An introduction to Meta-F*
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?
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 → T)) (ensures (λ h () h' → T)) =
   let r = alloc 1 in
   incr r;
   let v = !r in
   assert (v == 2)
∀ (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))))))

(* our assertion *)
∀ (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))))))
∀ (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 $\varphi = \text{Pure unit (requires } \top \text{) (ensures } (\lambda () \to \varphi )))$

```haskell
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 $\varphi = \text{Pure unit (requires } T\text{) (ensures } (\lambda () \rightarrow \varphi))$

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 $\varphi = \text{Pure unit (requires} \top \text{) (ensures } \lambda () \rightarrow \varphi ))$

let lemma_carry_limb_unrolled (a0 a1 a2 : nat) : Lemma $(a0 \mod p44 + p44 \times ((a1 + a0 \div p44) \mod p44) + p88 \times (a2 + ((a1 + a0 \div p44) \div p44)) == a0 + p44 \times a1 + p88 \times a2)$

=  
  $\rightarrow$ pow2_plus 44 44;
  $\rightarrow$ lemma_div_mod (a1 + a0 \div p44) p44;
  $\rightarrow$ lemma_div_mod a0 p44:
  distributivity_add_right p88 a2 ((a1 + a0 \div p44) \div p44);
  distributivity_add_right p44 ((a1 + a0 \mod p44) \mod p44) (p44 \times ((a1 + a0 \div p44) \div p44));
  distributivity_add_right p44 a1 (a0 \div p44);
When SMT doesn't cut it

Note: **Lemma** $\varphi = \text{Pure unit (requires } \top \text{) (ensures } (\lambda () \rightarrow \varphi ))$

```plaintext
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 $\varphi = \text{Pure unit}$

```
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 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

```ocaml
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 h0 r1 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)
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
Meet Meta-\texttt{F*}

A tactics and metaprogramming language for \texttt{F*}

- Embedded into \texttt{F*} as an \textit{effect}: \texttt{Tac}
  - Metaprograms are terms with \texttt{Tac} effect
  - Exceptions, divergence and \textbf{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*

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

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 written and typechecked as any other kind of effectful term:

```ocaml
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 written and typechecked as any other kind of effectful term:

```ocaml
let mytac () : Tac unit =
    let h1 : binder = implies_intro () in
    rewrite h1;
    reflexivity ()

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

Goal 1/1

<table>
<thead>
<tr>
<th>a b : int</th>
<th>h0 : a &gt; 0</th>
</tr>
</thead>
</table>

\[
a = b \implies f b == f a
\]
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 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 \rightarrow f b == f a)
  by (mytac ())
```

Goal 1/1

\[
\begin{align*}
& a \ b : \text{int} \quad h0 : a > 0 \\
& h1 : a = b \\
\hline
& f b == f b
\end{align*}
\]
Metaprograms are written and typechecked as any other kind of effectful term:

```ocaml
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*

```ml
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*  

```coq
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)
```

```coq
  =
  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

\[ \forall n \ p \ r \ldots, \]
\[ \varphi_1 \implies \psi_1 \land \]
\[ \varphi_2 \implies \psi_2 \land \]
\[ \ldots \implies L = R \land \]
\[ L = R \implies \ldots \]
With `assert..by`, the VC will not contain the obligation, instead we get a goal

\[ \forall n \ p \ r \ \ldots, \]
\[ \varphi_1 \Rightarrow \psi_1 \land \]
\[ \varphi_2 \Rightarrow \psi_2 \land \]
\[ \ldots \Rightarrow L = R \land \]
\[ L = R \Rightarrow \ldots \]
Splitting assertions

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

\[
\forall n \ p \ r \ldots,
\varphi_1 \implies \psi_1 \land \\
\varphi_2 \implies \psi_2 \land \\
\ldots \\
L = R \land \\
H0 : \varphi_1 \\
H1 : \varphi_2 \\
\ldots
\]

\[
L = R
\]

Goal 1/1

\[
n : \text{int} \\
p : \text{int} \\
r : \text{int} \\
\vdots
\]

\[
H0 : \varphi_1 \\
H1 : \varphi_2 \\
\vdots
\]
Splitting assertions

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

\[
\forall n \ p \ r \ldots, \\
\varphi_1 \implies \psi_1 \land \\
\varphi_2 \implies \psi_2 \land \\
\ldots \implies L = R \land \\
L = R \implies \ldots \\
\]

Goal 1/1

n : int
p : int
r : int
\ldots
H0 : \varphi_1
H1 : \varphi_2
\ldots

\[
nf(L) = nf(R)
\]
Splitting assertions

With \textit{assert..by}, the VC will not contain the obligation, instead we get a goal:

\[ \forall n \ p \ r \ldots, \]
\[ \varphi_1 \implies \psi_1 \land \]
\[ \varphi_2 \implies \psi_2 \land \]
\[ \ldots \implies L = R \land \]
\[ L = R \implies \ldots \]

Goal 1/1

\begin{align*}
\text{nf}(L) &= \text{nf}(R) \\
\text{n : int} &
\text{p : int} \\
\text{r : int} &
\ldots \\
\text{H0 : } \varphi_1 &
\text{H1 : } \varphi_2 \\
\ldots &
\end{align*}
Splitting assertions

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

\[ \forall n \ p \ r \ldots, \]
\[ \varphi_1 \implies \psi_1 \land \]
\[ \varphi_2 \implies \psi_2 \land \]
\[ \ldots \implies L = R \land \]
\[ L = R \implies \ldots \]

Goal 1/1

\[ n : \text{int} \]
\[ p : \text{int} \]
\[ r : \text{int} \]
\[ \ldots \]
\[ H0 : \varphi_1 \]
\[ H1 : \varphi_2 \]
\[ \ldots \]

\[ nf(L) = nf(R) \]
Metaprogramming
Beyond proving, Meta-F* enables constructing terms

```ml
let f (x y : int) : int = _ by (exact (’42))
```
Metaprogramming: generating terms

Beyond proving, Meta-F* enables constructing terms

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

(* running exact ('42) *)

Goal 1/1
x : int
y : int

?u : int
Beyond proving, Meta-F* enables constructing terms

\[
\text{let } f (x \ y : \text{int}) : \text{int} = 42 \quad \text{No more goals}
\]
Beyond proving, Meta-F* enables constructing terms

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

No more goals

- Metaprogramming goals are **relevant** (can’t call `smt ()`).
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 ())
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
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
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
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
let mk_add () : Tac unit =
  let x = intro () in
  let y = intro () in
  apply ('(+));
  exact (quote y);
  exact (quote x)

let add : int \rightarrow int \rightarrow int =
\lambda x \rightarrow \lambda y \rightarrow y + ?u4

Goal 1/2
x : int
y : int

\hline
?u4 : int
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
type t1 =
  | A : int → int → t1
  | B : string → t1
  | C : t1 → t1

Similar to Haskell’s deriving and OCaml’s ppx_deriving, but completely in “user space.”
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* 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* can also be used to provide strategies for resolution of implicits.

```ocaml
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* can also be used to provide strategies for resolution of implicit arguments.

```ocaml
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)
```

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

Dictionary resolution, `tcresolve`, is a 20 line metaprogram.
Customizing implicit arguments

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

```ocaml
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* can also be used to provide strategies for resolution of implicits.

```ocaml
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.

```ocaml
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.

```ocaml
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

```haskell
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

```ocaml
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] (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])
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])
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

```ocaml
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-F* enables to extend F* in F* safely
  - Customize how terms are verified, typechecked, elaborated...
  - Native compilation allows fast extensions

Start with Intro.fst!
What are metaprograms?

- Use F∗’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 → 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

\[
\text{type error} = \text{exn} \ast \text{proofstate} \quad (* \text{error and proofstate at the time of failure} *)
\]

\[
\text{type result } a = | \text{Success} : a \rightarrow \text{proofstate} \rightarrow \text{result} a | \text{Failed} : \text{error} \rightarrow \text{result} a
\]

\[
\text{let} \quad \text{tac } a = \text{proofstate} \rightarrow Dv \ (\text{result} a) \quad (* Dv: \text{possibly diverging} *)
\]

\[
\text{let} \quad \text{t\_return } (x:\alpha) = \lambda ps \rightarrow \text{Success} \times ps
\]

\[
\text{let} \quad \text{t\_bind } (m:\text{tac } \alpha) (f:\alpha \rightarrow \text{tac } \beta) : \text{tac } \beta =
\quad \lambda ps \rightarrow \text{match } m \ ps \ \text{with} \ | \ \text{Success} \times ps' \rightarrow f \times ps' | \ \text{Error} \ e \rightarrow \text{Error} \ e
\]

\[
\text{new\_effect } \{ \ \text{TAC with} \ \text{repr} = \text{tac} ; \ \text{return} = \text{t\_return} ; \ \text{bind} = \text{t\_bind} \ \}
\]

\[
\text{sub\_effect } \text{DIV} \rightsquigarrow \text{TAC} = ...
\]

\[
\text{sub\_effect } \text{EXN} \rightsquigarrow \text{TAC} = ...
\]
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

```plaintext
type error = exn * proofstate (* error and proofstate at the time of failure *)
type result α = | Success : α → proofstate → result α | Failed : error → result α
let tac a = proofstate → Dv (result α) (* 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, ...

18 / 17
Goal 1/1

\[ n \ p \ r \ h \ r_0 \ r_1 \ h_0 \ h_1 \ h_2 \ s_1 \ d_0 \ d_1 \ d_2 \ h_\emptyset: \mathbb{Z} \]

**p:** pure_post unit

\[ uu___: p > 0 \land r_1 \geq 0 \land n > 0 \land 4 \times (n \times n) = p + 5 \land r = r_1 \times n + r_0 \land h = h_2 \times (n \times n) + h_1 \times n + h_\emptyset \land s_1 = r_1 + r_1 / 4 \land r_1 \% 4 = 0 \land d_0 = h_\emptyset \times r_\emptyset + h_1 \times s_1 \land d_1 = h_\emptyset \times r_1 + h_1 \times r_\emptyset + h_2 \times s_1 \land d_2 = h_2 \times r_\emptyset \land hh = d_2 \times (n \times n) + d_1 \times n + d_0 \land (\forall (pure\_result: \text{unit}). \ h \times r \% p = hh \% p \implies p \ pure\_result) \]

**return_val:** \( \mathbb{Z} \)

\[ uu___: \text{return_val} = p \]

**pure_result:** unit

\[ uu___: ((h_2 \times r_\emptyset) \times (n \times n) + (h_\emptyset \times ((r_1 / 4) \times 4) + h_1 \times r_\emptyset + h_2 \times (5 \times (r_1 / 4))) \times n +

(h_\emptyset \times r_\emptyset + 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_\emptyset) \times (n \times n) + (h_\emptyset \times ((r_1 / 4) \times 4) + h_1 \times r_\emptyset + h_2 \times (5 \times (r_1 / 4))) \times n +

(h_\emptyset \times r_\emptyset + h_1 \times (5 \times (r_1 / 4))) \% p \]

\[ p \]

squash \( (4 \times (h_2 \times (n \times (n \times (n \times (r_1 / 4)))))) + h_2 \times (n \times (n \times r_\emptyset)) +

(4 \times (n \times (n \times (h_\emptyset \times (r_1 / 4)))) + n \times (h_1 \times r_\emptyset)) +

(4 \times (n \times (h_\emptyset \times (r_1 / 4)))) + h_\emptyset \times r_\emptyset) =

h_2 \times (n \times (n \times r_\emptyset)) + (4 \times (n \times (h_\emptyset \times (r_1 / 4)))) + n \times (h_1 \times r_\emptyset) + 5 \times (h_2 \times (n \times (r_1 / 4)))) +

(h_\emptyset \times r_\emptyset + 5 \times (h_1 \times (r_1 / 4))) +

(4 \times (h_2 \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))))

(*?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 Ñ 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 Ñ trywith seen fuel (pack (Tv_FVar fv))) cands
  and trywith (seen:list term) (fuel:int) (t:term) : Tac unit =
    (λ () Ñ apply t) ‘seq’ (λ () Ñ tcresolve’ seen (fuel - 1))

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