

IFL 2026, GOTHENBURG, THURSDAY 29 OCTOBER

The Replay Lattice

A type system for floating-point reproducibility

Michael Haddon

politepixels, doi 10.5281/zenodo.23655048

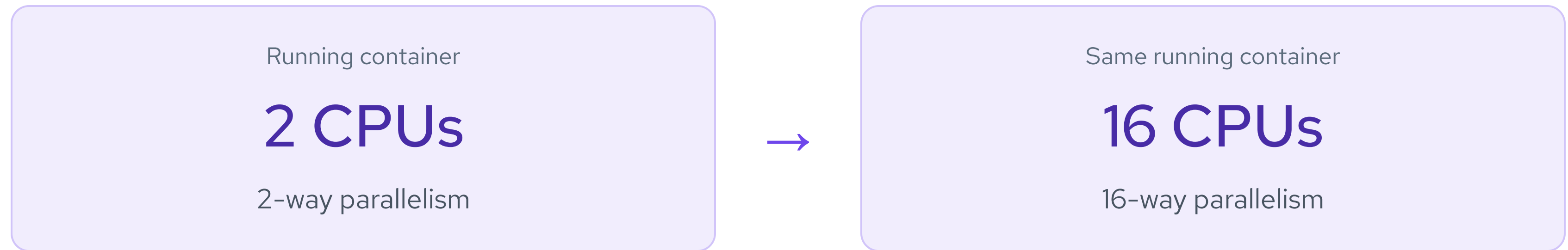
I run infrastructure

The system checks reproducibility, so nobody has to remember to.

- ✓ Container images pinned by digest
- ✓ Every shipped artefact hashed
- ✓ Replay recorded events to reproduce production failures
- ✓ The same code and inputs should give the same result in development and production

The runtime adapts. The answer should stay put.

Vertical autoscaling can change CPU limits without restarting the container.



Go 1.25 can adjust execution parallelism live as CPU limits change.

Java, Rust and C libraries can select different worker counts after restarting with a new CPU allocation.

You wanted the same answer, sooner. But now you've got **a different answer.**

Monday's run, Tuesday's run

Same code. Same data. A numerical simulation on autoscaled infrastructure.

```
Monday    total_energy = 1.734209461
```

```
Tuesday   total_energy = 1.734209458
```

The physics, or the **cluster**?

Did a parameter drift, or did the cluster scale? A decision about capacity has become a decision about what my application computes.

Nobody chose the schedule



The bits moved because the schedule moved. And nobody chose the schedule.

Two labels

A GRANT

“May contain nuts.”

Settled at the dish that carries it. Nothing added elsewhere can make it false.

The implementation may choose the order.

A DEMAND

“Contains no nuts.”

A claim about every ingredient that ever entered the pot. One nut anywhere falsifies it, and no later stirring takes it back out.

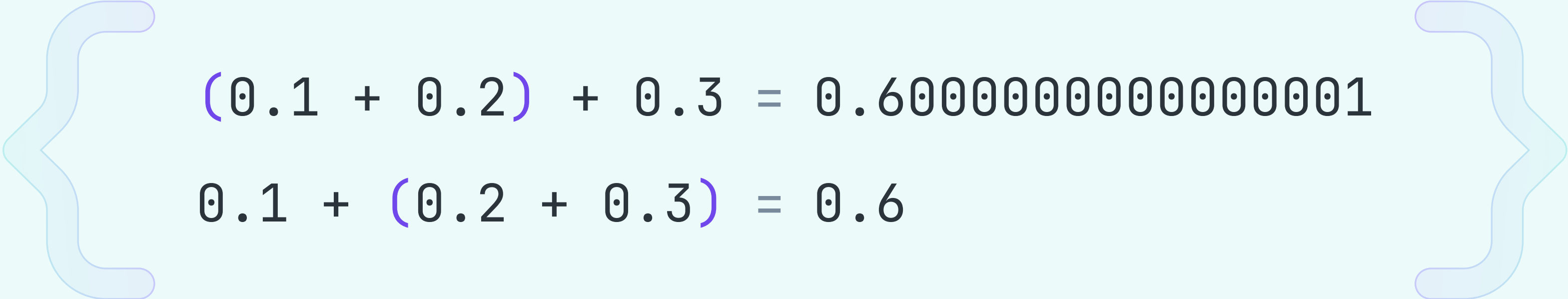
A promise about the result, so everything upstream has to keep it.

These bits must not depend on the schedule is the second kind of sentence. And unlike nuts, you cannot test the dish: one run looks fine, and it takes two runs to disagree.

Reproducibility is not a **grant**.

A grant is true where it is written. A demand was settled upstream, by everything that ever touched the value.

Two sums


$$\begin{aligned} & (0.1 + 0.2) + 0.3 = 0.600000000000000000000001 \\ & 0.1 + (0.2 + 0.3) = 0.6 \end{aligned}$$

Every addition in both sums is correctly rounded. IEEE 754 requires it.

Nothing here is a rounding bug. These are two different *trees*.

The census

SYSTEM	THE KNOB	ATTACHED TO	DIRECTION	ENFORCED?
LLVM / MLIR	reassoc, contract	an instruction	grant	no
SPIR-V	FPFastMathMode	an instruction	grant	no
Icing opt:	a scoped grant	an expression	grant	verified
Futhark reduce	a promise	an operator	grant	no
Halide rfactor	proved associativity	a directive	grant	in the reals
OpenMP	reduction	a construct	grant	no
MPI_REDUCE	implicit reorder licence	a collective	grant	no
CCCL require	a level	one call	demand	that call only
oneTBB reduce	split/join per partitioner	one call	demand	no
Batch invariance	order fixed by extent	one kernel	demand	no
Kernel contracts	three-level class	one kernel	demand	by invocation
IEEE 754	reproducibility attribute	a lexical block	demand	by the language
LS-DYNA	consistency flag	a run	demand	no
GLSL precise	a qualifier	a value + producers	demand	no computed class
This work	a lane	a value	demand	whole path

The lattice

Three classes, carried on every value of a pure dataflow array language. One such label on one value is a **lane**.

exact

The bits equal the **serial reference**.

$\sqsubseteq$

replay

The bits **must not move** with the schedule. Tomorrow's run agrees with today's.

$\sqsubseteq$

approx

The bits **may move** with the schedule. The default, because only some results need the promise.

exact does not mean correct.

It matches the serial reference, rounding error and all.

It's not enough to just be **deterministic**.

Determinism today does not mean determinism tomorrow.

The join

When two values meet, the result takes the worse of the two.

```
19 x := indices(count: 100000.0) exact
```

```
20 approx pre := sum(x * 0.25) approx
```

```
21 replay demanded := sum(x + pre) approx
```

$x + pre$ exact $\sqcup$ approx = approx

$sum, fixed\ tree$ replay $\sqcup$ approx = approx

$demanded$ asked for replay, got approx

THE POISONING LAW, THEOREM 2

The label is set by the worst thing that ever touched the value, and it never improves. One free fold is the nut in the pot.

NO ESCAPE HATCH

Once the rounding has happened, the low bits are gone.

You wrote **replay**. The history says **approx**. **Refused**.

Refused, and it names names

```
$ ./build/bin/lanelang check src/tests/lanelang/reject_poison.lane
src/tests/lanelang/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
src/tests/lanelang/reject_poison.lane:20:19: note: 'sum' is a fold/free, so it contributes approx to
the join
    20 |      approx pre := sum(x * 0.25)
        |      ^^^
lanelang: 1 determinism lane violation
(exit 1)
```

The checker

WHAT YOU WRITE

One **qualifier**, on the one value you care about.

```
replay demanded := sum(x + pre)
```

WHAT IT DOES

- 1 Heals what nothing else can read
- 2 Refuses what it cannot prove
- 3 Names names

One word deleted, one producer healed

The same program, one word deleted.

```
BEFORE 20 approx pre := sum(x * 0.25)
```

```
AFTER 20 pre := sum(x * 0.25)
```

```
$ ./build/bin/lanelang check talk/demo/heal_poison.lane  
accepted: 1 declaration checked, 1 producer healed, result lane replay  
(exit 0)
```

Why pay for exact?

With **replay**, the backend picks the order. With **exact**, your program picks it.

	approx	replay	exact
Who picks the order	each run	the backend	your program: serial, by default
Same bits	never promised	every rerun on the same setup, any worker count	anything that computes the same order, in principle
What can change your answer	any run	a new GPU, library or kernel	only an edit to your code
Speed	fastest	tuned to vector and warp width	serial, by default
Reach for it when	nothing depends on the bits	shipping; reruns and audits on the same setup	matching a plain loop; goldens shared across machines; a CPU checking a GPU

exact does not make machines agree. It gives them **something to agree on**.

STILL OPEN ACROSS MACHINES

Floating-point environment

rounding mode

flush-to-zero

x86 and Arm flush differently

a library resetting it after load

Instructions and code

FMA contraction

vendor estimate instructions

GPU fast division and square root

instruction-set level

maths library

compiler version

Pay only where you ask

The lane goes on the call, not inside the function.

```
func energy(v Column[Float]) Float {  
    return sum(v * 2.0)  
}  
  
exact expected := energy(x)    // the test: serial fold  
replay shipped := energy(x)    // production: fixed tree
```

Same function. Only the test pays for **exact.**

Both demands are accepted, and the checker keeps the two kernels apart.

One waiver

The schedule gets none. The input bytes get exactly one.

THE SCHEDULE

The lattice handles it. Once the rounding has happened, nothing can waive it.

THE INPUT BYTES

A **source** is **ungrounded**: the program does not fix its bytes, so **replay** over it is refused.

```
@bless(0x1234abcd) x := source(count: 1000.0)
replay energy := sum(x * 2.0)
```

Bare, it is a promise.

“These bytes will be the same next run.”

With a checksum, a **receipt.**

The run checks it before computing anything.

Refused: the input is not fixed

The last slide's program, without the blessing. The bits move with the world, not the schedule.

```
$ ./build/bin/lanlang check src/tests/lanlang/reject_ungrounded_replay.lane
src/tests/lanlang/reject_ungrounded_replay.lane:19:5: error: replay demands a cone the program fixes
or a blessing vouches for, and this value is neither. The bits do not move with the schedule here,
they move with the world: vouch for it with @bless(...) on the binding that names it, take the value
as a parameter, which makes it the caller's obligation, or weaken the demand to approx
    19 |      replay energy := sum(x * 2.0)
        |      ^^^^^^
src/tests/lanlang/reject_ungrounded_replay.lane:18:10: note: 'source' is a leaf this program does
not fix, so its value is not a function of the program
    18 |      x := source(count: 1000.0)
        |      ^^^^^^
lanlang: 1 ungrounded demand
(exit 1)
```

For compiler writers

The lane is part of the value's identity.

STILL LEGAL

- ✓ Fusion
- ✓ Tiling
- ✓ Any reordering that keeps dependences intact

FORBIDDEN

- ✗ Regrouping a pinned sum
Splitting it into partial sums changes the bits.
- ✗ Merging identical sums with different lanes
Identical-looking sums need separate kernels when their contracts differ.

The lane is part of the value's identity, so common-subexpression elimination **can't merge them**.

WHAT EACH LANE COMPILES TO

- approx** → a reduction with unconstrained grouping
- replay** → a reduction with a fixed tree
- exact** → a serial fold

0

cost to code that asks for nothing

Unannotated code lowers to the identical free kernel.

1 pass

compile time

At most $d + 2$ passes across d carried edges.

A sentence the compiler checks

“These bits must not depend on the schedule.”

The foundation of a **determinism-safe** language.

[doi 10.5281/zenodo.23655048](https://doi.org/10.5281/zenodo.23655048)

github.com/politepixels/replay-lattice-paper

IF ASKED

What does **replay** cost?

VENDOR-PRICED
REPRODUCIBLE REDUCE

$\geq 15\%$

LS-DYNA, pinned order, shared-memory build, serial assembly: "at least 15 percent".

20-30%

CCCL `gpu_to_gpu`, large problem sizes: "20% to 30%".

UNDEMANDED CODE

0, checked

The unannotated program and its **approx** twin lower to the identical free kernel. Pinned by two fixtures.

COMPILE TIME

One pass

With no recurrence. At most $d + 2$ passes when lane information crosses d carried edges.

IF ASKED

Why recurrence needs more passes

d counts the carried edges lane information must cross. The ceiling is $d + 2$.

NESTED

```
total := iterate(..., with: (outer, t) => {  
  nested := iterate(..., with: (inner, u) => {  
    return inner + sum(x)  
  })  
  return outer + nested  
})
```

Two recurrences, one inside the other.

$d = 2 \rightarrow$ at most 4 passes

CHAINED

```
a := iterate(..., with: (acc, t) => {  
  return acc + sum(x)  
})  
b := iterate(..., with: (acc, t) => {  
  return acc + a  
})  
c := iterate(..., with: (acc, t) => {  
  return acc + b  
})
```

Three recurrences in sequence; each reads the previous result.

$d = 3 \rightarrow$ at most 5 passes

The checker stops when a pass changes no lanes. These are graph-analysis passes, not loop iterations. The sketches omit iteration settings.

IF ASKED

The second pass exposes the demand

Both passes visit the whole function. The colours show the information discovered so far.

Pass 1: discover approx

```
func main() Float {  
  x := indices(count: 1000.0)  
  total := iterate(  
    times: 3.0, from: 0.0,  
    with: (acc, t) => {  
      replay hello := acc  
      approx part := sum(x)  
      return hello + part  
    })  
  return total  
}
```

Body and total: approx. The earlier demand still appears satisfied.

Pass 2: feedback reaches acc

```
func main() Float {  
  x := indices(count: 1000.0)  
  total := iterate(  
    times: 3.0, from: 0.0,  
    with: (acc, t) => {  
      replay hello := acc  
      approx part := sum(x)  
      return hello + part  
    })  
  return total  
}
```

Body and total: approx. The replay demand on hello is violated.

Amber: approx - Green: demand appears satisfied - Red: demand violated

Pass 3 confirms no change.

IF ASKED

When healing must refuse

A demand can change its private producers, but cannot change a value observed outside its scope.

Private producer - accepted

```
func main() Float {  
  x := indices(count: 1000.0)  
  s := sum(x)  
  t := s + 1.0  
  replay e := sum(x * t)  
  return e  
}
```

s reaches the result only through e. The demand permits pinning s.

Replay demand satisfied.

Outside read - rejected

```
func main() Float {  
  x := indices(count: 1000.0)  
  s := sum(x)  
  t := s + 1.0  
  replay e := sum(x * t)  
  return t + e  
}
```

t reaches the result without passing through e. Pinning s could change that outside result.

Replay demand cannot be met.

IF ASKED

The operation-class table

Kernel class sets the strongest guarantee. Input lanes can weaken the result.

Kernel	Its bits depend on	Class	Witnessing check
Elementwise arithmetic	No reassociable combination; host compiler outside S	neutral	L2 (combination)
Data movement, literals	No combination	neutral	L2
External source	Outside world; grounding refuses unsupported demands	neutral, ungrounded	L3, CC1, L2
Carried-state recurrence	Strictly sequential; no schedule choice of its own	neutral	L3, L4, L2
Fixed-tree reduction	Data and shape	replay	L4, L2, CC2
Fixed-block scan	Data and shape; serial block scans and serial base pass	replay	L3, L4, CC3, L2
Oracle-order strict fold	Data; matches the serial reference by construction	exact	L4 (reference), CC4, L2
Exact-integer reduction	Data; matches serial while integers’ absolute values sum below 2^{53} . Evaluation fails outside that range.	exact	L3, L4, L2 (chunking)
Free reduction, free scan	Schedule; reassociation allowed	approx	L4, L2
Parallel scatter	Unchecked: no parallel scatter implemented	approx	None
Transcendental functions	Unchecked: no maths library specified	approx	None

Neutral kernels add no weakening. The table covers the demonstration language’s kernels.

IF ASKED

Theorem 1: schedule agreement

An accepted demand delivers the determinism it asks for.

What the theorem needs

1. Graph properly ordered
2. Kernels honour their classes
3. Each node's lane follows the join rule
4. Choose any two schedules for which:
 - 4a. At each recurrence step, the carried bits match where required
 - 4b. The declared lane promises matching bits across these two schedules
5. The computed lane satisfies the declared demand

Same program. Same input bits.



Schedule A
e.g. 2 workers
demanded bits

Schedule B
e.g. 8 workers
demanded bits

Identical bits

Replay covers deployment schedules. Exact also agrees with the serial reference.

IF ASKED

Theorem 2: the poisoning law

A stronger kernel cannot repair a weaker input guarantee.



The join keeps the weakest guarantee among the kernel and all its inputs.

Every downstream dependency inherits approx, however strong its own kernel.

This is a law about labels. It does not claim the actual answers must differ.