Setup

Go to:

https://github.com/FStarLang/pulse-tutorial-24

And follow instructions to get a working setup with VS Code

Pulse

Proof-oriented Programming in Concurrent Separation Logic

Thibault Dardinier, Megan Frisella, Guido Martinez,
Tahina Ramananandro, Aseem Rastogi, Nik Swamy

For the past two decades …

Many in the PL community have been studying:

What would a general-purpose programming language with dependent types at its core look like?

Cayenne, Liquid Haskell, F*, Idris, Dependent Haskell, ..., Lean, Coq, Agda

Not fully settled yet, but many closely related, successful designs, for purely functional languages

(and with varying degrees of support for reasoning about effects)

F*: A Programming Language with Dependent Types

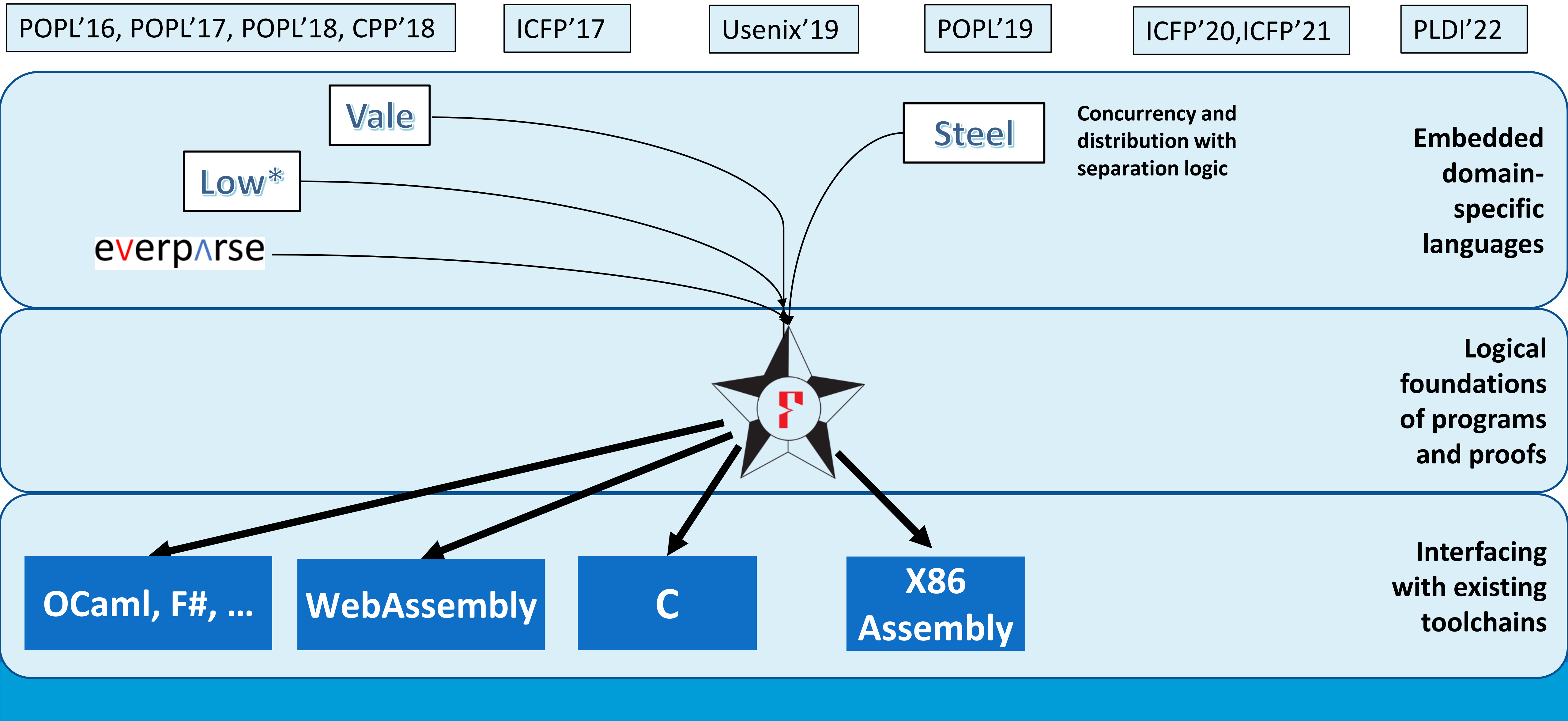
```
let perm l m = forall x. count l x == count m x

let sorted l = match l with
    | [] -> true | [x] -> true
    | x::y::rest -> x <= y && sorted (y::rest)

val mergesort : l:list int -> m:list int{sorted m && perm l m}
```

```
let mergesort l = match l with
  | []  -> []
  | [x] -> [x]
  | _   ->
    let l1, l2 = partition l in
    let l1_sorted = mergesort l1 in
    let l2_sorted = mergesort l2 in
    merge l1_sorted l2_sorted
```

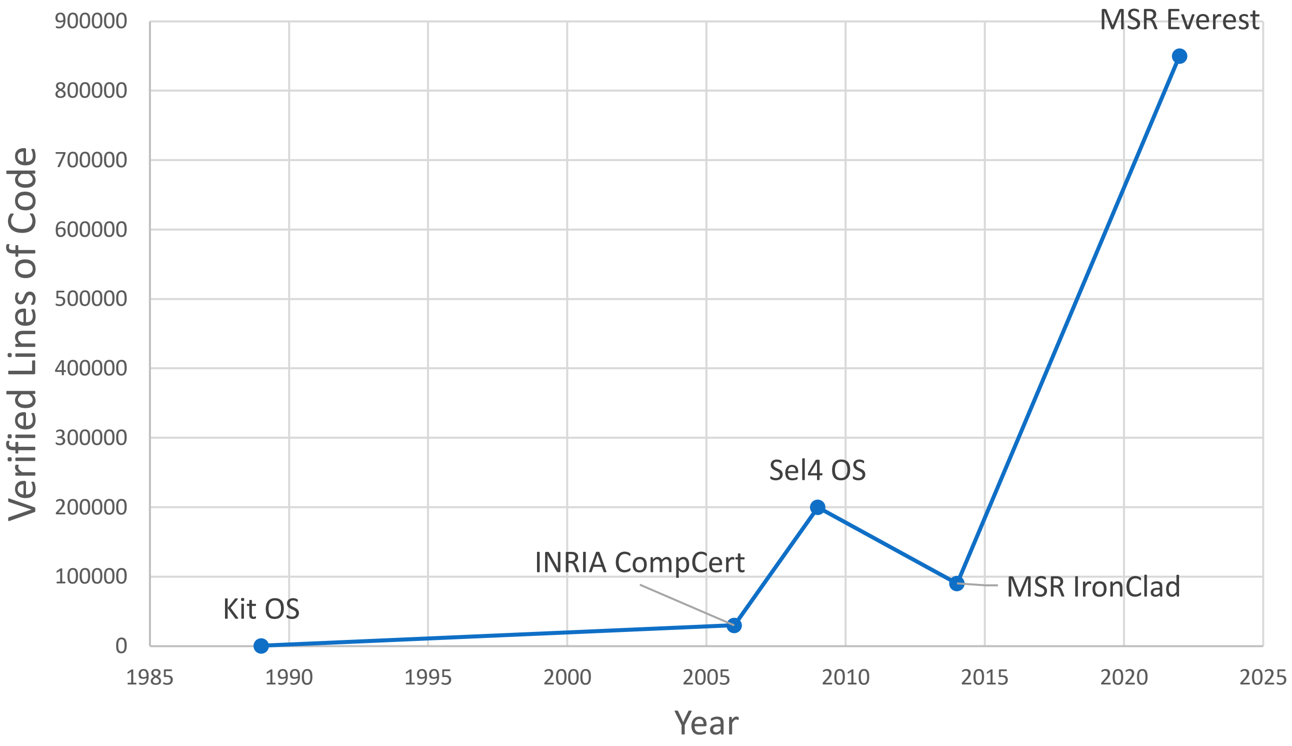
Correctness specification

But, dependent types alone are not enough for high-performance, effectful programming
How to specify and prove programs using mutable state, concurrency, distribution, … ?

Many DSLs embedded in F* for proofs of effectful programs



Deployments of artifacts proven in F*



~850,000 and growing lines of verified code by 50+ developers

Under CI, built on every push

High-assurance software components:
parsers and serializers, standardized firmware (DICE), cryptographic libraries, …

Microsoft Hyper-v

WireGuard — FAST, MODERN, SECURE VPN TUNNEL

Signal

eBPF

Proof-oriented parsers in Hyper-V have been in production for about 2 years now

Every network message passing through the Azure platform is first parsed by EverParse code

But:

Reasoning about mutable state and heaps etc. is heavily reliant on SMT solving

- Proofs can be brittle, requires a lot of hand-holding of the solver

All our deployed code is inherently sequential

- Some exceptions, e.g., SIMD crypto etc.

```
#push-options "–z3rlimit 100"
let h0 = HST.get () in
HST.push_frame ();
let hs0 = HST.get () in
B.fresh_frame_modifies h0 hs0;
let deviceID_priv: B.lbuffer byte_sec 32 = B.alloca (u8 0x00) 32ul in
let hs01 = HST.get () in
let authKeyID: B.lbuffer byte_pub 20 = B.alloca 0x00uy 20ul in
let hs02 = HST.get () in
let _h_derive_deviceID_pre = HST.get () in
B.modifies_buffer_elim cdi  B.loc_none h0 _h_derive_deviceID_pre;
B.modifies_buffer_elim fwid B.loc_none h0 _h_derive_deviceID_pre;
B.modifies_buffer_elim deviceID_label B.loc_none h0 _h_derive_deviceID_
B.modifies_buffer_elim deviceID_label B.loc_none h0 _h_derive_deviceID_
derive_deviceID
  (cdi) (fwid)
  (deviceID_label_len) (deviceID_label)
  (aliasKey_label_len) (aliasKey_label)
  (deviceID_pub) (deviceID_priv)
  (aliasKey_pub) (aliasKey_priv)
  (authKeyID);
let _h_derive_deviceID_post = HST.get () in
B.modifies_trans B.loc_none h0 _h_derive_deviceID_pre (
  B.loc_buffer deviceID_pub  `B.loc_union`
  B.loc_buffer deviceID_priv `B.loc_union`
  B.loc_buffer aliasKey_pub  `B.loc_union`
  B.loc_buffer aliasKey_priv `B.loc_union`
  B.loc_buffer authKeyID
) _h_derive_deviceID_post;
let _h_step2_pre = _h_derive_deviceID_post in
```

Low*

What would a general-purpose programming language with concurrent separation logic at its core look like?

Rust: Borrows ideas from linear types and CSL

- Linearly typed, systems programming can work in practice
- But, focuses on syntactic checks for safety and race freedom, not program correctness
- (And others before it, Cyclone, Mezzo, …)

CSL as a logic for modular reasoning about effectful programs

- E.g., Iris, building on many prior logics
- Iris and most other CSLs focus on doing proofs of programs, *after the fact*

Can we use CSL to also structure the construction of programs?

- To ease proofs; for proofs & specs to guide programming
- *Proof-oriented programming*

A first attempt: Steel

ICFP ‘20: We presented SteelCore

- A CSL embedded shallowly in F*
- With a fully formalized proof of soundness
- An impredicative separation logic, with dynamically allocated invariants
- Dependently typed, because it’s embedded in F*

ICFP ’21: Steel, a surface language for programming in SteelCore

- Implemented using combinators and tactics in F*
- Packaged using F*’s effect system
- But even we found it hard to use
- Hacking a dependent type checker to mimic a separation logic checker … hard to make it work well

Pulse

A general-purpose programming language with a CSL-based dependent, type system

Custom syntax, via F* support for syntax extension, a superset of F*'s

Embedded within F* and gaining from:

- Reusing a lot of the core machinery developed for dependently typed programming
- Pulse checker handles separation logic connectives (mainly *); F* takes care of all pure reasoning
- A foundational semantics formalized in F* (SteelCore, ICFP '20)
- SMT & tactic-based automation
- Reusing libraries for functional programming and proving already in F*
- With a checker implemented in F* itself with an (in-progress) proof of correctness

Supporting

- Mutable state, concurrency, asynchrony, etc.
- Extensibility with assurance, using F* to formalize new features
- Deployability, with extraction to OCaml and C (like F* and Low*), and now also Rust

With inspiration from Rust, Iris, Hoare Type Theory, Verifast, Viper, …

A taste of Pulse

```
fn incr (x:ref int)
requires pts_to x 'i
ensures pts_to x ('i + 1)
{
    let v = !x;
    x := v + 1;
}
```

Separation logic annotations

Imperative syntax

```
fn par_incr (x y:ref int)
requires pts_to x 'i ** pts_to y 'j
ensures pts_to x ('i + 1) ** pts_to y ('j + 1)
{
   par (fun _ -> incr x)
       (fun _ -> incr y)
}
```

Pulse Architecture

Certified Libraries

(refs, arrays, locks, …)

Pulse DSL

```
fn incr (x:ref int)
requires pts_to x 'i
ensures pts_to x ('i + 1)
{
    let v = !x;
    x := v + 1;
}
```

Pulse Typechecker

Programmed in Meta-F*

Certified* to produce
well-typed derivations
in SteelCore

Extraction Backends

OCaml

C

The Rust Programming Language

Proofs erasure

A Certified Checker for Pulse (in progress)

pulse_checker elaborates a term, computing its type, and building a Pulse typing derivation, or fails with an error

```
val pulse_checker (g:env) (src:tm)
: Tac (e:tm & c:ty & pulse_typing g e c)
```

A theorem in F* to show that Pulse typing derivations can be elaborated into F* typing derivations

```
val pulse_typing_sound (d:pulse_typing g e c)
: fstar_typing g (elab d) (elab_ty c)
```

`fstar_typing`: *A specification of F*'s typing judgment as an inductive type in Meta F**

Pulse is fresh!

We’ve been working on it for the past year or so, and just made our first release for this tutorial

You’ll be the first to use Pulse, aside from us and our interns this past summer

Many things can be improved …

Looking to get early feedback from the community

Pulse is also active and real

We've been building two new systems & libraries in it, targeting production usage:

- A verified, measured boot protocol standard called DICE Protection Engine
- A new library for parsing and serialization of CBOR formats

And have other projects building on Pulse in the works

We'd be happy for you to consider using it in your research

- E.g., to build verified systems; or, on the PL side, to build new CSL abstractions, proof procedures etc.

We also have been writing quite a lot of documentation and code samples

- Maybe you'd be interested to use it in teaching, e.g., for a grad level PL or verification seminar

Today's tutorial

A quick tour of Pulse
Cherry-picking our way through the online docs
Read along, interact with code samples

https://fstar-lang.org/tutorial/book/index.html

Session 1: Pulse basics

- Basic connectives of the logic, references, arrays, loops, etc.

Session 2: Concurrency

- Atomics, invariants, spin locks, ghost state, parallel increment

Proof-Oriented Programming in F*

Search docs

CONTENTS:

Pulse: Proof-oriented Programming in Concurrent Separation Logic

Many F* projects involve building domain-specific languages with specialized programming and proving support. For example, Vale supports program proofs for a structured assembly language; Low* provides effectful programming in F* with a C-like memory model; EverParse is a DSL for writing low-level parsers and serializers. Recently, F* has gained new features for building DSLs embedded in F* with customized syntax, type checker plugins, extraction support, etc., with *Pulse* as a showcase example of such a DSL.

Pulse is a new programming language embedded in F*, inheriting many of its features (notably, it is higher order and has dependent types), but with built-in support for programming with mutable state and concurrency, with specifications and proofs in Concurrent Separation Logic.

As a first taste of Pulse, here's a function to increment a mutable integer reference.

```
fn incr (x:ref int)
requires pts_to x 'i
ensures pts_to x ('i + 1)
{
    let v = !x;
    x := v + 1;
}
```

And here's a function to increment two references in parallel.

```
fn par_incr (x y:ref int)
requires pts_to x 'i ** pts_to y 'j
ensures pts_to x ('i + 1) ** pts_to y ('j + 1)
{
   par (fun _ -> incr x)
       (fun _ -> incr y)
}
```

You may not have heard about separation logic before—but perhaps these specifications already make intuitive sense to you. The type of `incr` says that if "x points to 'i" initially, then when `incr` returns, "x points to 'i + 1"; while `par_incr` increments the contents of `x` and `y` in parallel by using the `par` combinator.

Concurrent separation logic is an active research area and there are many such logics out there, all https://steelpulse.com/

Pulse basics

````
// regular F* code
let fstar_five : int = 5


```pulse
fn five ()
  requires emp
  returns x:int
  ensures pure (x == 5)
{
    fstar_five
}
```
````

Separation logic triples:

$$\{\, P \,\}\, s \,\{\, \lambda x.\, Q \,\}$$

where P and Q are heap predicates of type `vprop`

`five` can be seen as a proof for the triple

$\{\, emp \,\}$ `fstar_five` $\{\, \lambda x.\, pure\,(x == 5) \,\}$

Separating conjunction

$P \ {**}\ Q$ Specifies a state that can be split into two disjoint fragments satisfying P and Q resp.

- Commutative
- Associative
- `emp` is the left and right identity

Frame reasoning rule:

If $\{\ P\ \}\ s\ \{\ Q\ \}$ then $\{\ P\ \ {**}\ R\ \}\ s\ \{\ Q\ \ {**}\ R\ \}$

Allows for modular and compositional reasoning. R is called the frame

Exercise

Example: compare two arrays

An imperative algorithm that loops over the two arrays comparing them pointwise

As soon as there is an index where the arrays are not equal, return false

Otherwise, if the arrays are equal at all the indices, return true

But we need some more machinery to write this …

While loop

```
// while loop with loop invariant

while (cond) invariant b. p { body }
```

$$\frac{\{\, \exists\, b.\, inv\ b \,\}\ c\ \{\, \lambda b.\, inv\ b \,\} \qquad \{\, inv\ true \,\}\ s\ \{\, \exists\, b.\, inv\ b \,\}}{\{\, \exists\, b.\, inv\ b \,\}\ while\ c\ inv\ s\ \{\, inv\ false \,\}}$$

Example: compare two arrays

Walkthrough in vscode

Extraction

Pulse programs can be extracted to Rust, C, and OCaml

Each backend providing different characteristics

Rust backend:

Immutable and mutable refs
Arrays and slices
Generics

Need to be aware of the ownership discipline

C backend:

Goes through the Karamel tool
Monomorphization etc.
Pulse -> C is similar to Low* -> C

OCaml backend:

Most general

Uses the OCaml GC
No explicit memory management
E.g., `let mut` become heap allocations

A little background on user-defined predicates

Concurrency in Pulse

Three kinds of computations

General purpose computations that can loop forever

- This is what we've mainly seen so far

Ghost computations, always terminate, used for specifications and proofs

- We saw a bit of this, e.g., with v. assert; introduce exists* etc.
- But, ghost computations can do more, e.g., read and write ghost state

Atomic computations: Computations that are guaranteed to be executed in a single-step without interruption by other threads.

General purpose computations

General purpose computations that can loop forever

```
fn do_something()
requires p
returns x:t
ensures q
```

Pulse computation type

```
stt (a:Type) (p:vprop) (q:a -> vprop) : Type
```

```
val do_something (_:unit) : stt t (requires p) (ensures fun x -> q)
```

Ghost computations

```
ghost
fn share #t #p (x:ref t)
requires pts_to x #p ‘v
ensures pts_to x #(half_perm p) ‘v ** pts_to x #(half_perm p) ‘v
```

```
stt_ghost (t:Type) (p:vprop) (q:t -> vprop) : Type
```

Ghost computation type

Ghost computations

```
ghost
fn gather #t #p1 #p2 (x:ref t)
requires pts_to x #p1 ‘v1 ** pts_to x #p2 ‘v2
ensures pts_to x #(sum_perm p1 p2) ‘v1 ** pure (‘v1 == ‘v2)
```

Ghost state

```
module GR = Pulse.Lib.GhostReference

ghost
fn alloc (x:t)
requires emp
returns r:GR.ref t
ensures GR.pts_to r #full_perm x
```

```
ghost
fn ( ! ) …

ghost
fn ( := ) …

ghost
fn share, gather, free
```

Ghost PCM Reference

Actually, for any partial commutative monoid p:pcm a

Can create ghost state that contains elements of that pcm

With frame-preserving updates

Pulse.Lib.GhostReference: a special case using the PCM of fractional permissions

Atomic computations

```
atomic
fn atomic_read (r:ref U32.t)
requires pts_to x #p ‘v
returns x:U32.t
ensures
 pts_to x #p ‘v ** pure (x == ‘v)
```

```
atomic
fn cas (r:ref U32.t) (m n:U32.t)
requires pts_to x ‘v
returns b:bool
ensures exists* k.
   pts_to x k **
   pure (if b
         then k == n /\
             ‘v == m
         else k == ‘v)
```

Composing different kinds of computations

Ghost computations compose on the left and right with atomic computations, so long as the ghost computation result is non-informative

- `stt_ghost unit ; stt_atomic a    : stt_atomic a`
- `stt_atomic _    ; stt_ghost unit : stt_atomic unit`
- `stt_ghost bool ; stt_atomic a` ← not allowed; since the Boolean result is erased at runtime

Ghost also composes with `stt` in the same way

Atomic computations can be lifted to general purpose computations

`stt_atomic a <: stt a`

** There's a fourth kind of computation: `stt_unobservable`, but we'll not cover that today*

Why distinguish atomic computations?

They have one unique privilege:

Atomic computations can open invariants

```
atomic
fn f (…)
requires p
returns x:a
ensures q
opens i
```

```
stt_atomic (a:Type) (i:inames) (p:vprop) (q: a -> vprop)
```

The invariants that an atomic computation may open

Invariants

From Pulse.Lib.Core:

```
val inv (p:vprop) : Type

val name_of_inv #p (i:inv p) : iname
```

`inv p`: The type of an *invariant*

- A token which guarantees that p is true in the current state and in all future states of the program
- Every invariant has a *name*, `name_of_inv i`

Creating an invariant

```
fn new_invariant (p:vprop)
requires p
returns i:inv p
ensures emp
```

If p is true now, turn it into an invariant to make it true forever

But, lose permission on p

new_invariant is unobservable

Using an invariant

An atomic computation can:

- Assume the invariant at the start of the computation,
- Must restore the invariant when the step completes

Since atomic computations run in a single, uninterruptible step, this ensures that the invariant always remains valid

When:

```
    i : inv p
    e : stt_atomic t j (p ** r) (fun x -> p ** s x)
    i not-in j
```

e gets to assume p, so long as it also restores p

```
with_invariants i { e }
: stt_atomic t (j U {i}) r s
```

the context does not have to prove p, since i:inv p guarantees that it is valid

But, we have to record that the invariant i was opened

General style of using an invariant

```
{ q }

    with_invariants i,j,k {
      { pi ** pj ** pk ** q }

          zero or more ghost or pure steps;

          zero or one atomic step;

          zero or more ghost or pure steps;

      { pi ** pj ** pk ** s }
    }

{ s }
```

Since ghost computations compose with atomic computations, one can precede and follow atomic step with ghost steps