The Pulse Programming Language

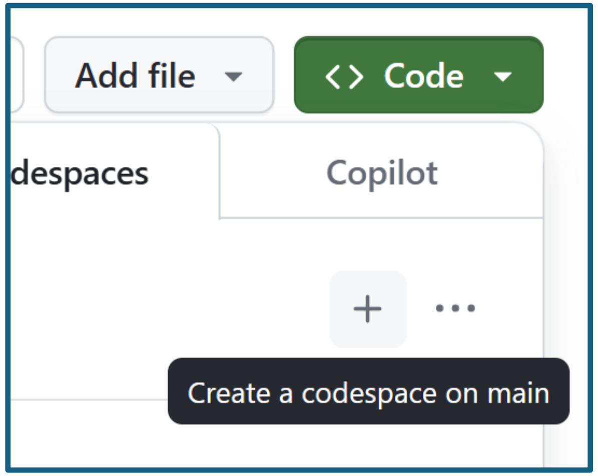
4th Iris Workshop

Aseem Rastogi, Gabriel Ebner, *Guido Martínez*, Nik Swamy, Tahina Ramananandro

Megan Frisella, Thibault Dardinier, and soon Jonas Fiala

June 2024

Elevator Pitch



- Devcontainer available!

  https://tinyurl.com/gopulse

- CSL-based imperative language, embedded in F*
  - F* expressions/types + imperative commands
  - Built over the PulseCore logic
    - Pulse ≈ PulseCore + syntax + automation + extraction
- Pulse has its own typechecker, with “knowledge” about SL
  - The typechecker is itself verified in F* (using Meta-F*)
  - Any well-typed Pulse program represents a well-typed F* program
  - No addition to TCB (modulo extraction)

F* Basics

F*: Proof-oriented Programming

```
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)

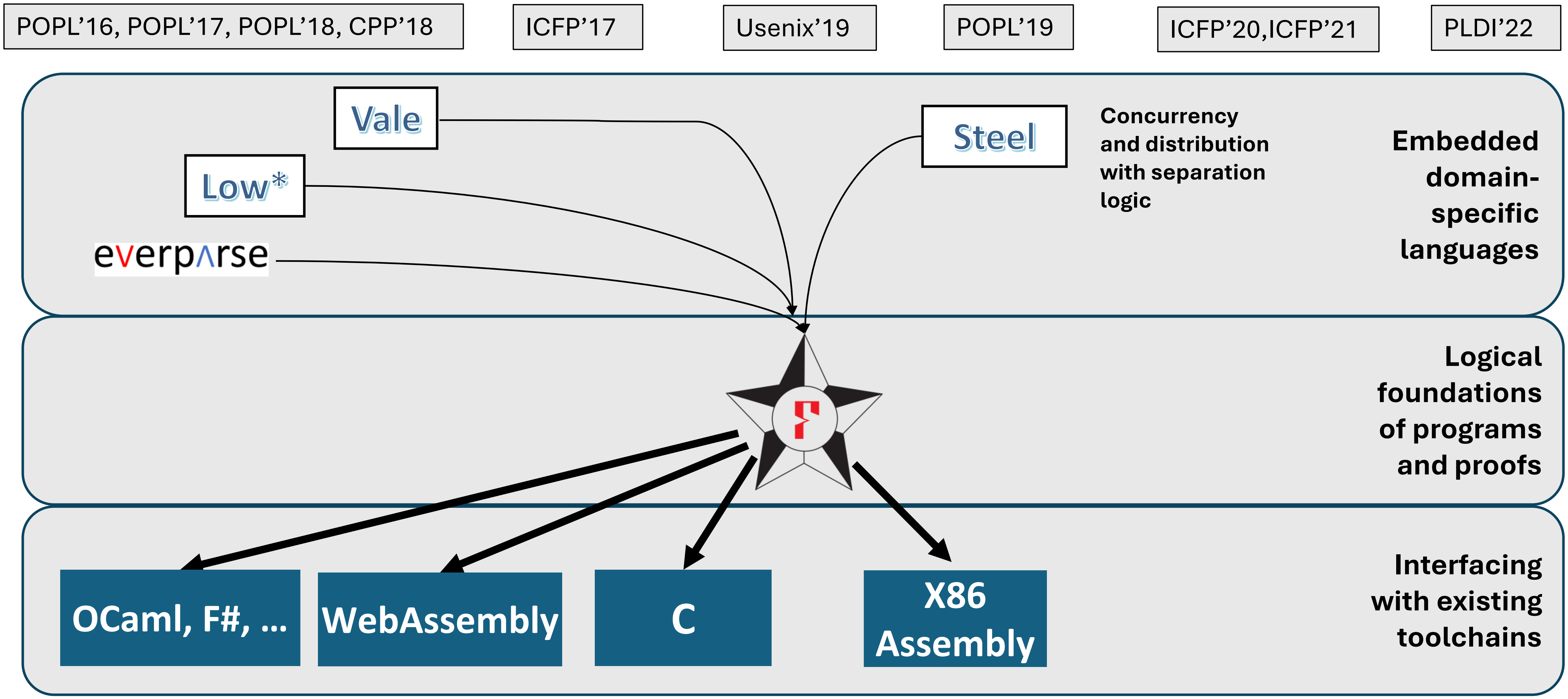
val quicksort : l:list int ->
                m:list int{sorted m && perm l m}
```

Correctness specification

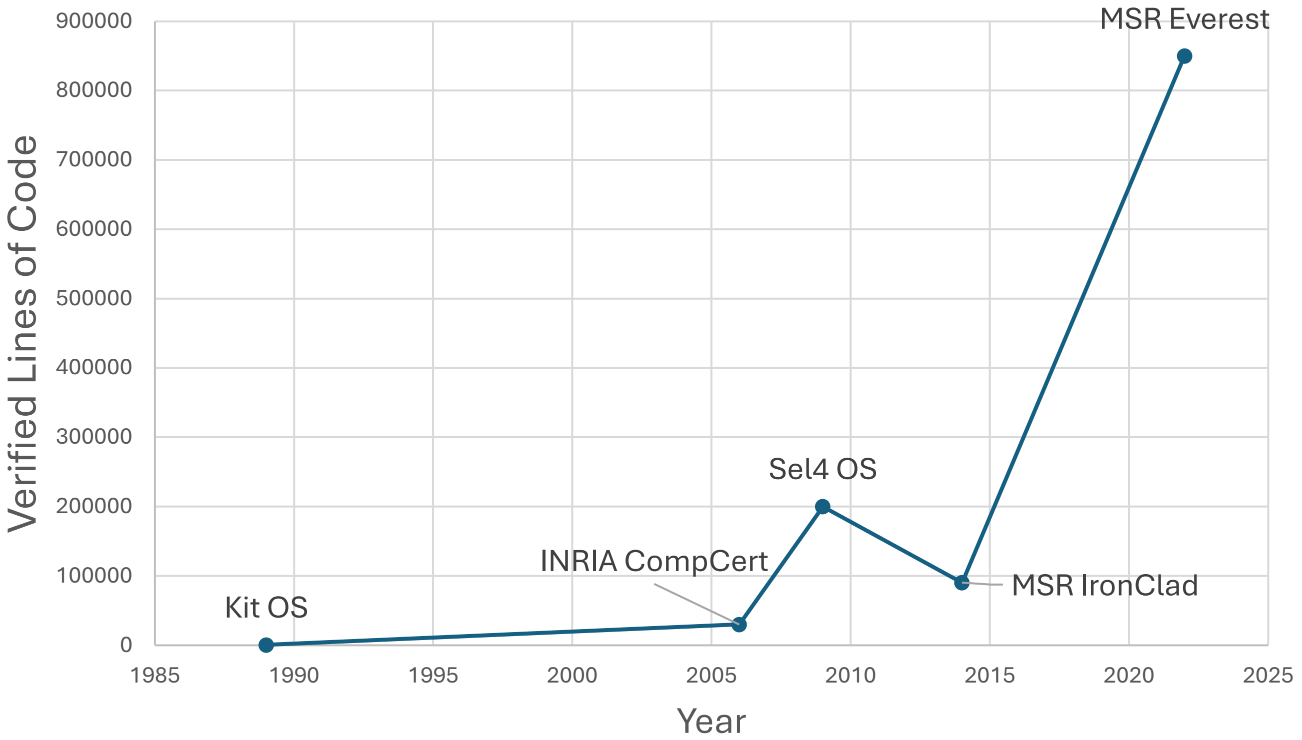
```
let quicksort l = match l with
  | []  -> []
  | [x] -> [x]
  | p::xs   ->
    let l1 = filter (<p) xs in
    let l2 = filter (>=p) xs in
    quicksort l1 @ p :: quicksort l2
```

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

Many F* DSLs for effectful program verification



Deployments of artifacts proven in F*



~1M 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 of our deployed code is inherently sequential
  - Boot firmware: DICE, DPE
  - Parsing: CBOR, CDDL, COSE
  - Crypto primitives: HACL*
  - Device attestation: SPDM, TDISP
  - Some exceptions, e.g., SIMD crypto etc.

Low*

```
#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_pre
B.modifies_buffer_elim deviceID_label B.loc_none h0 _h_derive_deviceID_pre
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
```

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 full correctness (though of course see Verus)
- (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****: dependent types, higher-order, full verification, foundational.*

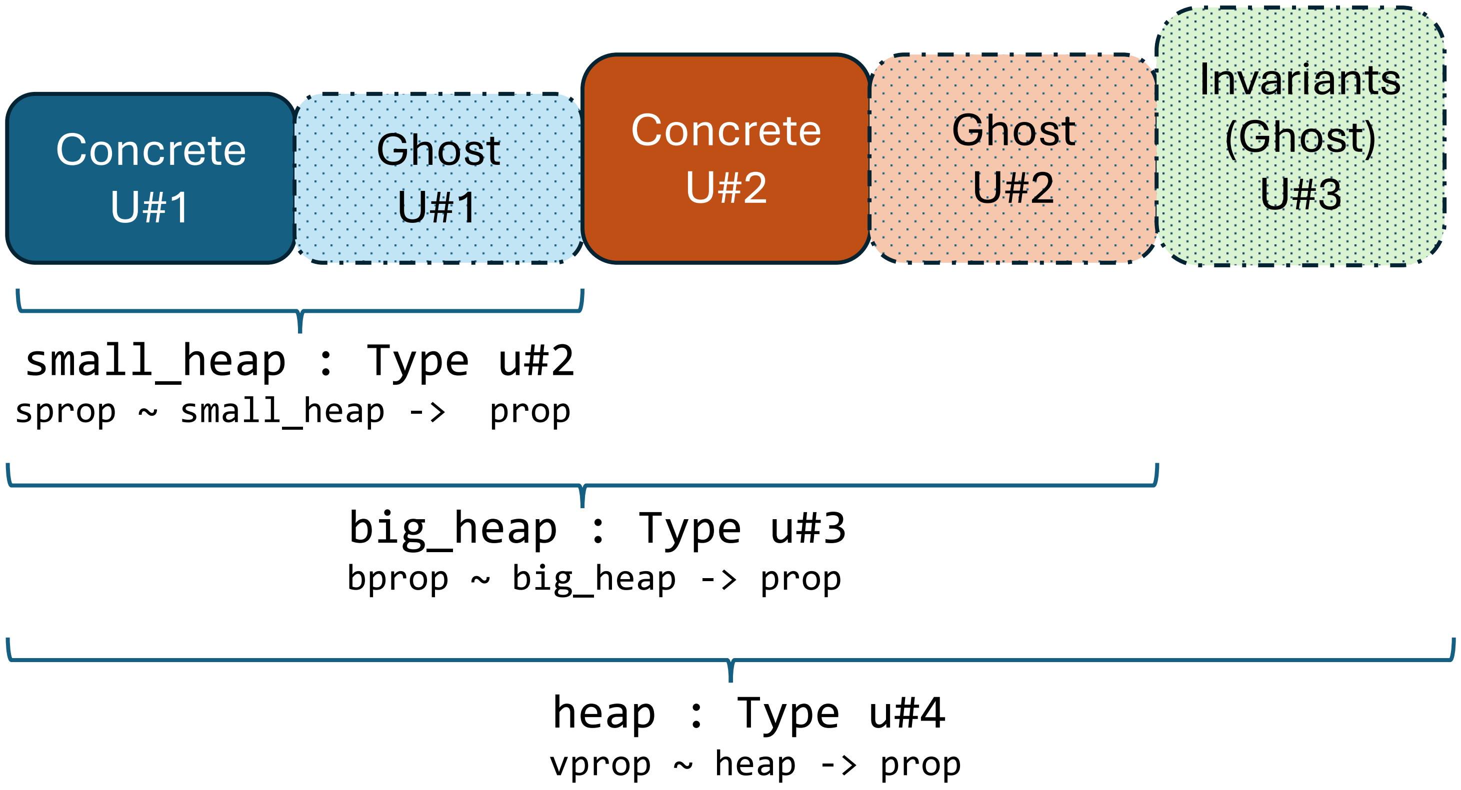
Quick Demo

Basics + lock implementation

PulseCore

A shallowly-embedded, stratified, dependently-typed separation logic

Stratified heaps



A heap is a map from ref names (nats) into values in a given universe

- All cells follow a user-chosen PCM
- Separation logic propositions are shallow
    - A `heap -> prop` predicate, with proper restrictions (affine)
- Support for (predicative) higher-order ghost state
    - big_heap can store sprop
    - vprop is just another F* type

A final heap allows to allocate invariants over the lower heaps

PulseCore computations

- `stt:` a-returning computations, from `pre` to `post`
  - Intrinsically-typed actions trees under the hood
  - Interleaving of atomic actions
  - Partial correctness
- Atomic and ghost variants as well
  - Tracking opened invariants
  - `stt_ghost` is computationally irrelevant all the way down
  - Must be total

```
val stt
  (a:Type u#a)
  (pre:vprop) (post:a -> vprop) : Type0

val stt_atomic
  (a:Type u#a)
  (opens:inames)
  (pre:vprop) (post:a -> vprop) : Type u#(max 4 a)

[@@ erasable]
val stt_ghost
  (a:Type u#a)
  (opens:inames)
  (pre:vprop) (post:a -> vprop) : Type u#(max 4 a)
```

Invariants

- Somewhat similar to Iris

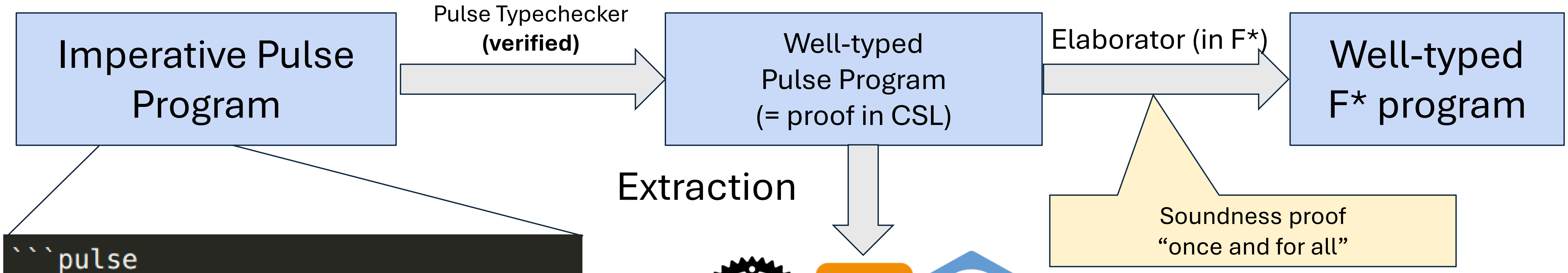
```
val new_invariant (p:vprop { is_big p })
: stt_ghost iref emp_inames p (fun i -> inv i p)
```

- Invariants can only be created over “big” vprops, cannot contain other invariants
- Some consequences:
  - Predicativity: for instance, locks cannot protect locks (unless it’s a “higher lock”)
  - No nesting of invariants

The Pulse Typechecker

A certified separation-logic type-checker

Architecture




```
```pulse
fn swap (#t: Type) (a b: ref t)
  requires pts_to a x ** pts_to b y
  ensures pts_to a y ** pts_to b x
{
  let va = !a;
  let vb = !b;
  a := vb;
  b := va;
  ()
}
```
```

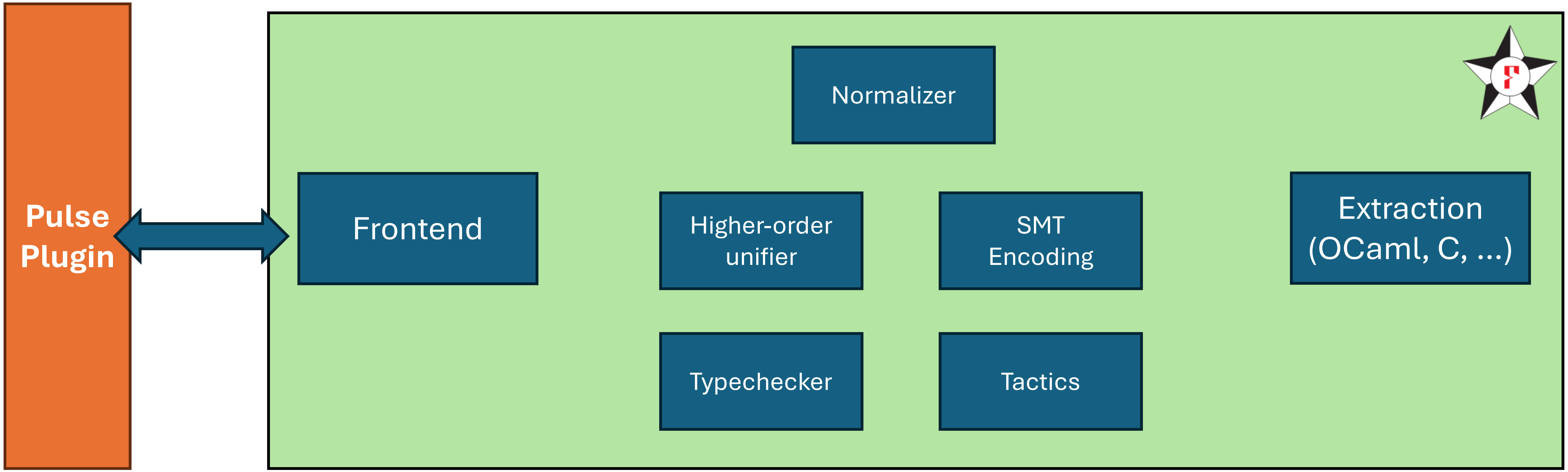
Assertions in SL

Imperative Syntax

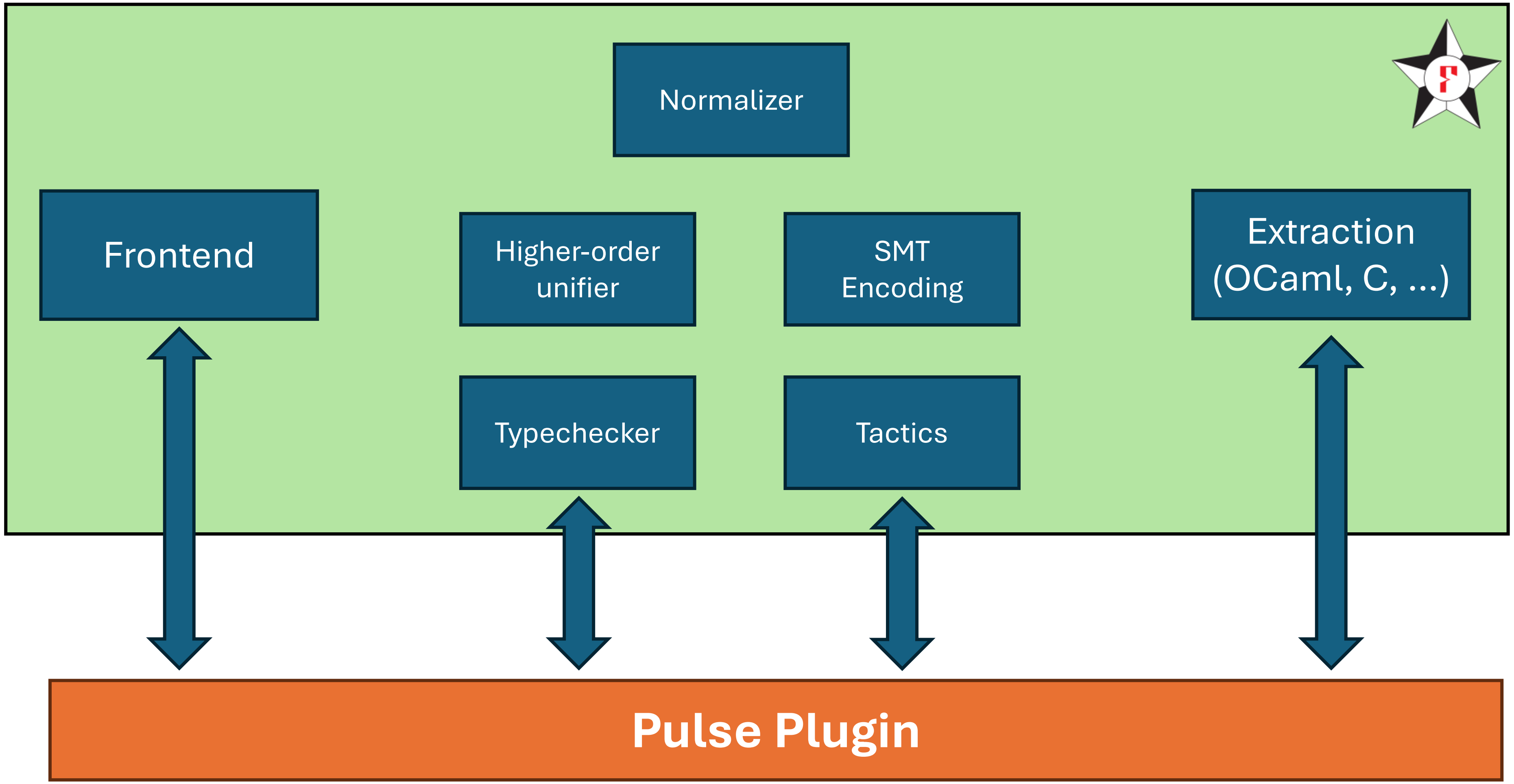
Steel: Proof-oriented Programming in a Dependently Typed Concurrent Separation Logic

AYMERIC FROMHERZ, Carnegie Mellon University, USA
ASEEM RASTOGI, Microsoft Research, India
NIKHIL SWAMY, Microsoft Research, USA
SYDNEY GIBSON, Carnegie Mellon University, USA
GUIDO MARTÍNEZ, CIFASIS-CONICET, Argentina
DENIS MERIGOUX, Inria Paris, France
TAHINA RAMANANANDRO, Microsoft Research, USA

F* as a library



F* as a library



F* Typing Reflection

```
type typing : env -> term -> comp_typ -> Type0 =
  | T_Abs :
    g:env ->
    x:var { None? (lookup_bvar g x) } ->
    ty:term ->
    body:term { ~(x `Set.mem` freevars body) } ->
    body_c:comp_typ ->
    u:universe ->
    pp_name:pp_name_t ->
    q:aqualv ->
    ty_eff:T.tot_or_ghost ->
    typing g ty (ty_eff, tm_type u) ->
    typing (extend_env g x ty) (open_term body x) body_c ->
    typing g (pack_ln (Tv_Abs (mk_binder pp_name 0 ty q) body))
             (T.E_Total,
              pack_ln (Tv_Arrow (mk_binder pp_name 0 ty q)
                                (mk_comp (close_comp_typ body_c x))))

  | T_App :
    g:env ->
    e1:term ->
    e2:term ->
    x:binder ->
    t:term ->
    eff:T.tot_or_ghost ->
    typing g e1 (eff, pack_ln (Tv_Arrow x (mk_comp (eff, t)))) ->
    typing g e2 (eff, binder_sort x) ->
    typing g (pack_ln (Tv_App e1 (e2, binder_qual x)))
             (eff, open_with t e2)
```

```
val mkfoo : g:env -> Tac (tm:term &
                          ty:typ{typing env tm ty})

%splice_t [foo] (mkfoo)
```

- Metaprogram the term ***and*** its typing derivation at once.
  - No re-checking needed.
  - Especially useful in an *extensional* type theory
- Roll your own syntax, typechecker, elaborator
  - Verified or not!

The Pulse checker

- Checker: Pulse AST ~> Pulse Typing derivation
  - The typing judgment is defined by the "user"
- The *prover* takes care of proving preconditions of each statement
  - It can solve unification variables from the separation logic context (which F* cannot/should not do)
  - Eliminates existentials from context, introduces them in goals (when needed). Same for pure resources.
  - Calls SMT as needed.
- All of this with a (WIP) proof of soundness

````
val ( ! ) (#a:Type) (r:ref a) (#n:erased a) (#p:perm)
  : stt a
        (pts_to r #p n)
        (fun x -> pts_to r #p n ** pure (reveal n == x))

val ( := ) (#a:Type) (r:ref a) (x:a) (#x0:erased a)
  : stt unit
        (pts_to r x0)
        (fun _ -> pts_to r x)

```pulse
fn make_even (r:ref int)
  requires
    exists* v. pts_to r v
  ensures
    exists* v. pts_to r v ** pure (v % 2 == 0)
{
  let v = !r;
  r := 2 * v;
}
```
````

Eliminate existential, instantiate implicit

Instantiate, prove pre and frame rest, obtain pure

Introduce existential, prove pure

Matcher and Toggles

- The *matcher* is the part of the checker that shuffles resources between context and preconditions, and computes frames.
  - It can match resources in many ways, including proving them equal via an SMT call.
  - SMT calls are expensive: the user gets to choose matching strategy
- Also, some WIP on user-extensible automation
  - E.g. share/gather
  - Hopefully via the typeclass system

```
(* only use term_eq *)
val equate_syntactic : unit
(* only use unifier, without zeta *)
val equate_strict    : unit
(* allow up to full SMT queries *)
val equate_by_smt    : unit
```

```
val pts_to
  (#a:Type)
  ([@@@equate_strict] r:ref a)
  (#[T.exact (`1.0R)] p:perm)
  (n:a) : vprop
```

Task-parallelism in Pulse

Quicksort

¿Can we parallelize it?

```
fn rec quicksort (a : array int) (lo hi : nat)
  requires a[lo..hi] |-> s
  ensures a[lo..hi] |-> sort s
{
  if (hi – lo < 2) return;
  let (p,q) = partition a lo hi;
   { a[lo..p] |-> s1 ** a[q..hi] |-> s2 ** ... }
  quicksort a lo p;
   { a[lo..p] |-> sort s1 ** a[q..hi] |-> s2 ** ... }
  quicksort a q hi;
   { a[lo..p] |-> sort s1 ** a[q..hi] |-> sort s2 ** ... }
  quicksort_lemma a lo hi p;
   { a[lo..hi] |-> sort s }
}
```

Quicksort... with **tasks**?

```
fn rec t_quicksort (p : pool)
        (a : array int) (lo hi : nat)
  requires a[lo..hi] |-> s
  ensures a[lo..hi] |-> sort s
{
  if (hi – lo < 2) return;
  let (p,q) = partition a lo hi;
  spawn p { t_quicksort a lo p; };
  t_quicksort a q hi;
}
```

```
fn quicksort (a : array int)
                    (lo hi : nat)
{
  let p = setup_pool (nproc());
  t_quicksort p a lo hi;
  wait_pool p
}
```

Pledges: reasoning about the future

Resource

d ~> v

"when *d* holds, you can trade in this pledge to ***also*** get *v*"

```
ghost fn redeem_pledge (d v : vprop)
  requires d ** d ~> v
  ensures d ** v
```

```
ghost fn return_pledge (d v: vprop)
  requires v
  ensures d ~> v
```

```
ghost fn join_pledges (d v1 v2 : vprop)
  requires d ~> v1 ** d ~> v2
  ensures d ~> (v1 ** v2)
```

```
ghost fn split_pledge (d v1 v2 : vprop)
  requires d ~> (v1 ** v2)
  ensures d ~> v1 ** d ~> v2
```

```
ghost fn squash_pledge (d v: vprop)
  requires d ~> (d ~> v)
  ensures d ~> v
```

... and others ...

- New connective
- Fully verified in Pulse, no axioms

Task pool

```
fn setup_pool (n : nat)
  requires emp
  returns p:pool
  ensures alive p

fn spawn (p : pool)
         (f : unit -> stt unit pre post)
  requires alive p ** pre
  ensures done p ~> post

fn stop_pool (p : pool)
  requires alive p
  ensures done p
```

Roughly inspired by OCaml5's TaskPool

- Also similar to Cilk, OpenMP, etc.

In the actual implementation…

- **Join** for tasks
- Tasks can return values
- Pool can be shared
- **Implemented and verified**, not just an interface
  - With some caveats… no impredicativity
  - If you're interested let's talk?

Quicksort with **tasks**!

```
fn rec t_quicksort (p : pool)
        (a : array int) (lo hi : nat)
  requires a[lo..hi] |-> s
  ensures done p ~> a[lo..hi] |-> sort s
{
  if (hi – lo < 2) return;
  let (p,q) = partition a lo hi;
  { a[lo..p] |-> s1 ** a[hi..q] |-> s2 ** ... }
  spawn p { t_quicksort a lo p; };
 { done p ~> (done p ~> a[lo..p] |-> sort s1)
   ** a[hi..q] |-> s2 ** ... }
  t_quicksort a q hi;
 { done p ~> (done p ~> a[lo..p] |-> sort s1)
   ** done p ~> (a[hi..q] |-> sort s2) ** ... }
  squash_pledge ...;
  join_pledge ...;
  under (done p) (quicksort_lemma a lo hi p);
  { done p ~>a[lo..hi] |-> sort s }
}
```

```
fn quicksort (a : array int) (n : nat)
 requires a |-> s
  ensures a |-> sort s
{
  let p = setup_pool (nproc());
  { alive p }
  t_quicksort p a lo hi;
  { alive p ** done p ~> a[0..n] |-> sort s }
  wait_pool p;
  { done p ** done p ~> a[0..n] |-> sort s }
  redeem_pledge (done p) (a |-> sort s)
}
```

I lied a bit: we also need to split and track permissions to the pool. Doable but very boring!

GPU Kernels in Pulse

GPU kernel programming (very green)

- Landscape is complicated
  - Cuda,cuBLAS, Intel MKL, Py${TOOL}, MSSCL
  - Database implementations

- A **safe** GPU kernel programming language
  - No footguns: cannot read uninitialized memory, no data races
  - For the brave, support to do functional verification
  - No liveness nor reasoning about performance (for now?)

- Extending separation logic to model GPU device and memory
  - Device vs host pointers and references
  - Access is restricted (GPU cannot in general access CPU memory)
  - A *mode* system for the logic, each function runs in CPU or GPU

GPU kernel programming (very green)

```
(* Token for being in CPU code *)
val cpu : vprop

(* Token for being in GPU code *)
val gpu : vprop

val launch_kernel_1
  (#pre : vprop)
  (#post : vprop)
  (f : unit -> stt unit (gpu ** pre) (fun _ -> gpu ** post))
  : stt unit (cpu ** pre) (fun _ -> cpu ** post)

val launch_kernel_n
  (nthr : pos)
  (#pre  : (tid:nat{tid < nthr} -> vprop))
  (#post : (tid:nat{tid < nthr} -> vprop))
  (f : ((tid:nat{tid < nthr}) ->
        stt unit (gpu ** pre tid) (fun _ -> gpu ** post tid)))
  : stt unit (cpu ** bigstar 0 nthr pre)
             (fun _ -> cpu ** bigstar 0 nthr post)
```

- Abstract resources (cpu/gpu) encode the mode
  - Not ideal, want a mode system
- Kernel calls allow to call gpu code from cpu
  - Not the other way around
- An n-way kernel call requires n-way split of pre and post

GPU allocation and data movement

```
val gpu_pts_to_array
  (#a:Type u#0)
  (#sz:nat)
  (x:gpu_array a sz)
  (#[exact (`1.0R)] f : perm)
  (v : seq a)
: vprop
```

Resource encoding that array x is live, and pointing to a given sequence v. Fractional permissions allow to share the array for read-only access.

Types indicate where the data lives

```
fn gpu_array_alloc
  (#a:Type u#0)
  (sz:nat)
  requires cpu
  returns  x : gpu_array a sz
  ensures  exists* (s:seq a). cpu ** x |-> #1.0R s
```

Uninitialized allocation: called from CPU, you get back a gpu array pointer for *some* sequence s.

GPU allocation and data movement

```
fn memcpy_host_to_device
  (arr : array a)
  (#f : perm)
  (#v : erased (seq a))
  (garr : gpu_array a sz)
  (#gv : erased (seq a))
  requires cpu ** arr |-> #f v ** garr |-> #1.0R gv
  ensures  cpu ** arr |-> #f v ** garr |-> #1.0R v

fn memcpy_device_to_host
  (arr : array a)
  (#f : perm)
  (#v : erased (seq a))
  (garr : gpu_array a sz)
  (#gv : erased (seq a))
  requires cpu ** arr |-> #1.0R v  ** garr |-> #f gv
  ensures  cpu ** arr |-> #1.0R gv ** garr |-> #f gv
```

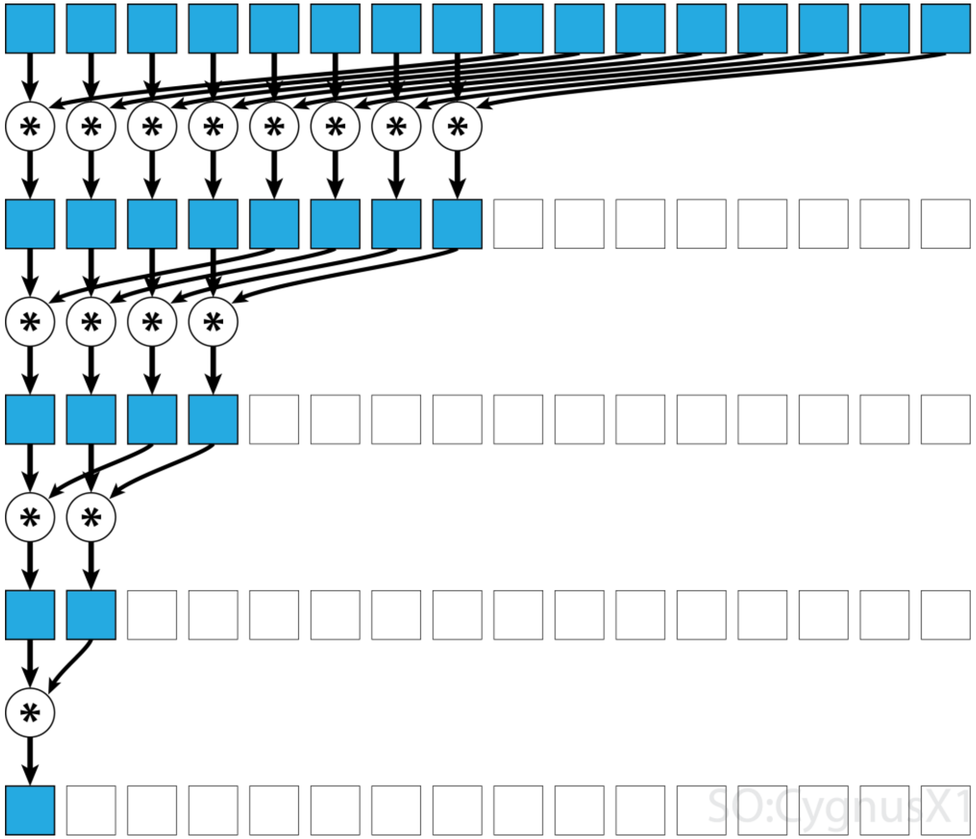
- From CPU code, copy both ways.
- Need full permission on target, only a fraction on source
- Preserve knowledge on contents

Barriers and resource movement

```
val barrier
  (n:nat)
  (p : nat -> vprop)
  (q : nat -> vprop)
  : Type0

fn mk_barrier
  (n : nat)
  (p : nat -> vprop)
  (q : nat -> vprop)
  (pf : unit -> ghost unit (requires bigstar 0 n p) (ensures bigstar 0 n q))
  requires emp
  returns  b : barrier n p q
  ensures  barrier_alive n p q b ** bigstar 0 n (barrier_tok b)

fn barrier_wait
  (#n : nat)
  (#p : nat -> vprop)
  (#q : nat -> vprop)
  (b : barrier n p q)
  (#i : erased nat)
  requires barrier_alive n p q b ** barrier_tok b i ** p i
  ensures  barrier_alive n p q b ** barrier_tok b i ** q i
```



Brief example: dot product

Closing

- Consider trying Pulse out!
  - See https://github.com/FStarLang/pulse
  - Try it in your browser! Or locally with no setup.

- Pulse chapters in F* book
  - Also the best way to get started with F*
  - https://fstar-lang.org/tutorial/
  - I am also teaching a class for undergrads, if you want the materials let me know

- If you're interested in task-parallelism or GPUs let's talk?

Thank You!