An introduction to Meta-F*

Nik Swamy Guido Martínez

ECI 2019

Two camps of program verification

Interactive Theorem Provers (ITPs): Coq, Agda, Lean, Idris, ...

- Usually for pure programs
- Very expressive
- Have traditionally relied on tactics for doing proofs

Two camps of program verification

Interactive Theorem Provers (ITPs): Coq, Agda, Lean, Idris, ...

- Usually for pure programs
- Very expressive
- Have traditionally relied on tactics for doing proofs

Program Verifiers: Dafny, VCC, Liquid Haskell, ...

- Verification conditions (VCs) computed and sent to SMT solvers
- Simple proofs often automatic

Two camps of program verification

Interactive Theorem Provers (ITPs): Coq, Agda, Lean, Idris, ...

- Usually for pure programs
- Very expressive
- Have traditionally relied on tactics for doing proofs

Program Verifiers: Dafny, VCC, Liquid Haskell, ...

- Verification conditions (VCs) computed and sent to SMT solvers
- Simple proofs often automatic
- When the solver fails, no good way out

Two camps of program verification

Interactive Theorem Provers (ITPs): Coq, Agda, Lean, Idris, ...

- Usually for pure programs
- Very expressive
- Have traditionally relied on tactics for doing proofs

Program Verifiers: Dafny, VCC, Liquid Haskell, ...

- Verification conditions (VCs) computed and sent to SMT solvers
- Simple proofs often automatic
- When the solver fails, no good way out
    - Need to tweak the program (to call lemmas, etc)
    - No automation
    - No good way to inspect or transform the proof environment

Two camps of program verification

Interactive Theorem Provers (ITPs): Coq, Agda, Lean, Idris, ...

- Usually for pure programs
- Very expressive
- Have traditionally relied on tactics for doing proofs

Program Verifiers: Dafny, VCC, Liquid Haskell, ...

- Verification conditions (VCs) computed and sent to SMT solvers
- Simple proofs often automatic
- When the solver fails, no good way out
    - Need to tweak the program (to call lemmas, etc)
    - No automation
    - No good way to inspect or transform the proof environment

> Can we retain the comfort of automation while avoiding the solver's issues?

F* basics

- Functional and effectful programming language / program verifier
  - A member of the ML family
  - Extracts to OCaml or F#; a subset (Low*) can also extract to C
  - Used for crypto implementations (e.g. EverCrypt)

F* basics

- Functional and effectful programming language / program verifier
  - A member of the ML family
  - Extracts to OCaml or F#; a subset (Low*) can also extract to C
  - Used for crypto implementations (e.g. EverCrypt)
- Full dependent types
  - As in Coq, Agda, Lean, Idris, etc

F* basics

- Functional and effectful programming language / program verifier
  - A member of the ML family
  - Extracts to OCaml or F#; a subset (Low*) can also extract to C
  - Used for crypto implementations (e.g. EverCrypt)
- Full dependent types
  - As in Coq, Agda, Lean, Idris, etc
- Rich specifications over both pure and effectful computations
  - Proof automation via an SMT solver (Z3)

$\mathrm{F}^\star$ basics

- Functional and effectful programming language / program verifier
  - A member of the ML family
  - Extracts to OCaml or F#; a subset (Low*) can also extract to C
  - Used for crypto implementations (e.g. EverCrypt)
- Full dependent types
  - As in Coq, Agda, Lean, Idris, etc
- Rich specifications over both pure and effectful computations
  - Proof automation via an SMT solver (Z3)
- **Now with a tactics and metaprogramming engine:** Meta-$\mathrm{F}^\star$
  - Automate hard proofs
  - Generate verified programs (and fragments) automatically
  - Language extensions in $\mathrm{F}^\star$

An easy example

```
let incr (r : ref int) =
    r := !r + 1

let f () : ST unit (requires (λ h → ⊤)) (ensures (λ h () h' → ⊤)) =
    let r = alloc 1 in
    incr r;
    let v = !r in
    assert (v == 2)
```

The easy VC

```
∀ (p: st_post_h heap unit) (h: heap).
  (∀ (h: heap). p () h) ⟹
  (∀ (r: ref int) (h2: heap).
      r ∉ h ∧ h2 == alloc_heap r 1 h ⟹
        r ∈ h2 ∧
        (∀ (a: int) (h3: heap).
            a == h2.[r] ∧ h3 == h2 ⟹
              (∀ (b: int).
                  b == a + 1 ⟹
                    r ∈ h3 ∧
                    (∀ (h4: heap).
                        h4 == upd h3 r b ⟹
                          r ∈ h4 ∧
                          (∀ (v: int) (h5: heap).
                              v == h4.[r] ∧ h5 == h4 ⟹
                                v == 2 ∧
                                  (v == 2 ⟹
                                      p () h5))))))
```

The easy VC

```
∀ (p: st_post_h heap unit) (h: heap).
  (∀ (h: heap). p () h) ⟹
  (∀ (r: ref int) (h2: heap).
      r ∉ h ∧ h2 == alloc_heap r 1 h ⟹
        r ∈ h2 ∧
        (∀ (a: int) (h3: heap).
            a == h2.[r] ∧ h3 == h2 ⟹
              (∀ (b: int).
                  b == a + 1 ⟹
                    r ∈ h3 ∧
                    (∀ (h4: heap).
                        h4 == upd h3 r b ⟹
                          r ∈ h4 ∧
                          (∀ (v: int) (h5: heap).
                              v == h4.[r] ∧ h5 == h4 ⟹
                                v == 2 ∧ (* our assertion *)
                                  (v == 2 ⟹
                                      p () h5))))))
```

The easy VC

```
∀ (p: st_post_h heap unit) (h: heap).
  (∀ (h: heap). p () h) ⟹
  (∀ (r: ref int) (h2: heap).
      r ∉ h ∧ h2 == alloc_heap r 1 h ⟹
        r ∈ h2 ∧
        (∀ (a: int) (h3: heap).
            a == h2.[r] ∧ h3 == h2 ⟹
              (∀ (b: int).
                  b == a + 1 ⟹
                    r ∈ h3 ∧
                    (∀ (h4: heap).
                        h4 == upd h3 r b ⟹
                          r ∈ h4 ∧
                          (∀ (v: int) (h5: heap).
                              v == h4.[r] ∧ h5 == h4 ⟹
                                v == 2 ∧ (* our assertion *)
                                  (v == 2 ⟹
                                      p () h5))))))
```

When SMT doesn't cut it

Note: Lemma φ = Pure unit (requires $\top$) (ensures (λ () $\rightarrow \varphi$))

```
let lemma_carry_limb_unrolled (a0 a1 a2 : nat)
  : Lemma (a0 % p44 + p44 * ((a1 + a0 / p44) % p44) + p88 * (a2 + ((a1 + a0 / p44) / p44))
            == a0 + p44 * a1 + p88 * a2)

  = ()
```

When SMT doesn't cut it

Note: Lemma φ = Pure unit (requires $\top$) (ensures (λ () $\rightarrow \varphi$))

```
let lemma_carry_limb_unrolled (a0 a1 a2 : nat)
  : Lemma (a0 % p44 + p44 * ((a1 + a0 / p44) % p44) + p88 * (a2 + ((a1 + a0 / p44) / p44))
            == a0 + p44 * a1 + p88 * a2)

  =
      pow2_plus 44 44;
      lemma_div_mod (a1 + a0 / p44) p44;
      lemma_div_mod a0 p44:
      distributivity_add_right p88 a2 ((a1 + a0 / p44) / p44);
      distributivity_add_right p44 ((a1 + a0 / p44) % p44) (p44 * ((a1 + a0 / p44) / p44));
      distributivity_add_right p44 a1 (a0 / p44)
```

When SMT doesn't cut it

Note: Lemma φ = Pure unit (requires $\top$) (ensures (λ () $\to \varphi$))

```
let lemma_carry_limb_unrolled (a0 a1 a2 : nat)
  : Lemma (a0 % p44 + p44 * ((a1 + a0 / p44) % p44) + p88 * (a2 + ((a1 + a0 / p44) / p44))
            == a0 + p44 * a1 + p88 * a2)

  =
⟶ pow2_plus 44 44;
⟶ lemma_div_mod (a1 + a0 / p44) p44;
⟶ lemma_div_mod a0 p44:
    distributivity_add_right p88 a2 ((a1 + a0 / p44) / p44);
    distributivity_add_right p44 ((a1 + a0 / p44) % p44) (p44 * ((a1 + a0 / p44) / p44));
    distributivity_add_right p44 a1 (a0 / p44)
```

When SMT doesn't cut it

Note: Lemma φ = Pure unit (requires $\top$) (ensures (λ () $\rightarrow \varphi$))

```
let lemma_carry_limb_unrolled (a0 a1 a2 : nat)
  : Lemma (a0 % p44 + p44 * ((a1 + a0 / p44) % p44) + p88 * (a2 + ((a1 + a0 / p44) / p44))
             == a0 + p44 * a1 + p88 * a2)
  =
  → pow2_plus 44 44;
  → lemma_div_mod (a1 + a0 / p44) p44;
  → lemma_div_mod a0 p44;
  → distributivity_add_right p88 a2 ((a1 + a0 / p44) / p44);
  → distributivity_add_right p44 ((a1 + a0 / p44) % p44) (p44 * ((a1 + a0 / p44) / p44));
  → distributivity_add_right p44 a1 (a0 / p44)
```

When SMT doesn't cut it

Note: Lemma φ = Pure u

let lemma_carry_limb_unro
: Lemma (a0 % p44 + p4
== a0 + p44 *

=

→ pow2_plus 44 44;
→ lemma_div_mod (a1 -
→ lemma_div_mod a0 p
→ distributivity_add_righ
→ distributivity_add_righ
→ distributivity_add_righ

When SMT *really* doesn't cut it

```
let lemma_poly_multiply (n p r h r0 r1 h0 h1 h2 s1 d0 d1 d2 hh : int)
  : 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 == h0 ∗ r1 + h1 ∗ r0 + h2 ∗ s1 ∧
               d2 == h2 ∗ r0 ∧ hh == d2 ∗ (n ∗ n) + d1 ∗ n + d0)
     (ensures (h ∗ r) % p == hh % p)
  =
  let r1_4 = r1 / 4 in
  let h_r_expand = (h2 ∗ (n ∗ n) + h1 ∗ n + h0) ∗ ((r1_4 ∗ 4) ∗ n + r0) in
  let hh_expand = (h2 ∗ r0) ∗ (n ∗ n) + (h0 ∗ (r1_4 ∗ 4) + h1 ∗ r0 + h2 ∗ (5 ∗ r1_4)) ∗ n
                  + (h0 ∗ r0 + h1 ∗ (5 ∗ r1_4)) in
  let b = ((h2 ∗ n + h1) ∗ r1_4) in
  modulo_addition_lemma hh_expand p b;
  assert (h_r_expand == hh_expand + b ∗ (n ∗ n ∗ 4 + (- 5)))
```

When SMT *really* doesn't cut it

```
let lemma_poly_multiply (n p r h r0 r1 h0 h1 h2 s1 d0 d1 d2 hh : int)
  : 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 == h0 ∗ r1 + h1 ∗ r0 + h2 ∗ s1 ∧
              d2 == h2 ∗ r0 ∧ hh == d2 ∗ (n ∗ n) + d1 ∗ n + d0)
    (ensures (h ∗ r) % p == hh % p)
  =
  let r1_4 = r1 / 4 in
  let h_r_expand = (h2 ∗ (n ∗ n) + h1 ∗ n + h0) ∗ ((r1_4 ∗ 4) ∗ n + r0) in
  let hh_expand = (h2 ∗ r0) ∗ (n ∗ n) + (h0 ∗ (r1_4 ∗ 4) + h1 ∗ r0 + h2 ∗ (5 ∗ r1_4)) ∗ n
                  + (h0 ∗ r0 + h1 ∗ (5 ∗ r1_4)) in
  let b = ((h2 ∗ n + h1) ∗ r1_4) in
  modulo_addition_lemma hh_expand p b;
  assert (h_r_expand == hh_expand + b ∗ (n ∗ n ∗ 4 + (- 5)))
```

- The last assertion involves **41** distributivity/associativity steps

When SMT *really* doesn't cut it

```
let lemma_poly_multiply (n p r h r0 r1 h0 h1 h2 s1 d0 d1 d2 hh : int)
  : 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 = ...
              d0 == h0 ∗ r0 + h1 ∗ s1 ∧ d1 == h0 ∗ r1 + h1 ∗ r ... ∗ s1 ∧ ...
              d2 == h2 ∗ r0 ∧ hh == d2 ∗ (n ∗ n) + d1 ∗ n + d0)
    (ensures (h ∗ r) % p == hh % p)
  =
  let r1_4 = r1 / 4 in
  let h_r_expand = (h2 ∗ (n ∗ n) + h1 ∗ n + h0) ∗ ((r1_4 ∗ 4) ∗ n ...
  let hh_expand = (h2 ∗ r0) ∗ (n ∗ n) + (h0 ∗ (r1_4 ∗ 4) + h1 ∗ ... ∗ (5 ∗ r1_ ...
                  + (h0 ∗ r0 + h1 ∗ (5 ∗ r1_4)) in
  let b = ((h2 ∗ n + h1) ∗ r1_4) in
  modulo_addition_lemma hh_expand p b;
  assert (h_r_expand == hh_expand + b ∗ (n ∗ n ∗ 4 + (- 5)))
```

- The last assertion involves **41** distributivity/associativity steps

Meet Meta-F*

A tactics and metaprogramming language for F*

- Embedded into F* as an *effect*: Tac
  - Metaprograms are terms with Tac effect
  - Exceptions, divergence and **proof state** manipulations
  - Transformations of the proof state allowed only via primitives for soundness

Meet Meta-F⋆

A tactics and metaprogramming language for F⋆

- Embedded into F⋆ as an *effect*: Tac
  - Metaprograms are terms with Tac effect
  - Exceptions, divergence and **proof state** manipulations
  - Transformations of the proof state allowed only via primitives for soundness

```
val exact : term → Tac unit
val apply_lemma : term → Tac unit
val intro : unit → Tac binder
```

Meet Meta-F*

A tactics and metaprogramming language for F*

- Embedded into F* as an *effect*: Tac
  - Metaprograms are terms with Tac effect
  - Exceptions, divergence and **proof state** manipulations
  - Transformations of the proof state allowed only via primitives for soundness

```
val exact : term → Tac unit
val apply_lemma : term → Tac unit
val intro : unit → Tac binder
```

- F* internals exposed to metaprograms
  - Inspired by Idris and Lean
  - Typechecker, normalizer, unifier, etc., are all exposed via an API
  - Inspect, create and manipulate terms and environments

Meet Meta-F*

A tactics and metaprogramming language for F*

- Embedded into F* as an *effect*: Tac
  - Metaprograms are terms with Tac effect
  - Exceptions, divergence and **proof state** manipulations
  - Transformations of the proof state allowed only via primitives for soundness

```
val exact : term → Tac unit
val apply_lemma : term → Tac unit
val intro : unit → Tac binder
```

- F* internals exposed to metaprograms
  - Inspired by Idris and Lean
  - Typechecker, normalizer, unifier, etc., are all exposed via an API
  - Inspect, create and manipulate terms and environments

```
val tc : term → Tac term
val normalize : config → term → Tac term
val unify : term → term → Tac bool
```

Metaprograms are first-class citizens

Metaprograms are written and typechecked as any other kind of effectful term:

```
let mytac () : Tac unit =
    let h1 : binder = implies_intro () in
    rewrite h1;
    apply_lemma (`eq_refl)

let test (a : int{a>0}) (b : int) =
    assert (a = b ==> f b == f a)
        by (mytac ())
```

Metaprograms are first-class citizens

Metaprograms are written and typechecked as any other kind of effectful term:

```
let mytac () : Tac unit =
    let h1 : binder = implies_intro () in
    rewrite h1;
    apply_lemma (‘eq_refl)

let test (a : int{a>0}) (b : int) =
    assert (a = b ⟹ f b == f a)
        by (mytac ())
```

```
Goal 1/1
a b : int
h0 : a > 0
────────────────────
a = b ⟹ f b == f a
```

Metaprograms are first-class citizens

Metaprograms are written and typechecked as any other kind of effectful term:

```
let mytac () : Tac unit =
    let h1 : binder = implies_intro () in  ←
    rewrite h1;
    apply_lemma (`eq_refl)

let test (a : int{a>0}) (b : int) =
    assert (a = b ⟹ f b == f a)
        by (mytac ())
```

```
Goal 1/1
a b : int
h0 : a > 0
h1 : a = b
────────────────────
f b == f a
```

Metaprograms are first-class citizens

Metaprograms are written and typechecked as any other kind of effectful term:

```
let mytac () : Tac unit =
    let h1 : binder = implies_intro () in
    rewrite h1; ←
    apply_lemma (`eq_refl)

let test (a : int{a>0}) (b : int) =
    assert (a = b ==> f b == f a)
        by (mytac ())
```

```
Goal 1/1
a b : int
h0 : a > 0
h1 : a = b
--------------------------------------------
f b == f b
```

Metaprograms are first-class citizens

Metaprograms are written and typechecked as any other kind of effectful term:

```
let mytac () : Tac unit =
    let h1 : binder = implies_intro () in
    rewrite h1;
    apply_lemma (`eq_refl) ←

let test (a : int{a>0}) (b : int) =
    assert (a = b ⟹ f b == f a)
        by (mytac ())
```

No more goals

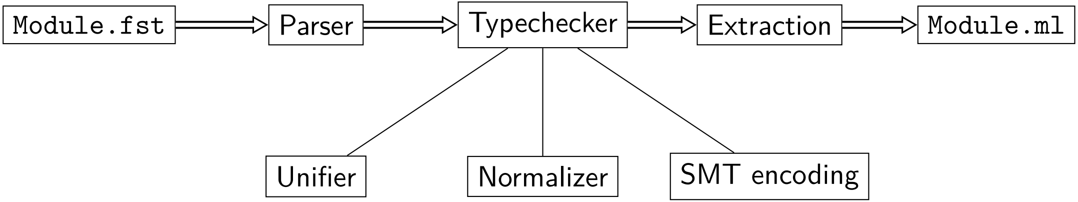
Metaprograms are first-class citizens

Further:

- Higher-order combinators and recursion
- Exceptions
- Reuse existing pure and exception-raising code

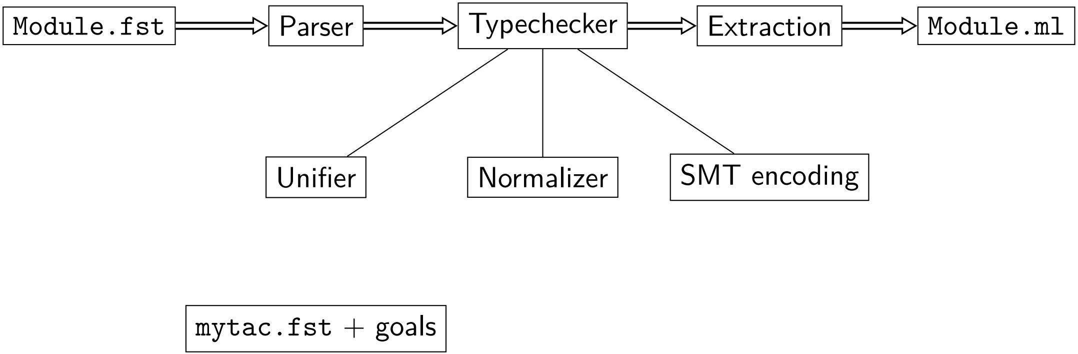
Metaprogram execution

The usual compiler pipeline:



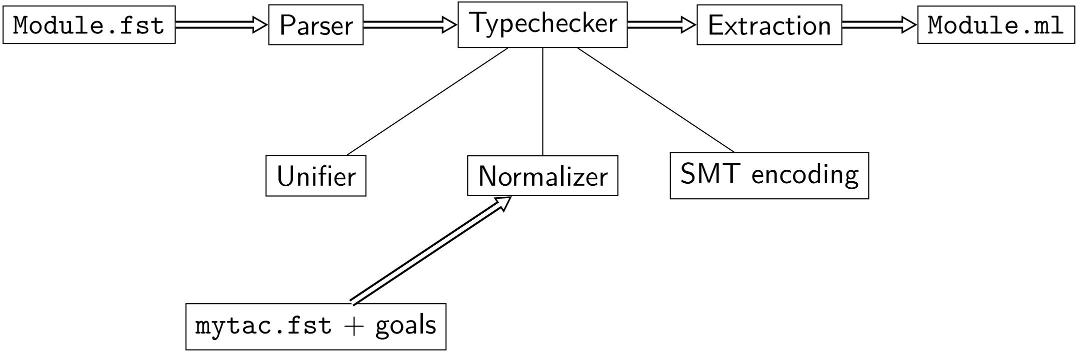
Metaprogram execution

The usual compiler pipeline:



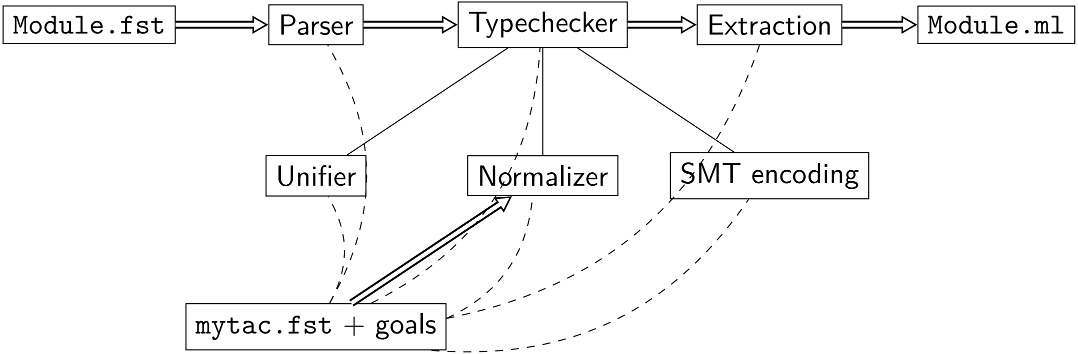
Metaprogram execution

The usual compiler pipeline:



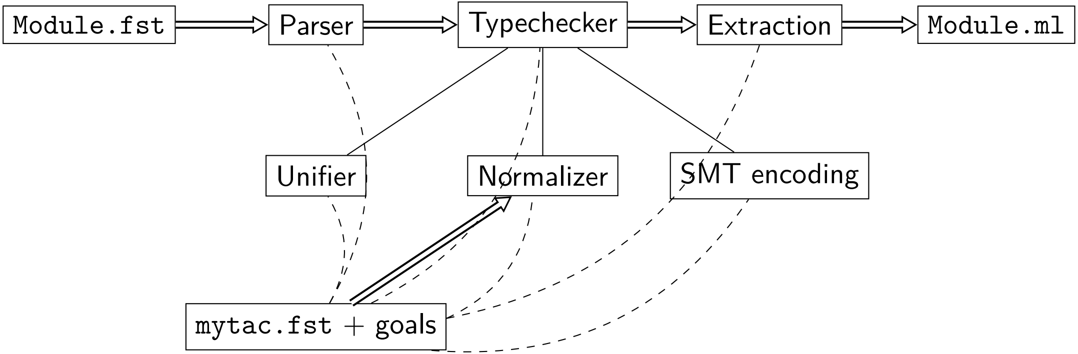
Metaprogram execution

The usual compiler pipeline:



Metaprogram execution

The usual compiler pipeline:



Metaprograms are safe **compiler scripts**

Now, let's use use Meta-F⋆

```
let lemma_poly_multiply (n p r h r0 r1 h0 h1 h2 s1 d0 d1 d2 hh : int)
  : 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 == h0 ∗ r1 + h1 ∗ r0 + h2 ∗ s1 ∧
               d2 == h2 ∗ r0 ∧ hh == d2 ∗ (n ∗ n) + d1 ∗ n + d0)
     (ensures (h ∗ r) % p == hh % p)
  =
  let r1_4 = r1 / 4 in
  let h_r_expand = (h2 ∗ (n ∗ n) + h1 ∗ n + h0) ∗ ((r1_4 ∗ 4) ∗ n + r0) in
  let hh_expand = (h2 ∗ r0) ∗ (n ∗ n) + (h0 ∗ (r1_4 ∗ 4) + h1 ∗ r0 + h2 ∗ (5 ∗ r1_4)) ∗ n
                  + (h0 ∗ r0 + h1 ∗ (5 ∗ r1_4)) in
  let b = ((h2 ∗ n + h1) ∗ r1_4) in
  modulo_addition_lemma hh_expand p b;
  assert (h_r_expand == hh_expand + b ∗ (n ∗ n ∗ 4 + (- 5)))
```

Now, let's use use Meta-F⋆

```
let lemma_poly_multiply (n p r h r0 r1 h0 h1 h2 s1 d0 d1 d2 hh : int)
  : 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 == h0 ∗ r1 + h1 ∗ r0 + h2 ∗ s1 ∧
               d2 == h2 ∗ r0 ∧ hh == d2 ∗ (n ∗ n) + d1 ∗ n + d0)
     (ensures (h ∗ r) % p == hh % p)
  =
  let r1_4 = r1 / 4 in
  let h_r_expand = (h2 ∗ (n ∗ n) + h1 ∗ n + h0) ∗ ((r1_4 ∗ 4) ∗ n + r0) in
  let hh_expand = (h2 ∗ r0) ∗ (n ∗ n) + (h0 ∗ (r1_4 ∗ 4) + h1 ∗ r0 + h2 ∗ (5 ∗ r1_4)) ∗ n
                  + (h0 ∗ r0 + h1 ∗ (5 ∗ r1_4)) in
  let b = ((h2 ∗ n + h1) ∗ r1_4) in
  modulo_addition_lemma hh_expand p b;
  assert (h_r_expand == hh_expand + b ∗ (n ∗ n ∗ 4 + (- 5))) by (canon_semiring int_cr; smt ())
```

Splitting assertions

With assert..by, the VC will not contain the obligation, instead we get a *goal*

$$\forall \mathsf{n}\ \mathsf{p}\ \mathsf{r}\ \ldots,$$
$$\varphi_1 \implies \psi_1 \wedge$$
$$\varphi_2 \implies \psi_2 \wedge$$
$$\ldots \implies \mathsf{L} = \mathsf{R} \wedge$$
$$\mathsf{L} = \mathsf{R} \implies \ldots$$

Splitting assertions

With assert..by, the VC will not contain the obligation, instead we get a *goal*

$$\forall \mathsf{n}\ \mathsf{p}\ \mathsf{r}\ \ldots,$$
$$\varphi_1 \Longrightarrow \psi_1 \land$$
$$\varphi_2 \Longrightarrow \psi_2 \land$$
$$\ldots \Longrightarrow \sout{\mathsf{L} = \mathsf{R}} \land$$
$$\mathsf{L} = \mathsf{R} \Longrightarrow \ldots$$

Goal 1/1
n : int
p : int
r : int
...
H0 : φ_1
H1 : φ_2
...

L = R

Splitting assertions

With assert..by, the VC will not contain the obligation, instead … *al*

$\forall$n p r ...,

$\varphi_1 \Longrightarrow \psi_1 \wedge$

$\quad \varphi_2 \Longrightarrow \psi_2 \wedge$

$\quad\quad \ldots \Longrightarrow$ ~~L = R~~ $\wedge$

$\quad\quad\quad$ L = R $\Longrightarrow \ldots$

Goal 1/1
n : int
p : int
r : int
...
H0 : φ_1
H1 : φ_2
...

L = R

Splitting assertions

With assert..by, the VC will not contain the obligation, instea… *al*

∀n p r ...,
$\varphi_1 \Longrightarrow \psi_1 \land$
$\quad \varphi_2 \Longrightarrow \psi_2 \land$
$\quad\quad \ldots \Longrightarrow$ ~~L = R~~ $\land$
$\quad\quad\quad$ L = R $\Longrightarrow \ldots$

Goal 1/1
n : int
p : int
r : int
...
H0 : φ_1
H1 : φ_2
...

$nf(\mathsf{L}) = nf(\mathsf{R})$

Splitting assertions

With assert..by, the VC will not contain the obligation, instea… *al*

∀n p r …,

$\varphi_1 \Longrightarrow \psi_1 \land$
$\quad \varphi_2 \Longrightarrow \psi_2 \land$
$\quad\quad$ … $\Longrightarrow$ ~~L = R~~ $\land$
$\quad\quad\quad$ L = R $\Longrightarrow$ …

Goal 1/1
n : int
p : int
r : int
…
H0 : φ_1
H1 : φ_2
…

---

nf(L) = nf(R)

Splitting assertions

With assert..by, the VC will not contain the obligation, instead … *al*

$\forall$n p r ...,
$\varphi_1 \Longrightarrow \psi_1 \land$
$\varphi_2 \Longrightarrow \psi_2 \land$
$\ldots \Longrightarrow$ ~~L = R~~ $\land$
L = R $\Longrightarrow \ldots$

Goal 1/1
n : int
p : int
r : int
...
H0 : φ_1
H1 : φ_2
...

---

nf(L) = nf(R)

Metaprogramming: generating terms

Beyond proving, Meta-F* enables constructing terms

```
let f (x y : int) : int = _ by (exact (`42))
```

Metaprogramming: generating terms

Beyond proving, Meta-$F^\star$ enables constructing terms

```
let f (x y : int) : int = ?u
```

(∗ running exact (‘42) ∗)

Goal 1/1
x : int
y : int

?u : int

Metaprogramming: generating terms

Beyond proving, Meta-$\mathrm{F}^\star$ enables constructing terms

```
let f (x y : int) : int = 42
```

No more goals

Metaprogramming: generating terms

Beyond proving, Meta-$\mathrm{F}^\star$ enables constructing terms

```
let f (x y : int) : int = 42
```

No more goals

- Metaprogramming goals are **relevant**.
- Proving goals are **irrelevant**, they have no operational meaning.
- SMT can only be called on irrelevant goals.

Customizing implicit arguments

- Meta-$\mathrm{F}^\star$ can also be used to provide strategies for resolution of implicits.

```
let diag (x:int) (#[same_as x] y : int) : int * int = (x, y)
```

Customizing implicit arguments

- Meta-$F^\star$ can also be used to provide strategies for resolution of implicits.

```
let diag (x:int) (#[same_as x] y : int) : int * int = (x, y)

diag 42 == (42, 42)
```

Customizing implicit arguments

- Meta-$F^\star$ can also be used to provide strategies for resolution of implicits.

```
let diag (x:int) (#[same_as x] y : int) : int * int = (x, y)
```

```
diag 42 == (42, 42)
```

```
diag 42 #50 == (42, 50)
```

Customizing implicit arguments

- Meta-$\mathrm{F}^\star$ can also be used to provide strategies for resolution of implicits.

```
let diag (x:int) (#[same_as x] y : int) : int * int = (x, y)

diag 42 == (42, 42)

diag 42 #50 == (42, 50)
```

- We combine this with some metaprogramming to implement typeclasses completely in **user space**.

Customizing implicit arguments

- Meta-F* can also be used to provide strategies for resolution of implicits.

```
let diag (x:int) (#[same_as x] y : int) : int * int = (x, y)

diag 42 == (42, 42)

diag 42 #50 == (42, 50)
```

- We combine this with some metaprogramming to implement typeclasses completely in **user space**.
- Dictionary resolution, tcresolve, is a 20 line metaprogram

Summary

- Mixing SMT and Tactics, use each for what they do best
  - Simplify proofs for the solver
  - No need for full decision procedures
- Meta-$\mathrm{F}^\star$ enables to extend $\mathrm{F}^\star$ in $\mathrm{F}^\star$ safely
  - Customize how terms are verified, typechecked, elaborated...
  - Native compilation allows fast extensions