Agentic Proof-oriented Programming

Nik Swamy
Microsoft Research

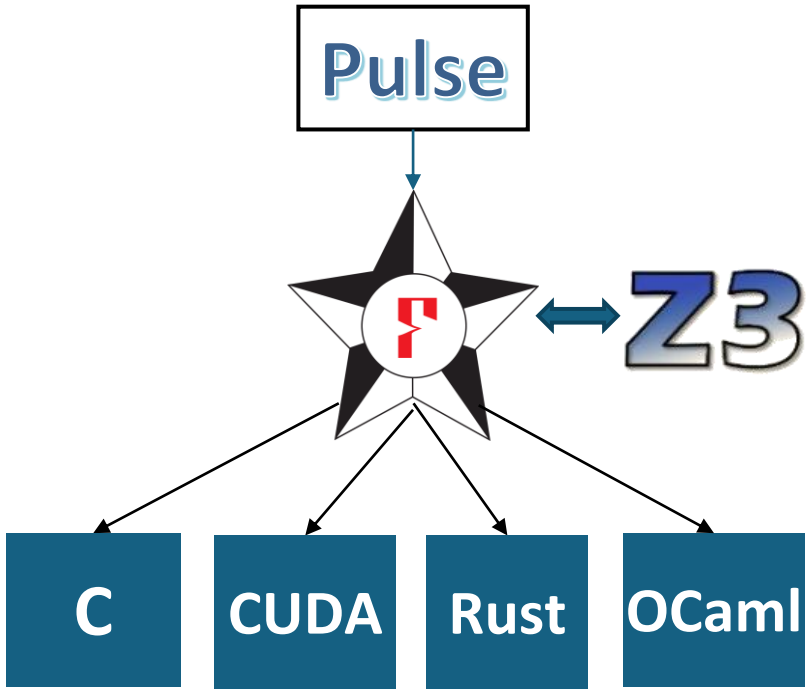
IIT Madras
April 6, 2026

Proof-oriented Programing

- Programs & proofs co-developed, providing mathematical guarantees of system correctness

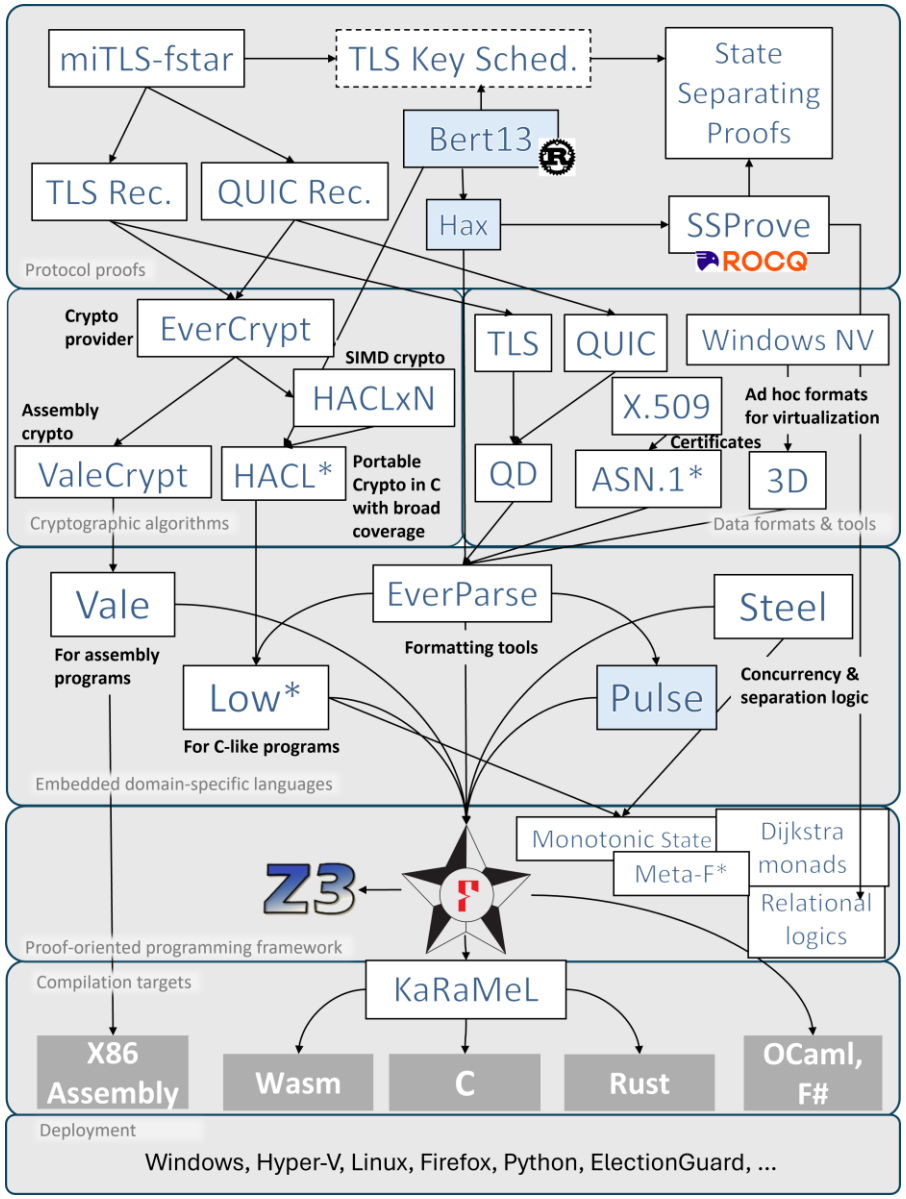
F*: A General-Purpose Proof-oriented Programming Language

Full-fledged programming & proving, with concurrency, shared memory, message passing, higher-order functions, effects, ...




```
fn max (n:SZ.t) (a:array int)
preserves a |-> s
requires pure (length s == n)
return res:SZ.t { res < n }
ensures forall i. i < n ==> s[i] <= s[res]
{
  let mut i : SZ.t = 0sz;
  let mut max : SZ.t = 0sz;
  while (!i < n)
  invariant live i
  invariant live max
  invariant pure (
    !i <= n /\
    !max <= i /\
    (forall (j:nat). j < !i ==> s[j] <= s[!max])
  )
  decreases n - !i
  {
    let v = a.(!i);
    if (a.(!i) > a.(!max)) {
      max := !i;
    }
    i := !i + 1sz;
  };
  !max;
}
```

Project Everest: A Large Stack of Provably Correct Software Built using F*



Including production systems, such as:

Verified data formats:

- CoreOS: vmswitch, http.sys, netvsc, ...
- Hyper-V: ICs, data formats, ...
- Ebpf-for-windows

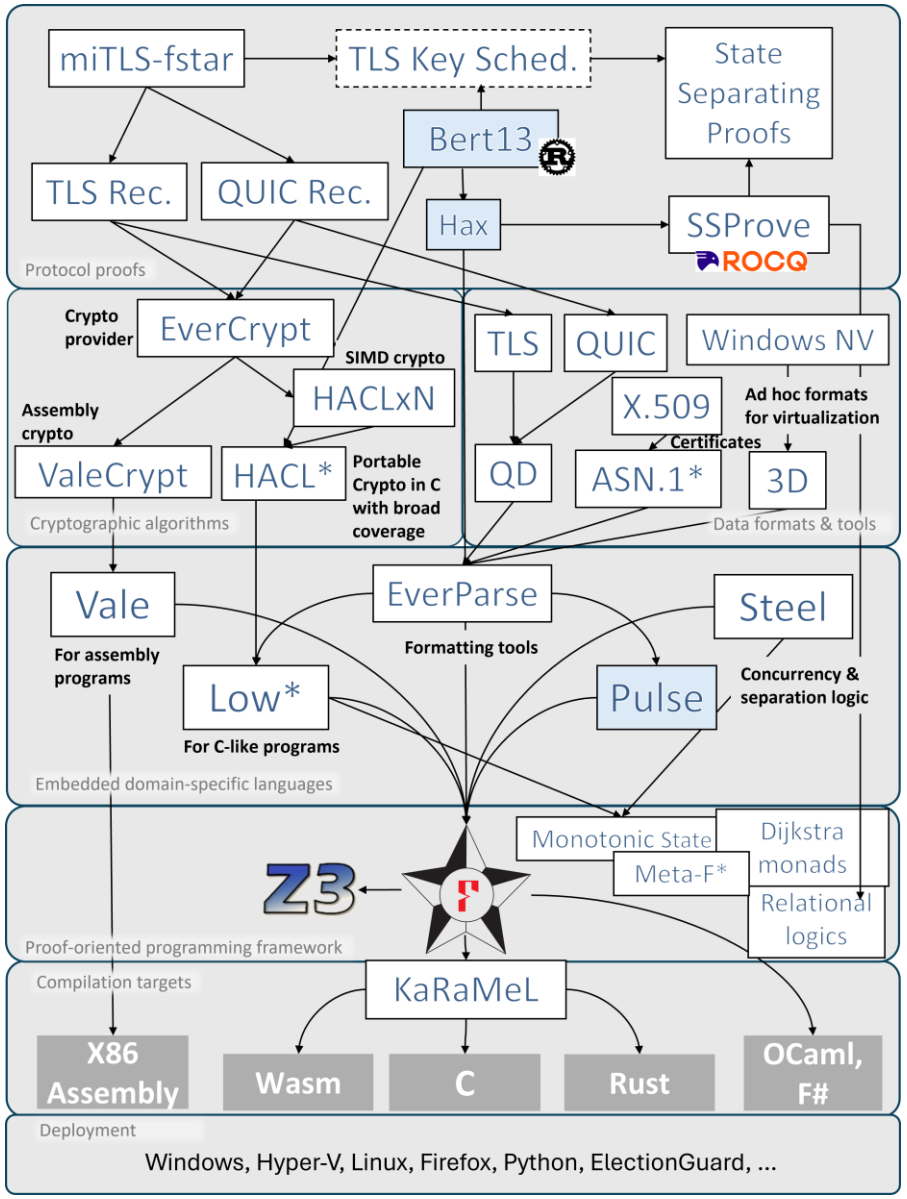
Verified crypto libraries: python boringssl

- Linux, Python, Firefox/NSS, BoringSSL PQ Crypto, ...

Verified HW/SW secure capability systems: FUNGIBLE

- Fungible DPU

Project Everest: A Large Stack of Provably Correct Software Built using F*



Including production systems, such as:

Verified data formats:

CoreOS: vmswitch, http.sys, netvsc, ...
Hyper-V: ICs, data formats, ...
Ebpf-for-windows

Verified crypto libraries: python boringssl

- Linux, Python, Firefox/NSS, BoringSSL PQ Crypto, ...

Verified HW/SW secure capability systems: FUNGIBLE

- Fungible DPU

But, about ~100 person years of PhD+ level of expertise

- **1.5 Million Lines of Code & Proof**
- **About 200,000 lines of executable C & Asm code**
- **Lots of reusable bits**
- **But is this level of effort really cost-effective?**

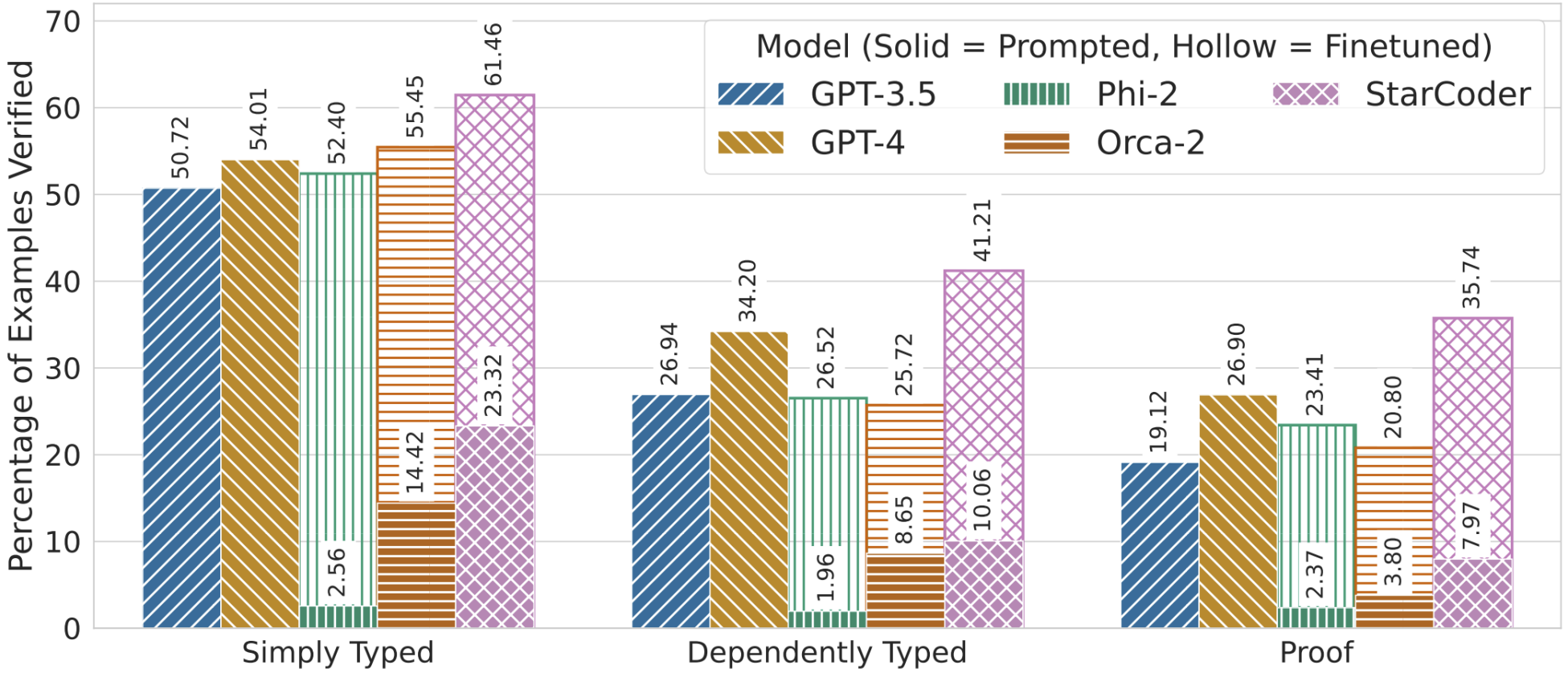
Can AI Help?

- All this code is open source on GitHub
- AI models have them in their pretraining …
- But, are they actually useful at doing proofs?

- Since 2021, my group has been studying this. Trying to assemble datasets, train models, etc

Best paper at ICSE 2024 …
but, frankly, still useless for real proof engineering

Success rate (verified @ 10)

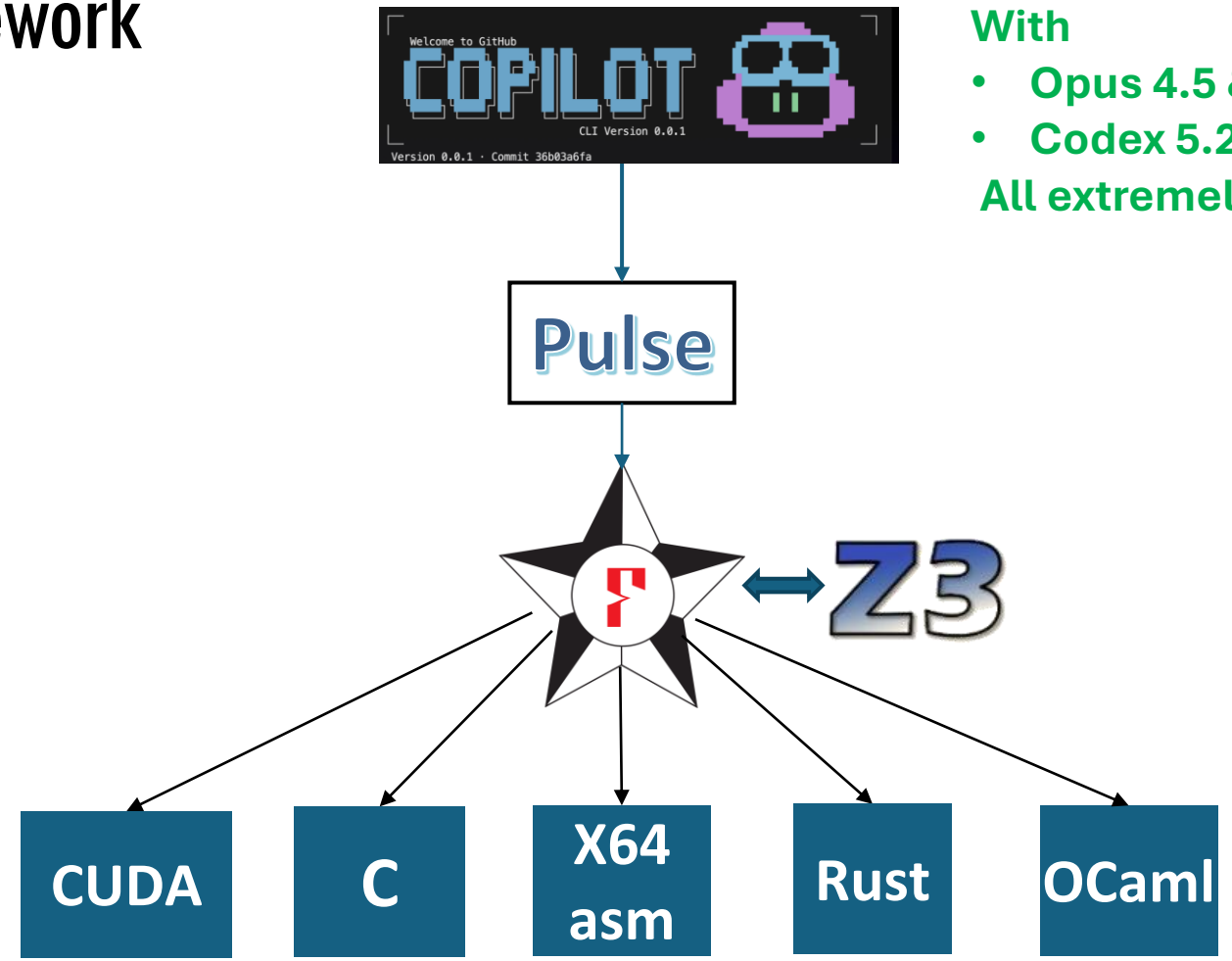


GPT models are reasonably good at F* out of the box
But, smaller finetuned models match or outperform GPT-4 on definition synthesis!

But, since December 2025

- Reasoning models reach a new level of performance
- Models are also very well-trained for tool use
- Combining the two in an agentic coding environment ➔ 
  - GitHub Copilot CLI, Claude Code, …

Our Agentic Proof-oriented Programming Framework



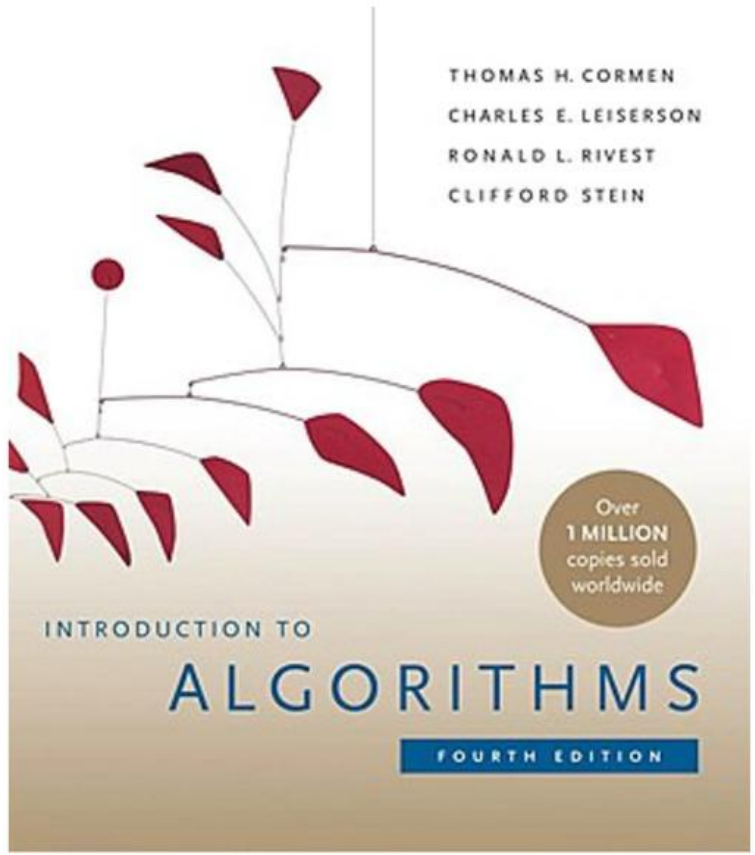
With

- **Opus 4.5 & 4.6**
- **Codex 5.2, 5.3**

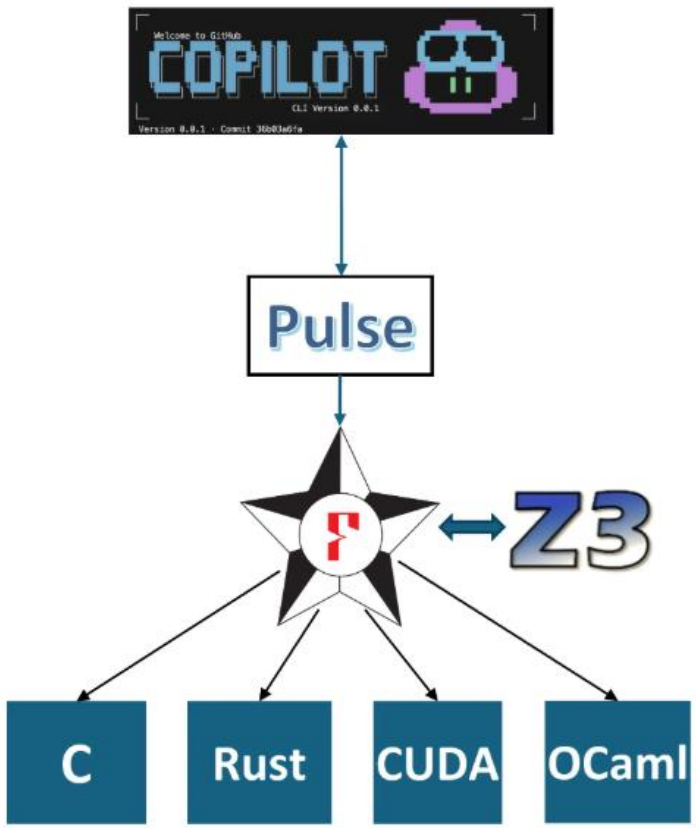
All extremely good at F* and Pulse

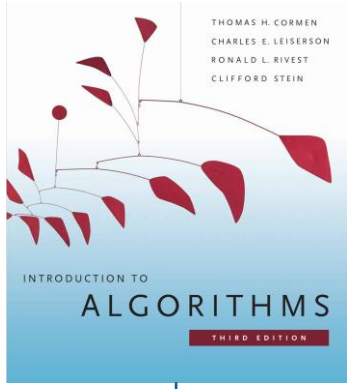
- `Install F*: curl -fsSL https://aka.ms/install-fstar | bash -s -- --nightly`
- `Install copilot: curl -fsSL https://gh.io/copilot-install | bash`
- `Install proof-copilot: copilot plugin install FStarLang/proof-copilot`

Demo

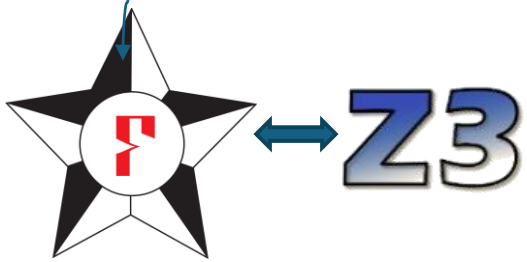

THOMAS H. CORMEN
CHARLES E. LEISERSON
RONALD L. RIVEST
CLIFFORD STEIN
Over 1 MILLION copies sold worldwide
INTRODUCTION TO
ALGORITHMS
FOURTH EDITION

&


COPILOT
Pulse
Z3
C
Rust
CUDA
OCaml



Pulse



120KLOC of code & proof for 52 algos & data structures

#	Algorithm	CLRS	Spec	Complexity
1	Insertion Sort	§2.1	sorted ∧ permutation	Linked $O(n^2)$
2	Merge Sort	§2.3	sorted ∧ permutation	Linked O(n lg n)
3	Binary Search	§2.3	found ⟺ key in array	Linked O(lg n)
4	Kadane (Max Subarray)	§4.1	result = max contiguous sum	Linked O(n)
5	D&C Max Subarray	§4.1	result = max contiguous sum	Pure O(n lg n)
6	Matrix Multiply	§4.2	C = A · B	Linked $O(n^3)$
7	Strassen	§4.2	Strassen = standard mult	Pure O(n^{2.81})
8	Heapsort	§6.4	sorted ∧ permutation	Linked O(n lg n)
9	Partition	§7.1	elements partitioned ∧ perm	Linked Θ(n)
10	Quicksort	§7.1	sorted ∧ permutation	Linked $O(n^2)$
11	Counting Sort	§8.2	sorted ∧ permutation	—
12	Counting Sort (Stable)	§8.2	stable sort ∧ perm	—
13	Radix Sort	§8.3	sorted ∧ permutation	—
14	Bucket Sort	§8.4	sorted ∧ permutation	—
15	Min / Max	§9.1	result ∈ array ∧ is min/max	Linked Θ(n−1)
16	Simultaneous Min-Max	§9.1	min ∧ max of array	Linked ⌊3(n−1)/2⌋
17	Quickselect	§9.2	k-th smallest at position k	Linked $O(n^2)$
18	Partial Selection Sort	§9.2	first k sorted, k-th correct	Linked O(nk)
19	Stack	§10.1	ghost list matches (LIFO)	—
20	Queue	§10.1	ghost list matches (FIFO)	—
21	Singly Linked List	§10.2	ghost list matches contents	Linked O(n)
22	Doubly-Linked List	§10.2	segment pred ∧ ghost list	—
23	Hash Table (Linear Probing)	§11.4	key ↦ value lookup	—
24	BST (Pointer-based)	§12.1–3	key set ∧ BST invariant	Linked O(h)
25	BST (Array-based)	§12.1–3	key set ∧ BST invariant	Linked O(h)
26	RB Tree (Okasaki)	§13.1–4	BST ∧ RB invariants	Pure O(lg n)
27	RB Tree (CLRS-style)	§13.1–4	BST ∧ RB invariants + delete	—
28	Rod Cutting	§15.1	optimal revenue	Linked $O(n^2)$
29	LCS	§15.4	LCS length	Linked O(mn)
30	Matrix Chain	§15.2	optimal parenthesization cost	Pure $O(n^3)$
31	Activity Selection	§16.1	greedy = optimal count	Linked O(n)
32	Huffman Coding	§16.3	WPL-optimal prefix-free tree	Pure $O(n^2)$
33	Union-Find	§21.3	find/union maintain forest	—
34	BFS	§22.2	distances ∧ parent tree	Linked $O(V^2)$
35	DFS	§22.3	timestamps ∧ classification	Linked $O(V^2)$
36	Kahn Topological Sort	§22.4	valid topological order	Linked $O(V^2)$
37	Kruskal MST	§23.2	forest of graph	Pure $O(V^3)$
38	Prim MST	§23.2	spanning tree + key correct	Pure $O(V^2)$
39	MST Theory	§23.1	cut/cycle properties	—
40	Bellman-Ford	§24.1	SSSP distances	Linked $O(V^3)$
41	Dijkstra	§24.3	SSSP distances	Linked $O(V^2)$
42	Floyd-Warshall	§25.2	APSP distances	Linked $Θ(n^3)$
43	Edmonds-Karp (Max Flow)	§26.2	valid flow ∧ max flow	Pure $O(VE^2)$
44	GCD / Extended GCD	§31.2	gcd ∧ Bézout coefficients	Linked O(lg b)
45	Modular Exponentiation	§31.6	result = b^e mod m	Linked O(lg e)
46	Naive String Match	§32.1	all match positions	Linked O(nm)
47	KMP	§32.4	all match positions	Linked O(n+m)
48	Rabin-Karp	§32.2	all match positions	Pure O(nm)
49	Segments (Primitives)	§33.1	correct intersection test	—
50	Graham Scan	§33.3	convex hull (all left turns)	—
51	Jarvis March	§33.3	convex hull vertices	—
52	Vertex Cover (2-approx)	§35.1	cover ≤ 2 · OPT	—

Zoom in on Dijkstra’s shortest path algorithm

```
fn dijkstra
  (weights: A.array int)
  (n: SZ.t)
  (source: SZ.t)
  (dist: A.array int)
  (pred: A.array SZ.t)
  (ctr: GR.ref nat)
  requires
    A.pts_to weights sweights **
    A.pts_to dist sdist **
    A.pts_to pred spred **
    GR.pts_to ctr c0 **
    pure (
      SZ.v n > 0 /\
      SZ.v source < SZ.v n /\
      Seq.length sweights == SZ.v n * SZ.v n /\
      Seq.length sdist == SZ.v n /\
      Seq.length spred == SZ.v n /\
      SZ.fits (SZ.v n * SZ.v n) /\
      all_weights_non_negative sweights /\
      weights_in_range sweights (SZ.v n)
    )
  ensures exists* sdist' spred' (cf: nat).
    A.pts_to weights sweights **
    A.pts_to dist sdist' **
    A.pts_to pred spred' **
    GR.pts_to ctr cf **
    pure (
      Seq.length sdist' == SZ.v n /\
      (forall (v: nat). v < SZ.v n ==>
        Seq.index sdist' v == SP.sp_dist sweights (SZ.v n) (SZ.v source) v) /\
      shortest_path_tree spred' sweights (SZ.v n) (SZ.v source) /\
      dijkstra_complexity_bounded cf (reveal c0) (SZ.v n)
    )
```

```
let sp_dist_optimal (weights: Seq.seq int) (n: nat) (s v: nat) (p: path)
  : Lemma
    (requires n > 0 /\ s < n /\ v < n /\ Seq.length weights == n * n /\
              path_source p == s /\ path_dest p == v /\
              path_edges p <= n - 1 /\
              path_valid p n /\ path_all_edges_exist p weights n /\
              all_weights_non_negative weights)
    (ensures sp_dist weights n s v <= path_weight p weights n)
```

```
let sp_dist_achievable  (weights: Seq.seq int) (n: nat) (s v: nat)
  : Pure path
    (requires n > 0 /\ s < n /\ v < n /\ Seq.length weights == n * n /\
              sp_dist weights n s v < inf)
    (ensures fun p ->
      path_source p == s /\ path_dest p == v /\
      path_edges p <= n - 1 /\
      path_valid p n /\ path_all_edges_exist p weights n /\
      path_weight p weights n == sp_dist weights n s v) = ...
```

```
let shortest_path_tree (spred: Seq.seq SZ.t) (sweights: Seq.seq int) (n source: nat) : prop
  Seq.length spred == n /\
  Seq.length sweights >= n * n /\
  source < n /\
  SZ.v (Seq.index spred source) == source /\
  (forall (v: nat). v < n /\ v <> source /\ SP.sp_dist sweights n source v < SP.inf ==>
    (let p = SZ.v (Seq.index spred v) in
     p < n /\
     SP.sp_dist sweights n source v ==
       SP.sp_dist sweights n source p + Seq.index sweights (p * n + v)))
```

High automation, but not autonomy

Specification review is key

Initial version: Almost trivial

This first version had an extremely weak specification, mainly just proving that in the output `dist` array, the distance from the source vertex to itself is less than or equal to `0`, a laughably weak specification.

```
ensures exists* sdist'.
  A.pts_to weights sweights **
  A.pts_to dist sdist' **
  pure (
    Seq.length sdist' == SZ.v n /\
    (SZ.v source < Seq.length sdist' ==> Seq.index sdist' (SZ.v source) <= 0)
  )
```

Judging specifications with Proof-oriented Tests

Two-sided specifications:

- The specifications of a component must be precise enough to allow a proof of a client

An idea from Shuvendu Lahiri: S. Lahiri, Evaluating LLM-driven User-Intent Formalization for Verification-Aware Languages

- For client programs, prove the correctness of test drivers on small concrete instances

```
fn test() {
  let mut a = [|0; 3sz|];
  a.(0sz) <- 3;  a.(1sz) <- 1;  a.(2sz) <- 2;
  quicksort a 3sz;
  assert (a.(0sz) == 1);
  assert (a.(1sz) == 2);
  assert (a.(2sz) == 3)
}
```

- The precondition of quicksort must be satisfiable on a typical input
- The postcondition must be precise enough to derive the expected output
- Small concrete inputs are good enough:
  - It's easy to predict the expected output
  - And specification errors are typically not due to over specialization

```
fn test_quicksort_3 ()
  requires emp
  returns r: bool
  ensures pure (r == true)
{
  // Input: [3; 1; 2]
  let v = V.alloc 0 3sz;
  V.to_array_pts_to v;
  let arr = V.vec_to_array v;
  rewrite (A.pts_to (V.vec_to_array v) (Seq.create 3 0)) as (A.pts_to arr (Seq.create 3 0
  arr.(0sz) <- 3;
  arr.(1sz) <- 1;
  arr.(2sz) <- 2;

  // Bind input ghost
  with s0. assert (A.pts_to arr s0);

  // y = quicksort(x)
  quicksort arr 3sz;

  // assert(y == expected)
  with s. assert (A.pts_to arr s);
  // Reveal opaque CLRS.permutation to get SP.permutation
  reveal_opaque (`%SS.permutation) (SS.permutation s0 s);
  // Now Z3 sees: SP.permutation int s0 s, and knows s0 == [3;1;2] from array writes
  completeness_sort3 s;

  // Read and verify each element (runtime checks that survive extraction)
  let v0 = arr.(0sz);
  let v1 = arr.(1sz);
  let v2 = arr.(2sz);
  assert (pure (v0 == 1));
  assert (pure (v1 == 2));
  assert (pure (v2 == 3));
  let ok = int_eq v0 1 && int_eq v1 2 && int_eq v2 3;

  // cleanup
  with s2. assert (A.pts_to arr s2);
  rewrite (A.pts_to arr s2) as (A.pts_to (V.vec_to_array v) s2);
  V.to_vec_pts_to v;
  V.free v;
  ok
```

Initial review report: Evaluating CLRS formalization on this criterion. It has good discriminating power!

02	Insertion Sort	Deterministic	✅ Precise
02	Merge Sort	Deterministic	✅ Precise
04	Binary Search	Deterministic	✅ Precise
04	Max Subarray (Kadane)	Deterministic	✅ Precise
04	Matrix Multiply	Deterministic	✅ Precise
06	Heapsort	Deterministic	✅ Precise
07	Partition	**Relational**	✅ Precise
07	Quicksort	Deterministic	✅ Precise
08	Counting Sort (2 variants)	Deterministic	✅ Precise
09	Min / Max	Deterministic	✅ Precise
09	Simultaneous MinMax	Deterministic	✅ Precise
09	Quickselect	Deterministic	✅ Precise

11	Hash Table	Relational/Precise	⚠️ Weak insert	**Comprehensive**	7 tests	0	Insert postcondition doesn't guarantee success
12	BST (Pointer)	Deterministic	✅ Precise	**Comprehensive**	14 pure + Pulse	0	None
12	BST (Array)	Relational/Existential	❌ **Too weak**	Minimal	Partial	0	No reachability; insert→search broken
21	Union-Find	Relational	⚠️ Moderate	Minimal	1 union on 3 elts	0	Rank bound not preserved in postcondition
22	BFS	**Relational/Imprecise**	⚠️ **Weak**	Minimal	1 (3-vertex chain)	0	**No shortest-path optimality in postcondition**
22	DFS	Relational	⚠️ **Weak**	Minimal	1 (3-vertex chain)	0	**Spec↔Impl disconnect**; theorems not exposed
22	Topological Sort	**Relational**	✅ Precise	Moderate	1 (3-vertex DAG)	0	None — correctly relational
23	Kruskal	Relational	⚠️ **Weak**	Minimal	1 (3-vertex triangle)	0	**No spanning tree or MST property**
23	Prim	Functional but weak	❌ **Critical**	Minimal	1 (3-vertex triangle)	0	**Postcondition admits incorrect outputs**
24	Bellman-Ford	Relational/Conditional	⚠️ Moderate	Adequate	1 (3-vertex, neg wts)	0	Correctness conditional on runtime boolean

A typical specification gap: Underspecified error conditions

```
fn hash_insert
  (table: A.array int)
  (#s: erased (Seq.seq int))
  (size: SZ.t)
  (key: int{key >= 0 /\ SZ.fits key})
  (ctr: GR.ref nat)
  (#c0: erased nat)
  requires
    A.pts_to table s **
    GR.pts_to ctr c0 **
    pure (SZ.v size > 0 /\ Seq.length s == SZ.v size /\
          valid_ht s (SZ.v size))
  returns result: bool
  ensures exists* s' cf.
    A.pts_to table s' **
    GR.pts_to ctr cf **
    pure (
      Seq.length s' == SZ.v size /\
      Seq.length s' == Seq.length s /\
      valid_ht s' (SZ.v size) /\
      (if result
       then (key_in_table s' (SZ.v size) key /\
             key_findable s' (SZ.v size) key /\
             (exists (idx: nat). idx < SZ.v size /\
               (Seq.index s idx == -1 \/ Seq.index s idx == -2) /\
               Seq.index s' idx == key /\
               seq_modified_at s s' idx))
       else s' == s) /\
      cf >= reveal c0 /\ cf - reveal c0 <= SZ.v size
    )
```

Does not say when result is allowed to be false

```
// Insert key 0
let b = hash_insert table 3sz 0 ctr;

if b {
  // === Insert succeeded ===
  // From insert postcondition:
  //   key_in_table s' 3 0 /\ key_findable s' 3 0 /\ valid_ht s' 3

  // Search for key 0 — should find it
  let r = hash_search table 3sz 0 ctr;

  // Postcondition precision: search must find key 0.
  //
  // From insert: key_in_table s' 3 0
  // From search postcondition: SZ.v r == 3 ==> ~(key_in_table s' 3 0)
  // Contrapositive: key_in_table s' 3 0 ==> SZ.v r != 3
  // Combined with SZ.v r <= 3: SZ.v r < 3
  assert (pure (SZ.v r < 3));

  // Cleanup
  GR.free ctr;
  with s. assert (A.pts_to table s);
  rewrite (A.pts_to table s) as (A.pts_to (V.vec_to_array tv) s);
  hash_table_free tv;
} else {
  // === Insert failed ===
  // Postcondition: s' == s (table unchanged)
  // This branch represents a spec weakness: the postcondition of hash_insert
  // does not guarantee success when empty slots are available.
```

Unable to prove that this branch is unreachable

```
  // Cleanup
  GR.free ctr;
  with s. assert (A.pts_to table s);
  rewrite (A.pts_to table s) as (A.pts_to (V.vec_to_array tv) s);
  hash_table_free tv;
}
```

```
//SNIPPET_START: ht_hash_insert
// Insert a key into the hash table
// Returns true if successful, false if table is full
// Proves both correctness and O(n) complexity
fn hash_insert
  (table: A.array int)
  (#s: erased (Seq.seq int))
  (size: SZ.t)
  (key: int{key >= 0 /\ SZ.fits key})
  (ctr: GR.ref nat)
  (#c0: erased nat)
  requires
    A.pts_to table s **
    GR.pts_to ctr c0 **
    pure (SZ.v size > 0 /\ Seq.length s == SZ.v size /\
          valid_ht s (SZ.v size))
  returns result: bool
  ensures exists* s' cf.
    A.pts_to table s' **
    GR.pts_to ctr cf **
    pure (
      Seq.length s' == SZ.v size /\
      Seq.length s' == Seq.length s /\
      valid_ht s' (SZ.v size) /\
      (if result
       then (key_in_table s' (SZ.v size) key /\
             key_findable s' (SZ.v size) key /\
             (exists (idx: nat). idx < SZ.v size /\
               (Seq.index s idx == -1 \/ Seq.index s idx == -2) /\
               Seq.index s' idx == key /\
               seq_modified_at s s' idx))
       else (s' == s /\
             (forall (q: nat). {:pattern (hash_probe_nat key q (SZ.v size))}
               q < SZ.v size ==>
                 Seq.index s (hash_probe_nat key q (SZ.v size)) =!= -1 /\
                 Seq.index s (hash_probe_nat key q (SZ.v size)) =!= -2))) /\
      cf >= reveal c0 /\ cf - reveal c0 <= SZ.v size
    )
//SNIPPET_END: ht_hash_insert
```

Strengthened postcondition
Allows proving that the result must be true in this test case

```
fn test_insert_then_search ()
  requires emp
  returns res: bool
  ensures pure (res == true)
{
  let tv = hash_table_create 3sz;
  let table = V.vec_to_array tv;
  rewrite (A.pts_to (V.vec_to_array tv) (Seq.create 3 (-1)))
          as (A.pts_to table (Seq.create 3 (-1)));
  let ctr = GR.alloc #nat 0;

  // Insert key 0
  let b = hash_insert table 3sz 0 ctr;
  trigger_insert_empty 3 0;
  assert (pure (b == true));

  // Search for key 0 — must find it
  let r = hash_search table 3sz 0 ctr;
  assert (pure (SZ.v r < 3));

  // Compute runtime-observable result
  let ok = (b && (r <^ 3sz));
  assert (pure (ok == true));

  // Cleanup
  GR.free ctr;
  with s. assert (A.pts_to table s);
  rewrite (A.pts_to table s) as (A.pts_to (V.vec_to_array tv) s);
  hash_table_free tv;

  ok
}
```

02	Insertion Sort	Deterministic	✅ Precise
02	Merge Sort	Deterministic	✅ Precise
04	Binary Search	Deterministic	✅ Precise
04	Max Subarray (Kadane)	Deterministic	✅ Precise
04	Matrix Multiply	Deterministic	✅ Precise
06	Heapsort	Deterministic	✅ Precise
07	Partition	**Relational**	✅ Precise
07	Quicksort	Deterministic	✅ Precise
08	Counting Sort (2 variants)	Deterministic	✅ Precise
09	Min / Max	Deterministic	✅ Precise
09	Simultaneous MinMax	Deterministic	✅ Precise
09	Quickselect	Deterministic	✅ Precise
09	Partial Selection Sort	Deterministic	✅ Precise
10	Stack	Deterministic	✅ Precise
10	Queue	Deterministic	✅ Precise
10	Doubly Linked List	Deterministic	✅ Precise
10	Singly Linked List	Deterministic	✅ Precise
11	Hash Table	Deterministic	✅ Precise
12	BST (Pointer)	Deterministic	✅ Precise
12	BST (Array)	Deterministic	✅ Precise
13	RB-Tree (Okasaki)	Deterministic	✅ Precise
13	RB-Tree (CLRS)	Deterministic	✅ Precise
15	Rod Cutting	Deterministic	✅ Precise
15	Matrix Chain	Deterministic	✅ Precise
15	LCS	Deterministic	✅ Precise
16	Activity Selection	**Precise Relational**	✅ Precise
16	Huffman Tree	**Precise Relational**	✅ Precise
16	Huffman Codec	Deterministic	✅ Precise
21	Union-Find	Relational	✅ Precise
22	BFS	Relational	✅ Precise
22	DFS	Relational	✅ Precise
22	Topological Sort	**Relational**	✅ Precise
23	Kruskal	Relational	✅ Precise
23	Prim	Relational	✅ Precise
24	Bellman-Ford	Deterministic (conditional)	✅ Precise
24	Dijkstra	Deterministic	✅ Precise
25	Floyd-Warshall	Deterministic	✅ Precise
26	Max Flow (Edmonds-Karp)	Deterministic + Optimality	✅ Precise
31	GCD	Deterministic	✅ Precise
31	Extended GCD	Deterministic	✅ Precise
31	ModExp (R-to-L)	Deterministic	✅ Precise
31	ModExp (L-to-R)	Deterministic	✅ Precise
32	Naive String Match	Deterministic	✅ Precise
32	KMP	Deterministic	✅ Precise
32	Rabin-Karp	Deterministic	✅ Precise
33	Segments (primitives)	Deterministic	✅ Precise
33	Graham Scan	Deterministic	✅ Precise
33	Jarvis March	Deterministic	✅ Precise
35	Vertex Cover (2-approx)	**Relational**	✅ Precise

After iterating with the model to fix specs to prove all test cases

A Second Experiment

A Verified Mark & Sweep GC and Allocator for OCaml

- Based on work from IIT Madras!
- About 50,000 lines of code & proof in F*
  - Using an old, deprecated dialect of F* called Low*
- Upgrading and porting this code to Pulse: A lot of work!
- Also, the existing proofs are very monolithic
  - A single 35,000 line file contains most of the proof!

- Can F* proof-copilot modernize and modularize the development?

https://github.com/FStarLang/pulse-verified-gc

- ~25,000 lines of F* and Pulse
- Including both a mark & sweep collector and an allocator
- Extracting to C code, integrated with Ocaml's bytecode compiler

- All code written by agents
- But, with many rounds of specification review from proof experts
- Still, would benefit from more review from domain experts
  - (e.g., Sheera, KC et al)

Some reflections

- Close the loop!
  - Agents coupled with objective feedback (e.g., from program verifiers) can provide extreme program synthesis & proof automation
  - Others have observed this too, cf. John Regehr's zero degrees of freedom

- Agents allow devs to produce 100K lines of code a day
  - But, how can anyone review that volume of code
  - Program proof as a review load reduction technique
    - Review spec not code
  - Specs are not absolute

- Massive amounts of data on agentic usage of PL tools
  - Learn from that data, improve the tools, improve the agents

Some reflections

- Agents do months of PhD level work in hours! What does that mean?
  - Devaluing human effort?
    - Ironic, these agents got good by training on human artifacts
  - Refocusing human effort?
    - Aim hirer?

- What should we be teaching?
  - Can you become a good software engineering without writing and debugging code?
  - Can you become a good proof engineer without wrestling with proof tools?
    - Maybe?
  - Specification review, debugging techniques, …
  - *“Proof-oriented Architecture”?*

- Access inequalities
  - Not everyone has access to or can afford state of the art models
  - Does this accelerate only the few?
  - Can research institutions get large token grants?

Parting thoughts

- A lot has changed very quickly
  - Some things that were previously very costly are no longer so
    - E.g., the cost of proof engineering
  - Some other things that were hard, will always remain hard
    - E.g., how to precisely formalize user intent

- This is a great time to be a PL researcher
  - Tools that offer strong symbolic guarantees are more needed than ever
    - To reduce degrees of freedom for agentic engines
  - How do we think of PL design, balancing agent and human needs?

- Aim high, question long-held premises, but don’t give up on long-held principles
  - This is the beginning of the future, let’s shape it together!

```
•  Install F*: curl -fsSL https://aka.ms/install-fstar | bash -s -- --nightly
•  Install copilot: curl -fsSL https://gh.io/copilot-install | bash
•  Install proof-copilot: copilot plugin install FStarLang/proof-copilot
```