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
- Usually automate proofs via tactics

Two camps of program verification

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

- Usually for pure programs
- Very expressive
- Usually automate proofs via tactics

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

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

Two camps of program verification

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

- Usually for pure programs
- Very expressive
- Usually automate proofs via tactics

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

- Verification conditions (VCs) computed and sent to SMT solvers
- Simple proofs are often fully 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
- Usually automate proofs via tactics

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

- Verification conditions (VCs) computed and sent to SMT solvers
- Simple proofs are often fully 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
- Usually automate proofs via tactics

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

- Verification conditions (VCs) computed and sent to SMT solvers
- Simple proofs are often fully 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 automation while avoiding these issues?

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 $\varphi =$ 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 () $\rightarrow \varphi$))

```
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 () $\rightarrow \varphi$))

```
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)
```

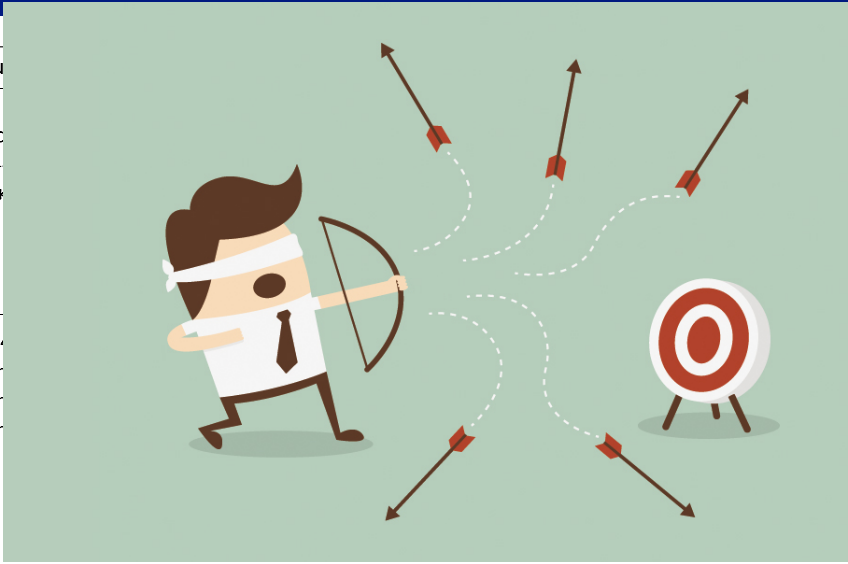
When SMT doesn't cut it

Note: Lemma $\varphi =$ 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-$F^\star$

A tactics and metaprogramming language for $F^\star$

- Embedded into $F^\star$ 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 trivial : unit → Tac unit (* solve goal if trivial *)
val apply_lemma : term → Tac unit (* use a lemma to solve the goal *)
val split : unit → Tac unit (* split a ∧ b oal into two goals *)
```

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 trivial : unit → Tac unit (* solve goal if trivial *)
val apply_lemma : term → Tac unit (* use a lemma to solve the goal *)
val split : unit → Tac unit (* split a ∧ b oal into two goals *)
```

- 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-$F^\star$

A tactics and metaprogramming language for $F^\star$

- Embedded into $F^\star$ 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 trivial : unit → Tac unit (* solve goal if trivial *)
val apply_lemma : term → Tac unit (* use a lemma to solve the goal *)
val split : unit → Tac unit (* split a ∧ b oal into two goals *)
```

- $F^\star$ 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 (* check the type of a term *)
val normalize : config → term → Tac term (* evaluate a term *)
val unify : term → term → Tac bool (* call the unifier *)
```

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;
    reflexivity ()

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;
    reflexivity ()

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;
    reflexivity ()

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; ←
    reflexivity ()

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;
    reflexivity () ←

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

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*

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

Splitting assertions

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

$\forall$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
...

L = R

Splitting assertions

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

$\forall$n p r ...,
$\varphi_1 \Longrightarrow \psi_1 \wedge$
$\varphi_2 \Longrightarrow \psi_2 \wedge$
$\ldots \Longrightarrow$ ~~L = R~~ $\wedge$
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, instead … *al*

$\forall$n p r …,

$\varphi_1 \implies \psi_1 \land$

$\varphi_2 \implies \psi_2 \land$

$\ldots \implies$ ~~L = R~~ $\land$

L = R $\implies \ldots$

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 V… will not contain the obligation, instea… *…al*

$\forall$n p r ...,

$$\varphi_1 \implies \psi_1 \land$$
$$\varphi_2 \implies \psi_2 \land$$
$$\ldots \implies \sout{L = R} \land$$
$$L = R \implies \ldots$$

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 V… ntain the obligation, instea… *al*

$\forall$n p r ...,

$\varphi_1 \implies \psi_1 \land$

$\quad \varphi_2 \implies \psi_2 \land$

$\quad\quad ... \implies$ ~~L = R~~ $\land$

$\quad\quad\quad$ L = R $\implies$ ...

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

---

nf(L) = nf(R)

Metaprogramming

Metaprogramming: generating terms

Beyond proving, Meta-$\mathrm{F}^\star$ 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-$F^\star$ 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** (can't call smt ()!).

Metaprogramming: generating terms

```
let mk_add () : Tac unit =
  let x = intro () in
  let y = intro () in
  apply (`(+));
  exact (quote y);
  exact (quote x)

let add : int → int → int =
    _ by (mk_add ())
```

Metaprogramming: generating terms

```
let mk_add () : Tac unit =
  let x = intro () in
  let y = intro () in
  apply (`(+));
  exact (quote y);
  exact (quote x)

let add : int → int → int =
    ?u
```

Goal 1/1

?u : int → int → int

Metaprogramming: generating terms

```
let mk_add () : Tac unit =
  let x = intro () in  ←
  let y = intro () in
  apply (`(+));
  exact (quote y);
  exact (quote x)

let add : int → int → int =
    λx → ?u1
```

Goal 1/1
x : int

?u1 : int → int

Metaprogramming: generating terms

```
let mk_add () : Tac unit =
  let x = intro () in
  let y = intro () in  ←
  apply (`(+));
  exact (quote y);
  exact (quote x)

let add : int → int → int =
    λx → λy → ?u2
```

```
Goal 1/1
x : int
y : int
_______________________
?u2 : int
```

Metaprogramming: generating terms

```
let mk_add () : Tac unit =
  let x = intro () in
  let y = intro () in
  apply (`(+)); ←
  exact (quote y);
  exact (quote x)

let add : int → int → int =
    λx → λy → ?u3 + ?u4
```

Goal 1/2
x : int
y : int

?u3 : int

Goal 2/2
x : int
y : int

?u4 : int

Metaprogramming: generating terms

```
let mk_add () : Tac unit =
  let x = intro () in
  let y = intro () in
  apply (‘(+));
  exact (quote y); ←
  exact (quote x)

let add : int → int → int =
    λx → λy → y + ?u4
```

Goal 1/2
x : int
y : int

?u4 : int

Metaprogramming: generating terms

```
let mk_add () : Tac unit =
  let x = intro () in
  let y = intro () in
  apply (`(+));
  exact (quote y);
  exact (quote x) ←

let add : int → int → int =
    λx → λy → y + x
```

No more goals

Deriving code from types

```
type t1 =
  | A : int → int → t1
  | B : string → t1
  | C : t1 → t1
```

Deriving code from types

```
type t1 =
  | A : int → int → t1
  | B : string → t1
  | C : t1 → t1

let t1_print : t1 → string = _ by (derive_printer ())
```

Deriving code from types

```
type t1 =
  | A : int → int → t1
  | B : string → t1
  | C : t1 → t1

let t1_print : t1 → string = _ by (derive_printer ())

let rec t1_print (v : t1) : Tot string =
  match v with
   | A x y → "(A " ^ string_of_int x ^ " " ^ string_of_int y ^ ")"
   | B s → "(B " ^ s ^ ")"
   | C x → "(C " ^ t1_print x ^ ")"
```

Deriving code from types

```
type t1 =
  | A : int → int → t1
  | B : string → t1
  | C : t1 → t1

let t1_print : t1 → string = _ by (derive_printer ())

let rec t1_print (v : t1) : Tot string =
  match v with
   | A x y → "(A " ^ string_of_int x ^ " " ^ string_of_int y ^ ")"
   | B s → "(B " ^ s ^ ")"
   | C x → "(C " ^ t1_print x ^ ")"
```

Similar to Haskell's `deriving` and OCaml's `ppx_deriving`, but completely in "user space".

Customizing implicit arguments

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

```
let id (#a:Type) (x:a) : Tot a = x
let ten = id 10 (* implicit solved to int by unifier *)
```

Customizing implicit arguments

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

```
let id (#a:Type) (x:a) : Tot a = x
let ten = id 10 (* implicit solved to int by unifier *)

let bad (x:int) (#y : int) : Tot (int * int) = (x, y)
let wontwork = bad 10 (* no information to solve y *)
```

Customizing implicit arguments

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

```
let id (#a:Type) (x:a) : Tot a = x
let ten = id 10 (* implicit solved to int by unifier *)

let bad (x:int) (#y : int) : Tot (int * int) = (x, y)
let wontwork = bad 10 (* no information to solve y *)

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

Customizing implicit arguments

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

```
let id (#a:Type) (x:a) : Tot a = x
let ten = id 10 (* implicit solved to int by unifier *)

let bad (x:int) (#y : int) : Tot (int * int) = (x, y)
let wontwork = bad 10 (* no information to solve y *)

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

diag 42 == (42, 42) (* metaprogram runs to find solution *)
```

Customizing implicit arguments

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

```
let id (#a:Type) (x:a) : Tot a = x
let ten = id 10 (* implicit solved to int by unifier *)

let bad (x:int) (#y : int) : Tot (int * int) = (x, y)
let wontwork = bad 10 (* no information to solve y *)

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

diag 42 == (42, 42) (* metaprogram runs to find solution *)

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

Customizing implicit arguments

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

```
let id (#a:Type) (x:a) : Tot a = x
let ten = id 10 (* implicit solved to int by unifier *)

let bad (x:int) (#y : int) : Tot (int * int) = (x, y)
let wontwork = bad 10 (* no information to solve y *)

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

diag 42 == (42, 42) (* metaprogram runs to find solution *)

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 id (#a:Type) (x:a) : Tot a = x
let ten = id 10 (* implicit solved to int by unifier *)

let bad (x:int) (#y : int) : Tot (int * int) = (x, y)
let wontwork = bad 10 (* no information to solve y *)

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

diag 42 == (42, 42) (* metaprogram runs to find solution *)

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

Typeclasses

```
class additive a = { zero : a; plus : a → a → a; }
    (* val zero : #a:Type → (#[tcresolve] _ : additive a) → a *)
    (* val plus : #a:Type → (#[tcresolve] _ : additive a) → a → a → a *)
```

Typeclasses

```
class additive a = { zero : a; plus : a → a → a; }
    (* val zero : #a:Type → (#[tcresolve] _ : additive a) → a *)
    (* val plus : #a:Type → (#[tcresolve] _ : additive a) → a → a → a *)

instance add_int : additive int = ...
instance add_bool : additive bool = ...
instance add_list a : additive (list a) = ...
```

Typeclasses

```
class additive a = { zero : a; plus : a → a → a; }
    (* val zero : #a:Type → (#[tcresolve] _ : additive a) → a *)
    (* val plus : #a:Type → (#[tcresolve] _ : additive a) → a → a → a *)

instance add_int : additive int = ...
instance add_bool : additive bool = ...
instance add_list a : additive (list a) = ...

let _ = assert (plus 1 2 = 3)
let _ = assert (plus true false = true)
let _ = assert (plus [1] [2] = [1;2])
```

Typeclasses

```
class additive a = { zero : a; plus : a → a → a; }
    (* val zero : #a:Type → (#[tcresolve] _ : additive a) → a *)
    (* val plus : #a:Type → (#[tcresolve] _ : additive a) → a → a → a *)

instance add_int : additive int = ...
instance add_bool : additive bool = ...
instance add_list a : additive (list a) = ...

let _ = assert (plus 1 2 = 3)
let _ = assert (plus true false = true)
let _ = assert (plus [1] [2] = [1;2])

let sum_list (#a:Type) [|additive a|] (* <- this is (#[tcresolve] _ : additive a) *)
                       (l : list a) : a = fold_right plus l zero
```

Typeclasses

```
class additive a = { zero : a; plus : a → a → a; }
    (* val zero : #a:Type → (#[tcresolve] _ : additive a) → a *)
    (* val plus : #a:Type → (#[tcresolve] _ : additive a) → a → a → a *)

instance add_int : additive int = ...
instance add_bool : additive bool = ...
instance add_list a : additive (list a) = ...

let _ = assert (plus 1 2 = 3)
let _ = assert (plus true false = true)
let _ = assert (plus [1] [2] = [1;2])

let sum_list (#a:Type) [|additive a|] (* <- this is (#[tcresolve] _ : additive a) *)
                     (l : list a) : a = fold_right plus l zero

let _ = assert (sum_list [1;2;3] == 6)
let _ = assert (sum_list [false; true] == true)
let _ = assert (sum_list [[1]; []; [2;3]] = [1;2;3])
```

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

Start with `Intro.fst`!

What are metaprograms?

- Use $\mathrm{F}^\star$'s effect extension machinery to make new effect: TAC

What are metaprograms?

- Use F*'s effect extension machinery to make new effect: TAC
  - Representation: proofstate $\rightarrow$ either error (a $*$ proofstate)
  - Completely standard and user-defined...
  - ... except for the assumed primitives

What are metaprograms?

- Use F⋆'s effect extension machinery to make new effect: TAC
  - Representation: proofstate → either error (a * proofstate)
  - Completely standard and user-defined...
  - ... except for the assumed primitives

```
type error = exn * proofstate (* error and proofstate at the time of failure *)
type result a = | Success : a → proofstate → result a | Failed : error → result a
let tac a = proofstate → Dv (result a) (* Dv: possibly diverging *)
let t_return (x:α) = λps → Success x ps
let t_bind (m:tac α) (f:α → tac β) : tac β=
            λps → match m ps with | Success x ps' → f x ps' | Error e → Error e

new_effect { TAC with repr = tac ; return = t_return ; bind = t_bind }

sub_effect DIV ⇝ TAC = ...
sub_effect EXN ⇝ TAC = ...
```

What are metaprograms?

- Use F⋆'s effect extension machinery to make new effect: TAC
  - Representation: proofstate → either error (a ∗ proofstate)
  - Completely standard and user-defined...
  - ... except for the assumed primitives

```
type error = exn ∗ proofstate (∗ error and proofstate at the time of failure ∗)
type result a = | Success : a → proofstate → result a | Failed : error → result a
let tac a = proofstate → Dv (result a) (∗ Dv: possibly diverging ∗)
let t_return (x:α) = λps → Success x ps
let t_bind (m:tac α) (f:α → tac β) : tac β=
            λps → match m ps with | Success x ps' → f x ps' | Error e → Error e

new_effect { TAC with repr = tac ; return = t_return ; bind = t_bind }

sub_effect DIV ⇝ TAC = ...
sub_effect EXN ⇝ TAC = ...
```

- No put operation, can only modify proofstate via primitives:

  exact, apply, intro, tc, raise, catch, ...

```
Goal 1/1
 n p r h r0 r1 h0 h1 h2 s1 d0 d1 d2 hh: ℤ
 p: pure_post unit
 uu___: p > 0 ∧ r_1 ≥ 0 ∧ n > 0 ∧ 4 × (n × n) == p + 5 ∧ r == r_1 × n + r_0 ∧
 h == h_2 × (n × n) + h_1 × n + h_0 ∧ s_1 == r_1 + r_1 / 4 ∧ r_1 % 4 == 0 ∧ d_0 == h_0 × r_0 + h_1 × s_1 ∧
 d_1 == h_0 × r_1 + h_1 × r_0 + h_2 × s_1 ∧ d_2 == h_2 × r_0 ∧ hh == d_2 × (n × n) + d_1 × n + d_0 ∧
 (∀ (pure_result: unit). h × r % p == hh % p ⟹ p pure_result)

 return_val: ℤ
 uu___: return_val == p
 pure_result: unit
 uu___: ((h_2 × r_0) × (n × n) + (h_0 × ((r_1 / 4) × 4) + h_1 × r_0 + h_2 × (5 × (r_1 / 4))) × n +
   (h_0 × r_0 + h_1 × (5 × (r_1 / 4))) +
   ((h_2 × n + h_1) × (r_1 / 4)) × p) %
 p =
 ((h_2 × r_0) × (n × n) + (h_0 × ((r_1 / 4) × 4) + h_1 × r_0 + h_2 × (5 × (r_1 / 4))) × n +
   (h_0 × r_0 + h_1 × (5 × (r_1 / 4)))) %
 p
 ────────────────────────────────────────────────────────────────
 squash (4 × (h_2 × (n × (n × (n × (r_1 / 4))))) + h_2 × (n × (n × r_0)) +
     (4 × (n × (n × (h_1 × (r_1 / 4)))) + n × (h_1 × r_0)) +
     (4 × (n × (h_0 × (r_1 / 4))) + h_0 × r_0) ==
     h_2 × (n × (n × r_0)) + (4 × (n × (h_0 × (r_1 / 4))) + n × (h_1 × r_0) + 5 × (h_2 × (n × (r_1 / 4)))) +
     (h_0 × r_0 + 5 × (h_1 × (r_1 / 4))) +
     (4 × (h_2 × (n × (n × (n × (r_1 / 4))))) + -5 × (h_2 × (n × (r_1 / 4))) +
     (4 × (n × (n × (h_1 × (r_1 / 4)))) + -5 × (h_1 × (r_1 / 4)))))
 (*?u4857*) _
```

A peek at tcresolve

```
let rec tcresolve' (seen:list term) (fuel:int) : Tac unit =
  if fuel ⩽ 0 then
      fail "out of fuel";
  let g = cur_goal () in
  if FStar.List.Tot.Base.existsb (term_eq g) seen then
      fail "loop";
  let seen = g :: seen in
    local seen fuel ‘or_else‘ global seen fuel
and local (seen:list term) (fuel:int) () : Tac unit =
  let bs = binders_of_env (cur_env ()) in
  first (λ b → trywith seen fuel (pack (Tv_Var (bv_of_binder b)))) bs
and global (seen:list term) (fuel:int) () : Tac unit =
  let cands = lookup_attr (‘tcinstance) (cur_env ()) in
  first (λ fv → trywith seen fuel (pack (Tv_FVar fv))) cands
and trywith (seen:list term) (fuel:int) (t:term) : Tac unit =
  (λ () → apply t) ‘seq‘ (λ () → tcresolve' seen (fuel - 1))

let tcresolve () : Tac unit = tcresolve' [] 16
```