Each year Microsoft Research hosts hundreds of influential speakers from around the world including leading scientists, renowned experts in technology, book authors, and leading academics, and makes videos of these lectures freely available.

2016 © Microsoft Corporation. All rights reserved.
Extending F* in F*

Proof automation and Metaprogramming for Typeclasses, Concurrency, Optimizations and More

Guido Martínez
advised by Nikhil Swamy
RiSE / Everest

MSR Redmond, September 2018
What's this \( F^* \) 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)
What's this F\(^{\ast}\) 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)

```%
let incr (r:ref int) = r := !r + 1
let f () =
  let r = alloc 1 in
  incr r;
  let v = !r in
  assert (v == 2) (* succeeds due to the logical context *)
```
When automation fails

What does the user do when the solver fails?
When automation fails

What does the user do when then solver fails?

val lemma_mod_spec: a:int → p:pos → Lemma (a / p = (a - (a % p)) / p)
What does the user do when then solver fails?

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

What does the user do when then solver fails?

```haskell
val lemma_mod_spec : a:int \rightarrow p:pos \rightarrow Lemma (a / p = (a - (a \ mod\ p)) / p)
let lemma_mod_spec a p =
  lemma_div_mod a p;
  assert (a = p \times (a / p) + a \ mod\ p);
  assert (a \ mod\ p = a - p \times (a / p));
  assert (a - (a \ mod\ p) = p \times (a / p));
  lemma_mod_plus 0 (a / p) p;
  assert ((p \times (a / p)) \ mod\ p = 0);
  lemma_div_exact (p \times (a / p)) p
```

Some proofs are just out of the solver's scope
When automation fails

What does the user do when then solver fails?

```plaintext
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
```

Some proofs are just out of the solver's scope
Motivated a tactics engine, a way to tweak proof obligations before sending to SMT.
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

$$\Gamma_0 \vdash ?_0 : t_0$$
Metaprograms, in general, perform correct transformations of the proofstate. Basically a set of "goals" to be fulfilled:

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

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

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

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

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

- The proofstate is transformed by certain primitives, which are faithful to F*'s typing judgment.
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

\[ \text{squash} \left( r_1 \geq 0 \land n > 0 \land 4 \times (n \times n) = p + 5 \land r = r_1 \times n + r_0 \land \\
 h_2 = \frac{r_2 \times (n \times n) + h_1 \times n + h_0 \land s_1}{r_1 + r_1 / 4 \land r_1 \% 4 = 0 \land d_0 = h_0 \times r_0 + h_1 \times s_1 \land \\
 d_1 = h_0 \times r_1 + h_1 \times r_0 + h_2 \times s_1 \land d_2 = h_2 \times r_0 \land hh = d_2 \times (n \times n) + d_1 \times n + d_0 \land \\
 (\forall \text{pure_result}: \text{unit}). \ h \times r \% p = hh \% p \Rightarrow p \text{ pure_result}) \]

pure_result: unit

\[ \text{squash} \left( \left( h_2 \times r_0 \right) \times (n \times n) + \left( h_0 \times \left( \left( r_1 / 4 \right) \times 4 \right) + h_1 \times r_0 + h_2 \times \left( 5 \times \left( r_1 / 4 \right) \right) \right) \times n + \\
 \left( \left( h_2 \times n + h_1 \right) \times \left( r_1 / 4 \right) \right) \times p \right) \%
\]

p =

\[ \left( \left( h_2 \times r_0 \right) \times (n \times n) + \left( h_0 \times \left( \left( r_1 / 4 \right) \times 4 \right) + h_1 \times r_0 + h_2 \times \left( 5 \times \left( r_1 / 4 \right) \right) \right) \times n + \\
 \left( h_0 \times r_0 + h_1 \times \left( 5 \times \left( r_1 / 4 \right) \right) \right) \right) \%
\]

h

\[ \text{squash} \left( 4 \times \left( h_2 \times \left( n \times \left( n \times \left( n \times \left( r_1 / 4 \right) \right) \right) \right) \right) \right) + h_2 \times \left( n \times \left( n \times r_0 \right) \right) + \\
 \left( 4 \times \left( n \times \left( n \times \left( h_1 \times \left( r_1 / 4 \right) \right) \right) \right) \right) + n \times \left( h_1 \times r_0 \right) + \\
 \left( 4 \times \left( n \times \left( h_0 \times \left( r_1 / 4 \right) \right) \right) \right) + h_0 \times r_0 \right) \right) = \\
 h_2 \times \left( n \times \left( n \times r_0 \right) \right) + \left( 4 \times \left( n \times \left( h_0 \times \left( r_1 / 4 \right) \right) \right) \right) + n \times \left( h_1 \times r_0 \right) + 5 \times \left( h_2 \times \left( n \times \left( r_1 / 4 \right) \right) \right) + \\
 \left( h_0 \times r_0 + 5 \times \left( h_1 \times \left( r_1 / 4 \right) \right) \right) + \\
 \left( 4 \times \left( h_2 \times \left( n \times \left( n \times \left( r_1 / 4 \right) \right) \right) \right) \right) + 5 \times \left( h_2 \times \left( n \times \left( r_1 / 4 \right) \right) \right) + \\
 \left( 4 \times \left( n \times \left( n \times \left( h_1 \times \left( r_1 / 4 \right) \right) \right) \right) \right) + 5 \times \left( h_2 \times \left( n \times \left( r_1 / 4 \right) \right) \right) + \\
 \left( 4 \times \left( n \times \left( n \times \left( h_1 \times \left( r_1 / 4 \right) \right) \right) \right) \right) + 5 \times \left( h_2 \times \left( n \times \left( r_1 / 4 \right) \right) \right) \]
What are metaprograms?

- Use DM4F machinery to make new custom effect: TAC
  - Representation: proofstate → either error (a * proofstate)
  - Completely standard and user-defined...
  - ... except for the assumed primitives
What are metaprograms?

- Use DM4F machinery to make new custom effect: TAC
  - Representation: proofstate → either error (a * proofstate)
  - Completely standard and user-defined...
  - ... except for the assumed primitives

```haskell
type error = string * proofstate (* error message 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 }
```
TAC is a first-class citizen of F*

Metaprograms are written as any other kind of program:

```fsharp
let mytac () : Tac unit =
    let h1 = implies_intro () in
    rewrite h1;
    apply_lemma ('mylem);
...
```
Higher-order combinators too:

```ocaml
define repeat (t : unit -> Tac α) : Tac (list α) =
  match catch t with
  | Inl _ -> []
  | Inr x -> x :: repeat t

define repeat1 (t : unit -> Tac α) : Tac (list α) =
  t () :: repeat t
```

TAC is a first-class citizen of F*
TAC is a first-class citizen of F*.

Inspect goals and types of terms:

\[
\begin{align*}
\text{val mapply : } \text{term } &\rightarrow \text{Tac unit} \\
\text{let rec mapply } t &= \\
&\begin{cases}
\text{let } g = \text{cur_goal } () \text{ in} \\
\text{let } ty = \text{tc } t \text{ in} \\
\text{let } (\_, c) = \text{collect_arr } ty \text{ in} \\
\text{match inspect_comp } c \text{ with} \\
&| \text{C_Lemma } \text{pre } \text{post } \rightarrow \\
&\quad \text{match trytac } (\lambda () \rightarrow \text{apply_lemma } t) \text{ with} \\
&\quad | \text{Some } \_ \rightarrow () \\
&\quad | \text{None } \rightarrow \text{let } \text{post } = \text{norm_term } [] \text{ post } \in \\
&\quad \quad \text{match term_as_formula' } \text{post } \text{with} \text{ (* Is it an implication? try to intro *)} \\
&\quad \quad | \text{Implies } p \ q \rightarrow \text{apply_lemma } (\text{`push1}); \text{apply_squash_or_lem } t \\
&\quad \quad | \_ \rightarrow ... \\
&| \text{C_Total } rt \_ \rightarrow .... \\
\end{cases}
\end{align*}
\]
let \( f \ x \ y \ z = \)

...  

assert (g x y == h z);  

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

let lemma_poly_multiply (p:pos) (n r h r0 r1 h0 h1 h2 s1 d0 d1 d2 h1 h2 hh : int) : Lemma
(requires r1 \geq 0 \land n > 0 \land 4 \ast (n \ast n) == p + 5 \land r == r1 \ast n + r0 \land
h == h2 \ast (n \ast n) + h1 \ast n + h0 \land s1 == r1 + (r1 / 4) \land r1 \% 4 == 0 \land
d0 == h0 \ast r0 + h1 \ast s1 \land d1 == h0 \ast r1 + h1 \ast r0 + h2 \ast s1 \land
d2 == h2 \ast r0 \land hh == d2 \ast (n \ast n) + d1 \ast n + d0)
(ensures (h \ast r) \% p == hh \% p) =
let r1_4 = r1 / 4 in
let h_r_expand = (h2 \ast (n \ast n) + h1 \ast n + h0) \ast ((r1_4 \ast 4) \ast n + r0) in
let hh_expand = (h2 \ast r0) \ast (n \ast n) + (h0 \ast (r1_4 \ast 4) + h1 \ast r0 + h2 \ast (5 \ast r1_4)) \ast n
+ (h0 \ast r0 + h1 \ast (5 \ast r1_4)) in
let b = ((h2 \ast n + h1) \ast r1_4) in
modulo_addition_lemma hh_expand p b;
assert (h_r_expand == hh_expand + b \ast (n \ast n \ast 4 + (-5)))
let lemma_poly_multiply (p:pos) (n r h r0 r1 h0 h1 h2 s1 d0 d1 d2 h1 h2 hh : int) : Lemma
(requires 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)
Beyond proving, Meta-F* enables the automatic construction of both terms and top-level definitions.

```fsharp
let one = _ by (exact ('1))
let one_plus_two = _ by (apply ('(+)'); exact ('1'); exact ('2'))
```
let lemma_poly_multiply (p:pos) (n r h r0 r1 h0 h1 h2 s1 d0 d1 d2 h1 h2 hh : int) : Lemma
(requires 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)))
let lemma_poly_multiply (p:pos) (n r h r0 r1 h0 h1 h2 s1 d0 d1 d2 h1 h2 hh : int) : Lemma (requires 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)
Beyond proving, Meta-F* enables the automatic construction of both terms and top-level definitions.

```ml
let one = _ by (exact '1)
let one_plus_two = _ by (apply '(+); exact '1; exact '2)
```
Beyond proving, Meta-F* enables the automatic construction of both terms and top-level definitions.

```ocaml
type test = | TA | TB | TC | TD

val p : parser_spec test
let p = synth (\() \rightarrow gen_enum_parser T.SMT ('test))

val q : parser_impl p
let q = synth (\() \rightarrow gen_parser_impl T.Goal)
```
Metaprogramming: generating terms

Beyond proving, Meta-F* enables the automatic construction of both terms and top-level definitions.

```ml
noeq

type t1 =
| A : int → string → t1
| B : t1 → int → t1
| C : t1
| D : string → t1
| E : t1 → t1
| F : (unit → t1) → t1

%splice (mk_printer ('t1))

(* val t1_print : t1 → string *)
```
Typing in F*

- Terms created by Meta-F* are well-typed, by the correctness of each primitive.
- Essentially, Meta-F* puts the entirety of F*'s typing judgment at hand, while F*'s typechecker is just one particular heuristic.
- This is useful since F*'s typing is in general undecidable: consider \( \cdot \vdash () : \text{squash } \varphi \); valid iff \( \varphi \) is a valid proposition.
- This aspect leveraged by the parser/generator tactics to verify them efficiently.
Customizing implicit arguments

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

```ocaml
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.

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

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

let _ = assert (diag 42 #21 == (42, 21))
Customizing implicit arguments

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

```fsharp
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))

let easy #a (#[easy_tactic ()] x : a) : a = x
```
Customizing implicit arguments

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

```fsharp
define diag (x:int) (y : int) : int * int = (x, y)
define assert (x : int) : unit = assert (diag x x)
define assert (x : int) : unit = assert (diag x (x + 1))
define assert (x : int) : unit = assert (diag x (x * 2))
define easy (a : Type) : a = a

define type monoid a zero op = {
    define idL : (x : a) -> Lemma (zero 'op' x == x);
    define idR : (x : a) -> Lemma (x 'op' zero == x);
    define assoc : (x y z : a) -> Lemma ((a 'op' b) 'op' c == a 'op' (b 'op' c));
}
define m : monoid int 0 (+) = {
    define idL = (\_ -> ());
    define idR = (\_ -> ());
    define assoc = (\_ \_ \_ -> ());
}
```
Customizing implicit arguments

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

```fsharp
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))

let easy #a (#[easy_tactic()] x : a) : a = x

noeq type monoid a zero op = {
    idL : (x : a) → Lemma (zero 'op' x == x);
    idR : (x : a) → Lemma (x 'op' zero == x);
    assoc : (x y z : a) → Lemma ((a 'op' b) 'op' c == a 'op' (b 'op' c));
}

let m : monoid int 0 (+) = {
    idL = easy;
    idR = easy;
    assoc = easy;
}
```
Putting it together: Typeclasses
Putting it together: Typeclasses

module FStar.Typeclasses
val mk_class : name \rightarrow Tac unit
val tcresolve : unit \rightarrow Tac unit
module FStar.Typeclasses
val mk_class : name → Tac unit
val tcresolve : unit → Tac unit

module MyMod

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

module FStar.Typeclasses
val mk_class : name → Tac unit
val tcresolve : unit → Tac unit

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

%ssplice (mk_class "MyMod.deq")
Putting it together: Typeclasses

module FStar.Typeclasses
val mk_class : name → Tac unit
val tcresolve : unit → Tac unit

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 ...) *)
module FStar.Typeclasses
val mk_class : name → Tac unit
val tcresolve : unit → Tac unit

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 ...) *)
module FStar.Typeclasses
val mk_class : name \rightarrow Tac unit
val tcresolve : unit \rightarrow Tac unit

module MyMod
class deq a = {
    eq : a \rightarrow a \rightarrow bool;
    eq_ok : (x:a) \rightarrow (y:a) \rightarrow Lemma (eq x y \iff x == y)
}
(* val eq : \#a:Type \rightarrow [\mid deq a [\mid] \rightarrow (a \rightarrow a \rightarrow bool) *)
(* val eq_ok : \#a:Type \rightarrow [\mid deq a [\mid] \rightarrow ((x:a) \rightarrow (y:a) \rightarrow Lemma ... ) *)

[@tcinstance] let eq_int : deq int = { eq = \lambda x y \rightarrow x = y; eq_ok = easy }
module FStar.Typeclasses
val mk_class : name → Tac unit
val tcresolve : unit → Tac unit

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 }
module FStar.Typeclasses
val mk_class : name $\rightarrow$ Tac unit
val tcresolve : unit $\rightarrow$ Tac unit

module MyMod
class deq a = {
  eq : a $\rightarrow$ a $\rightarrow$ bool;
  eq_ok : (x:a) $\rightarrow$ (y:a) $\rightarrow$ Lemma (eq x y $\iff$ x == y)
}

(* val eq : #a:Type $\rightarrow$ [\| deq a \|] $\rightarrow$ (a $\rightarrow$ a $\rightarrow$ bool) *)

(* val eq_ok : #a:Type $\rightarrow$ [\| deq a \|] $\rightarrow$ ((x:a) $\rightarrow$ (y:a) $\rightarrow$ Lemma ...) *)

instance eq_int : deq int = { eq = ($\lambda$ x y $\rightarrow$ x = y); eq_ok = easy }
instance enum_eq (#a:Type) (d : enum a) : deq a = {
  eq = ($\lambda$ (x y : a) $\rightarrow$ tolnt x = tolnt y);
  eq_ok = ($\lambda$ (x y : a) $\rightarrow$ inv1 x; inv1 y);
}
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 ('tinstance) (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))

[@plugin] let tcresolve () : Tac unit = tcresolve' [] 16
Putting it together: Typeclasses

module FStar.Typeclasses
val mk_class : name → Tac unit
val tcresolve : unit → Tac unit

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 }
instance enum_eq (#a:Type) (d : enum a) : deq a = {
  eq = (λ (x y : a) → toInt x = toInt y);
  eq_ok = (λ (x y : a) → inv1 x; inv1 y);
}
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))

[@plugin] let tcresolve () : Tac unit = tcresolve' [] 16
let t_inline () = compute (); trefl ()

[@ (postprocess_for_extraction_with t_inline)]
let t = synth (λ () → fib 8)
let t_inline () = compute (); trefl ()

[@ (postprocess_for_extraction_with t_inline)]
let t = synth (λ () → fib 8)

During verification: \( t = 1 + 1 + 1 + (1 + 1) + \ldots \)
let t_inline () = compute (); trefl ()

[@ (postprocess_for_extraction_with t_inline)]
let t = synth (\() -> fib 8)

During verification: t = 1 + 1 + 1 + (1 + 1) + ....
In extracted code: t = 34
let t_inline () = compute (); trefl ()

[@ (postprocess_for_extraction_with t_inline)]
let t = synth (λ () → fib 8)

During verification: \( t = 1 + 1 + 1 + (1 + 1) + \ldots \)
In extracted code: \( t = 34 \)
The equality is enforced by Meta-F*: verified transformations and optimizations.
Native execution

- By default, metaprograms are evaluated in F*'s normalizer. Primitives are evaluated by calling into compiler internals.
By default, metaprograms are evaluated in F*'s normalizer. Primitives are evaluated by calling into compiler internals.

As an alternative, we can reuse F*'s extraction mechanism to compile metaprograms as OCaml code.
By default, metaprograms are evaluated in F*’s normalizer. Primitives are evaluated by calling into compiler internals.

As an alternative, we can reuse F*’s extraction mechanism to compile metaprograms as OCaml code.

This code can then be compiled into dynamic libraries and loaded.
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!
let mytactic () = trefl (); rest
Coming back to this:

[@plugin] let tcresolve () = ...
Coming back to this:

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

- Typeclass resolution, a user metaprogram, is easily added as a native F* extension.
- Significant efficiency gains
Coming back to this:

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

- Typeclass resolution, a user metaprogram, is easily added as a native F* extension.
- Significant efficiency gains
- No addition to the TCB
Use case: Verifying concurrent programs

- Concurrent programs are challenging due to shared resources
- Crucial to track resources used by each function

Hats off to: Aseem, Monal
Use case: Verifying concurrent programs

- Concurrent programs are challenging due to shared resources
- Crucial to track resources used by each function
- Standard formalism: (concurrent) separation logic

Hats off to: Aseem, Monal
Concurrent programs are challenging due to shared resources
Crucial to track resources used by each function
Standard formalism: (concurrent) separation logic

Problem: separation logic requires to solve for framing, essentially a specific partition of the heap, at nearly every sequencing.
Use case: Verifying concurrent programs

- Concurrent programs are challenging due to shared resources
- Crucial to track resources used by each function
- Standard formalism: (concurrent) separation logic

Problem: separation logic requires to solve for framing, essentially a specific partition of the heap, at nearly every sequencing. By a careful design, we can make the initial and final heaps evident in the VCs, and have the tactic solve for them.

Hats off to: Aseem, Monal
Stateful primitives

val (!) (#a:Type) (r:ref a) :
  St a [r] (\lambda m \to T) (\lambda m \perp m' \to m == m' \land \perp == msel m r)

val ( := ) (#a:Type) (r:ref a) (v:a)
  St unit [r] (\lambda m \to T) (\lambda m () m' \to m' == mupd m r v)

val alloc (#a:Type) (v:a)
  St (ref a) [] (\lambda m \to T) (\lambda m r m' \to m' == (r \mapsto v * m))

val free (#a:Type) (r:ref a)
  St unit [r] (\lambda m \to T) (\lambda m () m' \to m' == emp)
\( \exists (ml: \text{memory}) \ (mr: \text{memory}). \)
\[ \begin{align*}
\text{defined} \ (ml \ast mr) \\
\wedge s \mapsto y \ast r \mapsto x \ast m1 = (ml \ast mr) \\
\wedge (\exists (x: \text{int}). \ ml = (r \mapsto x) \\
\wedge \text{defined} \ (r \mapsto 2 \ast mr) \implies \text{post} (\) \ (r \mapsto 2 \ast mr)) 
\end{align*} \]
\[
\exists (ml: \text{memory}) (mr: \text{memory}). \\
\quad \text{defined (} ml \ast mr \text{)} \\
\quad \land s \mapsto y \ast r \mapsto x \ast m1 \equiv (ml \ast mr) \\
\quad \land (\exists (x: \text{int}). ml \equiv r \mapsto x \\
\quad \quad \land \text{defined (} r \mapsto 2 \ast mr \text{)} \implies \text{post (} r \mapsto 2 \ast mr \text{)}) \\
\Rightarrow \\
\quad \text{defined (} r \mapsto x \ast s \mapsto y \ast m1 \text{)} (* \text{to SMT} *) \\
\quad \land s \mapsto y \ast r \mapsto x \ast m \equiv (r \mapsto x \ast s \mapsto y \ast m1) (* \text{trivial after canonicalization} *) \\
\quad \land (\exists (x': \text{int}). r \mapsto x \equiv r \mapsto x' (* \text{trivial after canonicalization} *) \\
\quad \quad \land \text{defined (} r \mapsto 2 \ast m1 \text{)} \implies \text{post (} r \mapsto 2 \ast m1 \text{)} (* \text{continuation} *)
\]
Concurrency

- Largely follow the same approach. New primitives:
  
  ```
  val par : (#a:Type) → (#b:Type) →
  (#wpa : st_wp a) → (#wpb : st_wp b) →
  (f : (unit → STATE a wpa)) →
  (g : (unit → STATE b wpb)) →
  STATE (a * b) (par_wp wpa wpb)
  ```

  ```
  type lock : list sref → (memory → Type0) → Type0
  ```

  ```
  val mklock (fp : list sref) (inv : memory → Type0)
  : St (lock fp inv) fp (λ m → inv m) (λ _ _ m' → m' == emp)
  ```

  ```
  val acquire (#fp: list sref) (#inv : (memory → Type0)) (l : lock fp inv)
  : St unit [] (λ _ → T) (λ _ _ m' → inv m')
  ```

  ```
  val release (#fp : list sref) (#inv : memory → Type0) (l : lock fp inv)
  : St unit fp (λ m → inv m) (λ _ _ m' → m' == emp)
  ```
let incr_both_swap (r s : ref int) = 
    let v1, v2 = !r, !s in 
    r := v2 + 1; s := v1 + 1 

let acq_step (#r #s : ref int) (l : lock [r; s] (\m -> m.r == m.s)) () = 
    acquire l; incr_both_swap r s; release l 

let test () =
    let r = alloc 1 in let s = alloc 1 in
    let l = mklock (#(\m -> m.r == m.s) [r; s] in
    let _ = par (acq_step l) (acq_step l) in
    acquire l;
    let v = !r in
    let u = !s in
    assert (v == u)
Next steps

- Widen usage of Meta-F*
- Efficiency, always a concern
- Bug extermination
- Use real exceptions for failure
- Minimize primitives, allow layering of user state and discard the goal stack
Next steps

- Widen usage of Meta-F*
- Efficiency, always a concern
- Bug extermination
- Use real exceptions for failure
- Minimize primitives, allow layering of user state and discard the goal stack

Thank you!
Layered DSLs for Verified Stateful Programming

Zoe Paraskevopoulou
Princeton University

with
Nikhil Swamy, Jonathan Protzenko, Tahina Ramananandro, Guido Martínez
Verification of stateful programs

Essential, yet hard!

• Explicit reasoning about the state
  allocation, deallocation, memory safety, aliasing, . . .

• Program Logics
  Hoare Logic, Separation Logic, Weakest precondition semantics

• Tools
  Frama C, Dafny, F*, VST, . . .
F*: Verification of stateful functional programs

- **Dependent Types**

  \[\forall A \ A, A \rightarrow B \rightarrow A\]

- **Monadic Encapsulation of Effects**

  ```
  let incr (l: ref int) : ST unit = l := !l + 1
  ```

- **Weakest Precondition Calculus with SMT backend**

  ```
  let incr (l : ref int) : ST unit (requires (\lambda h \rightarrow l \in h))
  (ensures (\lambda h' \rightarrow get l h' == l + get l h)) = ...
  ```
Low*: Verification of C programs in F*

- A subset of F* that can be translated to C
- Provides access to a CompCert-like memory model for C with the Stack and St effects
- Efficient low-level code, full verification power of F*

```fsharp
let swap (p1 : pointer Int32) (p2 : pointer Int32) : St ()
    (requires (\ (m:mem) \rightarrow live p1 m \land live p2 m))
    (ensures (\ (m:mem) (_:unit) (m':mem) \rightarrow
                      get p1 m = get p2 m' \land get p2 m = get p1 m')) =

let v = !p1 in
p1 := !p2;
p1 := v
```
Example: Crypto verification in Low*

```plaintext
let encrypt_aes_impl (key : barray aes_keylen)
  (iv : barray ivlen)
  (inp : barray inplen) // plain text
  (out : barray outlen) // cipher text
: St ()
  (requires (λ (m:mem) → live key m ∧ live iv m ∧ live in m ∧ live out ∧
    disjoint iv in ∧ disjoint iv out ∧ disjoint in out))
  (ensures (λ (m:mem) (_:_unit) (m’:mem) →
    live key m ∧ live iv m ∧ live in m ∧ live out ∧
    aes_spec (as_seq m’ key) (as_seq m’ iv)
    (as_seq m’ inp) (as_seq m’ out)))
= ...
Example: Crypto verification in Low

```plaintext
let encrypt_aes_impl (key : barray aes_keylen)
  (iv : barray ivlen)
  (inp : barray inplen) // plain text
  (out : barray outlen) // cipher text
  : St ()
  (requires (λ (m:mem) → live key m ∧ live iv m ∧ live in m ∧ live out ∧
              disjoint iv in ∧ disjoint iv out ∧ disjoint in out))
  (ensures (λ (m:mem) (_:unit) (m':mem) →
            live key m ∧ live iv m ∧ live in m ∧ live out ∧
            aes_spec (as_seq m' key) (as_seq m' iv)
            (as_seq m' inp) (as_seq m' out)))

= ...
```
Example: Crypto verification in Low

```plaintext
type aes_state = { key : seq Int32 aes_keylen;
                iv : seq Int32 ivlen;
                in : seq Int32 inlen;
                out : seq Int32 outlen }

let encrypt_aes () : State aes_state ()
               (requires (λ s → True))
               (ensures (λ s _ s' → aes_spec (key s') (iv s')
                           (in s') (out s'))))
```

= ...
Example: Crypto verification in Low*

```plaintext
type aes_state = { key : seq Int32 aes_keylen;
    iv : seq Int32 ivlen;
    in : seq Int32 inlen;
    out : seq Int32 outlen }

let encrypt_aes () : State aes_state ()
    (requires (λ s → True))
    (ensures (λ s__s' → aes_spec (key s') (iv s')
                (in s') (out s'))))

= ...  
```
Example: Crypto verification in Low*

```plaintext
let encrypt_aes_impl (key : barray aes_keylen)
  (iv : barray ivlen)
  (inp : barray inplen) // plain text
  (out : barray outlen) // cipher text
:
St ()
  (requires (\m : mem) \rightarrow live key m \land live iv m \land live in m \land live out \land disjoint iv in \land disjoint iv out \land disjoint in out))

(ensures (\m : mem) (_) : unit (m' : mem) \rightarrow
live key m \land live iv m \land live in m \land live out \land
let s' = encrypt_aes (as_aes_state m key iv inp out) in
m' == from_aes_state m key iv inp out s'))
```

= ...
Layered Approach

- The high-level program has the same computational behavior, but runs on a “customized” memory model
- Push the boilerplate to the low-level correspondence proof
- High-level: the “essence” of the proof

How much of the boilerplate code and proof can be automated?
High computations

Purely functional programs written in monadic style

\[ \text{hcomp } \alpha \text{ wp} \overset{\text{def}}{=} s_0 : \text{hstate } \rightarrow \text{Pure } (\alpha \times \text{hstate}) (\text{wp } s_0) \]

For any \( s_0 : \text{hstate} \) and \( \text{Post : hstate } \rightarrow (\alpha \times \text{hstate} \rightarrow \text{Type}) \rightarrow \text{Type} \), if

\[ \text{wp } s_0 \ \text{Post} \]

then

\[ c \ s_0 = (a, s_1) \]

and

\[ \text{Post } s_0 (a, s_1) \]
DSL for **High** computations

```
return : a → hcomp a (ret_wp a)
bind : hcomp a wp₁ → (a → hcomp b wp₂) → hcomp b (bind_wp wp₁ wp₂)
get : n:nat → hcomp int (wp_get n)
put : n:nat → v:int → hcomp unit (wp_put n v)
for : lo:nat → hi:nat → (nat → hcomp unit wp) → hcomp unit (wp_for lo hi wp)
```

Declare this as an **F**\(^*\) effect:

```
let add () : hcomp int wp_add =
  bind (get 1) (λ x₁ =>
       bind (get 2) (λ x₂ =>
         return (x₁ + x₂)))
```

```
let add () : HIGH int wp_add =
  let x₁ = get 1 in
  let x₂ = get 2 in
  x₁ + x₂
```
Example: Shift and sum two locations

```ocaml
let shift_and_sum_wp s0 post = let (i1, i2) = s0 in post ((), (i1+i2, i1))

val shift_and_sum : unit \rightarrow \textbf{HIGH} \text{ int} shift_and_sum_WP
let shift_and_sum () =
  let x0 = get 0 in
  let x1 = get 1 in
  put 0 (x0 + x1);
  put 1 x0
```
Changing views: From Low to High and back
Through monadic lenses

Low state
\( \text{\textit{lstate}} \) : pointers to \textit{mem} that correspond to the high-level view:

\[
\begin{align*}
\uparrow &: \text{mem} \times \text{\textit{lstate}} \rightarrow \text{\textit{hstate}} \\
\downarrow &: \text{mem} \times \text{\textit{lstate}} \rightarrow \text{\textit{hstate}} \rightarrow \text{mem}
\end{align*}
\]

such that

\[
\begin{align*}
\downarrow_{\text{\textit{(m,ls)}}}(\uparrow_{\text{\textit{(m, ls)}}}) &= \text{m} \\
\downarrow_{\text{\textit{(m,ls)}}}(\text{hs}_2) &= \downarrow_{\text{\textit{(m,ls)}}}(\text{hs}_1)(\text{hs}_2) \\
\uparrow_{\text{\textit{(m,ls)}}}(\text{hs}) &= \text{hs}
\end{align*}
\]
Low computations

Low* programs that refine a high computation

\[
\text{type } \text{lcomp } (a:\text{Type}) \text{ wp } (c: \text{comp } a \text{ wp}) = \\
\text{ls: lstate } \rightarrow \\
\text{St } a \\
\quad (\text{requires } (\lambda m \rightarrow \text{well_formed } m \text{ ls } \land \text{ wp } \uparrow(h, \text{ ls }) (\lambda _ \rightarrow \text{ True}))) \\
\quad (\text{ensures } (\lambda m \text{ r m }' \rightarrow \\
\quad \text{well_formed } m' \text{ ls } \land \\
\quad \text{let } (x, s1) = c \uparrow(m, \text{ ls }) \text{ in } \\
\quad m' = \uparrow_{(m, ls)} (s1) \land x = r)))
\]
DSL for Low computations

\[\begin{align*}
\text{lreturn} &: \ a \to \text{lcomp} \ a \ (\text{ret\_wp} \ a) \ (\text{return} \ a) \\
\text{lbind} &: \ \text{lcomp} \ a \ \text{wp}_1 \ m \to \ (a \to \text{lcomp} \ b \ \text{wp}_2 \ f) \to \\
&\quad \text{lcomp} \ b \ (\text{bind\_wp} \ \text{wp}_1 \ \text{wp}_2) \ (\text{bind} \ m \ f) \\
\text{lget} &: \ n : \text{nat} \to \text{lcomp} \ \text{int} \ (\text{wp\_get} \ n) \ (\text{get} \ n) \\
\text{lput} &: \ n : \text{nat} \to v : \text{int} \to \text{lcomp} \ \text{unit} \ (\text{wp\_put} \ n \ v) \ (\text{put} \ n \ v) \\
\text{lfor} &: \ \text{lo} : \text{nat} \to \text{hi} : \text{nat} \to (\text{nat} \to \text{lcomp} \ \text{unit} \ \text{wp}) \to \\
&\quad \text{lcomp} \ \text{unit} \ (\text{wp\_for} \ \text{lo} \ \text{hi} \ \text{wp}) \ (\text{for} \ \text{lo} \ \text{hi} \ f)
\end{align*}\]
Example: Shift and sum two locations

```ocaml
let hstate = (Int32, Int32)

val shift_and_sum : unit → HIGH int shift_and_sum_wp
let shift_and_sum () =
  let x0 = get 0 in
  let x1 = get 1 in
  put 0 (x0 + x1);
  put 1 x0

let lstate = (pointer, pointer)

val shift_and_sum_low : unit →
lcomp unit shift_and_sum_wp shift_and_sum
let shift_and_sum_low () =
  lbind (lget 0) (\ x0 →
  lbind (lget 1) (\ x1 →
  lbind (lput 0 (x0 + x1)) (\ _ →
  lbind (lput 1 x1) (\ _ →
  lreturn ()])))
```
A trivial Low computation

Every high computation gives rise to a low computation:

```
let morph (a:Type) wp (c: hcomp wp) : lcomp a wp c =
  \l \rightarrow
  let m = get () in // get current memory state
  let (x, hs) = c (↑(m, \l)) in
  put \l \hs; // update current memory
\n```

We know that \( \text{shift\_and\_sum\_low} \) has the same observable behavior as \( \text{morph} \) \( \text{shift\_and\_sum} \)

- How do we define \( \text{shift\_and\_sum\_low} \approx_{obs} \text{shift\_and\_sum} \)
- How can we rewrite with morphism laws to obtain \( \text{shift\_and\_sum\_low} \) from \( \text{morph} \text{shift\_and\_sum} \)
Equality on Low computations

// WPs for low computations

```
type lwp a = lstate → (mem → (a * mem → Type) → Type)
```

// High-level programs as low WPs

```
let as_lwp (#a:Type) wp (c:comp_wp a wp) : lwp a =
    λls m post →
    well_formed m ls ∧
    wp (λ _ → True) ∧
    (let (x, s) = c ↑(m,ls) in
    let m = ↓(m,ls) (s) in post (x, h1))
```
Equality on Low computations

```
let lwp_eq #a (wp1:lwp a) (wp2:lwp a) = 
  precise wp1 ∧ precise wp2 ∧
  (∀ ls h0 post. wp1 ls h0 post ↔ wp2 ls h0 post)

let l_eq (#a:Type)
  #wp1 (#c1:comp_wp a wp1) (lc1: lcomp_wp a wp1 c1)
  #wp2 (#c2:comp_wp a wp2) (lc2 : lcomp_wp a wp2 c2) =
  lwp_eq (as_lwp c1) (as_lwp c2)

assume val l_eq_axiom :
  (#a:Type) →
  wp1 → (#c1:comp_wp a wp1) → (lc1: lcomp_wp a wp1 c1) →
  wp2 → (#c2:comp_wp a wp2) → (lc2 : lcomp_wp a wp2 c2) →
  Lemma (requires (l_eq lc1 lc2)) (ensures (lc1 == lc2))
```
Morphism Laws

\[
\begin{align*}
morph (\text{return } x) &\equiv \text{lreturn } x \\
morph (\text{bind } c f) &\equiv \text{lbind } (morph c) (\lambda x \to \text{morh } (f x)) \\
morph (\text{get } n) &\equiv \text{lget } n \\
morh (\text{put } n v) &\equiv \text{lput } n v \\
morh (\text{for } lo hi f) &\equiv \text{lfor } lo hi (\lambda i \to \text{morh } (f i))
\end{align*}
\]
Obtain the Low program with successive rewrites

Apply the transformation using a tactic!

**Aim:** generic equality-based user-defined program transformation framework
Final Goal

Programmer:

```haskell
let encrypt_aes () : State aes_state ()
    (requires (\ s \rightarrow True))
    (ensures (\ s _ s' \rightarrow aes_spec (key s') (iv s') (in s') (out s'))

= ...
```

Layered DSL framework:

```haskell
: St () (requires (\ (m:mem) \rightarrow live key m ∧ live iv m ∧ live in m ∧ live out ∧
                disjoint iv in ∧ disjoint iv out ∧ disjoint in out))
    (ensures (\ (m:mem) (_:unit) (m':mem) \rightarrow 
                live key m ∧ live iv m ∧ live in m ∧ live out ∧
let s' = encrypt_aes (as_aes_state m key iv inp out) in
m' == from_aes_state m key iv inp out s'))
= ...
```
Part 2:
Normalization by evaluation in $\mathbb{F}^*$
Normalization by evaluation (NBE)

- Tactics: F* programs executed at type-checking time (interpreted)
- Idea: Interpret F* programs in terms of OCaml programs
- Use the evaluation mechanism of the host language
- Translate them back into F* values

```
let nbe e = readback (translate [] e)
...
let normalize cfg e =
  match e with
  ...
  | App f [x] when is_norm_request f && is_nbe_request f → normalize cfg (nbe f)
  ...
```
Normalization by evaluation (NBE)

- **Tactics:** F* programs executed at type-checking time (interpreted)
- **Idea:** Interpret F* programs in terms of OCaml programs
- Use the evaluation mechanism of the host language
- Translate them back into F* values

- Orders of magnitude faster than interpreted code
- Part of the POPL2019 submission

---

Obtain the \textbf{Low} program with successive rewrites

\[ s_0 \xrightarrow{c} s_1 \]

\[ \uparrow (ls, m_0) \]

\[ m_0 \xrightarrow{\text{morh c ls}} m_1 \]

\[ \approx_{\text{obs}} \]

\[ \downarrow (ls, m_0)^{s_1} \]

\[ \text{c_impl ls} \]

\[ m_0 \xrightarrow{} m_1 \]

Apply the transformation using a tactic!

\textbf{Aim:} generic equality-based user-defined program transformation framework