F* and Meta-F*

Formal Verification, Language Extensibility, and Proof automation

Jonathan Protzenko

Microsoft Research

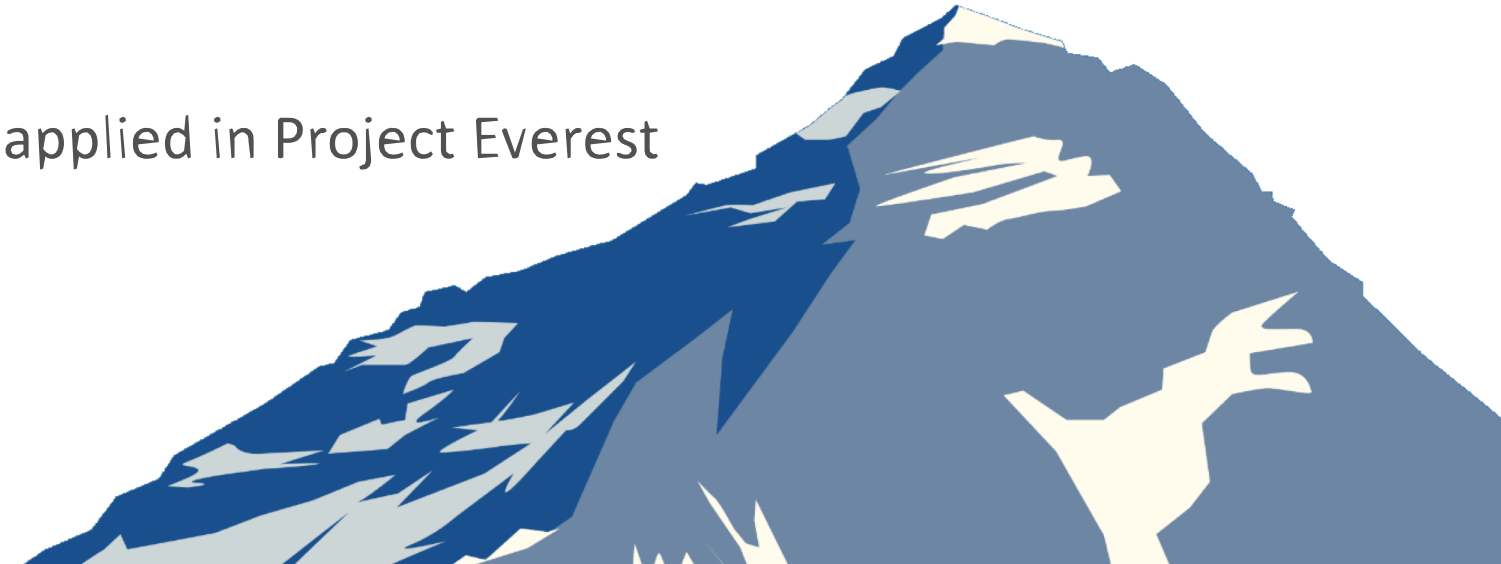
Inria INVENTEURS DU MONDE NUMÉRIQUE

Carnegie Mellon University

Microsoft Research - Inria JOINT CENTRE

https://fstar-lang.github.io
https://project-everest.github.io/

As applied in Project Everest

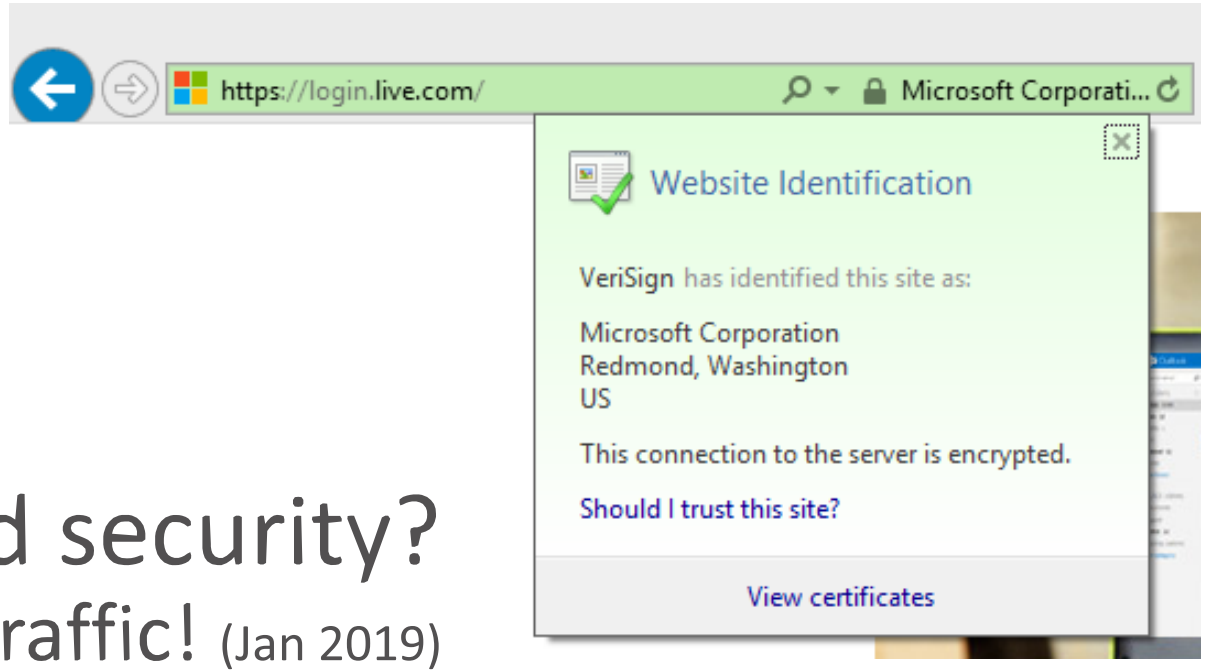


Goals of this lecture

- **Project Everest:** TLS verification at scale, using…
- **F*:** dependently-typed programming (context)
- **Meta-F*:** bridging interactive and automated theorem proving

Background on Project Everest

TLS: Transport Layer Security

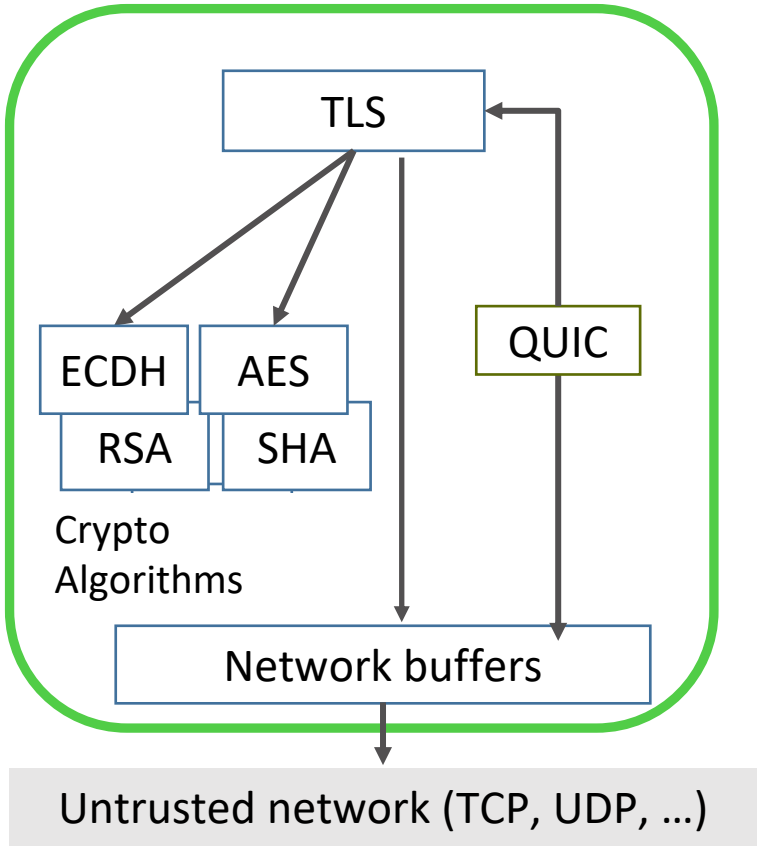


- Most widely deployed security?
  Now 73% of the internet traffic! (Jan 2019)
- Web, cloud, email, VoIP, 802.1x, VPNs, …

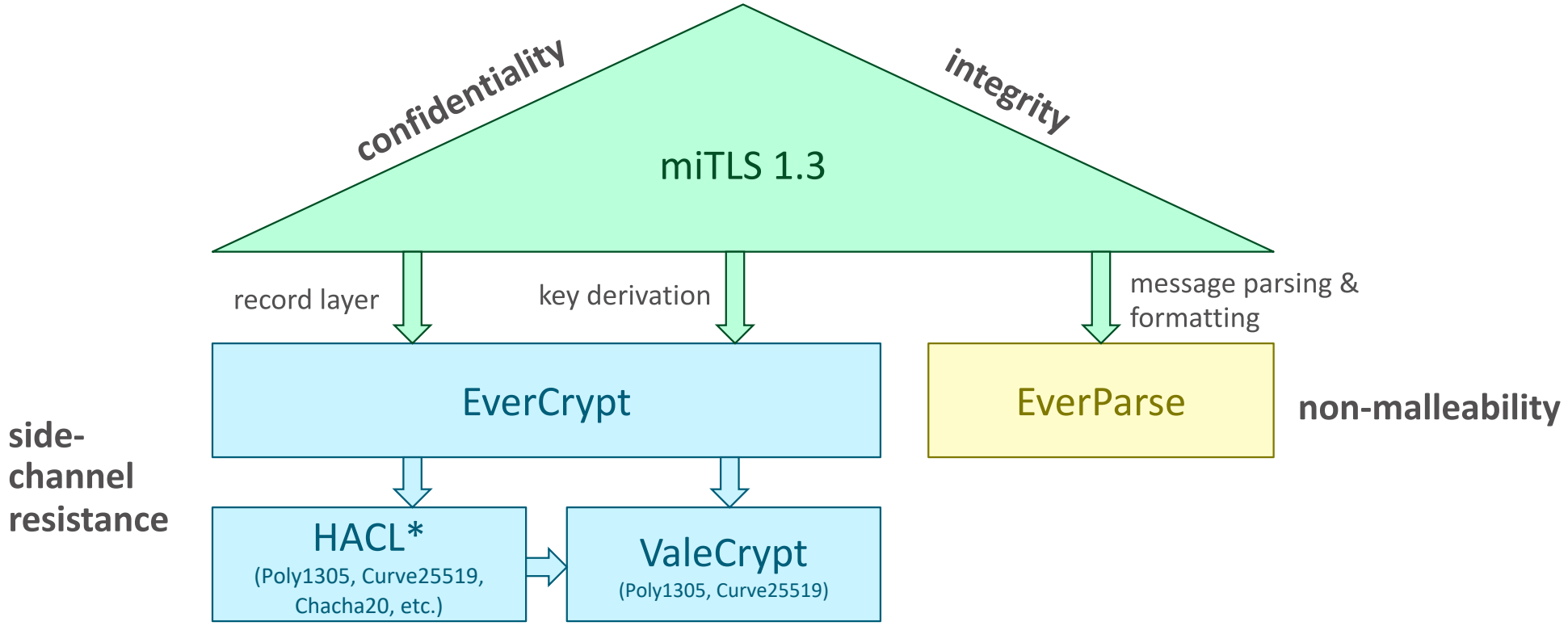
Project Everest

Verified Secure Components in the TLS Ecosystem

- Strong verified security
- Widespread deployment
- Trustworthy, usable tools
- Growing expertise in high-assurance software development
- Open source



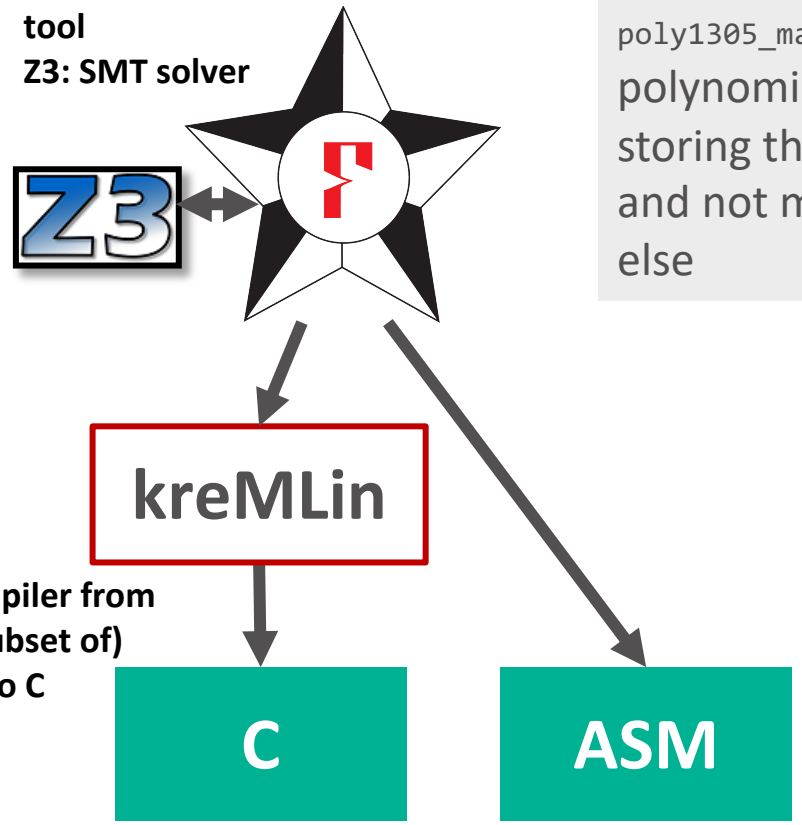
Everest: a verification pyramid (600,000 kloc and counting)



+ memory safety, functional correctness

Verification Tools and Methodology

F*: A general purpose programming language and verification tool
Z3: SMT solver



Compiler from (a subset of) F* to C

Math spec in F*

`poly1305_mac` computes a polynomial in GF(2^{130}-5), storing the result in `tag`, and not modifying anything else

```
val poly1305_mac: tag:nbytes 16 →
                  len:u32 →
                  msg:nbytes len{disjoint tag msg} →
                  key:nbytes 32 {disjoint msg key ∧ disjoint tag key} →
                     ST unit
  (requires (λ h → msg ∈ h ∧ key ∈ h ∧ tag ∈ h))
  (ensures (λ h0 _ h1 →
      let r=Spec.clamp h0.[sub key 0 16] in
      let s=h0.[sub key 16 16] in
      modifies {tag} h0 h1 ∧
      h1.[tag] == Spec.mac_1305 (encode_bytes h0.[msg]) r s))
```

Efficient ASM implementation

overhead

```
void poly1305_mac(uint8_t *tag, uint32_t len, uint8_t *msg, uint8_t *key)
{
  uint64_t tmp[10] = { 0 };
  uint64_t acc = tmp;
  uint64_t r = tmp + (uint32_t)5;
  uint8_t s[16] = { 0 };
  Crypto_Symmetric_Poly1305_poly1305_init(r, s, key);
  Crypto_Symmetric_Poly1305_poly1305_process(msg, len, acc, r);
  Crypto_Symmetric_Poly1305_poly1305_finish(tag, acc, s);
}
```

```
procedure{:quick}{:public}{:exportSpecs} Poly1305(
    ghost ctx_b:buffer64,
    ghost inp_b:buffer64,
    ghost len_in:nat64,
    ghost finish_in:nat64)
  reads
    ctx @= rdi; inp @= rsi; len @= rdx; finish @= rcx;
    h0 @= r14; h1 @= rbx; h2 @= rbp;
    ctx_in := (if win then rcx else ctx);
    inp_in := (if win then rdx else inp);
    n := 0x1_0000_0000_0000_0000;
    p := n * n * 4 - 5;
  modifies
    rax; rbx; rcx; rdx; rsi; rdi; rbp; rsp; r8; r9; r10; r11; r12; r13; r14; r15;
    efl; mem;
  ensures
    finish_in == 0 ==>
      modp(h210) == poly1305_hash_blocks
```

Everest in Action, so far

Production deployments of Everest Verified Cryptography

WinQUIC: Delivered Everest TLS 1.3 and crypto stack to Windows Networking, in the latest Windows developer previews

Mozilla NSS runs Everest verified crypto for several core algorithms

Everest verified crypto in the Linux kernel via WireGuard secure VPN (soon)

So what is this F* thing anyway?

A programming language
A proof assistant
A program verification tool

Two camps of program verification tools

Interactive proof assistants			Semi-automated verifiers of imperative programs	
Coq,	CompCert,	*air*	Dafny,	Verve,
Isabelle,	seL4,		FramaC,	IronClad,
Agda,	Bedrock,		Why3	miTLS
Lean, PVS, …	4 colors	*gap*		Vale

- **In the left corner**: Very expressive dependently-typed logics, but only purely functional programming
- **In the right**: effectful programming, SMT-based automation, but only first-order logic

F*: Bridging the gap

- **Functional programming language with effects**
  - Like OCaml, Haskell, F#, …
  - Compiles to OCaml or F#
  - A subset of F* compiled to C (with manual control over memory management)
- **With an expressive core dependent type theory**
  - Like Coq, Agda, Lean, …
- **Semi-automated verification using SMT**
  - Like Dafny, Vcc, Liquid Haskell, …
- **In-language extensibility and proof automation using metaprograms**

A first taste

- Write ML-like code

```
let rec factorial n =
  if n = 0 then 1
  else n * factorial (n - 1)
```

- Give it a specification, claiming that `factorial` is a total function from non-negative to positive integers.

```
val factorial: n:int{n >= 0} -> Tot (i:int{i >= 1})
```

- Ask F* to check it

```
fstar factorial.fst
Verified module: Factorial
All verification conditions discharged successfully
```

```
fstar factorial.fst
Verified module: Factorial
All verification conditions discharged successfully
```

F* builds a typing derivation of the form:

$$\Gamma_{\text{prelude}} \vdash \texttt{let factorial n = e} : t \Leftarrow \phi$$

- In a context Γ_{prelude} including definitions of F* primitives
- The program `let factorial n = e` has type t, given the validity of a logical formula ϕ
- ϕ is passed to Z3 (an automated theorem prover/SMT solver) to check for validity
- If the check succeds, then, from the metatheory of F*, the program is safe at type t

The functional core of F*

- Recursive functions

```
val factorial : int -> int
let rec factorial n = (if n = 0 then 1 else n * (factorial (n - 1)))
```

- Inductive datatypes (immutable) and pattern matching

```
type list (a:Type) =
  | Nil  : list a
  | Cons : hd:a -> tl:list a -> list a

val map : ('a -> 'b) -> list 'a -> list 'b
let rec map f x = match x with
  | [] -> []
  | h :: t -> f h :: map f t
```

- Lambdas (unnamed, first-class functions)

```
map (fun x -> x + 42) [1;2;3]
```

Refinement types

```
type nat = x:int{x>=0}
```

- Refinements introduced by type annotations (code unchanged)

```
val factorial : nat -> nat
let rec factorial n = (if n = 0 then 1 else n * (factorial (n - 1)))
```

- Logical obligations discharged by SMT (simplified)

```
n >= 0, n <> 0 |= n - 1 >= 0
n >= 0, n <> 0, factorial (n - 1) >= 0 |= n * (factorial (n - 1)) >= 0
```

- Refinements eliminated by **subtyping**: `nat<:int`

```
let i : int = factorial 42
let f : x:nat{x>0} -> int = factorial
```

Beyond pure code: weakest preconditions

- Reflects the semantics of an effectful (stateful) construct logically
- Reduces program verification to verifying the validity of a logical formula (sent to SMT)
- A weakest precondition is a **predicate transformer** from post-condition to precondition
- One WP per language construct

Beyond pure code: weakest preconditions

Examples of weakest preconditions:

- WP (return e) Q = Q e
- WP (bind e1 as x in e2) Q = WP e1 (**λ** x -> WP e2 Q x))

For any effectful expression, compute the WP.

Top-level functions are checked against initial pre-condition and desired post-condition.

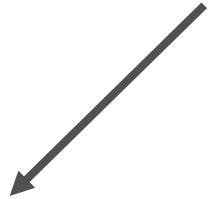
Beyond Pure Code

Effects

- Programmers model effects with monads
  - `st a = s -> a * s`
- F* derives a WP calculus for use with that monad
  - Also known as *Dijkstra* monads
  - `st_post a = a * s -> prop`
  - `st_pre = s -> prop`
  - `wp_st a = st_post a -> st_pre`
- Along with a computation type for effectful terms
  - Computations indexed by their own WPs: `STATE a (wp : wp_st a)`

Effectful programs with Hoare-style Specifications

only pure code in pre and post-conditions



```
let invert (r:ref int)
  : STExn unit
    (requires fun h0 -> True)
    (ensures fun h0 v h1 -> no_exn v ==> h0.[r] <> 0 /\ h1.[r] = 1 / h0.[r])
 =
    let x = !r in
    if x = 0 then
      raise Division_by_zero
    else r :=  1 / x
```

Effectful programs with Hoare-style Specifications

```
let invert (r:ref int)
  : STExn unit
   (requires fun h0 -> h0.[r] <> 0)
   (ensures fun h0 v h1 -> no_exn v /\ h1.[r] = 1 / h0.[r])
 =
   let x = !r in
   if x = 0 then
     raise Division_by_zero
   else r :=  1 / x
```

Exploiting Expressiveness & Extensibility

Low*: A subset of F* that compiles to C

- Embed within F* a CompCert C-like memory model
  - Low*: An effect for stateful programs manipulating a C-like memory model
  - Programs in the Low* effect are extracted to C by KreMLin, F*'s C backend
- Separate memory region for heaps and stacks
  - A region is a heterogeneous partial map
  - `region = addr -> option (a:Type & a)`
- With libraries modeling mutable arrays, pointers, structs, unions

Low* to C

And to support compilation to C, in nearly 1-1 correspondence, for auditability of our generated code

Designed to allow manipulating a C-like view of memory

Low* → COMPILED → C

```
1  let chacha20
2    (len: uint32{len ≤ blocklen}) (output: bytes{len = output.length})
3    (key: keyBytes) (nonce: nonceBytes{disjoint [output; key; nonce]}) (counter: uint32) :
4        Stack unit (requires (λ m0 → output ∈ m0 ∧ key ∈ m0 ∧ nonce ∈ m0))
5                   (ensures (λ m0 _m1 → modifies_1 output m0 m1)) =
6      push_frame ();
7      let state = Buffer.create 0ul 32ul in
8      let block = Buffer.sub state 16ul 16ul in
9      chacha20_init block key nonce counter;
10     chacha20_update output state len;
11     pop_frame ()
```

Erased specification

Stack allocation

Pointer arithmetic

```
let encrypt (#i:id) (e:writer i) (ad:bytes) (l:plainLen) (p:plain i l)
  : ST (cipher i l)
       (requires (λ h0 →
         l ≤ max_TLSPlaintext_fragment_length ∧
         sel h0 (ctr e.counter) < max_ctr))
       (ensures  (λ h0 c h1 →
         modifies [e.log_region] h0 h1 ∧
         h1 `HS.contains` (ctr e.counter) ∧
         sel h1 (ctr e.counter) == sel h0 (ctr e.counter) + 1 ∧
         (authId i ⟹
          let log = ilog e.log in
          let ent = Entry l c p in
          let n = Seq.length (sel h0 log) in
          h1 `HS.contains` log ∧
          witnessed (at_least n ent log) ∧
          sel h1 log == snoc (sel h0 log) ent))) =
  let h0 = get() in
  let ctr = ctr e.counter in
  HST.recall ctr; //lemma
  let text = if safeId i then create_ l 0z else repr i l p in
  let n = !ctr in
  lemma_repr_bytes_values n; //lemma
  let nb = bytes_of_int (AEAD.noncelen i) n in
  let iv = AEAD.create_nonce e.aead nb in
  lemma_repr_bytes_values (length text); //lemma
  let c = AEAD.encrypt e.aead iv ad text in
  if authId i then
    begin
    let ilog = ilog e.log in
    HST.recall ilog; //lemma
    let ictr: ideal_ctr e.region i ilog = e.counter in
    testify_seqn ictr;
    write_at_end ilog (Entry l c p);
    HST.recall ictr; //lemma
    increment_seqn ictr;
    HST.recall ictr //lemma
    end
  else
    ctr := n + 1;
  c
```

Dependently typed specs in the style of Hoare Type Theory / Coq

Dafny-ish proofs mostly automated by SMT, with a few well-chosen user-supplied lemmas

But SMT-based proofs can go awry

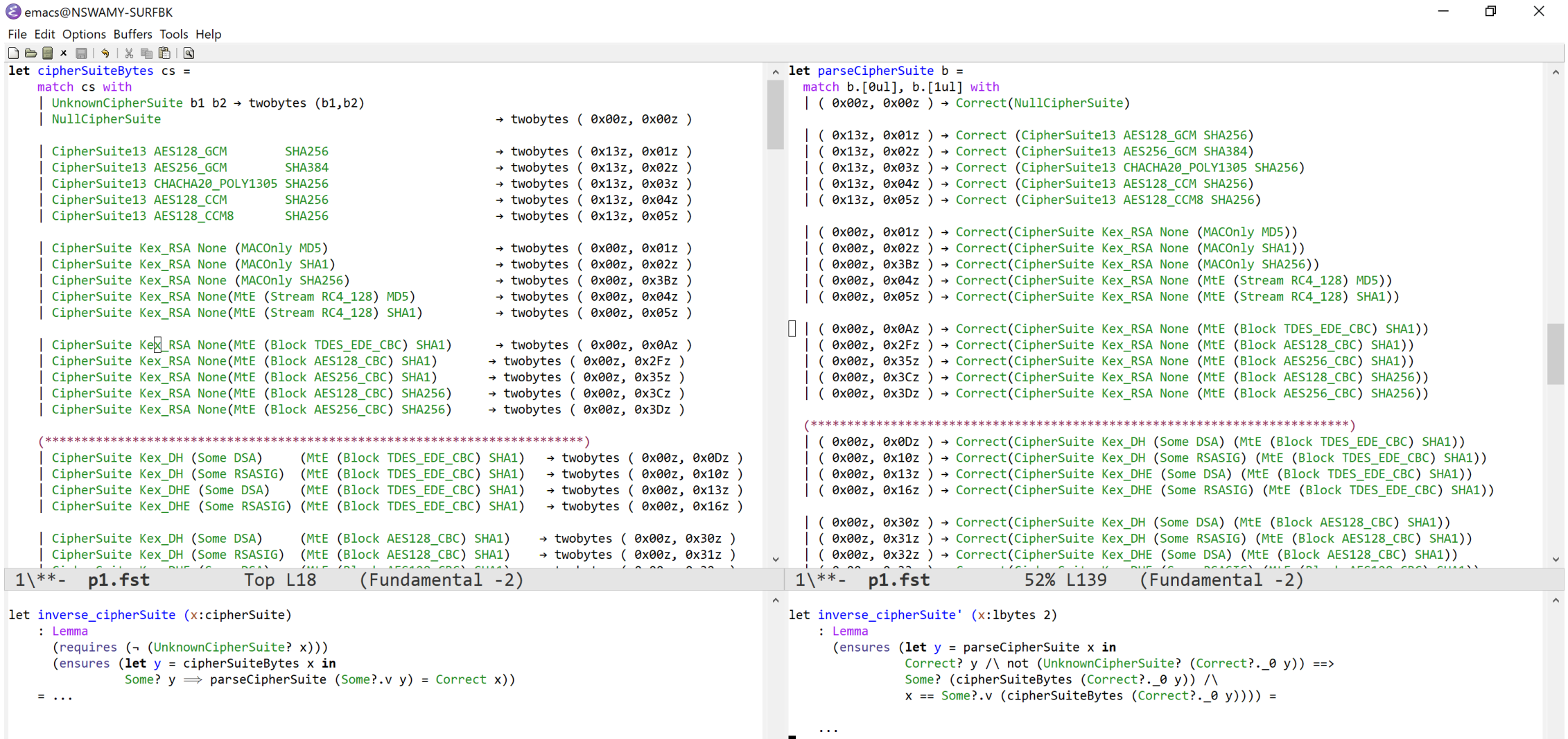
- E.g., when using theories like non-linear arithmetic

```
let lemma_carry_limb_unrolled (a0 a1 a2:nat) : Lemma
    (requires ⊤)
    (ensures (a0 % p44 + p44 * ((a1 + a0 / p44) % p44) + p88 * (a2 + ((a1 + a0 / p44) / p44))
            = a0 + p44 * a1 + p88 * a2)) =
  let open FStar.Math.Lemmas in
  let z = a0 % p44 + p44 * ((a1 + a0 / p44) % p44) + p88 * (a2 + ((a1 + a0 / p44) / p44)) in
  distributivity_add_right p88 a2 ((a1 + a0 / p44) / p44); (* argh! *)
  pow2_plus 44 44;
  lemma_div_mod (a1 + a0 / p44) p44;
  distributivity_add_right p44 ((a1 + a0 / p44) % p44) (p44 * ((a1 + a0 / p44) / p44)); (* argh! *)
  assert (p44 * ((a1 + a0 / p44) % p44) + p88 * ((a1 + a0 / p44) / p44) = p44 * (a1 + a0 / p44) );
  distributivity_add_right p44 a1 (a0 / p44); (* argh! *)
  lemma_div_mod a0 p44
```

Forced to write very detailed proof terms when SMT fails

And can be at a low level of abstraction

- Lots of boilerplate to define parsers/formatters and prove them mutually inverse



Automated (implicit?) theorem proving

- Unlike Coq, no direct access to the context
- Cannot stop, examine hypotheses and goals
- Proof-irrelevant theory: the SMT solver does not return a proof witness
- Non-modular: one big WP computed for each function

Enter Metaprogramming

Domain-specific languages, ad hoc proof automation, extensibility

Allow programmers to:

- Customize how programs are typechecked, producing domain-specific VCs
- Metaprogram programs and their proofs
- Define their own equivalence preserving program transformations
- …

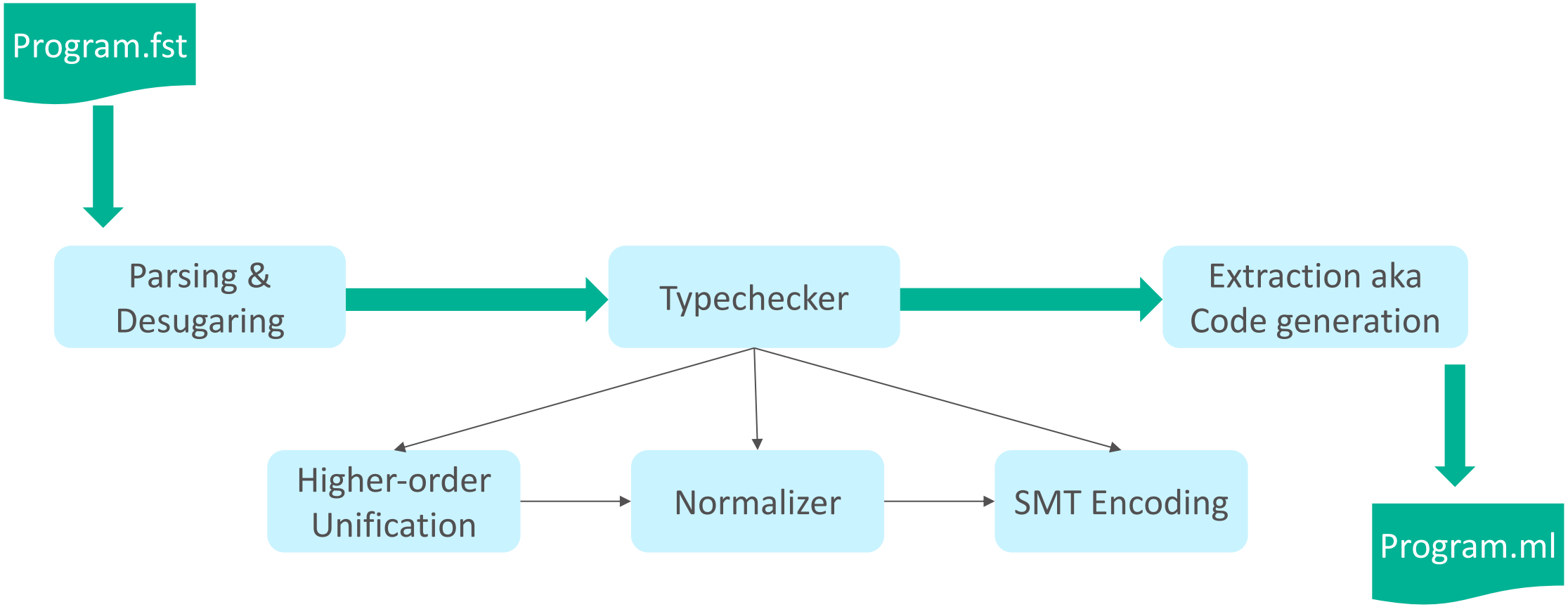
Domain-specific languages, ad hoc proof automation, extensibility

By treating the F* object language as its own metalanguage

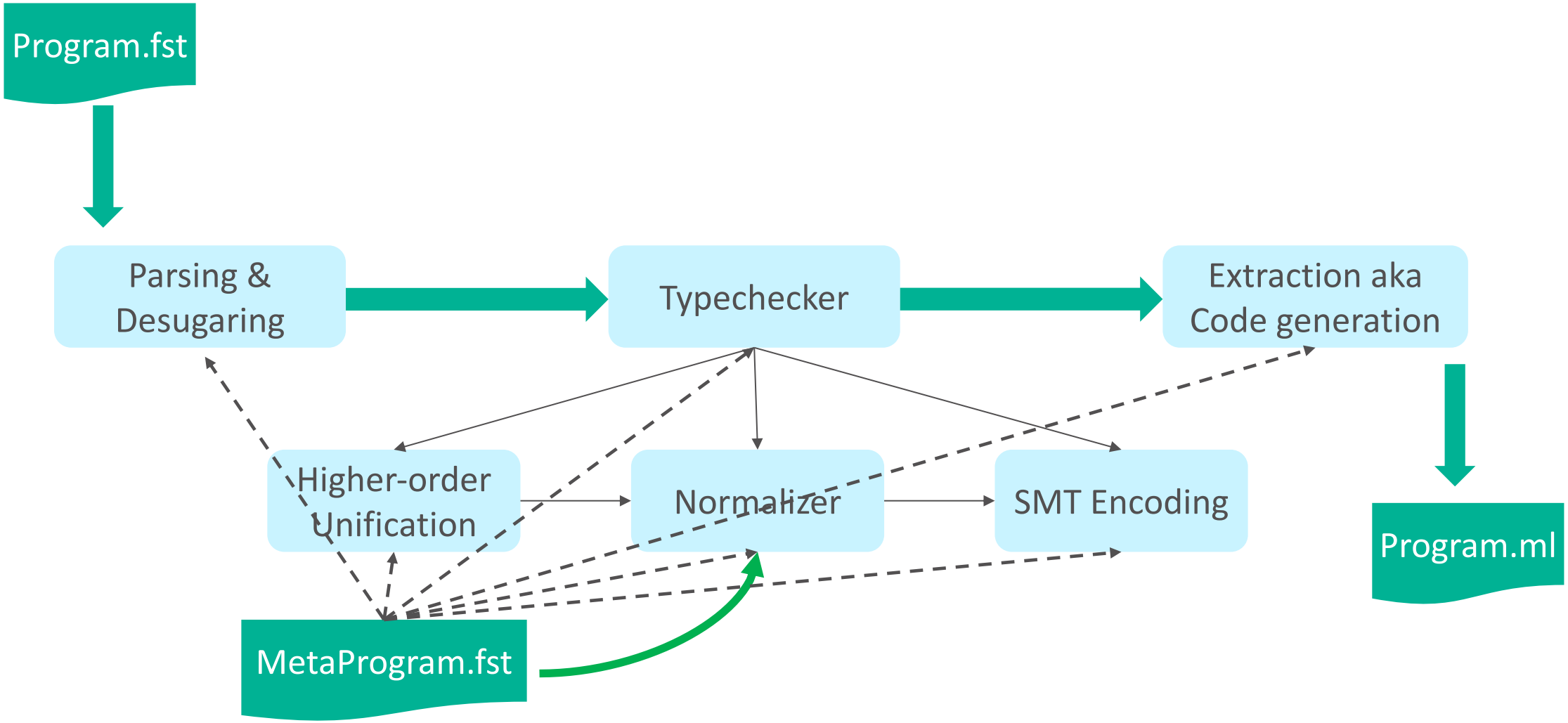
Building on *elaborator reflection* a great idea by

- David Christiansen and Edwin Brady (Idris)
- Leonardo de Moura (Lean)

A passive compiler pipeline



Scripting components with a metaprogram



Scripting a language implementation from within the language

Provide a API to compiler internals for F* (meta)programs to reflect and/or construct

- Syntax of terms
- Typechecking environment
- Typing derivations
- …

F* compiler runs F* metaprograms to build and typecheck other F* programs

From F* to Meta-F*, In three easy steps

1. Metaprogramming as a computational effect of *proof-state transformers*

2. Primitive operations to manipulate proof-states, using trusted compiler primitives

3. Reflecting on syntax: quotation and unquotation

Some terminology

- Meta-programs execute at any stage of the compiler
- The environment (meta-program state) **is** the compiler state
- When executing within the type-checker:
  - meta-program=tactic
  - environment=proof state

Recovering proof structure from WPs

- Synthesize a WP; get a verification condition
- Intercept it before sending it to the SMT solver
- A new construct: **assert ϕ by t**
- **assert ϕ; e** generates **ϕ** by t ∧ **(ϕ ⟹ VC e)**

- Traverse the VC, collecting hypotheses leading to **ϕ**
- Remove **ϕ** from the VC
- Run meta-program with hypotheses; goal = **ϕ**

Proof-state: A collection of typed holes

A hole is a missing program fragment in a context

$$\Gamma \vdash ?_0 : \tau$$

(aka a *goal)*

The proof-state is a collection of pending holes

$$\overline{\{\Gamma_i \vdash ?_i : \tau_i\}}$$

+ some internal persistent state

(e.g., unionfind graph of the unifier)

Metaprograms are proofstate transformers

```
let meta a = proofstate → Dv (either (a × proofstate) error)
```

- Uses an existing F* effect for non-termination: Dv
- The type of the state is an abstract type: proofstate
- error is the type of exceptions

State + Exception + Non-termination monad

Metaprogramming as a user-defined effect

```
let meta a = proofstate → Dv (either (a × proofstate) error)

(* Monad *)
let return a x : meta a = λ ps → Inl (x, ps)
let bind a b (f:meta a) (g: a → meta b) : meta b =
  λ ps → match f ps with
         | Inl (x, ps') → g x ps'
         | Inr err → Inr err

(* Actions *)
let get () : meta ps = λ ps → Inl (ps, ps)
let raise a err : meta a = λ ps → Inr (err, ps)
```

- Standard definitions of return, bind, get, raise
- Exceptions reset the state

Metaprogramming as a user-defined effect

```
let meta a = proofstate → Dv (either (a × proofstate) error)
```

Conspicuously, no **put** action

The proofstate is a compiler internal data structure.

A metaprogram can read it but cannot modify it arbitrarily (meta-result)

```
(* Actions *)
let get () : meta ps = λ ps → Inl (ps, ps)
let raise a err : meta a = λ ps → Inr (err, ps)

(* Ask F* to derive an effect from meta *)
new_effect { META : a:Type → Effect
             with repr = meta; bind; return; raise; get }
```

F* extended with meta-programs

- A new effect
  - regular **let** and implicit return
  - WP semantics derived *automatically* (Dijkstra Monads for free)
- Running it
  - automatically translated back to a functional implementation (state-passing)
  - executed on the F* normalizer (general reduction machinery for pure programs)

Abstraction for efficient execution

- F* meta-programs are deeply-embedded in the normalizer
- Need to embed/de-embed

- ```
  let rec eval (t: term) = match t with
    | Tm_App (Tm_Fvar “addition”, e1, e2) ->
      match eval e1, eval e2 with
      | Tm_constant c1, Tm_constant c2 ->
          Tm_constant (c1 + c2)
      | e1, e2 -> Tm_App (Tm_Fvar “addition”, e1, e2)
    | Tm_App (Tm_Builtin lid, e1) ->
      // how to represent the state e1?
    | Tm_App (f, x) -> subst ...
  ```
  ```

Abstraction for efficient execution

- The proof state is **abstract** from the point of view of meta-programs
- No need to **embed** and **de-embed**
- Jump back and forth:
  - outside of the normalizer into the type-checker code when a primitive action is called
  - from the type-checker code back into the (suspended) computation in the normalizer when a primitive action returns

Step 2

Primitive operations on `proofstate`

- Every typing rule (read backwards) provided as a proofstate primitive

$$\frac{\Gamma, x{:}t_1 \vdash ?_1 : t_2}{\Gamma \vdash ?_o : (x{:}t_1 \to t_2)}\ \text{[TC-Abs]} \qquad \textit{where}\ \ ?_0 \coloneqq \lambda\, x{:}t_1.?_1$$

```
intro : unit -> Meta binder
intro ()  (Γ ⊢ ?_o : (x:t_1 → t_2)  :: rest) =
  Inl (x, (Γ, x:t_1 ⊢ ?_1 : t_2      :: rest))
  where fresh x and ?_0 := fun (x:t_1)->?_1
intro () _ = raise (Failure "Goal is not an arrow")
```

Step 3

Reflecting on syntax

- Locally nameless abstract syntax of F* terms provided as a datatype to metaprograms

- Quotation: Builds the syntax of a term

```
`(0 + 1) : term
```

- Unquotation
  - Typechecks syntax at a given type

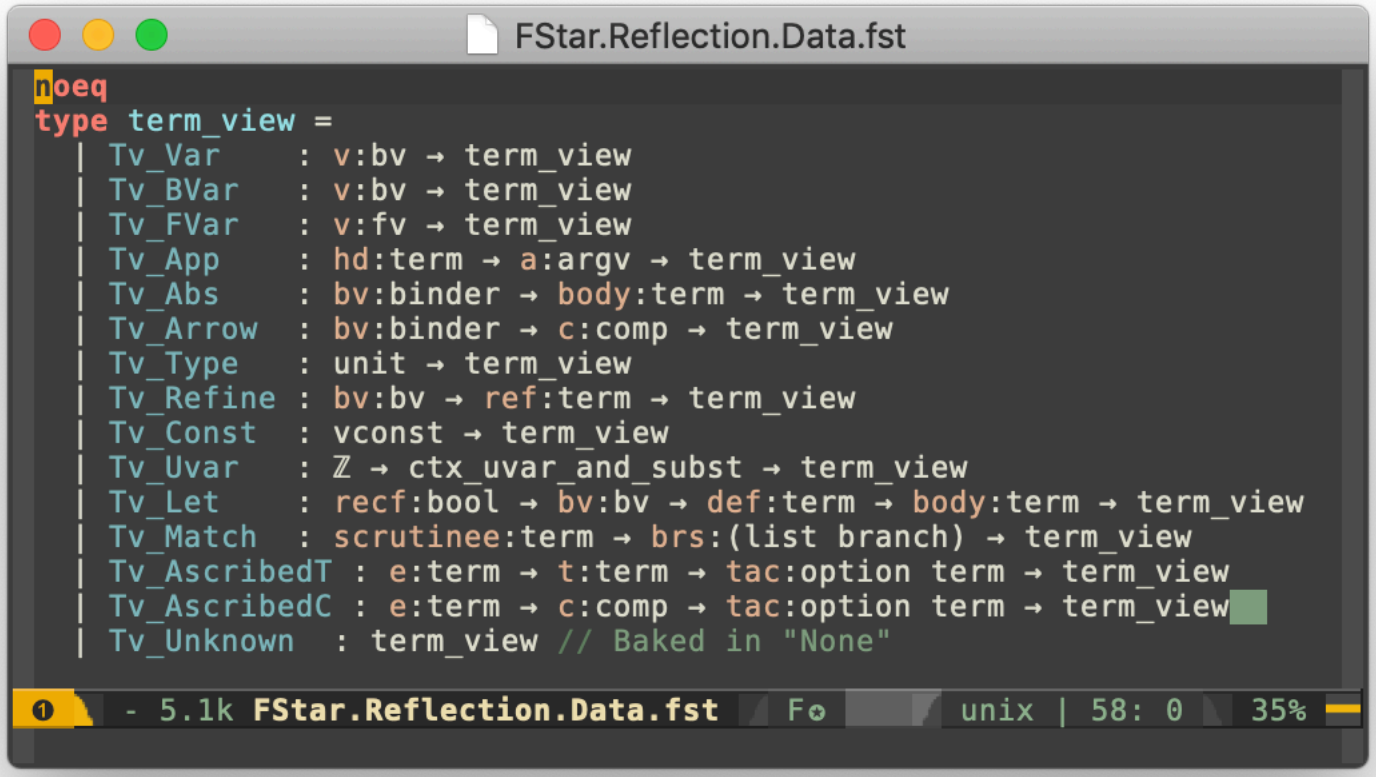
```
type term =
  | T_Var     : v:bv → term
  | T_BVar    : v:bv → term
  | T_FVar    : v:fv → term
  | T_App     : hd:term → a:argv → term
  | T_Abs     : bv:binder → body:term → term
  | T_Arrow   : bv:binder → c:comp → term
  | T_Type    : unit → term
  | T_Refine  : bv:bv → ref:term → term
  | T_Const   : vconst → term
  | T_Uvar    : ℤ → ctx_uvar_and_subst → term
  | T_Let     : recf:bool → bv:bv → def:term → body:term → term
  | T_Match   : scrutinee:term → brs:(list branch) → term
  | T_AscribedT : e:term → t:term → tac:option term → term
  | T_AscribedC : e:term → c:comp → tac:option term → term
```

```
val unquote: a:Type -> x:term -> Meta a
```

Step 3

Reflecting on syntax (but, better)

- Directly reflecting **term** is inefficient; need to convert potentially large terms
- Instead, use **term views** to unroll only the head constructor (old idea)
- Apply the same optimization with **type term** being abstract (no embedding)



Putting it together

Entrypoint: Decorate an implicit with a metaprogram (program synthesis demo)

```
let id : (a:Type) -> a -> a =
  _ by (
       let a = intro () in
       let x = intro () in
       hyp x
       )
```

$[\cdot \vdash \_ : (a : \mathrm{Type}) \to a \to a]$

$[a : \mathrm{Type} \vdash \_ : a \to a]$

$[a : \mathrm{Type}, x : a \vdash \_ : a]$

$[]$

And can be at a low level of abstraction

- Lots of boilerplate to define parsers/formatters and prove them mutually inverse

Metaprogramming mutually inverse parsers and formatters

```
type color = | Red | Blue | Green | Yellow
type pallette = nlist 18 (color × u8)
let ps : (p:parser pallette & serializer p) =
  _ by (gen_parser_serializer (`pallette))
```

Where, the types capture that parser/serializer are mutual inverses

```
let parser (t:Type) = bytes → option t
let serializer (#t:Type) (p:parser t) =
  s : (x:t → b:bytes) {
    (∀ x. p (s x) = Some x) ∧
    (∀ b. match p b with
          | Some x → s x = b
          | _ → ⊤)}
```

Putting it together

Entrypoint: Decorate an assertion with a metaprogram

```
let f (x:nat) =
  if x > 1 then
    assert (x * x > x)
        by tac;
    …
```

$$x{:}\,nat, h{:}\,(x > 1) \vdash \_ : (x * x > x)$$

- Assertions produce a goal in a context including control flow hypotheses
- Tackled by the metaprogram `tac`

SMT: Just one of F*'s tactic primitives

```
val smt: unit -> Meta unit
```

```
let f (x:nat) =
  if x > 1 then
    assert (x * x > x)
        by (smt());
    …
```

$$x{:}\,nat, h{:}\,(x > 1) \vdash \_ : (x * x > x)$$

- Assertions produce a goal in a context including control flow hypotheses
- Tackled by the metaprogram `tac`

But SMT-based proofs can go awry

- E.g., when using theories like non-linear arithmetic

```
let lemma_carry_limb_unrolled (a0 a1 a2:nat) : Lemma
    (requires ⊤)
    (ensures (a0 % p44 + p44 * ((a1 + a0 / p44) % p44) + p88 * (a2 + ((a1 + a0 / p44) / p44))
            = a0 + p44 * a1 + p88 * a2)) =
  let open FStar.Math.Lemmas in
  let z = a0 % p44 + p44 * ((a1 + a0 / p44) % p44) + p88 * (a2 + ((a1 + a0 / p44) / p44)) in
  distributivity_add_right p88 a2 ((a1 + a0 / p44) / p44); (* argh! *)
  pow2_plus 44 44;
  lemma_div_mod (a1 + a0 / p44) p44;
  distributivity_add_right p44 ((a1 + a0 / p44) % p44) (p44 * ((a1 + a0 / p44) / p44)); (* argh! *)
  assert (p44 * ((a1 + a0 / p44) % p44) + p88 * ((a1 + a0 / p44) / p44) = p44 * (a1 + a0 / p44) );
  distributivity_add_right p44 a1 (a0 / p44); (* argh! *)
  lemma_div_mod a0 p44
```

Remember this?

Forced to write very detailed proof terms when SMT fails

SMT + Tactics for more automated, robust proofs

Key lemma in poly1305, a MAC algorithm,
doing multiplication in the prime field $2^{130} - 5$

```
let poly_multiply (n p r h r0 r1 h0 h1 h2 s1 d0 d1 d2 h1 h2 hh : ℤ)
    : Lemma
      (requires p > 0 ∧ r1 ≥ 0 ∧ n > 0 ∧ 4 × (n × n) == p + 5 ∧ r == r1 × n + r0 ∧
                h == h2 × (n × n) + h1 × n + h0 ∧ s1 == r1 + (r1 / 4) ∧ r1 % 4 == 0 ∧
                d0 == h0 × r0 + h1 × s1 ∧ d1 ==
                d2 == h2 × r0 ∧ hh == d2 × (n
      (ensures (h × r) % p == hh % p) =
  let r14 = r1 / 4 in
  let h_r_expand = (h2 × (n × n) + h1 × n + h0)
  let hh_expand = (h2 × r0) × (n × n) + (h0 × (
                  (5 × r14)) × n + (h0 × r0 + h
  let b = (h2 × n + h1) × r14 in
  modulo_addition_lemma hh_expand p b;
  assert (h_r_expand == hh_expand + b × (n × n
      by (canon_semiring int_csr;
          smt())
```

- A reflective metaprogram to canonicalize terms in commutative semirings
- Simplifies goal into a form that smt can then solve using linear arithmetic only
- Prior manual proof required 41 steps of explicit rewriting lemmas (!)

Partial evaluation for Low* productivity

This is not Low*:

```
val compress:
  a:sha_alg -> state a -> array u8 -> Stack unit
```

Reason:

```
let state a = function
  | SHA2_224 | SHA2_256 -> array u32
  | SHA2_384 | SHA2_512 -> array u64
```

This *could* be compiled as a union. However, this is not **idiomatic.** Instead, we rely on **partial evaluation:**

```
let compress_224 =
  FStar.Tactics.(synth_by_tactic (specialize (compress SHA2_224)
  [`% ... ]))
let compress_256 =
  FStar.Tactics.(synth_by_tactic (specialize (compress SHA2_256)
  [`% ... ]))
let compress_384 =
  FStar.Tactics.(synth_by_tactic (specialize (compress SHA2_384)
  [`% ... ]))
let compress_512 =
  FStar.Tactics.(synth_by_tactic (specialize (compress SHA2_512)
  [`% ... ]))
```

Higher-order combinators

- It is frequent to **combine** core operations to generate a **construction** (counter-mode, hash, etc.)

Example:

- Given any (abstract) *compression function*, one can define a **Merkle-Damgård construction**
- Construction: **folding** the core compression function over multiple **blocks** of input data
- We **do not** want to write this n times...

An example of a higher-order combinator

```
// Write once; this is not Low*
noextract inline_for_extraction
let mk_compress_blocks (a: hash_alg)
  (compress: compress_st a)
  (s: state a)
  (input: blocks)
  (n: u32 { length input = block_size a * n })
=
  C.Loops.for 0ul n (fun i ->
    compress s (Buffer.sub input (i * block_size a) (block_size a)))


// Specialize many times; now this is Low*
let compress_blocks_224 = FStar.Tactics.(synth_by_tactic (specialize SHA2_224) [`%...])
...
let compress_md5 = FStar.Tactics.(synth_by_tactic (specialize MD5) [`%...])
...
```

Language extensions / DSLs in userland

- Typeclass resolution as a user-provided program
  - a hook to synthesize a term if F* fails to infer an implicit argument
  - dedicated syntactic sugar
  - optional arguments
- Syntax for base64 array literals
  - encoded via a string
  - rewritten via a tactic into a suitable Low* array declaration
- Deriving
  - printers, marshalling functions, etc.
- More
  - tactics as an F* frontend for DSLs

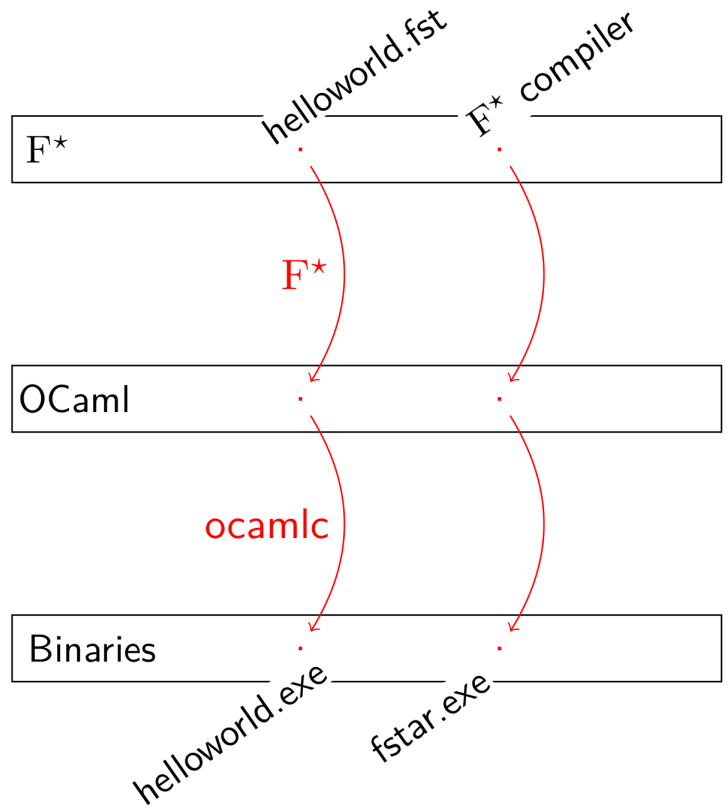
Executing tactics (reminder)

- Tactics can be **reified** as proof-state-passing programs
- F* features a **normalizer** (for higher-order unification)
- Upon seeing a tactic, the F* type-checker **interprets** the tactic program using the normalizer (KAM)
- Tactics are written against an **abstract** state
- Primitive actions are type-checker **primitives**

Works for any metaprogram, really.

Executing metaprograms natively

- By default, metaprograms are interpreted on F*'s normalizer
- But, F* is implemented in F* and all F* programs can be compiled to OCaml

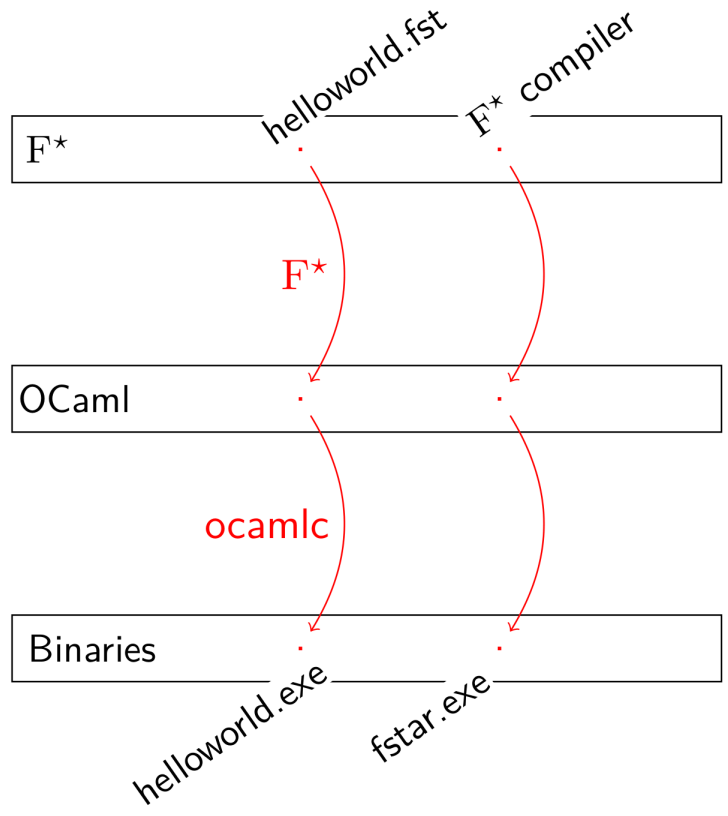


Dynamic loading of plugins

- Tactic is type-checked then extracted
- Fixed set of primitive type-checker actions; extend it with “dynlink”: dynamic linking to extend the compiler
- Register new primitive actions
- Detect at call-sites and trigger multi-language interop

Execution across language boundaries

- F* jumps back and forth between interpreter and native
- Conversions at the boundary: deeply embedded (`Tm_int “42”`) to native (`42`) and conversely.
- 10x performance boost



Some takeaways

- Freedom of expression
  - Tools for large-scale, full program verification need arbitrary expressive power
- Proof automation, Expressiveness, Control
  - SMT is great, but not a panacea: Eventually hit a complexity/undecidability wall
- Careful combination of tactics and SMT *improve* automation relative to either SMT or tactics alone
- Meta-F*: Self-scripting a PL implementation with reflective metaprogramming; tactics are just a special case