Why approx is the default.** Defaulting to replay would change every unannotated program's bits on the day the flag flips, and a discipline adopted only by a rewrite does not get adopted.

#### 3.2 The lattice and the join

The guarantees in §3.1 are nested, so their strength determines the ordering in Figure 1. The classes form a chain, and their join is max. Theorem 2 (§5.4) is then one line, because the order has a top. Read  $\sqsubseteq$  as substitutability, meaning a stronger guarantee may flow wherever a weaker one is demanded, and the reverse is refused.

#### 3.3 Where a lane attaches

A lane is demanded with a qualifier, written bare where a value gets its name, and the qualifier has three positions, one line each:

```
replay energy := sum(x)    // on a binding
func f(replay v Float)    // on a parameter
func f() replay Float      // in a return type
```

Three positions are enough because any part of an expression can be named with a binding and demanded through that name. Introducing the binding has no runtime cost, since equal subexpressions already share one node in the interned graph.

A demand adds a graph node and prevents otherwise identical folds with different lane contracts from sharing a computation. That rules out the opening incident. Consider this example:

```
offset := sum(y)
replay total := sum(x * y)
result := total + offset
```

Two scoping rules apply:

1. The qualifier on total does not impose a replay demand on offset in the later binding.
2. The qualifier covers the whole expression defining total, including the multiplication as well as the sum.

The demanded lane in force is the *ambient lane*. The nodes that feed a value, directly or through others, form its *cone*. When demands nest, the inner one governs its own scope.

A lane in a function's return type is a promise about the value it hands back, and the body is checked against the promise rather than covered by it. Folds in the body start free, and an inference pass, which we call healing, strengthens the ones private to the demand (§4.4). Every declared function is lowered and checked on its own, including one without callers, and an argument is checked in the caller's scope against the lane on the parameter it fills.

#### 3.4 The grammar and the typing rules

Figure 2 is the language the checker parses, with a superscript  $?$  marking an optional part, as in  $\ell^?$ , and an overline a sequence, as in  $\bar{a}$ . A call's argument labels and arities are checked later, when names are resolved to nodes. The three lane words and ungrounded are reserved, which stops a variable from shadowing them and a mistyped demand from silently becoming a variable name.

The rules define three judgements. Typing computes each value's lane and checks it wherever a demand is stated. Grounding decides whether the program fixes a value's bytes, and *waiver* whether every leaf it does not fix, and every ungrounded call, carries a blessing. All three read the resolved graph after healing (§4.4), where a bound name stands for the node its binding built.

```

 $\tau ::= \text{Float} \mid \text{Column}[\text{Float}]$ 
 $\ell ::= \text{exact} \mid \text{replay} \mid \text{approx}$ 
 $q ::= \text{ungrounded}^? \ell^?$ 
 $\beta ::= @\text{bless} \mid @\text{bless}(n)$ 
 $op ::= + \mid - \mid * \mid /$ 
 $red ::= \text{sum} \mid \text{isum}$ 
 $scn ::= \text{scan}$ 
 $gen ::= \text{indices} \mid \text{integers}$ 
 $src ::= \text{source}$ 
 $\lambda ::= (x, x) \Rightarrow e \mid (x, x) \Rightarrow \{ \bar{b} \text{ return } e \}$ 
 $a ::= e \mid x : e \mid \text{order} : \text{oracle} \mid \text{with} : \lambda$ 
 $e ::= n \mid x \mid (e) \mid e \text{ op } e \mid \text{red}(\bar{a}) \mid \text{scn}(\bar{a})$ 
 $\mid \text{gen}(\bar{a}) \mid \text{src}(\bar{a}) \mid \text{choose}(\bar{a}) \mid \text{iterate}(\bar{a}) \mid f(\bar{a})$ 
 $b ::= \beta^? \ell^? x := e$ 
 $p ::= \ell^? x \tau$ 
 $d ::= \text{func } f(\bar{p}) \text{ } q \tau \{ \bar{b} \text{ return } e \}$ 
 $P ::= \text{package } x \bar{d}$ 

```

Figure 2: The grammar.

The grammar's operations:

- indices and integers generate a column of given length
- source models an external input, sized by its count
- sum and integer isum reduce a column
- scan gives a column's running sums
- choose is a masked select, reading its mask and both arms
- iterate is a recurrence, repeating its with: step

| program                          | lane            | grounded | waived  |
|----------------------------------|-----------------|----------|---------|
| func main() replay Float {       | promises replay |          |         |
| @bless raw := source(count: 8.0) | exact           | no       | blessed |
| s := sum(raw)                    | replay, healed  | no       | yes     |
| total := sum(raw * s)            | replay, healed  | no       | yes     |
| return total                     | replay, met     | no       | yes     |
| }                                |                 |          |         |

Figure 3: One program, read by the three judgements.

In Figure 3, the return promises replay, so healing strengthens `s` and `total`, free folds private to that promise. The source `raw` is a column the program does not fix, so nothing reading it is grounded, and its blessing takes that obligation on. Demand replay on `total` instead, return `total + s` from an unqualified `main`, and `s` escapes the demand: healing may no longer strengthen it, and that variant is refused with blame at both ends (§4.3). The judgements take four forms, and typing has the first two and T-Fun's **ok**:

$$\Gamma; \ell_a \vdash e : \tau^\ell \quad \Gamma; \ell_a \vdash_{bind} b \dashv \Gamma' \quad \Gamma \vdash e \Downarrow g \quad \Gamma \vdash e \Downarrow_b w$$

Read the first as: under environment  $\Gamma$  and ambient lane  $\ell_a$ , which is  $\emptyset$  where no demand is in force, expression  $e$  has type  $\tau$  and lane  $\ell$ , its computed lane or a weaker one. The binding form extends  $\Gamma$  with the name at that lane. In the others,  $g$  is grounded or not, and  $w$  and a name's subscript  $c$  are waived or not.  $\beta$  marks  $e$ 's node blessed for B-Bless. Healing fixes a healed fold's kernel whatever  $\ell_a$  and  $o$  would pick, and  $\tau$ , one metavariable for both types, is never checked. In Figure 4,  $\ell_d$  is a demanded lane and  $\ell_f$  the lane a return's  $q$  writes; the ungrounded in  $q$  changes no lane. In T-Call,  $\ell_i$  is the lane a parameter declares and  $\ell'_i$  the lane its argument computes.  $o$  is the order dial, absent or oracle, and oracle pins one sum's order and states no demand, and isum refuses it. T-Fun's **ok** says a function's lanes check; acceptance also needs the waiver below.

Figure 4 leaves out the routine rules. A name has the lane its binding gave it; a literal, a generator and a source are exact; arithmetic

$$\begin{array}{c}
\frac{\Gamma; \ell_d \vdash e : \tau^\ell \quad \ell \sqsubseteq \ell_d \quad \Gamma \vdash e \Downarrow_b w \quad c = w}{\Gamma; \ell_a \vdash_{bind} \beta^? \ell_d x := e \dashv \Gamma, x : \tau_c^\ell} \text{T-BIND} \\
\\
\frac{\Gamma; \ell_a \vdash e : \tau^\ell \quad \Gamma \vdash e \Downarrow_b w \quad c = w}{\Gamma; \ell_a \vdash_{bind} \beta^? x := e \dashv \Gamma, x : \tau_c^\ell} \text{T-BINDFREE} \\
\\
\frac{\Gamma; \ell_a \vdash e : \tau^\ell \quad \kappa = \text{kernel}(\text{red}, \ell_a, o) \quad \ell_\kappa = \text{class}(\kappa)}{\Gamma; \ell_a \vdash \text{red}(e, (\text{order} : o)^?) : \tau^{\kappa \sqcup \ell}} \text{T-FOLD} \\
\\
\frac{\Gamma; \ell_a \vdash e : \tau^\ell \quad \kappa = \text{kernel}(\text{scn}, \ell_a) \quad \ell_\kappa = \text{class}(\kappa)}{\Gamma; \ell_a \vdash \text{scn}(e) : \tau^{\kappa \sqcup \ell}} \text{T-SCAN} \\
\\
\frac{\Gamma_0 = x_i : \tau_i^{\text{exact}} \quad (i = 1 \dots k) \quad \Gamma_{j-1}; \emptyset \vdash_{bind} b_j \dashv \Gamma_j \quad (j = 1 \dots m)}{\Gamma_m; \emptyset \vdash e : \tau^\ell \quad \ell \sqsubseteq \ell_f \text{ wherever } \ell_f \text{ is written}} \text{T-FUN} \\
\\
\frac{\begin{array}{c} f \text{ declares } \ell_i^? x_i : \tau_i \quad (i = 1 \dots k) \text{ and returns } q \tau \\ \Gamma; \ell_a \vdash a_i : \tau_i^{\ell'_i} \quad \ell'_i \sqsubseteq \ell_i \text{ wherever } \ell_i \text{ is written} \\ \Gamma; \ell_a \vdash \text{inline}(f, a_1 \dots a_k) : \tau^\ell \quad \ell \sqsubseteq \ell_f \text{ wherever } \ell_f \text{ is written} \end{array}}{\Gamma; \ell_a \vdash f(a_1 \dots a_k) : \tau^\ell} \text{T-CALL}
\end{array}$$

Figure 4: The binding, kernel and boundary rules.

and choose join their operands' lanes; and subsumption, printed in §4.1, lets any lane weaken along the order.

T-Bind checks a binding's demand. It types the right-hand side under the demanded lane, which becomes the ambient one, and binds the name at the computed lane, not the demanded one. The same rule without the comparison is T-BindFree, which keeps the ambient lane it inherits. A blessing is allowed only on a source or on a call to an ungrounded function (§4.6). It marks the node, not the name, so every name for that node reads waived, even one bound before it.

T-Fold is where the schedule enters. The ambient lane and the order dial pick which kernel the reduction lowers to, and the kernel's row in Table 2 supplies the class, so a demand can change which kernel is chosen but never a kernel's class. Because join is max, an approx fold overrides its inputs, while a replay fold leaves the result no stronger than replay. T-Scan is the same rule for a scan, which refuses the dial.

iterate joins like arithmetic, its lane a least fixed point over the carried edge.

**T-Fun's body is typed under no ambient lane**, so its folds start free and are compared against the declared lane. The parameters enter the environment at exact, the bottom of the chain, because a parameter's lane, like its grounding, is the caller's obligation. T-Call checks each declared parameter lane at the call. Its inline premise retypes the body under the caller's ambient lane and its arguments' lanes, so a promise the body keeps only for stronger arguments is refused at the call that supplies weaker ones. Outside both rules, a returned binding whose own qualifier is weaker than the declared lane is refused as malformed.

A leaf is grounded exactly when the program fixes its value. Any other node is grounded exactly when everything it reads is, the carried edge included, except an ungrounded call, which never is. Waiver adds one case: a blessed source, or a blessed ungrounded call over a waived body, counts as waived. An exact or replay demand, on a binding, a parameter or a return, is accepted only if its lane comparison holds and its value is waived.

**Table 2: The operation-class table: each kernel’s class, decided by what its output bits depend on.**

| Kernel                                  | Its bits are a function of                                                                                                                                                                                                                                 | Class                                 | Witnessing check                   |
|-----------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------|------------------------------------|
| elementwise arithmetic                  | nothing of its own: no reassociable combination, and the host compiler sits outside S (§5.1)                                                                                                                                                               | neutral                               | L2 (combination half)              |
| data movement, literals external source | nothing of its own: no combination at all the world outside the program, not the schedule; a demand over it is refused by the grounding judgement rather than weakened                                                                                     | neutral<br>neutral, <b>ungrounded</b> | L2<br>L3, CC1, L2                  |
| carried-state recurrence                | nothing of its own: strictly sequential, so it has exactly one schedule                                                                                                                                                                                    | neutral                               | L3, L4, L2                         |
| fixed-tree reduction                    | (data, shape)                                                                                                                                                                                                                                              | replay                                | L4, L2, CC2                        |
| fixed-block scan                        | (data, shape): each block scans serially and a serial base pass combines the totals                                                                                                                                                                        | replay                                | L3, L4, CC3, L2                    |
| oracle-order strict fold                | data alone, <b>and</b> equal to the serial reference by construction                                                                                                                                                                                       | exact                                 | L4 (reference comparison), CC4, L2 |
| exact-integer reduction                 | data alone, <b>and</b> equal to the serial reference: while the column holds integers whose absolute values sum below $2^{53}$ , addition is exact, so the free order and the serial order agree; outside that range, evaluation fails instead of rounding | exact                                 | L3, L4, L2 (chunking)              |
| free reduction, free scan               | the schedule: licensed to reassociate                                                                                                                                                                                                                      | approx                                | L4, L2                             |
| parallel scatter                        | <b>UNCHECKED</b> : no parallel scatter exists here; named again in §5.5                                                                                                                                                                                    | approx                                | none                               |
| transcendental functions                | <b>UNCHECKED</b> : lanelang names no maths library                                                                                                                                                                                                         | approx                                | none                               |

### 3.5 How each operation’s class is decided

So far, every rule moves classes around, and none of them decides one. A class is decided once per kernel, before any program exists, by one question: what are the kernel’s output bits a function of? If the schedule is anywhere in the answer, the class is approx. If it is not, the class is replay, or exact when the bits also equal the serial reference under the fixed build. A kernel that combines nothing is written *neutral*, and its lane is the join’s identity, exact, so it contributes nothing.

Table 2 applies this criterion to the demonstration language’s kernels in eleven rows, and other backends may supply more.

The kernel’s class is the strongest guarantee its result can carry. Its inputs may weaken the result’s guarantee, but cannot strengthen it beyond the kernel’s class. The typing rules use these classifications, written `class(kernel)`, rather than derive them. When an operation has several kernels, the lane demand and the order dial determine which kernel is selected. The selected kernel’s class is used in the join.

Why trust these classifications? The checker assumes each kernel satisfies its assigned class; §5 states the supporting assumptions.

## 4 Propagation, checking and blame

The site where a demand is written cannot settle it, so the checker brings the history to the site, folding everything upstream into one class that rides the value, and the declaration then needs only a local comparison.

### 4.1 The algorithm

The checker is a single forward pass over the graph, in dependency order, computing at each node:

$$\text{computed}(v) = \text{join}(\text{class}(v), \text{computed}(in_1), \dots, \text{computed}(in_k))$$

A recurrence’s carried value joins as one more input. At every node that states a demand, `computed(v)` must lie at or below `declared(v)` in the lattice’s order, or it is an error; nodes are constructed in dependency order, which is what lets one left-to-right sweep suffice.

**Except across a recurrence.** A recurrence sequentially carries its operand state across iterations, which a single sweep cannot track. The sweep therefore runs to a fixed point, repeating until

a full pass changes nothing. When a change must cross  $d$  carried edges, the sweep settles within  $d + 2$  passes, one to climb,  $d$  for the news to cross them, one to validate it is stable.

**Lemma 1: the pass computes the declarative type.** The typing rules say which programs should be accepted. The checker is what actually decides, and if the two could disagree, the proofs would certify a checker nobody ships. Lemma 1 says its sweep cannot: *for every node  $e$  of the healed graph,  $\text{computed}(e)$  is the least class the join and T-Sub can derive for it.* The statement has one half per verdict. Because the sweep’s answer is always derivable, every acceptance it grants is one the rules grant. Because it is the least derivable, every rejection it issues is one the rules issue.

$$\frac{\Gamma; \ell_a \vdash e : \tau^\ell \quad \ell \sqsubseteq \ell'}{\Gamma; \ell_a \vdash e : \tau^{\ell'}} \text{ T-SUB}$$

T-Sub lets a derivation answer any class above the join, so a sweep that returned anything higher would reject a program the rules accept. Hence the checker reaches the lane verdict the rules derive at every binding of the healed graph. Healing runs before the check, leaving Lemma 1 and §5.3 quantifying over the same graph. Both halves are stated over the acyclic fragment, and a fixed-point form of the lemma covers recurrence against the join alone.

### 4.2 Acceptance without demands

The checker enforces only what a programmer wrote.

*A demand-only checker rejects a program only for lane demands the program states, so one with none is never rejected.*

This follows from checking each typing rule in turn. A program stating no demand writes no qualifier, and every binding in it is then typed by T-BindFree, which states no at-or-below condition. Every rule that compares a computed lane against a demanded one applies only where a qualifier is written.

Accepting an unannotated corpus therefore says nothing about the checker’s ability to detect violations. Existing programs state no demands because the syntax did not exist when they were written. This is why §7 builds its evidence around things that can fail.

### 4.3 Blame

On violation, the blame rule walks back along the input, carrying the weakest lane, until it reaches a node whose class matches that lane. That node introduced the weakness itself rather than inheriting it, so it is provably a culprit, though not provably the nearest one. The checker reports both ends and the distance between them. Reporting only the demand site would invite the wrong repair, which would be a weakened demand.

Figure 5 shows the diagnostic. It fires on the program below, shown without its licence header and fixture directive; the figure counts lines of the file itself, header included, which is why the demand sits at line 21:

```
package app
func main() Float {
  x := indices(count: 100000.0)
  approx pre := sum(x * 0.25)
  replay demanded := sum(x + pre)
  return demanded
}
```

### 4.4 Healing

Introducing a name should not change a program's determinism. Here is one program, written two ways:

```
named: one producer healed
s := sum(x)
replay e := sum(x*s)
```

```
inline: nothing to heal
replay e := sum(x * sum(x))
```

Same program, same bits, one binding apart, and without healing the first form is rejected while the second is accepted. A control build without the healing pass must refuse the first form.

Making the two forms agree is healing's job. A declaration will often demand more than the producers above it promise, because upstream code carries no annotations for it. The checker could reject and make the programmer annotate every producer by hand. Instead it *heals*.

The checker can heal a producer only if it is *private to the demand*, with no live read-path out of the demanded cone. It rewrites the producer's class up to what the demand needs, as if the demand had been written at the source. The pass is ordinary annotation inference, and the privacy condition is what keeps it sound: pinning a private producer changes nothing an outsider can see. Healing never overrides a lane that a binding states, so a written `approx` stays free, and that is why §4.3's program is refused. If you drop the `approx` qualifier, the producer heals.

**Whether a producer is private is decided over every path that reads it, and its direct consumers alone cannot decide it.** One fixture, entire but for its licence header and directive:

```
package app
func main() Float {
  x := indices(count: 100000.0)
  partial := sum(x * 0.5)
  y := partial + 1.0
  replay demanded := sum(x * y)
  return demanded + y * 0.0
}
```

```
reject_poison.lane:21:5: error: determinism lane violation: this value
computes approx where replay is demanded. An unpinned schedule cannot
be recovered downstream: the rounding already happened. Pin the producer
by naming it with a replay or exact qualifier, or weaken the demand
21 |   replay demanded := sum(x + pre)
    |   ^^^^^ the demand is stated here
... the guarantee was lost 2 steps upstream:
20 |   approx pre := sum(x * 0.25)
    |   ~~~ introduced here
```

Figure 5: Blame from the checker, its note omitted.

partial has one direct consumer, `y`, and `y` feeds the demand, so a test on direct consumers calls `partial` private and heals it. But the return also reads `y`, so pinning `partial` would change bits an outsider sees, and the checker refuses with blame on `partial`. The checker follows every use of the producer, stopping at the demanded value. Changes observed through that value are allowed by the demand. The fixture is therefore refused.

**A demand's reach is decided by dataflow, not by where the text sits, and healing is what makes that true.** The qualifier itself covers only its right-hand side. Healing reaches the named producers above it.

A declared return lane covers nothing at all, a promise checked against the body rather than an ambient over it, so where no other lane is in force, a fold reaches the fixed tree by healing or not at all. A return demand heals the same way, through a node of its own that wraps the returned expression, and the body's producers are private to that node when nothing outside the body reads them.

### 4.5 Grounding

A column read from a file can carry replay as legitimately as the literal `3.0`, because reordering is not its problem, and that is the danger. Whether the file holds the same bytes next run is a fact about a deployment, not about arithmetic, and the lane has no way to ask. One fixture:

```
package app
func main() Float {
  x := source(count: 1000.0)
  replay energy := sum(x * 2.0)
  return energy
}
```

Everything the lane can see is faultless, and yet the fixture is refused, because the program has no way to know what bytes `x` will hold next run. *Grounding* is the judgement that refuses it. It asks whether the program fixes a value's bytes at all, and a value the program does not fix is *ungrounded*. Everything that enters from outside the program enters that way, because no rule inside the language can settle its bytes.

**grounded is one bit per node, and it is a conjunction where the lane is a join.** A leaf is grounded when the program fixes its value. Literals and generated columns are; an external source is not. Every other node is grounded exactly when all its reads are, including the carried edge, except a call to a function whose return type carries the ungrounded qualifier, which never is.

There are three ways out of the refusal: vouch for the input with a blessing, take it as a parameter so the obligation becomes the caller's, or weaken the demand to `approx`.

## 4.6 Blessing

Grounding’s refusal is right about the world and unfair to a programmer who knows their input is fixed. The lane over such a value is genuinely replay, because a fixed-tree fold is a fixed-tree fold whatever the bytes were, and yet the only spelling that compiles, approx, is a licence to reorder the programmer never meant to grant.

@bless closes that gap without weakening anything. It attaches to a binding whose value is a source or an ungrounded call, and it does *not* make that value grounded. What the blessing adds is a second way past grounding at the declaration: a demand whose every source and ungrounded call in its cone is blessed is waived. The refused fixture of §4.5, blessed with a checksum:

```
package app
func main() Float {
  @bless(0x1234abcd) x := source(count: 1000.0)
  replay energy := sum(x * 2.0)
  return energy
}
```

The fixture is now accepted. A blessing is written on a binding and covers the value that binding names, and no other.

**The conditional form matters, because it turns a claim into a receipt.** A bare @bless is an unverified human promise. A conditional @bless(...) states what must hold of the input, and the run refuses before any arithmetic over that column when it does not, which turns the promise into something a machine can check. Which conditions a language offers is the implementer’s choice; this artefact ships one, a checksum over the loaded column.

## 5 Determinism safety

So far a lane is a label the checker moves around, and no result says the label means anything about bits. Section 5 closes that gap: it defines which runs must agree (the semantics), states the two facts about arithmetic the proofs take on trust (the assumptions), and proves that an accepted program keeps its promise (both theorems). Reproducibility is a claim about *pairs* of runs, so the proofs walk two runs side by side (Figure 6).

| program                       | 2 workers  | 8 workers  | promise           |
|-------------------------------|------------|------------|-------------------|
| x := indices(count: 100000.0) | generated  | generated  | agree exact       |
| a := sum(x)                   | 2 chunks   | 8 chunks   | may differ approx |
| replay b := sum(x)            | fixed tree | fixed tree | agree replay      |
| return a - b                  | reads a, b | reads a, b | may differ approx |

**Figure 6: Both theorems on the opening program. The worker count picks how a free sum is chunked, and a fixed tree depends on the input’s shape alone. a - b may differ because it reads a, which lies outside b’s demand, so nothing heals it.**

### 5.1 The schedule-indexed semantics

A promise that bits will not move is empty until you say what may move. We draw that line where the artefact can witness it, and a

source of variation is inside the promise only if a check varies it and watches the bits.

Given that standard, let  $S$  be the checked schedule set: every combination of worker count and traversal order, the two settings the evaluation varies. A pair from  $S$  might be two workers under the fused traversal against eight workers under 7-wide tiles.  $S$  excludes exactly three things: the host compiler, the vector width, and the ABI version.

Then add one schedule no deployment ever runs, *oracle*, the serial reference fold under the fixed build, and write  $S^+$  for the enlarged set. By definition, *oracle* counts as its own schedule, even if some deployment schedule happens to compute the same bits. At *oracle* every combining kernel is replaced by the serial fold, whatever its class, and that replacement is the line between replay and exact, because a fixed tree agrees across  $S$  and may still differ at *oracle*. Write  $\text{bits}(e, s)$  for the bit pattern  $e$  computes under a schedule  $s$  from  $S^+$ , with the same inputs for every schedule.

We do not say how  $\text{bits}$  is computed. The proof treats it as a black box, and Assumption 1 is the only fact about the arithmetic it may use. Theorem 1 is therefore *parametric*: it holds for any arithmetic that satisfies Assumption 1, so its content is the lattice argument, not a fact about floating point. Such a theorem would be about nothing if no real arithmetic satisfied its assumption. A model evaluator, the artefact’s kernels restated in Coq, is the witness that one does, in a separate development with an audit of its own.

Each class names the set of schedule pairs its bits must agree across, written agrees:

$\text{agrees}(\text{exact})$  is every pair from  $S^+$ ,  $\text{agrees}(\text{replay})$  every pair from  $S$ , and  $\text{agrees}(\text{approx})$  is empty.

Each set strictly contains the next, and that strict containment is what puts exact at the bottom: the bigger a class’s agreement set, the stronger its promise, and the strongest sits lowest.

### 5.2 The two assumptions

The proofs take two facts about the arithmetic on trust, one over schedules and one over worlds.

**Assumption 1.** Each kernel honours its assigned class. With identical operands, its output agrees across the schedule pairs that class covers. For an operation  $o$ ,  $\text{class}(o)$  is the class of its selected kernel.

*Take any operation  $o$  and any schedule pair in  $\text{agrees}(\text{class}(o))$ . If the operands of  $o$  agree bitwise across the pair, the bits of  $o$  agree too.*

A recurrence’s carried value counts as an operand, and an approx operation covers no pair, so it owes nothing.

**Assumption 2.** The world has one door into a program, the external source, and every other operation computes from its operands and nothing else. It is Assumption 1 the other way round, with the schedule held fixed and the input bytes varied.

*Take any operation  $o$  other than an external source, and any pair of worlds at one schedule, a world being one assignment of bytes to every external input. If the operands of  $o$  agree bitwise across the pair, the bits of  $o$  agree too.*

Grounding relies on it, and it assumes nothing about the schedules or the rounding.

### 5.3 Theorem 1 (schedule agreement)

An accepted demand delivers the determinism it asks for. Run an accepted program twice, under any two schedules its demand covers, and the demanded bits agree. For a node  $u$  in the computation graph, the accepted-demand result carries these conditions:

- (1) *the program is well formed, in the sense that every operand of a node, its carried value aside, is an earlier node;*
- (2) *every operation in the program satisfies Assumption 1;*
- (3) *each node’s lane is  $\text{computed}(u)$ , the join of its own class with its inputs’, a recurrence’s carried value among them;*
- (4) *choose any schedule pair  $(s, s')$  for which:*
  - (a) *“carried agreement” holds: at every recurrence whose own lane demands agreement over  $(s, s')$ , the carried value the two runs read agrees;*
  - (b)  *$u$  is a node whose declared lane  $\ell_d$  covers the pair, meaning  $(s, s')$  lies in  $\text{agrees}(\ell_d)$ ;*
- (5)  *$u$  passes its lane check at  $\ell_d$ :  $\text{computed}(u)$  is at least as strong as the declared demand.*

Then the demanded bits of  $u$  agree across the pair.

Carried agreement is a separate premise because the carried edge can point forward, beyond what the graph induction reaches. The C evaluator runs recurrence steps strictly in index order, so the carried value a tick reads was settled the tick before. That meets premise (4a), but outside the mechanised proof, because the model does not represent those steps.

Figure 6 applies the theorem to the opening program at two workers and at eight. The pair of schedules lies in  $\text{agrees}(\text{replay})$ , so the theorem covers b, and the difference the return computes is exactly what the incident’s merge once destroyed.

**The result is the sentence a programmer relies on: declare a lane, be accepted, and the bits agree across every pair that lane covers.** Coq proves agreement over every pair the computed lane covers, and with condition (5) over every pair the declared lane covers, because a computed lane at least as strong as  $\ell_d$  covers every pair  $\ell_d$  covers.

**Adding a demand can change arithmetic order and therefore the result’s bits, by design.** The theorem applies to the healed graph. Healing can select stronger kernels for upstream producers, changing their arithmetic order and potentially their result bits.

The theorem makes a replay or exact value agree with itself across runs, while approx promises nothing, so while worker count and data delivery may differ, the demanded bits do not.

### 5.4 Theorem 2 (the poisoning law)

Once a value has reached approx, nothing downstream restores the guarantee. However strict the later operation, the bits it reads were already made under a free schedule, and no kernel recovers what a different order would have produced.

If  $\text{computed}(u) = \text{approx}$  and  $u$  is an input of  $v$ , then  $\text{computed}(v) = \text{approx}$ , whatever the operation class of  $v$ . The proof is one line: join is max on the chain and approx is its top.

**Consequence.** No per-call API is sufficient. A check at  $v$ ’s call site inspects  $v$ ’s operation and its operand values, neither of which records  $u$ ’s schedule history.

### 5.5 What is proved, and what is not

Fourteen results are mechanised in Coq: seven from nothing at all, and the other seven from only the eight axioms that frame the semantics, the schedule set, its oracle, worlds and bits. Proved:

- both theorems, and the consequence for declarations;
- Lemma 1 in both directions, its fixed-point form, a bound of twice the node count on iteration from the bottom;
- the chain’s two strict containments;
- five results about grounding, chief among them that a waived value’s bits agree across two worlds that agree on its blessed inputs, at one schedule.

Not proved: Assumptions 1 and 2 themselves, or carried agreement, across schedules (premise (4a)) or across worlds; that debt remains. Nine of Table 2’s eleven rows are proved of the model’s kernels. The two remaining, parallel scatter and transcendentals, are unchecked.

## 6 The lowering obligations

An accepted program is a promise the backend must keep, and the backend owes each class something different.

### 6.1 Where the type system’s responsibility ends

A checker can prove which kernel an operation lowers to. It cannot prove what that kernel does with the schedule, an arithmetic fact typing cannot reach. Treating the two as one obligation lets a determinism guarantee rest on a claim no check tested. In the artefact, two obligations are discharged by the evaluator’s kernels and one by its checksum guard. Table 3 lists the backend obligations alongside this artefact’s mechanisms, checks and remaining gaps. Every row but the last is imposed by a class; the last is owed by the human who wrote the blessing (§4.6).

**Table 3: Lowering obligations: what is owed, what discharges it, and what witnesses it.**

| Obligation                       | Lanes                   | Mechanism                                                                                                                  | Gate             | Status  |
|----------------------------------|-------------------------|----------------------------------------------------------------------------------------------------------------------------|------------------|---------|
| fixed reduction tree, block scan | replay                  | the evaluator’s fixed-tree kernel: blocks, then block results, folded strict left to right; scans in frozen 64-wide blocks | L4, L2, CC2, CC3 | checked |
| oracle-order fold, integer guard | exact                   | the evaluator’s serial strict fold; <code>isum</code> refuses at $2^{23}$                                                  | L4, L2, CC4      | checked |
| contraction (fused multiply-add) | replay, exact           | the build’s <code>-std=c11</code> , under which this toolchain does not contract; no gate reads the emitted code           | none             | unbuilt |
| disabled                         |                         | none: the only scatter here is serial                                                                                      | none             | unbuilt |
| canonical parallel scatter       | replay                  | none                                                                                                                       | none             | unbuilt |
| fixed math library               | replay across platforms | not pinned                                                                                                                 | none             | unbuilt |
| FTZ/DAZ and rounding mode        | all                     | the evaluator’s checksum guard, refusing the run before any arithmetic over the column                                     | L5               | checked |
| blessed input unchanged          | any                     |                                                                                                                            |                  |         |

## 6.2 The operation picks the kernel

**A result’s computed lane does not change the kernel that produces it.** The ambient lane, the order dial and healing pick the kernel before the checker runs. Lanes are computed afterwards for the checker; they do not feed back into kernel selection. If a computed lane could feed the choice, one class would denote two bit patterns depending on what was asked of it, which inverts the guarantee. This means that even when an input causes the result’s lane to be approx, the operation keeps the arithmetic order it was given.

In short, a sum can request a particular arithmetic order without demanding a stronger lane.

## 6.3 What a lane costs the optimiser

Lane demands constrain which optimisations are allowed. After a transformation, the checker recomputes the lanes and verifies that every remaining demand is satisfied.

Map-into-map fusion, tiling and dependence-preserving reordering are allowed when they preserve each element’s arithmetic and any required reduction grouping. Fusion needs additional care where a fold receives the fused work: parallelising a map and reduction as one loop can change how partial sums are formed and combined. Fusion itself is allowed if the required grouping stays fixed. Six programs, each run at five traversals and four worker counts, check that preservation: a value under a demand holds one bit pattern across all twenty cells, and a value under no demand holds up to four.

**For replay, reduction splitting must preserve a fixed arithmetic grouping.** Halide’s `rfactor` splits an associative reduction into partials under an associativity proved in the reals [40], and floating-point bits do not obey the reals. The control makes the grouping depend on tile size by accumulating a partial sum per tile; the schedule check catches changes in results that were required to stay identical.

**The class must also survive value numbering, and we learnt that from an incident.** The incident is §1’s four-line program. Its two reductions over the same column, one demanded replay and one not, are structurally identical and *must not* merge, because they have different lowering contracts. The compiler this work grew from interned nodes *before* lanes were stamped, so the twins merged and were re-stamped with the stronger lane, silently pinning a cone the program did not ask to pin. We now put the contract in the interning key; control CC5 tests the fix. A merge that changes no value can still lose a schedule contract. Spark states the discipline for source nondeterminism. We found none stating it for the floating-point schedule class.

**What adoption cost one compiler is reported, and a reader cannot rerun it.** In the production compiler the incident comes from, adoption touched an annotation on the syntax tree, a module-wide ambient lane, one integer on the IR node, three fields in the value-numbering key, one join pass, one heal pass, and a registry of sixty-nine rows. It changed nothing in that compiler’s type checker, and it found eight lane defects, two of them unsound acceptances.

Optimisation can change how work is scheduled, but it must preserve the arithmetic grouping required by a demand and keep distinct lowering contracts separate.

## 7 Evaluation

**What is evaluated, and by which figure.** The checker agrees with an exhaustive path enumeration on 58,245 node answers over random graphs and 47,738 over front-end-built ones, with zero mismatches, and 8,000 metamorphic assertions over three properties check that enforcement is monotone and mutates nothing. The evidence is then audited, through the ten occasions on which a passing check had proved nothing and the nineteen must-fail controls, each watched as it fires. Two experiments close the section, the composition gap inside NVIDIA’s own toolchain and the opening failure replayed with its repair beside it.

The assertion counts are machine-independent because every generator is a pure function of (seed, n, i). The evaluator itself is single-threaded, and a worker count selects the chunking, so every schedule in S is exercised as the bracketing it induces, never by concurrent execution. Reproduce with `./reproduce.sh` and `./controls.sh`; every check and its command is tabulated in the artefact (Appendix A).

Because the lattice runs at compile time, a demand’s execution cost comes from the kernel it selects. This single-threaded evaluator cannot measure that parallel cost. For context, vendor figures report overheads of 15 percent or more with LS-DYNA’s serial assembly [25] and 20 to 30 percent for NVIDIA’s cross-GPU reproducibility on large inputs [3]. These are external measurements, not timings of this artefact.

A *check* is one of the artefact’s numbered test programs, and a *control* is the deliberate fault of mutation testing [15], staged for a check to catch. A *receipt* is the status line a check prints, which is what a claim cites as its witness. But a check is another claim about the code, and a claim can rot like any other. Three things stop checks from needing checks of their own. Every control must fire, and a control that does not means its check is decorative and the run fails. §7.2 asks of every passing check what it could have caught. Finally, two pieces of evidence are not check-shaped at all, namely the exhaustive path enumeration in §7.1 and the mechanised proof in §5.5.

### 7.1 Testing the checker’s lane calculations

**How do we know whether the checker assigns the right lane to every value?** We test this by generating programs and comparing the default fast join algorithm with an independent calculation that follows every dependency path. We then check how its answers behave when demands change.

The two calculations work differently:

- **The default checker** calculates a node’s lane from its kernel class and the lanes already calculated for its inputs. It reuses those answers as it moves through the graph.
- **The exhaustive reference** traces every route from the original inputs to a node and takes the weakest class found along those routes. It follows each route separately instead of reusing an intermediate node’s calculated lane.

This differential test [27] found zero mismatches: L7 compared 58,245 node answers over 2,000 random graphs of 8 to 50 nodes. L6 also generated well-typed source programs, parsed, resolved and healed them, and compared 47,738 node answers with zero mismatches.

The methods share two primitives, so they test dependency handling rather than independently validate the join itself. Generated programs contain no function declarations or recurrences; those remain covered by fixtures.

Metamorphic tests [12] add 8,000 assertions: weakening all demands to approx yields no violations and never turns acceptance into rejection (M1); strengthening replay demands to exact never decreases the violation count (M2); and checking twice leaves the verdict and graph unchanged (M3).

## 7.2 When passing checks missed defects

In place of a no-regressions figure we report the opposite one: every occasion on which a check of ours passed although it could never have failed on the defect it was written to catch. There are ten, found across the twelve checks the artefact runs and the measurements and controls behind them. Four bear directly on the type system and follow; the rest are catalogued with the artefact (Appendix A).

**1. The invariance check measured the wrong graph.** §5.3 constrains the *healed* graph, and the check evaluated the graph as written. It went unseen while every program it ran annotated the fold directly, where no producer needs healing. The first program that needed a heal reported four bit patterns where the accepted program has one.

**2. A value test blind to a merge.** §6.3’s incident is two structurally identical reductions with different contracts collapsing into one, and no assertion over values can see it. The check needs a fact about the merge, and the check that catches it asserts over the emitted kernels instead.

**3. Every schedule-varying check asked whether a value agreed with itself.** Varying the schedule and comparing bits is a question a deterministic and consistently wrong kernel answers correctly. No check in the suite compared a value to the serial reference that exact names, which is the content of that class. L4’s reference comparison is the repair, and control CC4 is a build that discharges one lane with two different bit patterns while leaving every invariance assertion green.

**4. The round-trip check could not see the healing it names.** L6 heals each generated program, then compares every node’s lane against the path enumeration. Both sides read the healed graph, and deleting the pass changed no verdict. The repair also reads the lanes before the heal, and requires the generated programs to move some: 371 producers strengthened, and 2,809 of the 47,738 answers changed.

All ten were green when found, and none was found by a test failing.

## 7.3 The checker’s must-fail controls

The controls are nineteen in all, numbered CC1 to CC19. Seventeen break a build of the checker, the evaluator, or the mechanised proof. CC6 and CC8, a malformed program and the kill switch, an environment variable that disables the checker, assert directly on the shipped binary. CC17, for example, switches healing off, and that build must then reject the fixture `accept_healed`, while the normal build accepts.

## 7.4 Testing the limits of per-call guarantees

A per-call determinism guarantee may not cover the computations that produced its inputs. We test this composition boundary using NVIDIA’s CCCL: can a downstream call enforce agreement when its input comes from an unguaranteed reduction? Check L1 runs the experiment with the vendor’s compiler and headers.

Three programs are compiled with `nvcc` against the CCCL headers. Two test requests the library must reject locally. One combines an unsatisfiable determinism level, operator and element type; the other passes a determinism property without require. Both fail to compile, each with its own named diagnostic. The third takes the result of a `not_guaranteed` reduction and feeds it to a `run_to_run` one. It compiles without a word. The second reduction runs over one element, which no schedule can reorder. The case therefore rests entirely on what the result carries, and it carries nothing for the second call to check. The two rejections are what make the third case mean anything.

A LaneLang fixture tests the same dependency pattern. A free fold is passed to a parameter declared `replay` and is also read elsewhere by the caller, so healing cannot strengthen it. The checker refuses the call with blame at both ends. LaneLang therefore rejects this dependency where CCCL accepts it.

## 7.5 The opening case, replayed

The paper opened on 10,179 rerun datasets, and 31% changed trees. The authors’ stated mechanism is that parallel reductions re-associate as a function of the core count, and the drifted likelihoods then reach the comparisons that steer the search [39]. A drift in a low digit matters most once a branch reads it.

Check L4 replays that shape at this artefact’s scale, as a free fold feeding a recorded-value test:

```
package app
func main() Float {
  x := indices(count: 100000.0)
  approx s := sum(x * 0.5)
  picked := choose(when: s - 0.0042511812704049134,
    then: x * 0.0 + 3.0,
    else: x * 0.0 + 5.0)
  total := isum(picked)
  return total
}
```

The constant records the sum from a serial run, making this a regression test against a golden value. `indices` generates ill-conditioned cancellation pairs from a fixed seed, not 0, 1, 2. A non-zero mask picks then; zero picks else.

Across the twenty schedule cells, the free version takes both branches: every worker count but one differs from the recorded sum, in the eighth significant digit. Making the producer replay or exact gives a stable answer across all twenty schedules. The exact repair matches the recorded serial result and takes `else`; the replay repair consistently takes `then`, illustrating that repeatability does not require the serial answer.

Demand replay of the result while the mask stays free, and the checker refuses with two-span blame naming the fold (a refused fixture of L3). The regression test that would have broken under a different worker count is instead refused before it ever runs.

## 8 Limitations

One limit is the design’s reach as shipped: Table 2 leaves transcendental at approx, so no fold fed by an exp could demand replay. Every other limit here is a limit of the evidence, not of the lattice. *Each receipt is one machine, one toolchain*, so no claim is made across devices, GPU vendors or CPU ISAs; the evaluator’s kernels are the only backend anything here witnesses.

In addition, FTZ/DAZ (flush-to-zero and denormals-are-zero) and the rounding mode are not pinned, although an exact replay must run under the same flags, and no check covers them.

A demand over external data rests on a human (§4.6); an input that varies by nature, a live stream say, stays ungrounded.

Determinism safety is conditional on Assumption 1’s operation-class table and the fixed build; §5.5 states what is unproved.

The language is a demonstration, not a production compiler: pure, first-order, non-recursive, no transcendental, one numeric type, and no type checking over its two types, `Float` and `Column[Float]`.

## 9 Related work

Table 1 (§2) surveys the mechanisms that govern reordering.

**Determinism is already checked at plan time.** Flink, with its update-strategy check enabled, can reject a query whose column “can not satisfy the determinism requirement” [42], and Spark’s Catalyst computes a per-expression deterministic property as a meet over its children, whose `SemanticEquals` refuses to merge non-deterministic twins: §6.3’s lesson, already shipping [45]. Spark also grades each dataset determinate, unordered or indeterminate by whether a rerun returns the same rows, taking the least deterministic of its dependencies [44]. Determinism checking with blame over a dataflow plan is *not new*. Each of these grades records or source nondeterminism, imposed by the engine, and none grades the bits of arithmetic, as an open Spark report illustrates, in which a float sum differs by one bit on a retry and the retried stage duplicates and drops rows [43].

**Graded determinism as a checked type.** Mercury has had compiler-checked determinism declarations since the 1990s [38], and the graded systems after it each grade a different subject [10, 14, 17, 23, 30, 31, 37], while graded comonads and coeffects [35] grade a variable’s *use*, where this work grades a value’s whole *history*, joined over the dependency graph. The machinery is claimed from none of them, and the lattice frame itself is Denning’s [16]. The nearest floating-point analysis, Gu and Wahl’s, bounds per value what any compliant compiler could compute on a test input and proposes a fix, where this work checks a declaration [18].

The discharge into lowering is not novel either: EnerJ lowers its qualifiers into approximate storage and functional units [37]. Nor is a three-level vocabulary ours, since Veit’s kernel contracts publish `BITWISE | RUN_TO_RUN | NONE` for ML kernels [53], declared per kernel and checked by repeated invocation on one machine. In contrast, a lane is carried on a value. That work reaches composition and names it as future work in its own voice: what two contracted kernels compose to is, in general, not directly the conjunction of their contracts.

**Properties on array types.** SaC’s type patterns carry shape constraints per signature, re-established at every application and never carried on the value [1].

**Licences inside a value grammar.** Icing comes nearest, putting the optimisation licence *inside the value’s* grammar [8]; verified compilation of floating-point programs continues that line [9]. Icing is strictly sequential, quantifying over which compiler rewrites fired, whereas we quantify over which runtime schedule ran.

**Fixing the order.** Fixed-shape summation appeared independently at least four times between 1964 and 1970 [6, 24, 26, 36], and it is still being improved, from numpy’s partial pairwise summation, applied conditionally [51], to oneTBB’s deterministic reduce [52] to ReproRed [39]. Sixty years of implementations pin the order; what none of them publishes is the checked contract.

**Not fixing the order.** Where the family above pins a schedule, Demmel and Nguyen’s reproducible summation and ReproBLAS make the sum order-*independent* so that no schedule can disagree [2]. Although it looks like an exact mechanism, it is an alternative discharge of replay: a function of the data alone that can still differ from the serial fold’s answer.

**The nearest structural precedent, in security rather than numerics.** FaCT [11] grades a per-value secrecy label, joins it along the dependency path, checks it against a declared one, rejects where a linter would advise, and discharges declassify-free code into a constant-time lowering obligation: every element of §2.1, in another subject, in 2019. What differs is the quantifier.

Their pair of runs holds the program fixed and varies the secret, observing a leakage trace [4, 5, 7], whereas ours holds the input fixed and varies the schedule, observing the bits. Both are 2-safety properties, each blind to the other’s subject.

## 10 Conclusion

A demand carried on the value and joined over its history is enough to make reproducibility implementable with a type. The schedule is one subject amongst many, and a neighbour of this work already grades another. The Determinism Checker carries a three-point class of the same shape over nondeterminism in sequential code [31]. Swapping the operation-class table extends this design to other sources of nondeterminism in dataflow code.

The opening program’s free and replay folds now lower to two kernels, because the contract is part of each value’s identity and no merge can drop it. In such a language, *these bits must not depend on the schedule* is a sentence the compiler checks.

## A The artefact

The artefact is at <https://github.com/politepixels/replay-lattice-paper> and archived at <https://doi.org/10.5281/zenodo.23655048>. Check ids (L1 to L12) and control ids (CC1 to CC19) name entries in its registry, `ARTIFACT.md`, which gives each one’s command and expected result.

## References

- [1] Jordy Aaldering, Sven-Bodo Scholz, and Bernard van Gastel. 2023. Type Patterns: Pattern Matching on Shape-Carrying Array Types. In *IFL*. doi:10.1145/3652561.3652572
- [2] Willow Ahrens, Hong Diep Nguyen, and James Demmel. 2016. reproBLAS: Reproducible Basic Linear Algebra Sub-programs. <https://bebop.cs.berkeley.edu/reproblas/>.
- [3] Nader Al Awar and Srinivas Yadav Singanaboina. 2026. Controlling Floating-Point Determinism in NVIDIA CCCL. NVIDIA Technical Blog, 5 March 2026. <https://developer.nvidia.com/blog/controlling-floating-point-determinism-in-nvidia-cccl/>
- [4] José Bacerlar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. 2017. Jasmin: High-Assurance and High-Speed Cryptography. In *ACM CCS*. 1807–1823. doi:10.1145/3133956.3134078
- [5] José Bacerlar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying Constant-Time Implementations. In *USENIX Security*. 53–70.
- [6] Ivo Babuška. 1969. Numerical Stability in Mathematical Analysis. In *Proceedings of the IFIP Congress 68*. North-Holland, 11–23.
- [7] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. 2020. Formal Verification of a Constant-Time Preserving C Compiler. *Proc. ACM Program. Lang.* 4, POPL (2020), 7:1–7:30. doi:10.1145/3371075
- [8] Heiko Becker, Eva Darulova, Magnus O. Myreen, and Zachary Tatlock. 2019. Icing: Supporting Fast-Math Style Optimizations in a Verified Compiler. In *CAV (LNCS, Vol. 11561)*. 155–173. doi:10.1007/978-3-030-25543-5\_10
- [9] Heiko Becker, Robert Rabe, Eva Darulova, Magnus O. Myreen, Zachary Tatlock, Ramana Kumar, Yong Kiam Tan, and Anthony Fox. 2022. Verified Compilation and Optimization of Floating-Point Programs in CakeML. In *ECOOP (LIPIcs, Vol. 222)*. 1:1–1:28. doi:10.4230/LIPIcs.ECOOP.2022.1
- [10] Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. 2009. A Type and Effect System for Deterministic Parallel Java. In *OOPSLA*. 97–116. doi:10.1145/1640089.1640097
- [11] Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. 2019. FaCT: A DSL for Timing-Sensitive Computation. In *PLDI*. 96–111. doi:10.1145/3314221.3314605
- [12] T. Y. Chen, S. C. Cheung, and S. M. Yiu. 1998. *Metamorphic Testing: A New Approach for Generating Next Test Cases*. Technical Report HKUST-CS98-01. HKUST.
- [13] Michael R. Clarkson and Fred B. Schneider. 2010. Hyperproperties. *Journal of Computer Security* 18, 6 (2010), 1157–1210. doi:10.3233/JCS-2009-0393
- [14] Eva Darulova and Viktor Kuncak. 2014. Sound Compilation of Reals. In *POPL*. 235–248. doi:10.1145/2535838.2535874
- [15] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* 11, 4 (1978), 34–41. doi:10.1109/C-M.1978.218136
- [16] Dorothy E. Denning. 1976. A Lattice Model of Secure Information Flow. *Commun. ACM* 19, 5 (1976), 236–243. doi:10.1145/360051.360056
- [17] Joseph Devietti, Dan Grossman, and Luis Ceze. 2012. The Case For Merging Execution- and Language-level Determinism with MELD. In *WoDet*. [https://homes.cs.washington.edu/~djg/papers/wodet\\_meld.pdf](https://homes.cs.washington.edu/~djg/papers/wodet_meld.pdf)
- [18] Yijia Gu and Thomas Wahl. 2017. Stabilizing Floating-Point Programs Using Provenance Analysis. In *VMCAI (LNCS, Vol. 10145)*. 228–245. doi:10.1007/978-3-319-52234-0\_13
- [19] Horace He and Thinking Machines Lab. 2025. Defeating Nondeterminism in LLM Inference. Thinking Machines Lab: Connectionism. doi:10.64434/tml.20250910
- [20] IEEE. 2019. IEEE Standard for Floating-Point Arithmetic. IEEE Std 754.
- [21] William Kahan. 1965. Pracniques: Further Remarks on Reducing Truncation Errors. *Commun. ACM* 8, 1 (1965), 40. doi:10.1145/363707.363723
- [22] Ulrich Kulisch and Van Snyder. 2011. The Exact Dot Product as Basic Tool for Long Interval Arithmetic. *Computing* 91 (2011), 307–313. doi:10.1007/s00607-010-0127-7
- [23] Lindsey Kuper, Aaron Turon, Neelakantan R. Krishnaswami, and Ryan R. Newton. 2014. Freeze After Writing: Quasi-Deterministic Parallel Programming with LVars. In *POPL*. doi:10.1145/2535838.2535842
- [24] Peter Linz. 1970. Accurate Floating-Point Summation. *Commun. ACM* 13, 6 (1970), 361–362. doi:10.1145/362384.362498
- [25] Livermore Software Technology. 2021. LS-DYNA Keyword User's Manual, Volume I, R13. pp. 12-404 and 12-425 to 12-427.
- [26] Daniel D. McCracken and William S. Dorn. 1964. *Numerical Methods and Fortran Programming*. Wiley.
- [27] William M. McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (1998), 100–107.
- [28] Message Passing Interface Forum. 2023. MPI: A Message-Passing Interface Standard, Version 4.1. <https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf>.
- [29] Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. System Sci.* 17, 3 (1978), 348–375. doi:10.1016/0022-0000(78)90014-4
- [30] Alexandre Moine, Sam Westrick, and Joseph Tassarotti. 2026. All for One and One for All: Program Logics for Exploiting Internal Determinism in Parallel Programs. *Proc. ACM Program. Lang.* 10, POPL, Article 26 (2026). doi:10.1145/3776668
- [31] Rashmi Mudduluru, Jason Waataja, Suzanne Millstein, and Michael D. Ernst. 2021. Verifying Determinism in Sequential Programs. In *ICSE*. 37–49. doi:10.1109/ICSE43902.2021.00017
- [32] Arnold Neumaier. 1974. Rundungsfehleranalyse einiger Verfahren zur Summation endlicher Summen. *ZAMM* 54 (1974), 39–51. doi:10.1002/zamm.19740540106
- [33] NVIDIA. 2026. CCCL: CUDA Core Compute Libraries. v3.1.0 to v3.5.0. <https://github.com/NVIDIA/cccl>
- [34] OpenMP Architecture Review Board. 2021. OpenMP Application Programming Interface, Version 5.2, Section 5.5.6, Reduction Scoping Clauses.
- [35] Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. 2019. Quantitative Program Reasoning with Graded Modal Types. *Proc. ACM Program. Lang.* 3, ICFP, Article 110 (2019), 30 pages. doi:10.1145/3341714
- [36] Heinz Rutishauser. 1967. Description of ALGOL 60, Chapter 5: Miscellaneous Applications. In *Handbook for Automatic Computation, Vol. 1a*. Springer, 95–143. doi:10.1007/978-3-642-86934-1\_5
- [37] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 2011. EnerJ: Approximate Data Types for Safe and General Low-Power Computation. In *PLDI*. doi:10.1145/1993498.1993518
- [38] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. 1996. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming* 29, 1–3 (1996), 17–64.
- [39] Christoph Stelz, Lukas Hübner, and Alexandros Stamatakis. 2026. Bit-reproducible parallel phylogenetic tree inference. *Bioinformatics* 42, 2 (2026), btg044. doi:10.1093/bioinformatics/btag044
- [40] Patricia Suriana, Andrew Adams, and Shoaib Kamil. 2017. Parallel Associative Reductions in Halide. In *CGO*. doi:10.1109/CGO.2017.7863747
- [41] Tachio Terauchi. 2008. A Type System for Observational Determinism. In *IEEE CSF*.
- [42] The Apache Flink Project. 2022. Determinism in Continuous Queries. Flink documentation, Table API. <https://nightlies.apache.org/flink/flink-docs-release-1.20/docs/dev/table/concepts/determinism/>
- [43] The Apache Spark Project. 2024. SPARK-47520: Precision issues with sum of floats/doubles leads to incorrect data after repartition stage retry. Apache JIRA issue. <https://issues.apache.org/jira/browse/SPARK-47520>
- [44] The Apache Spark Project. 2024. RDD.outputDeterministicLevel and DeterministicLevel. Spark source, tag v3.5.3. <https://github.com/apache/spark/blob/v3.5.3/core/src/main/scala/org/apache/spark/rdd/RDD.scala>
- [45] The Apache Spark Project. 2026. Expression.deterministic and semanticEquals. Spark source, catalyst expressions, file Expression.scala.
- [46] The Futhark developers. 2026. Futhark prelude: reduce. Futhark 0.22.0 prelude documentation. <https://futhark-lang.org/docs/prelude/prelude/soacs.html>
- [47] The Khronos Group. 2023. The OpenGL Shading Language, Version 4.60.8. OpenGL Registry. <https://registry.khronos.org/OpenGL/specs/gl/GLSLangSpec.4.60.html>
- [48] The Khronos Group. 2024. SPV\_KHR\_float\_controls2. SPIR-V extension registry. [https://github.com/khronos.org/SPIR-Registry/extensions/KHR/SPV\\_KHR\\_float\\_controls2.html](https://github.com/khronos.org/SPIR-Registry/extensions/KHR/SPV_KHR_float_controls2.html)
- [49] The LLVM Project. 2026. Fast-Math Flags. LLVM Language Reference Manual (LLVM 24.0.0git, 29 August 2026), and lib/Transforms/Scalar/EarlyCSE.cpp.
- [50] The MLIR Project. 2026. 'arith' Dialect: FastMathFlagsAttr. MLIR documentation, [mlir.llvm.org/docs/Dialects/ArithOps/](https://mlir.llvm.org/docs/Dialects/ArithOps/).
- [51] The NumPy Project. 2026. numpy.sum. NumPy v2.5 Manual, reference documentation. <https://numpy.org/doc/stable/reference/generated/numpy.sum.html>
- [52] UXL Foundation. 2026. parallel\_deterministic\_reduce. oneAPI Specification 1.4-provisional-rev-1, oneTBB specification.
- [53] Cooper Veit. 2026. Kernel Contracts: A Specification Language for ML Kernel Correctness Across Heterogeneous Silicon. arXiv preprint. arXiv:2604.22032 [cs.LG]
- [54] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. 1996. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security* 4, 2-3 (1996), 167–187. doi:10.3233/JCS-1996-42-304
- [55] Steve Zdancewicz and Andrew C. Myers. 2003. Observational Determinism for Concurrent Program Security. In *IEEE CSFW*. 29–43. doi:10.1109/CSFW.2003.1212703