Tactics and Metaprogramming

An introduction to Meta-F*

Guido Martínez

F* PoP Up Seminar
Oct 11th 2022

This Talk

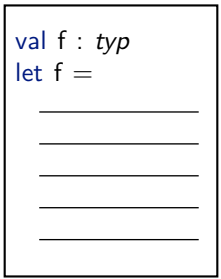
- High-level presentation + some live examples at the end.
- No way we cover everything today.
- See new chapter (in construction) in $\mathbf{F}^\star$ tutorial.

Main uses of Meta-F*

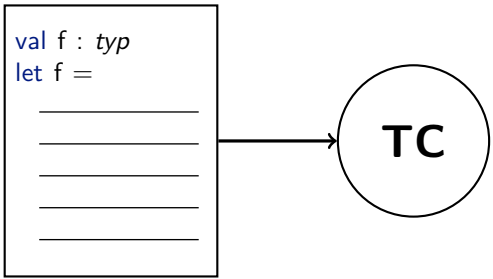
- Simplifying/proving assertions: assert φ by τ
- Metaprogramming, in various flavors.
- Solving implicit arguments (typeclasses).
- Program transformations (e.g. for efficient extraction).

Proving

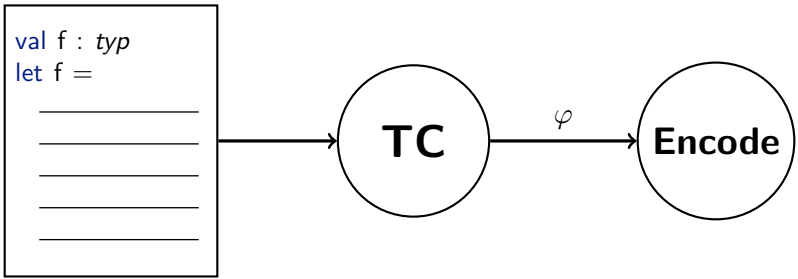
Auto-active Verification



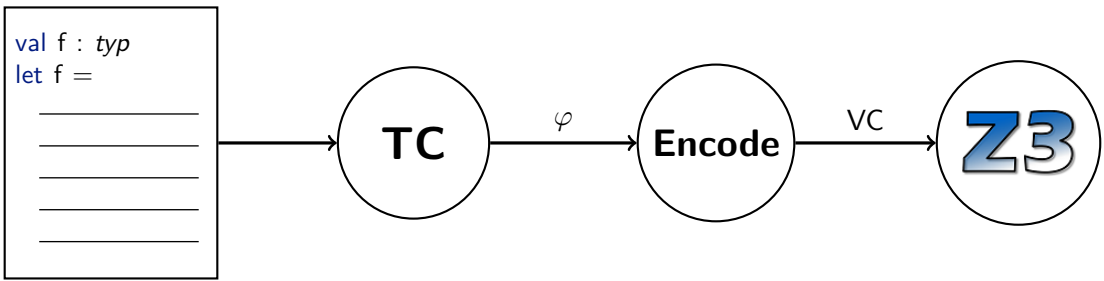
Auto-active Verification



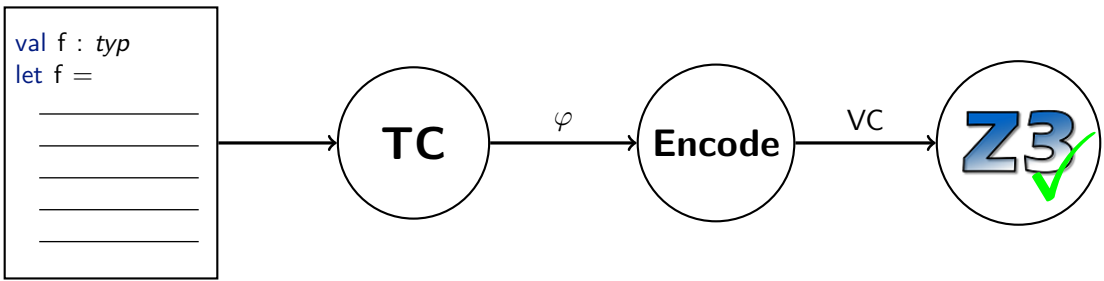
Auto-active Verification



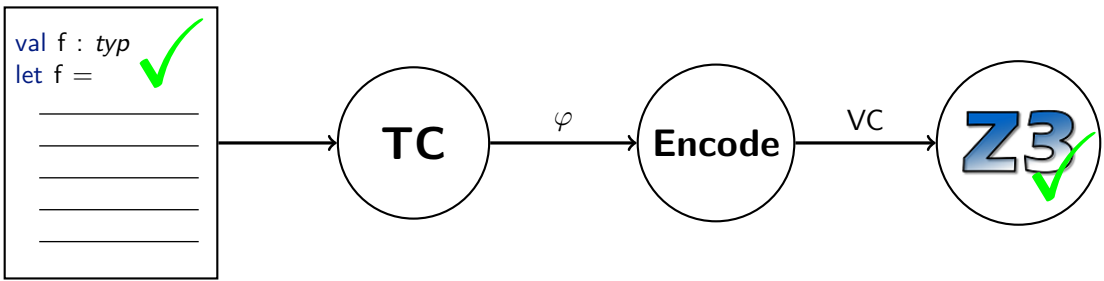
Auto-active Verification



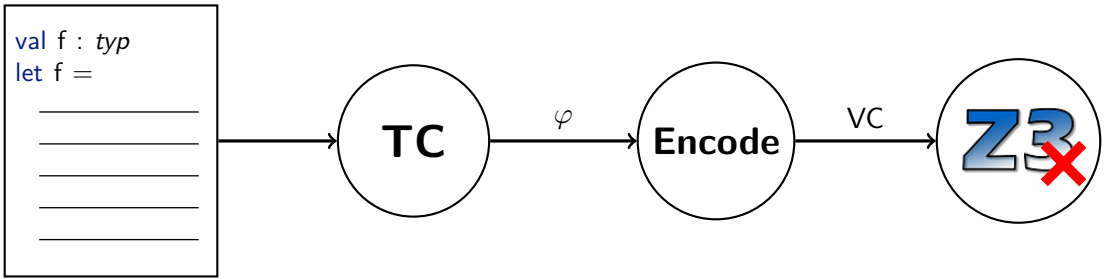
Auto-active Verification



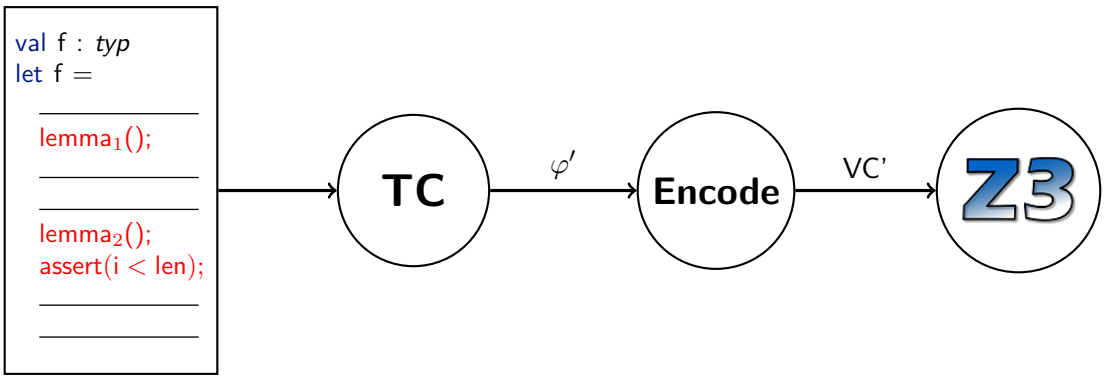
Auto-active Verification



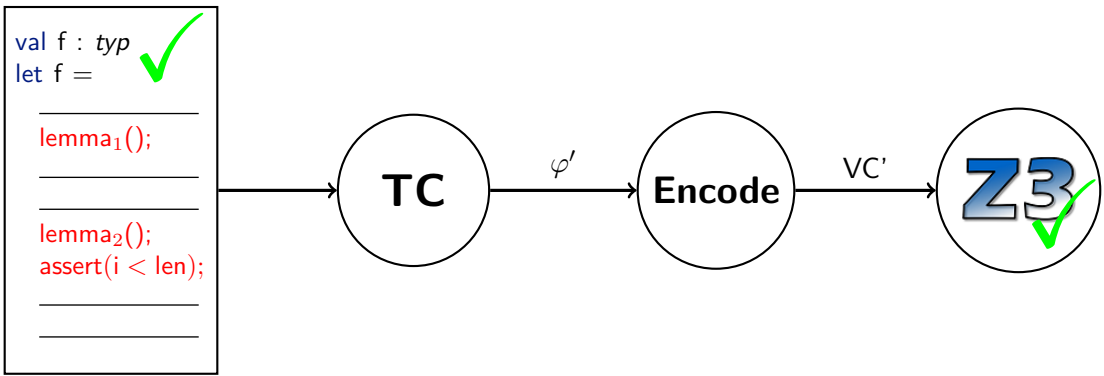
Auto-active Verification



Auto-active Verification



Auto-active Verification



Proofs by SMT

Z3 fleshes out many proofs easily. Normally, the "key insights" are all that is needed.

Proofs by SMT

Z3 fleshes out many proofs easily. Normally, the "key insights" are all that is needed. But SMT-only proofs can quickly become unwieldy...

```
val mod_add_both (a:int) (b:int) (x:int) (n:pos) : Lemma
    (requires a % n == b % n)
    (ensures (a + x) % n == (b + x) % n)
```

Proofs by SMT

Z3 fleshes out many proofs easily. Normally, the "key insights" are all that is needed. But SMT-only proofs can quickly become unwieldy...

```
val mod_add_both (a:int) (b:int) (x:int) (n:pos) : Lemma
    (requires a % n == b % n)
    (ensures (a + x) % n == (b + x) % n)

let mod_add_both (a:int) (b:int) (x:int) (n:pos) = ()
```

Proofs by SMT

Z3 fleshes out many proofs easily. Normally, the "key insights" are all that is needed. But SMT-only proofs can quickly become unwieldy...

```
val mod_add_both (a:int) (b:int) (x:int) (n:pos) : Lemma
    (requires a % n == b % n)
    (ensures (a + x) % n == (b + x) % n)

let mod_add_both (a:int) (b:int) (x:int) (n:pos) =
  lemma_div_mod a n;
  lemma_div_mod b n;
  lemma_div_mod (a + x) n;
  lemma_div_mod (b + x) n;
  let xx = (b + x) / n - (a + x) / n - b / n + a / n in
  distributivity_sub_left ((b + x) / n) ((a + x) / n) n;
  distributivity_sub_left ((b + x) / n - (a + x) / n) (b / n) n;
  distributivity_add_left ((b + x) / n - (a + x) / n - b / n) (a / n) n;
  lt_multiple_is_equal ((a + x) % n) ((b + x) % n) xx n
```

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 [illegible] (r1 / 4) ∧ r1 % [illegible] == 0 ∧
              d0 == h0 * r0 + h1 * s1 ∧ d1 == h0 * r1 + [illegible] + h2 * s[illegible]
              d2 == h2 * r0 ∧ hh == d2 * (n * n) + d1 * n [illegible]
    (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] 0)
  let hh_expand = (h2 * r0) * (n * n) + (h0 * (r1_4 * 4) + [illegible] + h2 * ([illegible] 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

Meta-F*: Tactics and Metaprogramming

Automate generation of proofs

Meta-F*: Tactics and Metaprogramming

Automate generation of proofs?

Meta-F*: Tactics and Metaprogramming

Automate generation of proofs?
Rather, pre-processing the VC before SMT.

Meta-F*: Tactics and Metaprogramming

Automate generation of proofs?
Rather, pre-processing the VC before SMT.

- Arithmetic canonicalization (nlarith)
- Simplification via rewrites, normalization
- Case splitting
- Proof environment manipulation (e.g. fact pruning)

Meta-F*: Tactics and Metaprogramming

Automate generation of proofs?
Rather, pre-processing the VC before SMT.

- Arithmetic canonicalization (nlarith)
- Simplification via rewrites, normalization
- Case splitting
- Proof environment manipulation (e.g. fact pruning)
- Fully automating proofs also possible

Tackling individual assertions

```
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)
```

Splitting assertions

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

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

Splitting assertions

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

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

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

L = R

- Can switch between part of a VC (formula) and a goal (type-theoretic hole)

Splitting assertions

VC will not contain the obligation,
instead we get a *goa*

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

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

L = R

- Can switch between part of a VC (formula) and a goal (type-theoretic hole)

Splitting assertions

VC will not contain the obligation,
instead we get a *goa*

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

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

---

nf(L) = *nf*(R)

- Can switch between part of a VC (formula) and a goal (type-theoretic hole)

Splitting assertions

VC will not contain the obligation,
instead we get a *goa*

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

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

---

nf(L) = *nf*(R)

- Can switch between part of a VC (formula) and a goal (type-theoretic hole)

What are metaprograms?

- New $\mathbf{F}^\star$ effect: TAC.

What are metaprograms?

- New $\mathbf{F}^\star$ effect: TAC.
  - Representation: proofstate $\rightarrow$ either error (a $*$ proofstate)
  - Completely standard and user-defined...
  - ... except for the assumed primitives

What are metaprograms?

- New **F*** effect: TAC.
  - Representation: proofstate $\rightarrow$ 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 *)
let tac a = proofstate → Dv ((a & proofstate) 'either' error) (* Dv: possibly diverging *)
let t_return (x:α) = ...
let t_bind (m:tac α) (f:α → tac β) : tac β= ...

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

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

What are metaprograms?

- New **F*** 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 *)
let tac a = proofstate → Dv ((a & proofstate) 'either' error) (* Dv: possibly diverging *)
let t_return (x:α) = ...
let t_bind (m:tac α) (f:α → tac β) : tac β= ...

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 F*

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 F*

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 F*

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 F*

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;
    ...
```

Metaprogramming

Metaprogramming: generating terms

Beyond proving, Meta-**F*** 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-**F*** 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-**F*** 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*** 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
```

Customizing implicit arguments

Meta-$\mathbf{F}^\star$ 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*** 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-$\mathbf{F}^\star$ 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 top-levels *)
val tcresolve : unit → Tac unit (* resolves dictionaries *)
```

Putting it together: Typeclasses

```
module FStar.Typeclasses
val mk_class : name → Tac unit (* creates method top-levels *)
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 top-levels *)
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 top-levels *)
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 top-levels *)
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 top-levels *)
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 top-levels *)
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 top-levels *)
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 }
```

Canonicalization

- Say we a complex goal involving list concatenation.
  (a@a)@(b@(c@d)) = ((a@(a@([]@b)))@c)@d

Canonicalization

- Say we a complex goal involving list concatenation.
  (a@a)@(b@(c@d)) = ((a@(a@([]@b)))@c)@d
- We can already apply lemmas repeatedly to solve it, but that's not much better.

```
val append_assoc (l1 l2 l3 : list α) : Lemma ((l1@l2)@l3 == l1@(l2@l2))
val append_nil_l (l1 : list α) : Lemma ([]@l1 == l1)
...
```

Canonicalization

- Say we a complex goal involving list concatenation.
  (a@a)@(b@(c@d)) = ((a@(a@([]@b)))@c)@d
- We can already apply lemmas repeatedly to solve it, but that's not much better.

```
val append_assoc (l1 l2 l3 : list α) : Lemma ((l1@l2)@l3 == l1@(l2@l2))
val append_nil_l (l1 : list α) : Lemma ([]@l1 == l1)
...
```

- First idea: write automation to inspect the goal and apply the lemma as needed.

Canonicalization

- Say we a complex goal involving list concatenation.
  (a@a)@(b@(c@d)) = ((a@(a@([]@b)))@c)@d
- We can already apply lemmas repeatedly to solve it, but that's not much better.

```
val append_assoc (l1 l2 l3 : list α) : Lemma ((l1@l2)@l3 == l1@(l2@l2))
val append_nil_l (l1 : list α) : Lemma ([]@l1 == l1)
...
```

- First idea: write automation to inspect the goal and apply the lemma as needed.
- This works, but is very slow due to the constant use of the typechecker/unifier.

Proof-by-reflection: 1 minute pitch

- Obtain an AST of the expression.

```
type atom = list α
type expr =
  | App : expr → expr → expr
  | Nil : expr
  | Val : atom → expr
```

Proof-by-reflection: 1 minute pitch

- Obtain an AST of the expression.

```
type atom = list α
type expr =
  | App : expr → expr → expr
  | Nil : expr
  | Val : atom → expr
```

- Denotation:

```
val denote : expr → list α
```

Proof-by-reflection: 1 minute pitch

- Obtain an AST of the expression.

```
type atom = list α
type expr =
  | App : expr → expr → expr
  | Nil : expr
  | Val : atom → expr
```

- Denotation:

```
val denote : expr → list α
```

- Write a *pure* canonicalizer of exprs:

```
val canon : expr → list atom
val canon_correct (e:expr)
 : Lemma (denote e == denote (canon e))
```

Proof-by-reflection: 1 minute pitch

- Obtain an AST of the expression.

```
type atom = list α
type expr =
  | App : expr → expr → expr
  | Nil : expr
  | Val : atom → expr
```

- Denotation:

```
val denote : expr → list α
```

- Write a *pure* canonicalizer of exprs:

```
val canon : expr → list atom
val canon_correct (e:expr)
 : Lemma (denote e == denote (canon e))
```

a@(b@c) = (a@b)@c

Proof-by-reflection: 1 minute pitch

- Obtain an AST of the expression.

```
type atom = list α
type expr =
  | App : expr → expr → expr
  | Nil : expr
  | Val : atom → expr
```

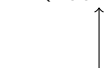
- Denotation:

```
val denote : expr → list α
```

- Write a *pure* canonicalizer of exprs:

```
val canon : expr → list atom
val canon_correct (e:expr)
 : Lemma (denote e == denote (canon e))
```

```
denote (App a' (App b' c'))
= denote (App (App a' b') c')
```



```
a@(b@c) = (a@b)@c
```

Proof-by-reflection: 1 minute pitch

- Obtain an AST of the expression.

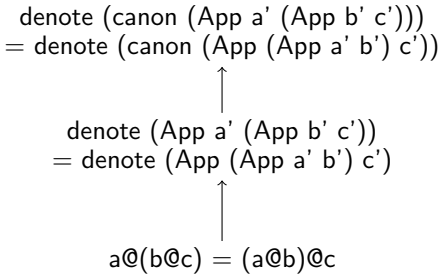
```
type atom = list α
type expr =
  | App : expr → expr → expr
  | Nil : expr
  | Val : atom → expr
```

- Denotation:

```
val denote : expr → list α
```

- Write a *pure* canonicalizer of exprs:

```
val canon : expr → list atom
val canon_correct (e:expr)
 : Lemma (denote e == denote (canon e))
```



Proof-by-reflection: 1 minute pitch

- Obtain an AST of the expression.

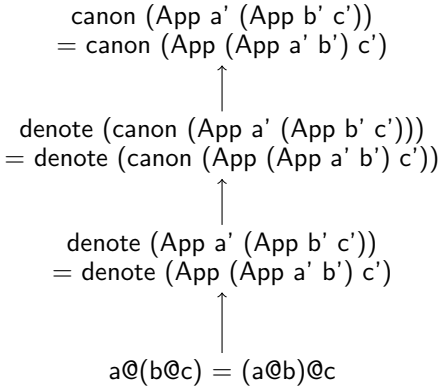
```
type atom = list α
type expr =
  | App : expr → expr → expr
  | Nil : expr
  | Val : atom → expr
```

- Denotation:

```
val denote : expr → list α
```

- Write a *pure* canonicalizer of exprs:

```
val canon : expr → list atom
val canon_correct (e:expr)
 : Lemma (denote e == denote (canon e))
```



Proof-by-reflection: 1 minute pitch

- Obtain an AST of the expression.

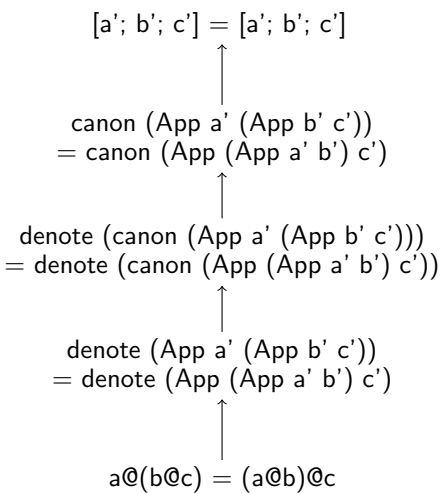
```
type atom = list α
type expr =
  | App : expr → expr → expr
  | Nil : expr
  | Val : atom → expr
```

- Denotation:

```
val denote : expr → list α
```

- Write a *pure* canonicalizer of exprs:

```
val canon : expr → list atom
val canon_correct (e:expr)
 : Lemma (denote e == denote (canon e))
```



Summary

- That was a super quick view of Meta-**F***
- Stay tuned for chapter in **F*** tutorial, this week.
- Also check out `examples/` and `tests/`

Other topics:

- Syntax inspection/handling: see tutorial, and also demo.
- Plugins: fast metaprograms via compilation.
- Steel: a heavy-duty use case for separation logic.

Summary

- That was a super quick view of Meta-**F***
- Stay tuned for chapter in **F*** tutorial, this week.
- Also check out `examples/` and `tests/`

Other topics:

- Syntax inspection/handling: see tutorial, and also demo.
- Plugins: fast metaprograms via compilation.
- Steel: a heavy-duty use case for separation logic.

Questions..?

Summary

- That was a super quick view of Meta-**F***
- Stay tuned for chapter in **F*** tutorial, this week.
- Also check out `examples/` and `tests/`

Other topics:

- Syntax inspection/handling: see tutorial, and also demo.
- Plugins: fast metaprograms via compilation.
- Steel: a heavy-duty use case for separation logic.

Questions..?

Demo!