# X Proofs: Checked Layouts and Memoised Parallel Calls

William Kirkman-Moller

https://github.com/WillKirkmanM/x-proofs

**Abstract**

X Proofs formalises record-layout translation and concurrent memoisation in Lean 4. For records stored in separate columns, we prove that writes through borrowed fields preserve execution outcomes, including the first invalid write. The theorem covers any finite record width and any Boolean invariant. For memoised calls, we prove that every completed response equals the uncached result when lookup and publication are atomic, even with simultaneous misses and colliding entries. We compose the two results to cover successful record updates followed by read-only calls. We also prove a validation criterion for a decoded write and its invariant guard, then apply it to selected LLVM output. Comparisons with compiled X programs and executable counterexamples test the owner tracking, immediate checks and complete-key comparisons required by the models.

## 1 Introduction

A programmer can request a different data layout or reuse a pure function's result while leaving the computation unchanged. Splitting a record into columns separates a field's address from the record whose invariant governs a write. Memoisation adds a mutable cache that parallel callers may share. Both transformations must preserve record checks and function results.

Sections 2 and 3 formalise the two transformations in Lean 4 [1]. The layout model translates a record mutation into a column update followed by a check on the same record. Its preservation theorem relates complete executions, including the first invalid write. The cache model separates lookup from publication so that computation can run outside the lock. We prove each completed response correct by maintaining a cache invariant, including when callers compute the same miss or replace entries in the same slot.

A third theorem composes the transformations. After a successful mutation finishes and the borrow is released, records in either layout supply the same keys to the cache protocol. We compare compiled X programs with the executable models and validate selected writes in the emitted LLVM code.

The implementation assumptions appear in Section 4; supporting models for X's ten constructs appear in Section 5.

# 2 Checked mutation across layouts

## 2.1 An invalid write must fail when it occurs

Consider two records initially equal to $(\texttt{lo}, \texttt{hi}) = (0, 10)$, with the invariant $\texttt{lo} \leq \texttt{hi}$. A program borrows the `lo` field of row zero, selects row one, and writes 11 through the existing borrow. The write still belongs to row zero, whose invariant now fails. Checking the newly selected row would miss the error.

A second program writes 11 and then 5 through a borrow of `lo`. The first write must trap: the second write is never reached. Checking only after both writes would accept the repaired record $(5, 10)$ and change the program's behaviour. Layout translation must therefore retain the borrowed field's record owner and check that record immediately after each write.

## 2.2 Source and target machines

The source store maps each of $n$ rows to a record $\mathrm{Fin}(m) \to \mathbb{Z}$, where $m$ is the number of fields. Its invariant is an arbitrary Boolean predicate on the whole record, so it may relate different fields. Its state also contains a mutable selected index, an optional borrowed projection, a list of observed values and a write count. Borrow creation copies the current index into the projection. Selecting another index afterwards leaves that projection unchanged. A second borrow is rejected until the first is released; direct observations require the loan to have ended.

A write through the source projection updates its field and immediately evaluates the record invariant. Success continues execution. Failure returns a trap containing the committed state, with no execution of subsequent commands. The target instead stores each field in a separate column. Its reference has a physical address, represented by a field and row, and a separately retained logical owner.

Translation expands each source store into two target instructions:

$$\mathrm{lower}(\mathrm{store}(v)) = [\mathrm{put}(v), \mathrm{guard}].$$

The first updates the addressed column. The second reconstructs all fields of the retained owner and evaluates the invariant. Between the write and the guard, the updated record may violate the invariant.

## 2.3 Preservation theorem

Let $E$ encode source records as columns and copy the frozen reference into the target address and owner. Let $S(P, s)$ execute source commands $P$ from state $s$, and let $T$ execute target instructions. Outcomes distinguish success, trap and rejected operations. The theorem `compile_preserves_execution` establishes

$$T(\mathrm{compile}(P), E(s)) = \mathrm{mapOutcome}(E, S(P, s)).$$

The theorem quantifies over finite command lists, source states, row and field counts, and Boolean record invariants in `WideCheckedLayout`. For success and trap outcomes, equality includes the complete store, observation list and write count. Rejected operations produce the same rejection outcome; the core records no diagnostic state for rejection.

The proof first shows that updating one column corresponds to updating the selected field in the logical store. Reconstructing the retained owner then gives the same record to the invariant predicate. A command-level lemma relates each source step to its target instruction list. Induction on commands, using an execution-concatenation lemma, extends this correspondence to whole programs. Trap and rejection cases terminate both executions before the remaining commands.

```
import XProof.WideCheckedLayout
open XProof.WideCheckedLayout
open XProof.CheckedLayout (execute)

example {n m : Nat} (invariant : Record m -> Bool)
    (program : List (Command n m)) (state : Source n m) :
    execute (targetStep invariant) (compile program) (encode state) =
      (execute (sourceStep invariant) program state).map encode :=
  compile_preserves_execution invariant program state
```

Listing 1: Execution preservation for any finite record width.

The theorem `first_failure_preserved` identifies the write at which both executions stop. If a command prefix succeeds and the next borrowed write makes its owner's invariant false, the compiled prefix and write return the corresponding trapped state for every remaining suffix. Later updates cannot repair the state before this failure.

### 2.4 Counterexamples and compiled programs

Lean evaluates the two opening examples and confirms that checking the wrong owner or delaying a guard changes the outcome. We test owner tracking and check placement in compiled X programs generated from eight command templates. Their expected outcomes come from `CheckedLayoutOracle.lean`. The experiments run on Windows x86-64. Each template runs on nine inputs with the invariant $0 \leq \texttt{lo} \leq \texttt{hi}$. The templates exercise both fields and rows, changes to the selected index, repeated stores, release and rebinding, and observations before a failing write. Compiling each under AoS and SoA at O0 and O3 gives 32 binaries and 288 executions. All agree with the oracle: 152 succeed and 136 fail with an illegal-instruction trap.

After each successful write, the programs flush their observations and a marker. This output identifies the write at which execution stops and can be compared with the oracle's observation prefix. Successful runs also report every record field. A trapped process exposes its output prefix and failure status. Equality of the committed stores is established by the model theorem.

The second experiment exercises records with one, three and eight fields across three rows. The invariant requires every field to be nonnegative and bounds their sum, so a check must inspect the whole record. Each program borrows the first or last field, observes the rows before mutation, and attempts a repair after the first input-dependent write. Twelve inputs, two layouts and two optimisation levels give 24 binaries and 288 executions. All agree with `WideCheckedLayoutOracle`: 144 succeed and 144 trap at the expected write. Lean computes the field values and observation prefixes for both outcomes.

The test arithmetic stays within the range of `i32`; the model uses mathematical integers.

### 2.5 Checking emitted write sites

The `LoweringValidation` module checks a decoded store and its immediate guard against the source write. Arithmetic expressions contain integer constants, field reads and addition; predicates contain comparisons and conjunctions. A site carries the frozen source owner $r$, projected field $j$, actual write address $a$ and actual guard $G$. Let $P$ be the independently supplied source predicate. Acceptance requires

$$a = (r, j) \quad \wedge \quad G = \mathrm{map}(\lambda f.\,(r, f), P).$$

The equality compares the complete predicate syntax, including its comparisons, constants and arithmetic operations.

The theorem `accepted_write_preserves` relates each accepted write and immediate guard to the encoded source outcome. It holds for every source store with the specified frozen loan and every integer written through it. Success and failure both preserve the complete committed store. Structural induction proves that mapping an expression or predicate to an owner's addresses preserves its evaluation; the site theorem then uses the layout write correspondence. The checker accepts the direct translation and rejects changes to the owner, field or predicate.

An adapter reconstructs write sites from LLVM output at O0. It follows constant GEP paths, field loads, additions, signed comparisons and short-circuit joins. The guard must precede any further store, output, opaque call or return, and its false branch must trap. Instruction ordering and dominance checks ensure that the guard reads fields after the write. The adapter also checks allocation shapes, the input call and the stored expression against the source template. Twelve modules, covering one, three and eight fields under both layouts, provide 24 sites that pass validation. All 140 mutated modules are rejected. The mutations alter addresses, guards or values, use stale field reads, introduce extra input calls, or change the allocation and control flow.

# 3 Memoised parallel calls

## 3.1 Cache validity

A cache key is a list of 64-bit words, and a result is a natural number. A cache maps each of 64 slots to either an empty entry or a key–value pair. For a total function $f : \mathrm{Key} \to \mathbb{N}$, cache validity requires every occupied entry to contain the corresponding result.

$$\mathrm{Valid}(f, C) \;\equiv\; \forall i, k, v.\; C(i) = \mathrm{some}(k, v) \implies v = f(k).$$

The slot selector $h$ maps keys to indices. A lookup examines $C(h(k))$ and compares the complete stored key with $k$. If they differ, the lookup misses even when they occupy the same slot. Storing $(k, f(k))$ replaces that slot and preserves validity.

The operation $\mathrm{invoke}(f, h, C, k)$ returns a result and a cache. A hit returns the stored value; a miss evaluates $f(k)$ and stores it. For $(v, C') = \mathrm{invoke}(f, h, C, k)$, the theorem `invoke_correct` establishes

$$\mathrm{Valid}(f, C) \implies v = f(k) \;\wedge\; \mathrm{Valid}(f, C').$$

The proof separates hits from misses. A hit uses the cache invariant and full-key equality. A miss uses the fact that storing the computed result preserves validity. Induction on a list of keys then proves that repeated invocation returns the same list as applying $f$ to every key.

Listing 2 derives result equality for an empty cache using the sequence theorem. The slot selector is arbitrary, including one that maps all keys to a single slot.

```
import XProof

open XProof.Memoise

theorem empty_cache_agrees (f : Key -> Nat)
    (slot : Key -> Slot) (keys : List Key) :
    (invokeMany f slot keys empty).1 = keys.map f :=
  (invokeMany_correct f slot keys empty
    (empty_valid f)).1
```

Listing 2: Result preservation for any sequence of cached calls.

Consider distinct keys $a$ and $b$ for which $h(a) = h(b)$. Starting from an empty cache, calls on $[a, b, a]$ store $f(a)$, replace it with $f(b)$, and then recompute $f(a)$. Their results are $[f(a), f(b), f(a)]$. The theorem `collision_misses` ensures that looking up $a$ after storing $b$ in the shared slot returns no value.

### 3.2 Interleaved lookup and publication

A source-pure function can have compiler-generated mutable state. The function $f : \mathrm{Key} \to \mathbb{N}$ is total, and its key contains every input that can influence its result. A protocol state consists of a cache $C$, a map $P$ of pending calls and a list $R$ of completed responses. Each response records its caller, key and result.

A request by an idle caller atomically looks up the complete key. A hit immediately records a response. A miss leaves the cache unchanged and places $(k, f(k))$ in that caller's pending entry. This abstracts pure computation outside the cache critical section. A later publication atomically stores that pending pair, records its response and clears the pending entry. A request by a caller already pending, or publication without a pending call, is rejected.

The state invariant requires

$$\begin{aligned} &\mathrm{Valid}(f, C), \\ &P(t) = \mathrm{some}(k, v) \implies v = f(k), \\ &r \in R \implies r.\mathrm{value} = f(r.\mathrm{key}). \end{aligned}$$

The theorem `step_preserves` proves the invariant for each accepted action. Lookup hits use full-key equality and cache validity; misses establish a correct pending pair. Publication uses that pair even if another caller has since replaced the slot. Induction on the action list gives `run_preserves`. Starting from an empty cache and no pending calls or responses, `every_completed_call_correct` follows.

```
import XProof.ConcurrentMemoise
open XProof.Memoise XProof.ConcurrentMemoise

example (f : Key -> Nat) (h : Key -> Slot)
    (actions : List Action) (final : State)
    (accepted : run f h actions initial = some final)
    (result : Response) (member : List.Mem result final.completed) :
    result.value = f result.key :=
  every_completed_call_correct f h actions final accepted result member
```

Listing 3: Correct responses for any accepted interleaving.

The corollary `schedules_agree` compares completed responses from any two accepted action lists. Responses with the same key have equal values regardless of the caller, completion order or final cache contents.

Consider two distinct keys $a = [1]$ and $b = [2, 3]$, the function $f(k) = |k|$, and a slot selector mapping every key to slot zero. Both callers can miss before either publishes. Their publications may overwrite each other, yet their responses remain 1 and 2; a later request for $a$ recomputes 1 after $b$ occupies the slot. Lean evaluates this schedule and a duplicate-miss schedule for the same key. A deliberately incorrect lookup that returns the occupied slot's value without comparing keys instead returns 2 for $a$. The actual lookup compares the keys and reports a miss.

A memoised function accesses shared memory and synchronises its cache operations. Source purity alone therefore cannot justify LLVM `memory(none)` or `nosync` attributes [6].

### 3.3 Mutation followed by read-only calls

The theorem `mutation_handoff_calls_correct` connects layout translation to the cache protocol. Let a mutation program finish in source state $s$, with its loan released. For any record-to-key function $K$, encoding and reconstructing records gives exactly the same job keys:

$$K(\text{gather}(E(s).\text{columns})(i)) = K(s.\text{records}(i)).$$

Jobs request a call using a record's key or publish a caller's pending result. The theorem permits every job list accepted by the cache protocol from its empty initial state. The target mutation reaches the encoded source state with no outstanding address or owner. Reconstructing the records produces identical cache actions, the cache remains valid, and every completed response satisfies $r.\text{value} = f(r.\text{key})$. The proof combines layout execution preservation, the gather/transposition identity and the cache invariant. The row count, field count, predicate, key function and total function are arbitrary.

The composition requires mutation to finish successfully and release its loan before workers read the records. Generated caches may still change during those reads. A mutation that fails earlier retains its original trap by `first_failure_preserved`.

### 3.4 Testing the transformations together

The combined experiment constructs 129 records with invariants, updates their keys from runtime input, and passes a read-only array view to workers. Each worker applies an integer recurrence with 32 steps and compares the result with a Python oracle. Four passes exercise cache reuse. One input distribution repeats eight keys; the other uses 129 distinct keys, forcing collisions in the cache's 64 slots. All record updates precede the workers.

We compile both layouts with ordinary or memoised calls, serial or parallel iteration, both key distributions, and O0 or O3, producing 32 binaries. Each runs with three seeds and either correct expectations or an incorrect expected value in the first, middle or final row. Parallel variants request one or four threads. All 576 executions have the expected outcome: 144 complete and 432 trap on the injected mismatch. OpenMP affinity output confirms four distinct workers in all 48 successful runs that request four threads. Injecting a wrong expected value checks that worker assertions remain effective even though the loop discards each worker's return value.

## 4 Scope and implementation assumptions

Table 1 records the proof coverage and implementation assumptions. Lean checks the model proofs and the theorem for decoded writes. The compiler and LLVM adapter are tested against these models but are not themselves verified.

The layout model excludes general heaps, nested records, escaping references and simultaneous aliases. Scheduler fairness, eventual completion and range coverage require separate arguments; the concurrent-cache theorem concerns completed responses. The adapter rejects unsupported LLVM forms. Its parser and JSON interface are unverified, as are the mapping to LLVM byte addresses, machine overflow, the cache lock, later optimisation and object emission. Compiled experiments use values within range to test the correspondence described above.

| Object | Established in Lean | Connection to implementation |
|---|---|---|
| Record-layout model | Values, observations and first invariant failure are preserved. | Mathematical integers, valid indices and one local borrowed projection. |
| Cache model and handoff | Completed calls return the original function's answer; mutation-to-read handoff preserves keys. | Total functions, complete keys and atomic lookup/publication; mutation finishes before reads. |
| Decoded LLVM write and guard | An accepted site preserves the specified source write, including its failure. | The unverified adapter reconstructs pointers, control flow and source correspondence. |
| Compiler-to-native path | No end-to-end theorem for this path. | Compiled comparisons and mutation tests of generated code. |

Table 1: Proof coverage and the connection to executable code. Validation covers selected decoded write sites.

## 5 Supporting semantic contracts

The library also contains individual models of X's semantic constructs. The `pure` model proves preservation of observable state and independence from its initial value while allowing local mutation. The `unique` model preserves pairwise loan compatibility under acquisition and release. The standalone `invariant` model stops at the first failed predicate, and the general `layout` model proves transposition commutes with field and snapshot record updates. The `parallel` model proves exactly-once range coverage for schedules that are permutations of the index list and world preservation for pure workers.

Other modules support X's call and binding rules. Specialisation preserves evaluation beneath runtime bindings; overload selection requires one matching full signature; tagged-result encoding preserves propagation and fallback. Accepted tail transfers preserve continuation depth and reject caller-local origins. Explicit tail calls through `become` also exist experimentally in Rust [7]; X's model specifies its own checked ABI and lifetime conditions.

## 6 Related work

Blazy and Leroy's Clight semantics supports the verified translations of CompCert [2]. Lem provides reusable executable definitions for multiple proof assistants and OCaml [3]. X Proofs uses executable Lean definitions to study record-layout translation and memoisation, then compares their behaviour with compiled programs.

RustBelt gives a semantic account of Rust's ownership discipline and verification conditions for unsafe libraries [4]. The layout model here records the owner of a projected borrow so that mutation checks the same invariant after the fields move into columns. Its simulation relates those updates and checks for every program in the core.

Herlihy and Wing develop linearizability and representation-invariant reasoning for concurrent objects [5]. Our protocol takes lookup and publication as atomic operations and proves that intervening misses and collisions preserve the function's answers.

Translation validation checks individual compiler outputs [8]. Alive2 validates LLVM transformations with bounded symbolic execution and models LLVM's undefined behaviours [9]. Our checker validates the borrowed field's address and the complete invariant guard at individual write sites. The local preservation theorem then applies to those decoded operations.

# 7 Conclusion

The Lean models establish preservation of checked record updates across layout changes and of function results under interleaved cache lookup and publication. Their composition covers the handoff from successful mutation to read-only calls. The counterexamples show why a field borrow must retain its owner, why each write needs an immediate guard, and why cache hits must compare complete keys. We test these requirements in compiled X programs and check selected writes in the emitted LLVM code.

# References


[1] Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language (system description). In *CADE-28*, pages 625–635, 2021.

[2] Sandrine Blazy and Xavier Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43(3):263–288, 2009.

[3] Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. Lem: Reusable engineering of real-world semantics. In *ICFP*, pages 175–188, 2014.

[4] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL), article 66, 2018.

[5] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[6] LLVM Project. LLVM language reference manual, function attributes `memory` and `nosync`. Accessed October 2026.

[7] Rust Project. The Rust Unstable Book: Explicit tail calls. Accessed October 2026.

[8] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *TACAS*, volume 1384 of LNCS, pages 151–166, 1998.

[9] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. Alive2: Bounded translation validation for LLVM. In *PLDI*, pages 65–79, 2021.