Pulse: Proof-Oriented Programming in a Dependently Typed Concurrent Separation Logic

Nikhil Swamy

Lecture 1 - *July 3, 2026*

1 F*: A Proof-Oriented Programming Language

F* is a general-purpose, *proof-oriented* programming language: programs and proofs are developed together in the same framework. It combines full dependent types, higher-order functions, and an extensible effect system supporting effects such as concurrency and shared mutable state. Proof obligations that arise while programming are by default discharged by the SMT solver Z3; when Z3 fails, F* also provides a tactic system, so one can always fall back to interactive proofs in the style of Lean or Rocq. Verified programs can then be compiled: every F* program extracts to OCaml by default, and specific subsets extract to C, Rust, and—recently—CUDA, via a DSL called Kuiper layered on top of Pulse for verified CUDA programming (presented at PLDI this year, and now being used in an effort to build a fully verified LLM stack).

1.1 Dependent Types

F* is fully dependently typed, so the return type of a function may depend on its argument:

```
let f (b:bool) : (if b then int else string)
  = if b then 0 else "hello"
```

Calling `f true`, F* knows the result is an `int`. The classic length-indexed vector also works: an inductive type `vec a n` carries a natural number index recording its length, and appending a `vec a n` to a `vec a m` yields a `vec a (n + m)`—with no additional bookkeeping. This is because F* is based on *extensional* type theory rather than the intensional type theory of Lean, Rocq, or Agda: conversions between provably equal types happen for free whenever the solver can establish the equality. The price is that type checking becomes undecidable, but since Z3 is already being used

Compiled By:
Markus de Medeiros, Dongyu Wu, John Rizkalla,
Carolyn Zech, and Weiyi Chen

on undecidable fragments, F* simply embraces undecidability. (The `#a` notation marks an implicit argument.)

Indexed vectors are nonetheless an anti-pattern in F*. The idiomatic way to track lengths is with *refinement types* on plain lists:

```
let rec append (#a:Type) (x y:list a)
  : z:list a { length z == length x + length y }
  = match x with
    | [] -> y
    | hd :: tl -> hd :: append tl y
```

F* proves termination of recursive functions using well-founded orders; by default it tries the lexicographic ordering of the arguments. If the recursive call fails to decrease, F* reports that it could not prove termination.

2 Pulse: Separation Logic for Imperative Programs

The core of F* is purely functional. Pulse is a language extension of F* (enabled per-file with `#lang-pulse`) for writing and verifying imperative, effectful programs in *concurrent separation logic*. Here is `swap` on two mutable references:

```
fn swap (#a:Type) (x y:ref a)
requires x |-> 'bx  **  y |-> 'by
ensures  x |-> 'by  **  y |-> 'bx
{
  let vx = !x;
  let vy = !y;
  x := vy;
  y := vx;
}
```

The precondition says `x` points to some value `'bx` and, *separately*, `y` points to `'by`; the double star `**` is the separating conjunction. The postcondition says the contents are exchanged. (Swap is in fact universe-polymorphic: like Lean and Rocq, F* has a countable hierarchy of predicative universes, and type inference fills in universes and type arguments at call sites.)

2.1 Live Proving

Pulse aims to give an *auto-active* verification experience but with full transparency into the proof state: at any point in a Pulse program one can ask "show me the proof state" and see exactly what the prover knows there—e.g., mid-way through `swap`, that both `x` and `y` point to `by`, so more work

remains. One works from the precondition toward the postcondition, inspecting progress along the way, much like tactic mode but while writing the program. This proof-state view is specific to Pulse; ordinary pure F* code has no proof state to display (though F*'s tactic language has a traditional interactive mode).

Stack allocation is written `let mut x = 1`, and `assert` can state any separation logic predicate—pure facts must be promoted explicitly, as in `assert (pure (x == 1))`.

2.2 Loops, Invariants, and Lemmas

While loops carry invariants, and loops containing `break` may also carry an `ensures` clause stating what holds on exit. A function `countdown_until` that decrements a reference `x` down to a lower bound `z` requires ownership `x |-> 'v` with `'v >= z`, maintains the invariant that the loop owns `x` (written `live x`) and that its current value is at least `z`, and ensures `x |-> z` on exit. Termination follows because the value of `x` decreases at each iteration; forgetting to decrement yields an error that the measure does not decrease.

Notably, `live`, `pure`, `while`, and even the termination measures are not built-in keywords but user-defined notions:

```
let live x = exists* v. x |-> v
```

Writing a dereference like `!x` inside an assertion is type-directed sugar: it desugars to `exists* c. x |-> c ** ...` with occurrences of `!x` replaced by `c`—which is also why an invariant mentioning `!x` must carry the corresponding permission `live x`.

Turing's own first example of program verification—multiplication by repeated addition—works the same way: allocate a counter and an accumulator on the stack and maintain an invariant relating them. A further example, which Rustan Leino had shown earlier in the week, proves Gauss's identity: a loop sums the first n numbers, and the postcondition states the result is $n(n+1)/2$. Since F* and Pulse are tightly integrated (every subterm of a Pulse program is an F* term), one proves a pure F* lemma `sum` $n = n(n+1)/2$ by recursion, and simply *calls* that lemma inside the Pulse function to finish the proof.

In short, Pulse aims to offer a Dafny-like experience in separation logic—you can start small and stay in a Dafny-looking world—but with a path all the way to Iris-style invariants, concurrency, and ghost state, plus extraction to C.

3 From Project Everest to Agentic Proving

Over the past decade, Project Everest built a stack of DSLs, verified cryptographic algorithms, and networking protocols on top of F*. This verified code runs nonstop in the Microsoft Azure cloud,

and the F* toolchain ships in every Windows developer build environment. But the roughly 1.5 million lines of verified code cost on the order of a hundred person-years of PhD-level work. Is that cost-effective outside of the most critical software? Could AI help?

For several years Swamy's group fine-tuned models (e.g., Qwen) with reinforcement learning on a curated dataset of F* proof tasks. The results beat GPT-3.5 and won best-paper awards—but were, frankly, still useless in practice. That changed dramatically around December 2025: since Opus 4.5, vanilla frontier models with only a light plugin have become startlingly good at proofs. (An analogy from the audience: like special-purpose hardware for functional languages losing to stock Intel CPUs, domain-specific proof models may be losing to general-purpose frontier models—the value of an LLM here is precisely its *world knowledge*, e.g., knowing what "involution" means.)

The setup: GitHub Copilot (or Claude Code) with an F*-specific plugin of agent and skill files. These files were produced by working interactively with an agent for two weeks *without* any plugin, then asking it to summarize what had been effective; the human then reviewed and tweaked the result. Given a prompt like "implement a file with a function to reverse a list and prove that reverse is an involution," the agent probes the environment with tool calls, invokes the F* checker in a feedback loop, and returns 29 lines of verified F*. But note the key problem: the agent *interpreted* the informal word "involution" as a formal statement, and the human must confirm that this formalization matches their intent.

3.1 AutoCLRS

As a capability experiment, Swamy gave an agent the PDF of CLRS (*Introduction to Algorithms*) and asked it to formalize the book. In about three weeks of interaction it produced **120,000 lines of Pulse code and proof for 52 algorithms and data structures**, including functional correctness and worst-case complexity—Dijkstra, Bellman-Ford, and also things like rod-cutting that required going back to the book to check. (The advanced data structures at the end of the book—B-trees, van Emde Boas trees, and such—were deliberately excluded from the experiment, though agents have since done B-trees within a week.) Most of the human time went into *reading the output* and judging whether the right thing was proved. A cautionary tale: the first proof of Dijkstra (with Opus 4.5) had as its postcondition only that the distance from the source to itself is at most zero. The proof checked; the specification was vacuous. *You must look at what it is proving.*

4 Rubrics, Templates, and Audits

Three emerging principles for structuring human–agent interaction in proof-oriented programming:

- **Rubrics** (as in grading rubrics) are set by a human in order to *assess* whether agentic output

is reasonable. Rubrics are for humans.

- **Templates** are information a human prepares and feeds *to* an agent in order to direct it—hints about how to find solutions.
- **Audits**: proofs only give you so much. One still needs to test the program, benchmark its performance, and judge, e.g., whether a claimed implementation of insertion sort really is insertion sort. There is an interesting design space of useful audits.

4.1 Agentic Proof Automation in the Small

The simplest case—agentic proving *in the small*: you write the exact statement of a lemma yourself (e.g., `rev_involutive : Lemma (rev (rev l) == l)`) and let the agent produce *any* well-typed proof. The statement *is* the rubric; an expert user formalized their intent, so there is no intent gap, and judging a solution is fully automated. This is the sweet spot of our proof checkers, and it is a powerful pattern: one can set a top-level theorem for something complex and let the agent go wild—it may produce many auxiliary lemmas, but as long as the top-level theorem is proved, you're good. If the agent also produces *definitions* (like `rev` itself), those need audits—e.g., does it correctly reverse a three-element list?

4.2 Shared Rubrics: Specifying Sorting Once

How do we turn principles from software- and proof-engineering best practice into rubrics? Left to its own devices on CLRS's sorting algorithms, the agent did something reasonable—shared definitions of sortedness and permutation, placed in a common library and used by all the sorting algorithms—but also something ugly: the definitions were specialized to integers (perhaps an artifact of how CLRS presents sorting), and heapsort's specification differed from insertion sort's, with extra preconditions and `sorted_upto` in place of `sorted`. Each sort should have the same spec. This is code smell—let's not tolerate it. Instead, set a rubric (one can even ask the agent to help write it): a typeclass that every sorting algorithm must instantiate, generic in the element type and in a total order:

```
class array_sort = {
  sort:
    (fn (#a:Type)
        (arr:array a) (len:SZ.t)
        (ord:Pulse.Lib.TotalOrder.total_order a)
        (#s:erased (Seq.seq a))
     requires arr |-> s ** pure (A.length arr == SZ.v len)
     ensures exists* s'.
        arr |-> s' **
        pure (sorted #_ #ord s' /\ permutation s s'))
}
```

Now auditing each algorithm's correctness is one line: check that it is an instance of `array_sort`. This is just good software engineering (factor and reuse common abstractions), and it remains good practice for agentic proof engineering. *If you use agents, don't check your brain at the door.*

4.3 Complexity Proofs and a Weak Rubric

CLRS also does complexity proofs, so we need rubrics for those too. The idea is quite simple: instrument the code with a *ghost counter* (a ghost reference to a natural number, existing only for specification), tick it at every operation to be counted, write invariants about the counter's value, and let the postcondition bound how many ticks happened—e.g., a quadratic bound for insertion sort.

But is this rubric good enough? Consider what one must review to be convinced this is a reasonable complexity result: that the implementation actually bumps the counter at every operation of interest, and that it never decrements the counter. That means reading the implementation in detail—which defeats the vision that one should be able to judge a program by studying its specification alone.

4.4 A Better Rubric via Abstraction and Parametricity

A better rubric combines two ideas:

Monotonic ghost state. An idea from separation logic: build *monotonic* ghost state that only allows the counter to be incremented, never decremented—so one no longer has to worry about decrements at all.

Parametricity. The sorting function is polymorphic, and instead of receiving the total order as a pure function it can call, the total order is *erased*—available only for specification purposes, not to the code. In its place the code receives an *instrumented* order `iord`: a function that is functionally equivalent to the total order but ticks the counter on every call. By parametricity, the only thing the code can do with array elements is call `iord` to compare them—so *every* comparison is accounted for, without ever reading the code.

The resulting rubric is a class `array_sort` indexed by a function `bound`, the complexity bound, sketched as:

```
class array_sort (bound: nat -> nat) = {
  sort:
    (fn (#a:Type)
        (arr:array a) (len:SZ.t)
        (ctr:ticks_t)                          // monotonic ghost counter
        (#ord:erased (total_order a))          // spec-only
        (iord:instrumented_total_order ord ctr)
```

```
        (#s:erased (Seq.seq a)) (#i:erased nat)
    requires arr |-> s ** ctr |-> i **
             pure (length arr == SZ.v len)
    ensures exists* s' ticks.
       arr |-> s' ** ctr |-> ticks **
       pure (sorted #_ #ord s' /\ permutation s s' /\
             ticks <= i + bound (Seq.length s)))
}
```

With an instance of this rubric in hand, the two questions above simply disappear. All one has to review at the end is, e.g.: insertion sort is an instance of `array_sort` with a quadratic bound; merge sort, with an $n \log n$ bound. (The agent's merge sort bound was still a bit messy—it needlessly branched on $n > 0$, though multiplying by zero would give zero anyway—but this is exactly what it produced.) In terms of judging correctness and complexity, this is far more tractable than before. The lesson: use what we have learned about proof engineering—abstraction, separation logic, parametricity, all the good stuff—to set good specs and make this way of programming actually productive.

5 Discussion

Concerns without easy answers. Using this technology requires a paid subscription and non-trivial token costs—a real accessibility problem, and agent sessions are bimodal bets: some produce nothing, while a lot of value concentrates in a few sessions. Better prompting and the techniques above may reduce that variance, and Swamy is exploring whether Microsoft could make token grants to universities so the research community can experiment. There are environmental questions about data center power and water use (the actual impact is hard to assess amid conflicting information, but better energy and cooling technology seem necessary), copyright questions, and pedagogical ones: if agents do everything, how do we teach intro CS? For early learners, plausibly one does not use these tools at first and comes to them later.

Do symbolic automation facilities still matter? F* has mechanisms like registered solver facts, unification hints, and tactics for backward chaining—automation designed to make proofs ergonomic for humans. The agents don't seem to care; they just blast out a proof. What really matters is the *specification*: if the proof checks out and the agent can maintain it as the code evolves, who cares how pretty the proof is? (Though tighter symbolic automation might still help agents find proofs with fewer tokens.)

Why general-purpose LLMs? World knowledge is essential—e.g., knowing what "involution" means when interpreting an informal request. Proofs are semantic constructs, not merely syntactic ones, and agents have shown a striking ability to make value judgments, such as selecting which lemmas are worth factoring out. Whether domain-specific proof models can beat the frontier general-purpose models is an empirical question—someone should try—but the bar is extremely

high right now.

What is this all for? A corpus of verified algorithms is useful in itself (verified components for building verified systems) and as an encyclopedia of vetted, idiomatic proofs that agents can leverage for other tasks. More broadly, the world is vibe-coding enormous amounts of software with no proofs; if software engineers churn out 100,000-line PRs over a weekend, the limiting factor on that productivity is *assurance*—and proofs may be able to corral it.