Meta-F*

Proof automation with SMT, Tactics, and Metaprograms

Guido Martínez Danel Ahman Victor Dumitrescu Nick Giannarakis
Chris Hawblitzel Catalin Hritcu Monal Narasimhamurthy
Zoe Paraskevopoulou Clément Pit-Claudel Jonathan Protzenko
Tahina Ramananandro Aseem Rastogi Nikhil Swamy

CIFASIS-CONICET Inria Paris University of Ljubljana MSR-Inria Joint Centre
Princeton University Microsoft Research University of Colorado Boulder MIT CSAIL

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?

$\mathrm{F}^\star$ 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)

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)

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)
- **Now with a tactics and metaprogramming engine:** Meta-F*
  - Automate hard proofs
  - Generate verified programs (and fragments) automatically
  - Language extensions in F*

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

$$
\begin{array}{l}
\forall\ (\mathsf{p}\colon \mathsf{st\_post\_h\ heap\ unit})\ (\mathsf{h}\colon \mathsf{heap}).\\
\quad (\forall\ (\mathsf{h}\colon \mathsf{heap}).\ \mathsf{p}\ ()\ \mathsf{h}) \Longrightarrow\\
\quad (\forall\ (\mathsf{r}\colon \mathsf{ref\ int})\ (\mathsf{h2}\colon \mathsf{heap}).\\
\qquad\quad \mathsf{r} \notin \mathsf{h}\ \land\ \mathsf{h2} == \mathsf{alloc\_heap\ r\ 1\ h} \Longrightarrow\\
\qquad\qquad \mathsf{r} \in \mathsf{h2}\ \land\\
\qquad\qquad (\forall\ (\mathsf{a}\colon \mathsf{int})\ (\mathsf{h3}\colon \mathsf{heap}).\\
\qquad\qquad\qquad \mathsf{a} == \mathsf{h2.[r]}\ \land\ \mathsf{h3} == \mathsf{h2} \Longrightarrow\\
\qquad\qquad\qquad\quad (\forall\ (\mathsf{b}\colon \mathsf{int}).\\
\qquad\qquad\qquad\qquad\quad \mathsf{b} == \mathsf{a} + 1 \Longrightarrow\\
\qquad\qquad\qquad\qquad\qquad \mathsf{r} \in \mathsf{h3}\ \land\\
\qquad\qquad\qquad\qquad\qquad (\forall\ (\mathsf{h4}\colon \mathsf{heap}).\\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{h4} == \mathsf{upd\ h3\ r\ b} \Longrightarrow\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{r} \in \mathsf{h4}\ \land\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (\forall\ (\mathsf{v}\colon \mathsf{int})\ (\mathsf{h5}\colon \mathsf{heap}).\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{v} == \mathsf{h4.[r]}\ \land\ \mathsf{h5} == \mathsf{h4} \Longrightarrow\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{v} == 2\ \land\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (\mathsf{v} == 2 \Longrightarrow\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{p}\ ()\ \mathsf{h5}))))))
\end{array}
$$

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 p4
→ 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 ∧ [illegible]= r1 ∗ n + r0 ∧
              h == h2 ∗ (n ∗ n) + h1 ∗ n + h0 ∧ s1 == r1 + ([illegible] ∧ r1 % 4 [illegible]
              d0 == h0 ∗ r0 + h1 ∗ s1 ∧ d1 == h0 ∗ r1 + h1 ∗ r[illegible] 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) ∗ [illegible]n
  let hh_expand = (h2 ∗ r0) ∗ (n ∗ n) + (h0 ∗ (r1_4 ∗ 4) + h1 ∗[illegible]2 ∗ (5 ∗ r1_[illegible]
                  + (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-$\mathrm{F}^\star$

A tactics and metaprogramming language for $\mathrm{F}^\star$

- Embedded into $\mathrm{F}^\star$ 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 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)
```

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)
```

```
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)
```

```
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)
```

```
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)
```

No more goals

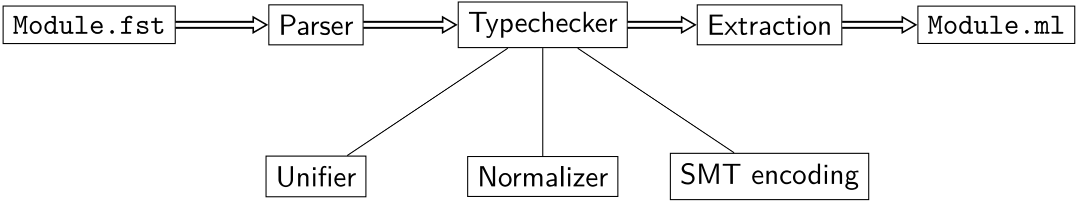
Metaprograms are first-class citizens

Further:

- Higher-order combinators and recursion
- Exceptions
- Reuse existing pure and exception-raising code
- “Lightweight” verification of metaprograms (see paper)

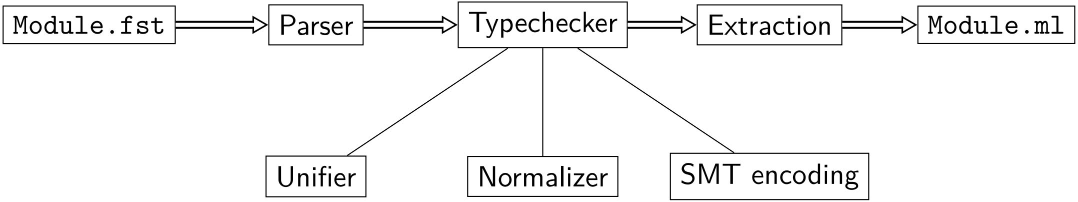
Metaprogram execution

The usual compiler pipeline:



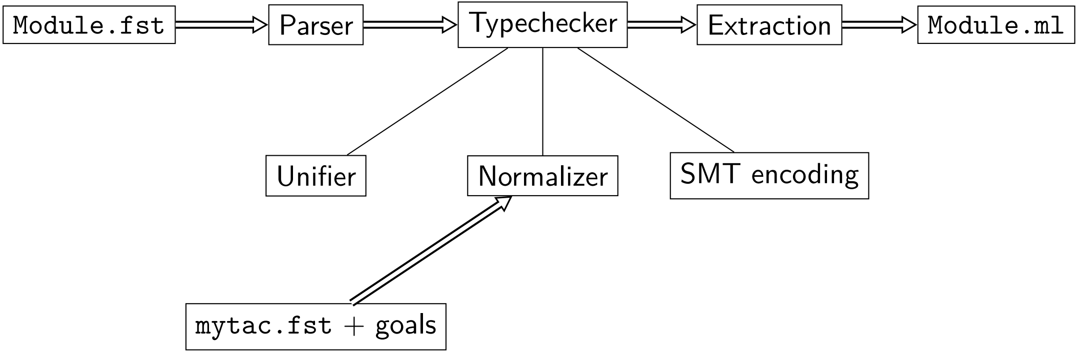
Metaprogram execution

The usual compiler pipeline:



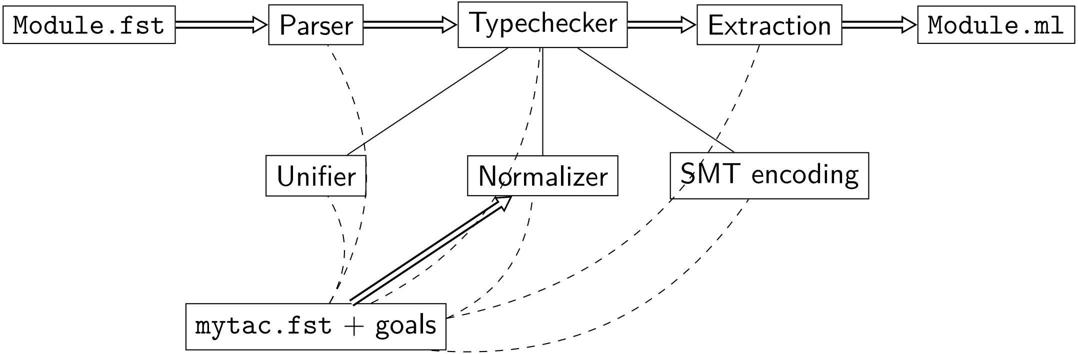
Metaprogram execution

The usual compiler pipeline:



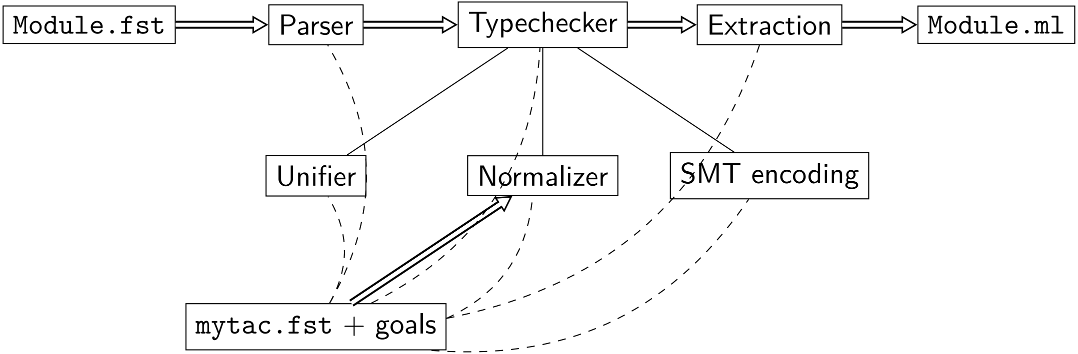
Metaprogram execution

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

VC will not contain the obligation,
instead we get a *goal* for it

$\forall$n p r ...,

$\varphi_1 \Longrightarrow \psi_1 \wedge$

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

$\quad\quad ... \Longrightarrow \mathsf{L} = \mathsf{R} \wedge$

$\quad\quad\quad \mathsf{L} = \mathsf{R} \Longrightarrow ...$

Splitting assertions

VC will not contain the obligation, instead we get a *goal* for it

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

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

L = R

Splitting assertions

VC will not contain the obligation,
instead we get a *goal* for it

$$
\begin{aligned}
&\forall \mathsf{n}\ \mathsf{p}\ \mathsf{r}\ \ldots, \\
&\varphi_1 \implies \psi_1 \land \\
&\qquad \varphi_2 \implies \psi_2 \land \\
&\qquad\qquad \ldots \implies \cancel{\mathsf{L} = \mathsf{R}} \land \\
&\qquad\qquad\qquad \mathsf{L} = \mathsf{R} \implies \ldots
\end{aligned}
$$

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

L = R

Splitting assertions

VC will not contain the obligation,
instead we get a *goal* for it

$$
\begin{array}{l}
\forall \mathsf{n\ p\ r} \ldots, \\
\varphi_1 \implies \psi_1 \wedge \\
\qquad \varphi_2 \implies \psi_2 \wedge \\
\qquad\qquad \ldots \implies \cancel{\mathsf{L} = \mathsf{R}} \wedge \\
\qquad\qquad\qquad \mathsf{L} = \mathsf{R} \implies \ldots
\end{array}
$$

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

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

- canon_semiring int_cr transforms the goal

Splitting assertions

VC will not contain the obligation, instead we get a *goal* for it

$$
\begin{aligned}
&\forall \mathsf{n}\ \mathsf{p}\ \mathsf{r}\ \ldots,\\
&\varphi_1 \Longrightarrow \psi_1 \wedge\\
&\qquad \varphi_2 \Longrightarrow \psi_2 \wedge\\
&\qquad\qquad \ldots \Longrightarrow \cancel{\mathsf{L} = \mathsf{R}} \wedge\\
&\qquad\qquad\qquad \mathsf{L} = \mathsf{R} \Longrightarrow \ldots
\end{aligned}
$$

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

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

- canon_semiring int_cr transforms the goal
- *partial* canonicalization: doesn't do AC

Splitting assertions

VC will not contain the obligation,
instead we get a *goal* for it

$$
\begin{aligned}
&\forall \mathsf{n\ p\ r}\ \ldots,\\
&\varphi_1 \Longrightarrow \psi_1 \land\\
&\qquad \varphi_2 \Longrightarrow \psi_2 \land\\
&\qquad\qquad \ldots \Longrightarrow \cancel{\mathsf{L} = \mathsf{R}} \land\\
&\qquad\qquad\qquad \mathsf{L} = \mathsf{R} \Longrightarrow \ldots
\end{aligned}
$$

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

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

- canon_semiring int_cr transforms the goal
- *partial* canonicalization: doesn't do AC
- Proof only requires linear arithmetic

Splitting assertions

VC will not contain the obligation,
instead we get a *goal* for it

$$
\begin{array}{l}
\forall \mathsf{n}\ \mathsf{p}\ \mathsf{r}\ \ldots, \\
\varphi_1 \Longrightarrow \psi_1 \wedge \\
\qquad \varphi_2 \Longrightarrow \psi_2 \wedge \\
\qquad\qquad \ldots \Longrightarrow \cancel{\mathsf{L} = \mathsf{R}} \wedge \\
\qquad\qquad\qquad \mathsf{L} = \mathsf{R} \Longrightarrow \ldots
\end{array}
$$

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

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

- canon_semiring int_cr transforms the goal
- *partial* canonicalization: doesn't do AC
- Proof only requires linear arithmetic
- Rest of proof automatic as before

Splitting assertions

VC will not contain the obligation,
instead we get a *goal* for it

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

Goal 1/1
n : int
p : int

	Success Rate
smt	0.5%
smt6x	10%
smt25x	16.5%
smt100x	24%
tactics	100%

- canon_semiring int_cr transforms the goal
- *partial* canonicalization: doesn't do AC
- Proof only requires linear arithmetic
- Rest of proof automatic as before

VC will not contain the obligation, instead we get a *goal* for it

Goal 1/1
n : int
p : int

$\forall$n p r ...,
$\varphi_1 \implies \psi_1 \land$
$\varphi_2 =$

A small tactic (+ SMT) goes a long way

- canon_semiring int_cr transforms the goal
- *partial* canonicalization: doesn't do AC
- Proof only requires linear arithmetic
- Rest of proof automatic as before

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-$\mathrm{F}^\star$ 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.

Metaprogramming parsers and serializers

```
let parser t = seq byte → option (t ∗ nat)
let serializer #t (p:parser t) = f:(t → seq byte){∀ x. p (f x) == Some (x, length (f x))}
type package t = { p : parser t ; s : serializer p }
```

Metaprogramming parsers and serializers

```
let parser t = seq byte → option (t ∗ nat)
let serializer #t (p:parser t) = f:(t → seq byte){∀ x. p (f x) == Some (x, length (f x))}
type package t = { p : parser t ; s : serializer p }

type sample = nlist 18 (u8 ∗ u8)
```

Metaprogramming parsers and serializers

```
let parser t = seq byte → option (t ∗ nat)
let serializer #t (p:parser t) = f:(t → seq byte){∀ x. p (f x) == Some (x, length (f x))}
type package t = { p : parser t ; s : serializer p }

type sample = nlist 18 (u8 ∗ u8)

let ps_sample : package sample = _ by (gen_specs (‘sample))
```

Metaprogramming parsers and serializers

```
let parser t = seq byte → option (t ∗ nat)
let serializer #t (p:parser t) = f:(t → seq byte){∀ x. p (f x) == Some (x, length (f x))}
type package t = { p : parser t ; s : serializer p }

type sample = nlist 18 (u8 ∗ u8)

let ps_sample : package sample = _ by (gen_specs (‘sample))

let p_low : parser_impl ps_sample.p = _ by gen_parser_impl
let s_low : serializer_impl ps_sample.s = _ by gen_serializer_impl
```

Metaprogramming parsers and serializers

```
let parser t = seq byte → option (t ∗ nat)
let serializer #t (p:parser t) = f:(t → seq byte){∀ x. p (f x) == Some (x, length (f x))}
type package t = { p : parser t ; s : serializer p }

type sample = nlist 18 (u8 ∗ u8)

let ps_sample : package sample = _ by (gen_specs (‘sample))

let p_low : parser_impl ps_sample.p = _ by gen_parser_impl
let s_low : serializer_impl ps_sample.s = _ by gen_serializer_impl
```

- All proofs done automatically by the tactic
- Generated parsers/serializers are in Low*, can be extracted to C

Parsers and serializers, before

Part of TLS' handshake:

```
let cipherSuiteBytesOpt (cs : ciperSuite) : option (lbytes 2) =
    match cs with
    | NullCipherSuite → Some (0x00uy, 0x00uy)
    | CipherSuite Kex_RSA None (MACOnly MD5) → Some (0x00uy, 0x01uy)
    | CipherSuite Kex_RSA None (MACOnly SHA1) → Some (0x00uy, 0x02uy)
    | CipherSuite Kex_RSA None (MACOnly SHA256) → Some (0x00uy, 0x3Buy)
    .... (* 55 more cases *)

let parseCipherSuite (b : lbytes 2) : result cipherSuite =
    match cbyte2 b with
    | (0x00uy, 0x00uy) → Correct (NullCipherSuite)
    | (0x00uy, 0x01uy) → Correct (CipherSuite Kex_RSA None (MACOnly MD5))
    | (0x00uy, 0x02uy) → Correct (CipherSuite Kex_RSA None (MACOnly SHA1))
    | (0x00uy, 0x3Buy) → Correct (CipherSuite Kex_RSA None (MACOnly SHA256))
    .... (* 55 more cases *)
```

Parsers and serializers, before

Part of TLS' handshake:

```
let cipherSuiteBytesOpt (cs : ciperSuite) : option (lbytes 2) =
  match cs with
  | NullCipherSuite → Some (0x00uy, 0x00uy)
  | CipherSuite Kex_RSA None (MACOnly MD5) → Some (0x00uy, 0x01uy)
  | CipherSuite Kex_RSA None (MACOnly SHA1) → Some (0x00uy, 0x02uy)
  | CipherSuite Kex_RSA None (MACOnly SHA256) → Some (0x00uy, 0x3Buy)
  .... (* 55 more cases *)

let parseCipherSuite (b : lbytes 2) : result cipherSuite =
  match cbyte2 b with
  | (0x00uy, 0x00uy) → Correct (NullCipherSuite)
  | (0x00uy, 0x01uy) → Correct (CipherSuite Kex_RSA None (MACOnly MD5))
  | (0x00uy, 0x02uy) → Correct (CipherSuite Kex_RSA None (MACOnly SHA1))
  | (0x00uy, 0x3Buy) → Correct (CipherSuite Kex_RSA None (MACOnly SHA256))
  .... (* 55 more cases *)

  (* + proof of mutual correspondance via SMT *)
```

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)
```

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)
```

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)
```

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

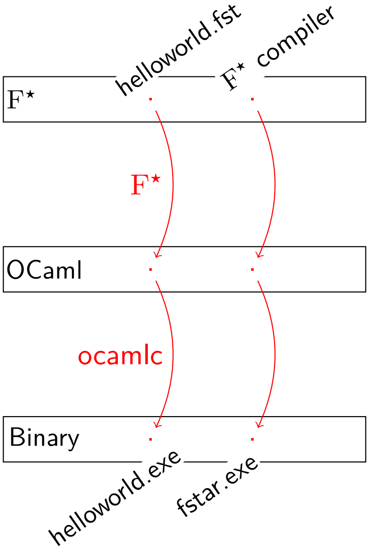
  let diag (x:int) (#[same_as x] y : int) : int $*$ int $=$ (x, y)

  diag 42 $==$ (42, 42)

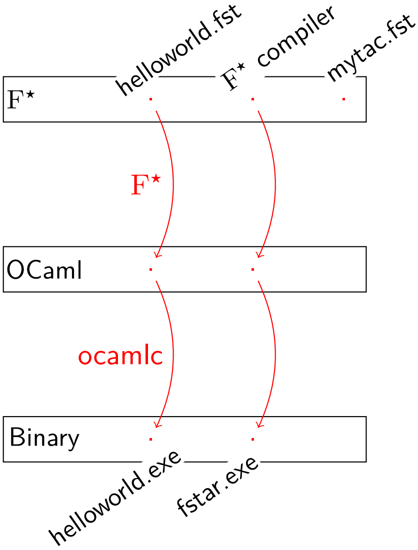
  diag 42 #50 $==$ (42, 50)

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

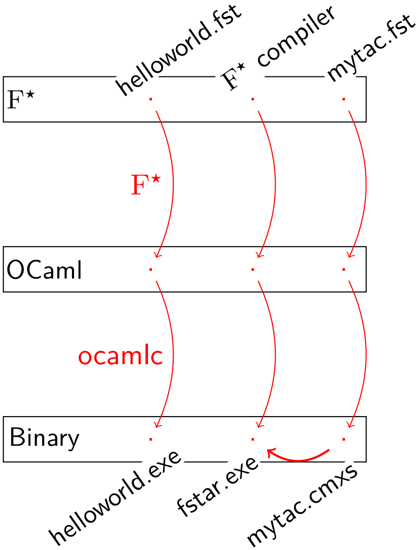
Native metaprograms



Native metaprograms

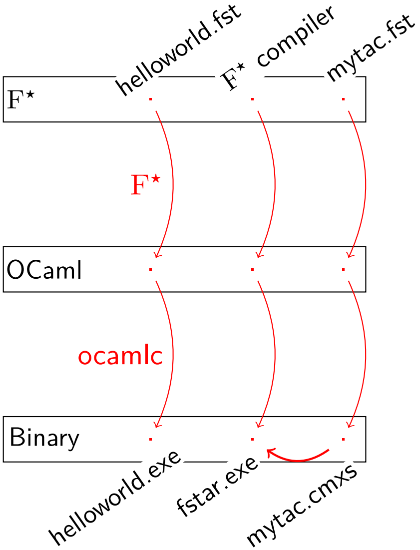


Native metaprograms



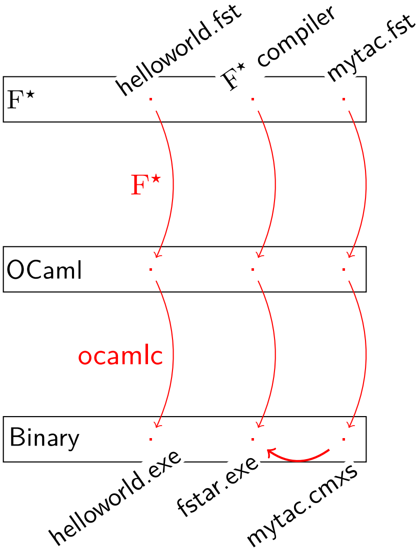
- Extract, compile, and load

Native metaprograms



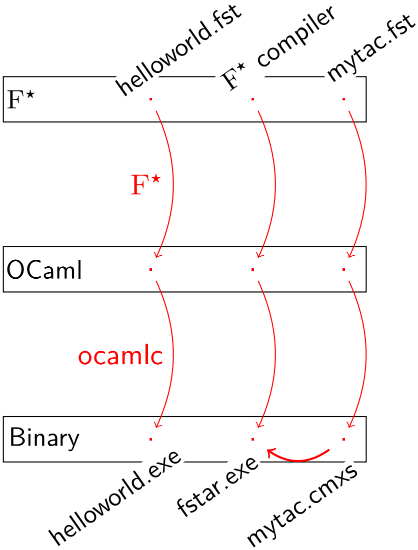
- Extract, compile, and load
- Run metaprogram natively, no interpretation needed
- New primitive: 10x speed gain!

Native metaprograms



- Extract, compile, and load
- Run metaprogram natively, no interpretation needed
- New primitive: 10x speed gain!
- Can be done for tcresolve: user level typeclasses running at native speed.

Native metaprograms



- Extract, compile, and load
- Run metaprogram natively, no interpretation needed
- New primitive: 10x speed gain!
- Can be done for tcresolve: user level typeclasses running at native speed.
  - Safe and fast compiler extensions!

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
- Using an effect for metaprograms increases reuse

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
- Using an effect for metaprograms increases reuse

In the paper:

- Details on implementation
- Trust argument and TCB
- Specifying metaprograms
- More examples

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
- Using an effect for metaprograms increases reuse

In the paper:

- Details on implementation
- Trust argument and TCB
- Specifying metaprograms
- More examples

Thank you!