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

Nikhil Swamy

Lecture 2 - *July 3, 2026*

The primary source of truth for this lecture is the Agentic Proof-Oriented Programming chapter of the F⋆ book [1].

1 Recap: Parametricity as Ghost State

Session 1 introduced three techniques for steering a coding agent to produce verified artifacts in F⋆:

- *Rubrics*: human-checkable statements of correctness, to easily allow a reader to evaluate aspects of an implementation's correctness without fully digesting the code,
- *Templates*: hints that steer the agent toward an effective solution shape, and
- *Audits*: techniques to ensure a generated artifact matches intent.

A rubric may be *too weak*: satisfiable by an implementation that does not do what we meant. We resume with the sharpest instance—specifying a *complexity-aware* sort in which every comparison is accounted for. One could imagine tracking the runtime of a program by instrumenting it with a (ghost) step accounting operation at every comparison. However, this is *too weak*. For this specification to have meaning, the increments must be correctly counted, and therefore the agent's code must be closely audited by a human expert.

The remedy is *parametricity*: we will specify that the algorithm must sort lists of an arbitrary type a, and provide it with an *instrumented total order* (ITO) for comparing elements of a, augmented with a ghost counter. By parametricity, it is not possible for the agent to generate a program which compares elements without logging that it did so.

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

Concretely, the ITO specification consists of the following:

- requires ownership of a ghost tick counter at value i;
- ensures the counter advances to $i + 1$; and
- ensures the Boolean answer agrees with an underlying pure total order.

The counter is a *monotonic* ghost reference, so the tally cannot be rolled back. The rubric then asserts:

- *sortedness* of the output;
- that the output is a *permutation* of the input; and
- a *tick bound* as a function of input length.

Using a rubric based on parametricity, a human expert does not need to digest agentically generated code in order to evaluate it.

2 Rubrics as Templates

A *rubric* is a pass/fail specification the agent optimizes against; a *template* is an exemplar whose shape it imitates. In this section we demonstrate how these two can overlap.

The organizing idea is *functional refinement*: relate an implementation-level data structure to a purely functional *model* which serves as its specification. For example, one can define a separation-logic predicate `isList` relating points-to Pulse connectives about the heap object (*representation*) to a pure F* list (its *model*). At this point, one verifies that each implementation refines the corresponding pure operation. The `search_structure` rubric below is exactly this `repr`/`model`/refinement, generalized from lists to arbitrary trees.

2.1 The `search_structure` Rubric

The book abstracts this pattern into a single typeclass, `search_structure`, whose fields constitute the rubric [1]:

- `repr`/`model`/`owns` — the representation type, its pure model, and the separation-logic ownership predicate tying an in-memory object to its model;
- `valid` — an invariant constraining the model itself (e.g., balancedness or the search-tree ordering);

- ghost find/insert/delete operations on the model, against which the imperative methods—`create`, `dispose`, `search`, `insert`, `delete`—are each given a Pulse specification stated purely in terms of the model;
- `height`/`size` — measures on the model in which complexity goals are stated ("logarithmic in size, linear in height"); and
- per-operation tick bounds, expressed as functions of those measures—the complexity claims we ultimately wish to establish.

It is noted that this technique involves a breadth of Pulse features: the object language comprises a diverse collection of primitives such as state and effects, whose state is described using an Iris-style separation logic, and whose specification is encoded by standard dependently-typed typeclasses.

The models are further constrained by a bundle of `search_structure_laws`: an inserted key is found, a deleted key is not, and other keys are unaffected. Without these, the rubric would be satisfiable by degenerate instances.

2.2 Auditing the Rubric with SPOTs

A rubric is only as good as our confidence that satisfying it entails correctness, so the rubric itself must be audited by writing proofs *against* it: *Small Proof-Oriented Tests* (SPOTs) [2]. A SPOT has two important properties:

- *Gap exposure.* A SPOT is a tiny client: in our list sorting example this could consist of a fixed three-element list. The desired property of a SPOT should be provable from the rubric alone. A well-chosen SPOT exposes weaknesses. Returning to our example, a specification asserting sortedness but not permutation may be satisfiable by the constant function returning `[0]`. Using a SPOT such as a three-element list could help us rule out this case in the same way that a conventional software test would rule out a bad implementation.
- *Instantiability.* A SPOT also checks that the rubric can be met at all. An empty rubric is vacuously safe but useless, and if possible a realistic SPOT helps gain confidence that the rubric is not impossibly strong.

2.3 Case Study: Red–Black Trees

Asked to instantiate `search_structure` with a balanced tree, the agent produced Okasaki's functional red–black trees [3] rather than the imperative CLRS formulation [4], at roughly 3100 lines. Since every method's specification speaks only of the model, any client-visible guarantee follows from the `model` and `laws` instances alone; to be convinced the instance is correct one reads those few dozen lines, not thousands of lines of proof.

3 Stateful Services

Many systems of interest are *communicating state machines*, not pure functions. The goal: build stateful services that are provably correct while keeping legible exactly *what* has been proved.

A first attempt at TLS by freeform prompting is a cautionary tale: it flames out, emitting some 50,000 lines of code that "works" with no tractable statement of what it achieves.

3.1 The Calculator Service

The disciplined route begins with a smaller exemplar, a *calculator service*. The calculator service maintains an internal state: a bounded stack with `push`, `peek`, and bounds-checked `add`/`sub`/`mul`/`div`, plus an agent-authored binary wire format for its operations. An agent is capable of supplying a Pulse implementation and a proof that it follows a pure specification. At this scale the result is directly readable, but that does not survive the jump to TLS. To scale up, we need a rubric whose audit surface is smaller still.

3.2 Abstraction: State Machines as Typeclasses

The solution is abstraction. The book factors the pattern into a small stack of typeclasses [1]:

- a `state_machine` class: an initial state, a type of states, and a transition *relation* over events (local or wire), from which one derives transitions, traces, and trace properties;
- a class of *wire formats*, with a round-trip law making parsing an inverse of serialization; and
- a class of *state-machine refinements* (`protocol_implementation`), engineered so that *any* instance is, by construction, a refinement of its abstract machine.

Tying the operational story to the abstract one is `pi_invariant_valid`: the implementation's invariant must entail that the byte history observed so far is a valid trace of the abstract machine. A companion lemma, `network_process_correct`, advances the invariant one step. The bytes consumed parse to an event, the machine takes the corresponding transition, and the output buffer holds exactly its serialized responses. A final ingredient is *monotonicity*: history may only grow. A `snapshot` of the underlying ghost state is a stable assertion that the machine was once in a given state, and a later `recall` obliges the current state to extend, not rewrite, that past.

Crucially, all of this scaffolding was authored by agents, instructed to produce code in a shape a human expert would recognize. Assembled, the pieces prove that the calculator instantiates its state-machine specification, and the audit is now substantially simpler. Over time, one hopes to accumulate a library of such rubrics distilled from experts. The extracted calculator is about 300 lines of C.

4 Scaling to TLS 1.3

Surprisingly, with the state machine pattern established for our calculator service, an agent is capable of scaling the same technique up to a verified implementation of TLS! Here, the *template* is the calculator server; the *rubric* is the state-machine machinery; the *audit* question is "what is TLS?"

We have two techniques to audit the TLS implementation. TLS is defined by an RFC [5], though in practice the operational notion of correctness is interoperation with existing TLS. Since F* can be compiled and executed, we are able to compile our TLS implementation and verify that it is *a TLS*.

The TLS state machine, transcribed from the RFC, runs to a few thousand lines of specification; that it genuinely *is* a state machine follows immediately from membership in the typeclass. Proof-oriented tests at this scale are large and messy, so instead of SPOTs we can leverage other property-oriented tests to establish properties of our state machine model. For example, we can prove that if a client and server complete a handshake, they agree on the same application key: this is discharged after several thousand lines of lemmas.

The final result rests on two explicit assumptions about the underlying cryptography: an ECDH (X25519) shared-secret agreement property and functional correctness of ChaCha20 open/close.

These remain assumptions only because of a language boundary (the existing proofs are in Low* as part of HACL* [6] rather than Pulse); absent that gap they would be theorems. The one top-level artifact to audit is `client_protocol_implementation`, the few lines instantiating the refinement typeclass: once it typechecks, the implementation refines the machine by construction. All further reasoning can ignore the implementation and proceed against the specification alone.

5 Concluding Remarks

The speed at which agents now produce verified code is startling, and it reopens questions of ambition. Historically, formal verification targeted *new* code rather than existing systems, because retrofitting proofs onto legacy code was prohibitively costly. As the cost collapses, several ambitions come into view:

- verifying existing systems at scale, such as operating systems;
- building the tooling such efforts demand, for instance PAL, a Proof-Annotated Language for C, which decorates C with pre/postconditions and invariants whose meaning is given in Pulse; and
- reaching beyond safety and functional correctness to properties harder even to *state*. Properties such as liveness and security are generally more difficult to prove; higher-level properties

such as the absence of cross-site-scripting attacks or hardware runtime guarantees may be possible to state and attack.

References

[1] N. Swamy, G. Martínez, and A. Rastogi. *Proof-Oriented Programming in F**. Online textbook. `https://fstar-lang.org/tutorial/`.
[2] N. Swamy. Spotting Specification Gaps with Small Proof-Oriented Tests. Blog post, F* project.
[3] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.
[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press.
[5] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, IETF, 2018.
[6] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche. HACL*: A Verified Modern Cryptographic Library. In *CCS*, 2017.