ML as a Tactic Language, Again

A look at Meta-$\mathrm{F}^\star$

The $\mathrm{F}^\star$ team

ML Workshop 2018

ML as a Tactic Language, Again

A look at Meta-$\mathrm{F}^\star$

The $\mathrm{F}^\star$ team

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

ML Workshop 2018

What's this $\mathrm{F}^\star$ thing?

A dependently-typed, effectful, functional programming language

- ML-like: higher-order, polymorphic, strict, safe
- Very expressive specs over effectful terms
- Proof-irrelevant logic: refinements and WP-calculus
- SMT backend (Z3)

An example

```
let incr (r:ref int) = r := !r + 1
let f () =
    let r = alloc 1 in
    incr r;
    let v = !r in
    assert (v == 2)
```

An example

```
let incr (r:ref int) = r := !r + 1
let f () =
    let r = alloc 1 in
    incr r;
    let v = !r in
    assert (v == 2)
```

```
st_wp a = (state → a → prop) → state → prop
```

```
val alloc : #a:Type → x:a → ST (ref a) (λ p h → ∀r. r ∉h ⟹ p (alloc_heap r x h) r)
val (!) : #a:Type → r:ref a → ST a (λ p h → h ∈p ∧ p h h.[r])
val (:=) : #a:Type → r:ref a → v:a → ST unit (λ p h → h ∈p ∧ p (upd h r v) ())
```

```
∀(p: st_post_h heap unit)
  (h: heap).
  (∀ (h1: heap). auto_squash (p a h1)) ⟹
  (∀ (a: mref int (trivial_preorder int))
      (h1: heap).
      fresh a h h1 ∧ modifies empty h h1 ∧
      sel h1 a == 1 ⟹
      (∀ (return_val: int).
          return_val == sel h1 a + 1 ⟹
          trivial_preorder int (sel h1 a) return_val ∧
              (h1: _: heap
                {trivial_preorder int (sel h1 a) return_val})
            .
              trivial_preorder int (sel h1 a) return_val ∧
              contains h1 a ∧
              modifies (singleton (addr_of a))
                h1
                h1 ∧ equal_dom h1 h1 ∧
              sel h1 a == return_val ⟹
              sel h1 a == 2 ∧
                  sel h1 a == 2 ⟹ p pure_result h1))
```

Doing proofs

What's the modus operandi?

Doing proofs

What's the modus operandi?

- User writes program
- $\mathrm{F}^\star$ computes VC
- $\mathrm{F}^\star$ queries the SMT solver

Doing proofs

What's the modus operandi?

- User writes program
- $\mathrm{F}^\star$ computes VC
- $\mathrm{F}^\star$ queries the SMT solver

All user input happens before VC generation: *auto-active verification*

Doing proofs

What's the modus operandi?

- User writes program
- $\mathrm{F}^\star$ computes VC
- $\mathrm{F}^\star$ queries the SMT solver

All user input happens before VC generation: *auto-active verification*
What does the user do when then solver fails?

Doing proofs

What's the modus operandi?

- User writes program
- $\mathrm{F}^\star$ computes VC
- $\mathrm{F}^\star$ queries the SMT solver

All user input happens before VC generation: *auto-active verification*
What does the user do when then solver fails?

```
val lemma_mod_spec: a:int → p:pos → Lemma (a / p = (a - (a % p)) / p)
```

Doing proofs

What's the modus operandi?

- User writes program
- $\mathrm{F}^\star$ computes VC
- $\mathrm{F}^\star$ queries the SMT solver

All user input happens before VC generation: *auto-active verification*
What does the user do when then solver fails?

```
val lemma_mod_spec: a:int → p:pos → Lemma (a / p = (a - (a % p)) / p)
let lemma_mod_spec a p = ()
```

Doing proofs

What's the modus operandi?

- User writes program
- $\mathrm{F}^\star$ computes VC
- $\mathrm{F}^\star$ queries the SMT solver

All user input happens before VC generation: *auto-active verification*

What does the user do when then solver fails?

```
val lemma_mod_spec: a:int → p:pos → Lemma (a / p = (a - (a % p)) / p)
let lemma_mod_spec a p =
    lemma_div_mod a p;
    assert (a = p * (a / p) + a % p);
    assert (a % p = a - p * (a / p));
    assert (a - (a % p) = p * (a / p));
    lemma_mod_plus 0 (a / p) p;
    assert ((p * (a / p)) % p = 0);
    lemma_div_exact (p * (a / p)) p
```

Interaction

Some proofs are just out of the solver's scope

Interaction

Some proofs are just out of the solver's scope
And the programmer's sanity scope...

Interaction

Some proofs are just out of the solver's scope
And the programmer's sanity scope...
Tactics engine: tweak proof obligations before SMT, *interactive*[1] verification

[1] WIP

Interaction

Some proofs are just out of the solver's scope
And the programmer's sanity scope...
Tactics engine: tweak proof obligations before SMT, *interactive*[1] verification

- User writes program
- $\mathrm{F}^\star$ computes VC
- Custom tactics apply
- $\mathrm{F}^\star$ *maybe* queries the SMT solver

[1] WIP

Interaction

Some proofs are just out of the solver's scope
And the programmer's sanity scope...
Tactics engine: tweak proof obligations before SMT, *interactive*[1] verification

- User writes program
- $\mathrm{F}^\star$ + custom tactics compute VC
- $\mathrm{F}^\star$ *maybe* queries the SMT solver

[1] WIP

The entrypoints

Proving:

... assert φ by τ ...

Catching entire VCs:

let e : C by τ = ...

Metaprogramming terms:

... (_ by τ) ...

Metaprogramming top-level definitions:

%splice τ

The entrypoints

Proving:

... assert φ by τ ... $\quad \Gamma \vdash \_ : \text{squash } \tau$

Catching entire VCs:

let e : C by τ = ... $\quad \Gamma \vdash \_ : \text{squash VC}$

Metaprogramming terms:

... (_ by τ) ... $\quad \Gamma \vdash \_ : \text{ty}$

Metaprogramming top-level definitions:

%splice τ $\quad \cdot \vdash \_ : \text{list def}$

Slapping a tactic on an assertion

```
let f x y z =
    ...
    assert (g x y == h z)
    ...
```

Slapping a tactic on an assertion

```
let f x y z =
    ...
    assert (g x y == h z)
        by (compute ();
            canon_semiring int_cr;
            trefl ());
    ...
```

Metaprogramming: generating terms

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

```
let one = _ by (exact (`1))
let one_plus_two = _ by (apply (`(+)); exact (`1); exact (`2))
```

Metaprogramming: generating terms

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

```
let one = _ by (exact (`1))
let one_plus_two = _ by (apply (`(+)); exact (`1); exact (`2))

let fib (i:nat) = if i < 2 then exact (`1)
                  else begin
                    apply (`(+));
                    fib (i - 1);
                    fib (i - 2)
                  end

let t = _ by (fib 8)
```

Metaprogramming: generating terms

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

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

%splice (mk_printer (`t1))
```

Metaprogramming: generating terms

Beyond proving, Meta-$F^\star$ enables constructing terms and top-level definitions

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

%splice (mk_printer (`t1))
```

```
let t1_print = λv →
  let rec ff_rec v_inner =
    (match v_inner with
      | A a1 a2 →
        concat "" [ "(";
                    concat "  " ["A"; print_int a1; print_string a2];
                    ")" ]
      | B a1 a2 → concat ""
                    ["("; concat "  " ["B"; ff_rec a1; print_int a2]; ")"]
      | C → "C"
      | ...
  in ff_rec v
```

Control flow

```
val f : n:int → Pure int (requires ⊤) (ensures (λ n' → n' > n))
let f (n:int) =
    if n > 0
    then begin
         assert (n + n > n) by τ;
         n + n
         end
    else 1 - n
```

Control flow

```
val f : n:int → Pure int (requires ⊤) (ensures (λ n' → n' > n))
let f (n:int) =
    if n > 0
    then begin
         assert (n + n > n) by τ;
         n + n
         end
    else 1 - n
```

Context in the proofstate contains hypothesis that `n > 0`.

Control flow

```
val f : n:int → Pure int (requires ⊤) (ensures (λ n' → n' > n))
let f (n:int) =
    if n > 0
    then begin
         assert (n + n > n) by τ;
         n + n
         end
    else 1 - n
```

Context in the proofstate contains hypothesis that $n > 0$.

```
Goal 1/1
n: int
p: pure_post int
...
uu____: squash (n > 0 == true)
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
squash (n + n > n)
```

Extracting sub-VCs

abstract with_tactic (τ : unit $\to$ Tac unit) (φ : prop) : prop = φ

assert φ by τ : Pure unit (requires (with_tactic $\tau\,\varphi$)) (ensures (λ _ $\to \varphi$)))

- In a positive position: with_tactic $\tau\,\varphi \rightsquigarrow \top$, run τ on φ and prove resulting goals too

Extracting sub-VCs

abstract with_tactic (τ : unit $\to$ Tac unit) (φ : prop) : prop $= \varphi$

assert φ by τ : Pure unit (requires (with_tactic $\tau\,\varphi$)) (ensures (λ _ $\to \varphi$)))

- In a positive position: with_tactic $\tau\,\varphi \rightsquigarrow \top$, run τ on φ and prove resulting goals too

$$\frac{\Gamma \vDash C[\top] \quad \Gamma' \vDash \varphi}{\Gamma \vDash C[\varphi]}$$

Extracting sub-VCs

abstract with_tactic (τ : unit $\to$ Tac unit) (φ : prop) : prop = φ

assert φ by τ : Pure unit (requires (with_tactic τ φ)) (ensures (λ _ $\to$ φ)))

- In a positive position: with_tactic τ φ $\rightsquigarrow$ $\top$, run τ on φ and prove resulting goals too

$$\frac{\Gamma \vDash C[\top] \quad \Gamma' \vDash \varphi}{\Gamma \vDash C[\varphi]} \qquad \tau\ [(\Gamma', \varphi)] \to^* [(\Gamma_i, \varphi_i)]$$

Extracting sub-VCs

abstract with_tactic (τ : unit $\to$ Tac unit) (φ : prop) : prop = φ

assert φ by τ : Pure unit (requires (with_tactic $\tau\,\varphi$)) (ensures (λ _ $\to \varphi$)))

- In a positive position: with_tactic $\tau\,\varphi \rightsquigarrow \top$, run τ on φ and prove resulting goals too

$$\frac{\Gamma \vDash C[\top] \qquad \dfrac{\Gamma_i \vDash \varphi_i}{\Gamma' \vDash \varphi}}{\Gamma \vDash C[\varphi]} \qquad\qquad \tau\ [(\Gamma', \varphi)] \to^* [(\Gamma_i, \varphi_i)]$$

Extracting sub-VCs

abstract with_tactic (τ : unit $\to$ Tac unit) (φ : prop) : prop = φ

assert φ by τ : Pure unit (requires (with_tactic $\tau\,\varphi$)) (ensures (λ _ $\to \varphi$)))

- In a positive position: with_tactic $\tau\,\varphi \rightsquigarrow \top$, run τ on φ and prove resulting goals too

$$\frac{\Gamma \vDash C[\top] \qquad \dfrac{\Gamma_i \vDash \varphi_i}{\Gamma' \vDash \varphi}}{\Gamma \vDash C[\varphi]} \qquad\qquad \tau\ [(\Gamma', \varphi)] \to^* [(\Gamma_i, \varphi_i)]$$

- In a negative position: with_tactic $\tau\,\varphi \rightsquigarrow \varphi$

The proofstate

- Metaprograms, in general, perform correct transformations of the *proofstate*.

The proofstate

- Metaprograms, in general, perform correct transformations of the *proofstate*.
- Basically a set of "goals" to be fulfilled

$$\Gamma_0 \vdash ?_0 : t_0$$

The proofstate

- Metaprograms, in general, perform correct transformations of the *proofstate*.
- Basically a set of "goals" to be fulfilled

$$\frac{\Gamma_1 \vdash ?_1 : t_1 \quad \Gamma_2 \vdash ?_2 : t_2 \quad \ldots \quad \Gamma_n \vdash ?_n : t_n}{\Gamma_0 \vdash ?_0 : t_0}\ R$$

The proofstate

- Metaprograms, in general, perform correct transformations of the *proofstate*.
- Basically a set of "goals" to be fulfilled

$$\frac{\Gamma_1 \vdash ?_1 : t_1 \quad \Gamma_2 \vdash ?_2 : t_2 \quad \ldots \quad \Gamma_n \vdash ?_n : t_n}{\Gamma_0 \vdash R[?_1, \ldots, ?_n] : t_0}\ R$$

The proofstate

- Metaprograms, in general, perform correct transformations of the *proofstate*.
- Basically a set of "goals" to be fulfilled

$$\frac{\Gamma_1 \vdash ?_1 : t_1 \quad \Gamma_2 \vdash ?_2 : t_2 \quad \ldots \quad \Gamma_n \vdash ?_n : t_n}{\Gamma_0 \vdash R[?_1, \ldots, ?_n] : t_0}\ R$$

- The proofstate is transformed by primitives, which implement typing rules in backwards style

The proofstate

- Metaprograms, in general, perform correct transformations of the *proofstate*.
- Basically a set of "goals" to be fulfilled

$$\frac{\Gamma_1 \vdash ?_1 : t_1 \quad \Gamma_2 \vdash ?_2 : t_2 \quad \ldots \quad \Gamma_n \vdash ?_n : t_n}{\Gamma_0 \vdash R[?_1, \ldots, ?_n] : t_0}\ R$$

- The proofstate is transformed by primitives, which implement typing rules in backwards style
- Essentially scripting typing derivations instead of term syntax

What are metaprograms?

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

What are metaprograms?

- Use $\mathrm{F}^\star$'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 $\mathrm{F}^\star$'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 $\mathrm{F}^\star$'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, ...

TAC is a first-class citizen of $\mathrm{F}^\star$

Metaprograms are written as any other kind of program:

```
let mytac () : Tac unit =
    let h1 = implies_intro () in
    rewrite h1;
    apply_lemma (`mylem);
    ...
```

TAC is a first-class citizen of $\mathrm{F}^\star$

Higher-order combinators too:

```
let rec repeat (t : unit → Tac α) : Tac (list α) =
    match catch t with
    | Inl _ → []
    | Inr x → x :: repeat t

let repeat1 (t : unit → Tac α) : Tac (list α) = t () :: repeat t
```

TAC is a first-class citizen of $\mathrm{F}^\star$

Use exceptions:

```
exception NotAForall

let forall_intro () : Tac binder =
    let g = cur_goal () in
    match term_as_formula g with
    | Forall _ _ → begin apply_lemma (`fa_intro_lem); intro () end
    | _ → raise NotAForall

let t () =
    try forall_intro () with
    | NotAForall → ()
    | e → raise e
```

TAC is a first-class citizen of $\mathrm{F}^\star$

Call into existing pure, diverging, and exception-raising code:

```
let f (seen:list term) : Tac unit =
    let g = cur_goal () in
    if FStar.List.Tot.Base.existsb (term_eq g) seen then
      raise Loop;
    ...
```

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

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

let _ = assert (diag 42 == (42, 42))
```

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)

let _ = assert (diag 42 == (42, 42))

let _ = assert (diag 42 #21 == (42, 21))
```

Putting it together: Typeclasses

Putting it together: Typeclasses

```
module FStar.Typeclasses
val mk_class : name → Tac unit (* creates method toplevels *)
val tcresolve : unit → Tac unit (* resolves dictionaries *)
```

Putting it together: Typeclasses

```
module FStar.Typeclasses
val mk_class : name → Tac unit (* creates method toplevels *)
val tcresolve : unit → Tac unit (* resolves dictionaries *)

module MyMod
noeq type deq a = {
  eq : a → a → bool;
  eq_ok : (x:a) → (y:a) → Lemma (eq x y ⟺ x == y)
}
```

Putting it together: Typeclasses

```
module FStar.Typeclasses
val mk_class : name → Tac unit (* creates method toplevels *)
val tcresolve : unit → Tac unit (* resolves dictionaries *)

module MyMod
noeq type deq a = {
  eq : a → a → bool;
  eq_ok : (x:a) → (y:a) → Lemma (eq x y ⟺ x == y)
}

%splice (mk_class "MyMod.deq")
```

Putting it together: Typeclasses

```
module FStar.Typeclasses
val mk_class : name → Tac unit (* creates method toplevels *)
val tcresolve : unit → Tac unit (* resolves dictionaries *)

module MyMod
noeq type deq a = {
  eq : a → a → bool;
  eq_ok : (x:a) → (y:a) → Lemma (eq x y ⟺ x == y)
}

%splice (mk_class "MyMod.deq")

(* val eq : #a:Type → (#[tcresolve] d : deq a) → (a → a → bool) *)
(* val eq_ok : #a:Type → (#[tcresolve] d : deq a) → ((x:a) → (y:a) → Lemma ...) *)
```

Putting it together: Typeclasses

```
module FStar.Typeclasses
val mk_class : name → Tac unit (* creates method toplevels *)
val tcresolve : unit → Tac unit (* resolves dictionaries *)

module MyMod
class deq a = {
  eq : a → a → bool;
  eq_ok : (x:a) → (y:a) → Lemma (eq x y ⟺ x == y)
}
(* val eq : #a:Type → (#[tcresolve] d : deq a) → (a → a → bool) *)
(* val eq_ok : #a:Type → (#[tcresolve] d : deq a) → ((x:a) → (y:a) → Lemma ...) *)
```

Putting it together: Typeclasses

```
module FStar.Typeclasses
val mk_class : name → Tac unit (* creates method toplevels *)
val tcresolve : unit → Tac unit (* resolves dictionaries *)

module MyMod
class deq a = {
  eq : a → a → bool;
  eq_ok : (x:a) → (y:a) → Lemma (eq x y ⟺ x == y)
}
(* val eq : #a:Type → [| deq a |] → (a → a → bool) *)
(* val eq_ok : #a:Type → [| deq a |] → ((x:a) → (y:a) → Lemma ...) *)
```

Putting it together: Typeclasses

```
module FStar.Typeclasses
val mk_class : name → Tac unit (* creates method toplevels *)
val tcresolve : unit → Tac unit (* resolves dictionaries *)

module MyMod
class deq a = {
  eq : a → a → bool;
  eq_ok : (x:a) → (y:a) → Lemma (eq x y ⟺ x == y)
}
(* val eq : #a:Type → [| deq a |] → (a → a → bool) *)
(* val eq_ok : #a:Type → [| deq a |] → ((x:a) → (y:a) → Lemma ...) *)

[@tcinstance] let eq_int : deq int = { eq = (λ x y → x = y); eq_ok = easy }
```

Putting it together: Typeclasses

```
module FStar.Typeclasses
val mk_class : name → Tac unit (* creates method toplevels *)
val tcresolve : unit → Tac unit (* resolves dictionaries *)

module MyMod
class deq a = {
  eq : a → a → bool;
  eq_ok : (x:a) → (y:a) → Lemma (eq x y ⟺ x == y)
}
(* val eq : #a:Type → [| deq a |] → (a → a → bool) *)
(* val eq_ok : #a:Type → [| deq a |] → ((x:a) → (y:a) → Lemma ...) *)

instance eq_int : deq int = { eq = (λ x y → x = y); eq_ok = easy }
```

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
```

Native execution

- By default, metaprograms are evaluated in $\mathrm{F}^\star$'s normalizer. Primitives are evaluated by calling into compiler internals.

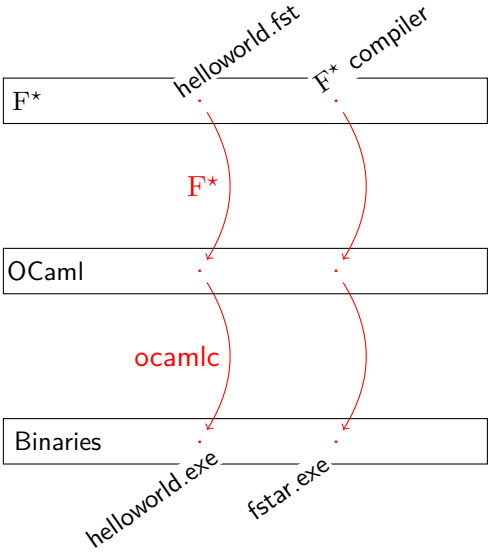
Native execution

- By default, metaprograms are evaluated in $\mathrm{F}^\star$'s normalizer. Primitives are evaluated by calling into compiler internals.
- As an alternative, we can reuse $\mathrm{F}^\star$'s extraction mechanism to compile metaprograms as OCaml code.

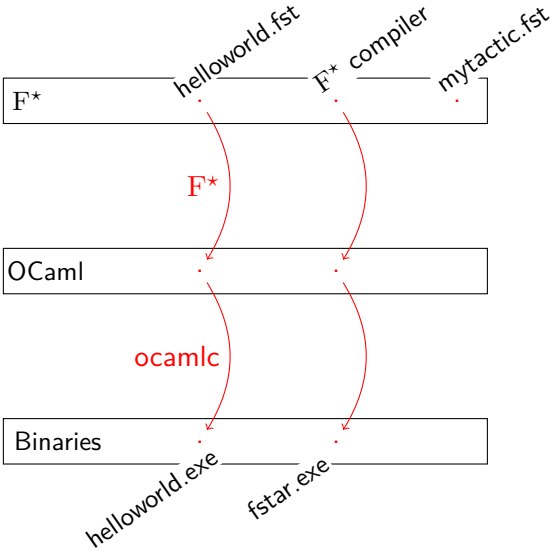
Native execution

- By default, metaprograms are evaluated in $\mathrm{F}^\star$'s normalizer. Primitives are evaluated by calling into compiler internals.
- As an alternative, we can reuse $\mathrm{F}^\star$'s extraction mechanism to compile metaprograms as OCaml code.
- This code can then be compiled into dynamic objects and loaded

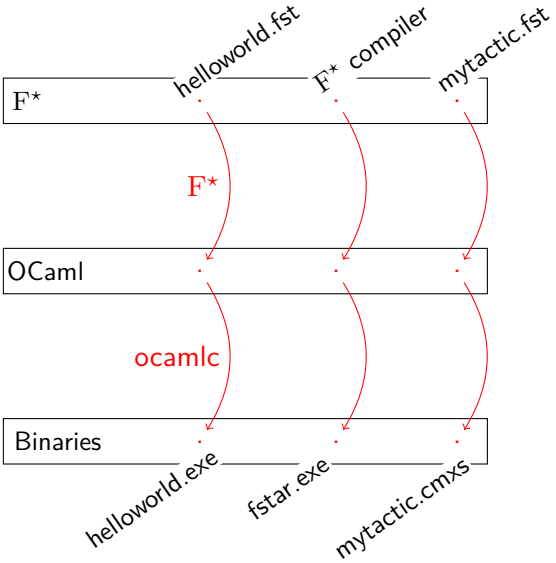
Compilation



Compilation

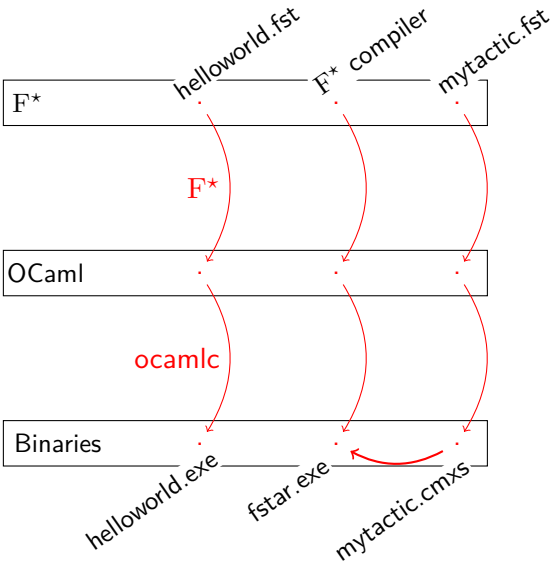


Compilation



- Extract the user tactic and compile it into a dynamic object

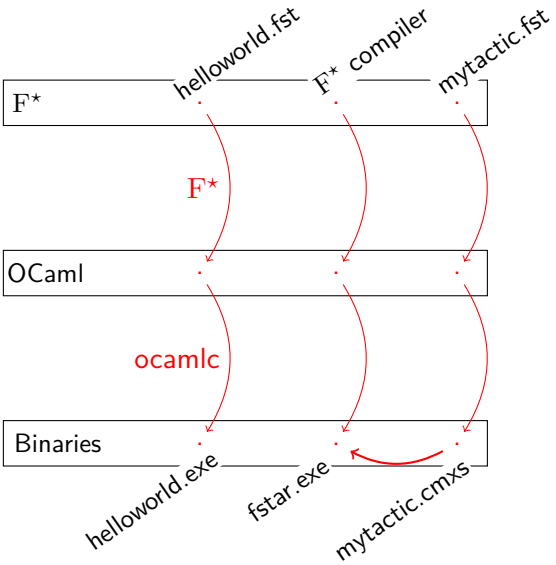
Compilation



- Extract the user tactic and compile it into a dynamic object
- Run

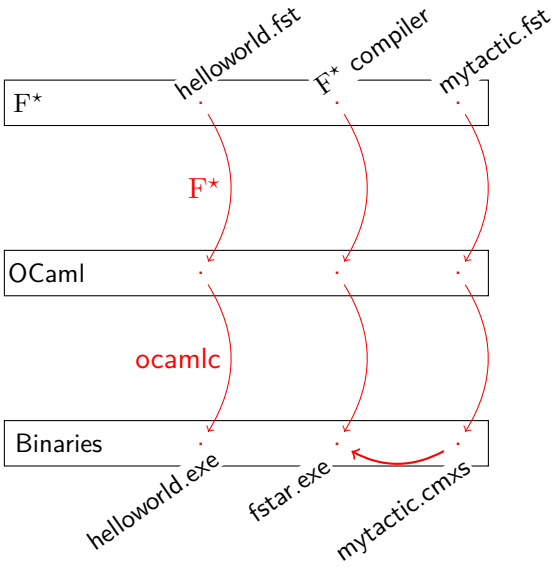
```
fstar.exe --load mytactic.cmxs
```

Compilation



- Extract the user tactic and compile it into a dynamic object
- Run
  `fstar.exe --load mytactic.cmxs`
- When loaded, the tactic registers a callback in the normalizer
- F⋆ is now extended with a new builtin: 10x speed gain!

Compilation



- Extract the user tactic and compile it into a dynamic object
- Run
  `fstar.exe --load mytactic.cmxs`
- When loaded, the tactic registers a callback in the normalizer
- F⋆ is now extended with a new builtin: 10x speed gain!
- Not metaprogram specific *at all*

Coming back to this:

```
let tcresolve () = ...
```

Fast and Safe extensions to $\mathrm{F}^\star$

Coming back to this:

```
[@plugin] let tcresolve () = ...
```

Fast and Safe extensions to $\mathrm{F}^\star$

Coming back to this:

```
[@plugin] let tcresolve () = ...
```

- Typeclass resolution, a user metaprogram, is easily added as a native $\mathrm{F}^\star$ extension.
- Significant efficiency gains

Fast and Safe extensions to $\mathrm{F}^\star$

Coming back to this:

[@plugin] let tcresolve () = ...

- Typeclass resolution, a user metaprogram, is easily added as a native $\mathrm{F}^\star$ extension.
- Significant efficiency gains
- tcresolve outside of the TCB

Summary

- Provides both automation and a “manual mode” to users
- Using an effect for metaprograms increases reuse
- Native compilation allows fast extensions
- Meta-$\mathrm{F}^\star$ opens many new possibilities in $\mathrm{F}^\star$

Summary

- Provides both automation and a “manual mode” to users
- Using an effect for metaprograms increases reuse
- Native compilation allows fast extensions
- Meta-$\mathrm{F}^\star$ opens many new possibilities in $\mathrm{F}^\star$

On the map:

- Improve user experience, REPL
- Minimize primitives
- Layering state on top of TAC

Summary

- Provides both automation and a "manual mode" to users
- Using an effect for metaprograms increases reuse
- Native compilation allows fast extensions
- Meta-$\mathrm{F}^\star$ opens many new possibilities in $\mathrm{F}^\star$

On the map:

- Improve user experience, REPL
- Minimize primitives
- Layering state on top of TAC

Thank you!

TRACE @ …Star.Tactics.Derived.fst(165,12-165,13) Fri Sep 7 10:49:00 2018

Goal 1/1

p: pos
n r h r0 r1 h0 _h1 _h2 s1 d0 d1 d2 h1 h2 hh: ℤ
p: pure_post unit
uu___: squash ($r_1 \geq 0 \wedge n > 0 \wedge 4 \times (n \times n) == p + 5 \wedge r == r_1 \times n + r_0 \wedge$
$h == h_2 \times (n \times n) + h_1 \times n + h_0 \wedge s_1 == r_1 + r_1 / 4 \wedge r_1 \,\%\, 4 == 0 \wedge d_0 == h_0 \times r_0 + h_1 \times s_1 \wedge$
$d_1 == h_0 \times r_1 + h_1 \times r_0 + h_2 \times s_1 \wedge d_2 == h_2 \times r_0 \wedge hh == d_2 \times (n \times n) + d_1 \times n + d_0 \wedge$
(∀ (pure_result: unit). $h \times r \,\%\, p == hh \,\%\, p \implies$ p pure_result))

pure_result: unit
uu___: squash $(((h_2 \times r_0) \times (n \times n) + (h_0 \times ((r_1 / 4) \times 4) + h_1 \times r_0 + h_2 \times (5 \times (r_1 / 4))) \times n +$
$(h_0 \times r_0 + h_1 \times (5 \times (r_1 / 4))) +$
$((h_2 \times n + h_1) \times (r_1 / 4)) \times p) \,\%$
$p =$
$((h_2 \times r_0) \times (n \times n) + (h_0 \times ((r_1 / 4) \times 4) + h_1 \times r_0 + h_2 \times (5 \times (r_1 / 4))) \times n +$
$(h_0 \times r_0 + h_1 \times (5 \times (r_1 / 4)))) \,\%$
$p)$

$$\begin{aligned}
&\text{squash } (4 \times (h_2 \times (n \times (n \times (n \times (r_1 / 4))))) + h_2 \times (n \times (n \times r_0)) + \\
&(4 \times (n \times (n \times (h_1 \times (r_1 / 4)))) + n \times (h_1 \times r_0)) + \\
&(4 \times (n \times (h_0 \times (r_1 / 4))) + h_0 \times r_0) == \\
&h_2 \times (n \times (n \times r_0)) + (4 \times (n \times (h_0 \times (r_1 / 4))) + n \times (h_1 \times r_0) + 5 \times (h_2 \times (n \times (r_1 / 4)))) + \\
&(h_0 \times r_0 + 5 \times (h_1 \times (r_1 / 4))) + \\
&(4 \times (h_2 \times (n \times (n \times (n \times (r_1 / 4)))))) + -5 \times (h_2 \times (n \times (r_1 / 4))) + \\
&(4 \times (n \times (n \times (h_1 \times (r_1 / 4))))) + -5 \times (h_1 \times (r_1 / 4)))))
\end{aligned}$$

(*?u3948*) _