Meta-F*  
Proof automation with SMT, Tactics, and Metaprograms

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

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

Interactive Theorem Provers (ITPs): Coq, Agda, Lean, Idris, ...

- Usually for pure programs
- Very expressive
- Have traditionally relied on tactics for doing proofs
Two camps of program verification

Interactive Theorem Provers (ITPs): Coq, Agda, Lean, Idris, ...
- Usually for pure programs
- Very expressive
- Have traditionally relied on tactics for doing proofs

Program Verifiers: Dafny, VCC, Liquid Haskell, ...
- Verification conditions (VCs) computed and sent to SMT solvers
- Simple proofs often automatic
Two camps of program verification

Interactive Theorem Provers (ITPs): Coq, Agda, Lean, Idris, ...

- Usually for pure programs
- Very expressive
- Have traditionally relied on tactics for doing proofs

Program Verifiers: Dafny, VCC, Liquid Haskell, ...

- Verification conditions (VCs) computed and sent to SMT solvers
- Simple proofs often automatic
- When the solver fails, no good way out
Two camps of program verification

Interactive Theorem Provers (ITPs): Coq, Agda, Lean, Idris, …
- Usually for pure programs
- Very expressive
- Have traditionally relied on tactics for doing proofs

Program Verifiers: Dafny, VCC, Liquid Haskell, …
- Verification conditions (VCs) computed and sent to SMT solvers
- Simple proofs often automatic
- When the solver fails, no good way out
  - Need to tweak the program (to call lemmas, etc)
  - No automation
  - No good way to inspect or transform the proof environment
Two camps of program verification

Interactive Theorem Provers (ITPs): Coq, Agda, Lean, Idris, ...

- Usually for pure programs
- Very expressive
- Have traditionally relied on tactics for doing proofs

Program Verifiers: Dafny, VCC, Liquid Haskell, ...

- Verification conditions (VCs) computed and sent to SMT solvers
- Simple proofs often automatic
- When the solver fails, no good way out
  - Need to tweak the program (to call lemmas, etc)
  - No automation
  - No good way to inspect or transform the proof environment

Can we retain the comfort of automation while avoiding the solver’s issues?
F* basics

• Functional and effectful programming language / program verifier
  • A member of the ML family
  • Extracts to OCaml or F#; a subset (Low*) can also extract to C
  • Used for crypto implementations (e.g. EverCrypt)
F* basics

- Functional and effectful programming language / program verifier
  - A member of the ML family
  - Extracts to OCaml or F#; a subset (Low*) can also extract to C
  - Used for crypto implementations (e.g. EverCrypt)

- Full dependent types
  - As in Coq, Agda, Lean, Idris, etc
**F* basics**

- Functional and effectful programming language / program verifier
  - A member of the ML family
  - Extracts to OCaml or F#; a subset (Low*) can also extract to C
  - Used for crypto implementations (e.g. EverCrypt)
- Full dependent types
  - As in Coq, Agda, Lean, Idris, etc
- Rich specifications over both pure and effectful computations
  - Proof automation via an SMT solver (Z3)
F* basics

• Functional and effectful programming language / program verifier
  • A member of the ML family
  • Extracts to OCaml or F#; a subset (Low*) can also extract to C
  • Used for crypto implementations (e.g. EverCrypt)

• Full dependent types
  • As in Coq, Agda, Lean, Idris, etc

• Rich specifications over both pure and effectful computations
  • Proof automation via an SMT solver (Z3)

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

```ocaml
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))))))}
∀ (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) \implies
(∀ (r: ref int) (h2: heap).
  r \notin h \land h2 == alloc_heap r 1 h \implies
  r \in h2 \land
  (∀ (a: int) (h3: heap).
   a == h2.[r] \land h3 == h2 \implies
    (∀ (b: int).
     b == a + 1 \implies
      r \in h3 \land
      (∀ (h4: heap).
       h4 == upd h3 r b \implies
        r \in h4 \land
        (∀ (v: int) (h5: heap).
         v == h4.[r] \land h5 == h4 \implies
          v == 2 \land (* our assertion *)
          (v == 2 \implies
            p () h5)))))))
Note: Lemma \( \varphi = \text{Pure unit (requires } \top) \text{ (ensures } \lambda \cdot \varphi)) \)

```lean
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 = $ Pure unit (requires $\top$) (ensures $(\lambda () \rightarrow \varphi)$)

let lemma_carry_limb_unrolled (a0 a1 a2 : nat) : Lemma (a0 $\%$ p44 + p44 $\times$ ((a1 + a0 / p44) $\%$ p44) + p88 $\times$ (a2 + ((a1 + a0 / p44) / p44)) == a0 + p44 $\times$ a1 + p88 $\times$ 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 $\times$ ((a1 + a0 / p44) / p44));
distributivity_add_right p44 a1 (a0 / p44)
When SMT doesn't cut it

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

```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)

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

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)

= [propositions]
  \rightarrow \text{pow2_plus 44 44};
  \rightarrow \text{lemma_div_mod (a1 + a0 / p44) p44};
  \rightarrow \text{lemma_div_mod a0 p44};
  \rightarrow \text{distributivity_add_right p88 a2 ((a1 + a0 / p44) / p44)};
  \rightarrow \text{distributivity_add_right p44 ((a1 + a0 / p44) % p44) (p44 * ((a1 + a0 / p44) / p44))};
  \rightarrow \text{distributivity_add_right p44 a1 (a0 / p44)}
Let `lemma_carry_limb_unrolled`:

\[
\text{Lemma}\ (a_0 \% p_{44} + p_{44} \times ((a_1 + a_0 / p_{44}) \% p_{44}) + p_{88} \times (a_2 + ((a_1 + a_0 / p_{44}) / p_{44})) = a_0 + p_{44} \times a_1 + p_{88} \times a_2)
\]

\[=\]

\[\rightarrow\] `pow2_plus 44 44;`
\[\rightarrow\] `lemma_div_mod (a_1 + a_0 / p_{44}) p_{44};`
\[\rightarrow\] `lemma_div_mod a_0 p_{44};`
\[\rightarrow\] `distributivity_add_right p_{88} a_2 ((a_1 + a_0 / p_{44}) / p_{44});`
\[\rightarrow\] `distributivity_add_right p_{44} ((a_1 + a_0 / p_{44}) \% p_{44}) (p_{44} \times ((a_1 + a_0 / p_{44}) / p_{44}));`
\[\rightarrow\] `distributivity_add_right p_{44} a_1 (a_0 / p_{44});`
When SMT *really* doesn't cut it

```ocaml
let lemma_poly_multiply (n p r 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 == hhexpand + b * (n * n * 4 + (-5)))
```
let lemma_poly_multiply (n p r 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

```ocaml
let lemma_poly_multiply (n p r 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 ∧ h = 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
```
Meet Meta-\textsc{F*}

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

- Embedded into \textsc{F*} as an \textit{effect}: \texttt{Tac}
  - Metapograms 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

```plaintext
val exact : term → Tac unit
val apply_lemma : term → Tac unit
val intro : unit → Tac binder
```
Meet Meta-F*  

A tactics and metaprogramming language for F* 

• Embedded into F* as an effect: Tac 
  - Metaprograms are terms with Tac effect 
  - Exceptions, divergence and proof state manipulations 
  - Transformations of the proof state allowed only via primitives for soundness 

val exact : term → Tac unit  
val apply_lemma : term → Tac unit  
val intro : unit → Tac binder 

• F* internals exposed to metaprograms 
  - Inspired by Idris and Lean 
  - Typechecker, normalizer, unifier, etc., are all exposed via an API 
  - Inspect, create and manipulate terms and environments
Meet Meta-$F^*$

A tactics and metaprogramming language for $F^*$

- Embedded into $F^*$ as an **effect**: $\text{Tac}$
  - Metaprograms are terms with $\text{Tac}$ effect
  - Exceptions, divergence and **proof state** manipulations
  - Transformations of the proof state allowed only via primitives for soundness

```
val exact : term $\rightarrow$ Tac unit
val apply_lemma : term $\rightarrow$ Tac unit
val intro : unit $\rightarrow$ Tac binder
```

- $F^*$ internals exposed to metaprograms
  - Inspired by Idris and Lean
  - Typechecker, normalizer, unifier, etc., are all exposed via an API
  - Inspect, create and manipulate terms and environments

```
val tc : term $\rightarrow$ Tac term
val normalize : config $\rightarrow$ term $\rightarrow$ Tac term
val unify : term $\rightarrow$ term $\rightarrow$ Tac bool
```
Metaprograms are first-class citizens

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;
  apply_lemma ('eq_refl)
```
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;
  apply_lemma ('eq_refl)
```

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 first-class citizens

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;
  apply_lemma ('eq_refl)
```

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:

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

Goal 1/1
a b : int
h0 : a > 0
h1 : a = b

________________________________________
f b == f b
```
Metaprograms are written and typechecked as any other kind of effectful term:

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

No more goals
Metaprograms are first-class citizens

Further:

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

The usual compiler pipeline:

Module.fst → Parser → Typechecker → Extraction → Module.ml

- Unifier
- Normalizer
- SMT encoding

mytac.fst + goals

Metaprograms are safe
Metaprogram execution

The usual compiler pipeline:

`Module.fst` → `Parser` → `Typechecker` → `Extraction` → `Module.ml`

- `Unifier`
- `Normalizer`
- `SMT encoding`

`mytac.fst + goals`
Metaprogram execution

The usual compiler pipeline:

```
Module.fst -> Parser -> Typechecker -> Extraction -> Module.ml
```

- Unifier
- Normalizer
- SMT encoding

```
mytac.fst + goals
```
Metaprogram execution

The usual compiler pipeline:

- **Module.fst** → **Parser** → **Typechecker** → **Extraction** → **Module.ml**
- **Unifier**
- **Normalizer**
- **SMT encoding**

```
mytac.fst + goals
```
Metaprogram execution

The usual compiler pipeline:

Module.fst → Parser → Typechecker → Extraction → Module.ml

Unifier → Normalizer → SMT encoding

mytac.fst + goals

Metaprograms are safe **compiler scripts**
Now, let’s use Meta-F*:

```lean
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*:

```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))) by (canon_semiring int_cr; smt ())
```

11 / 18


Splitting assertions

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

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

12 / 18
Splitting assertions

VC will not contain the obligation, instead we get a \textit{goal} for it

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

Goal 1/1
\begin{align*}
n &: \text{int} \\
p &: \text{int} \\
r &: \text{int} \\
\vdots \\
H_0 &: \varphi_1 \\
H_1 &: \varphi_2 \\
\vdots \\
L &= R
\end{align*}
Splitting assertions

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

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

Goal 1/1

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

L = R
Splitting assertions

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

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

Goal 1/1

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

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

- canon_semiring int_cr transforms the goal
Splitting assertions

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

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

Goal 1/1

n : int
p : int
r : int
...  
H0 : \varphi_1
H1 : \varphi_2
...  
\[ nf(L) = nf(R) \]

- canon_semiring int_cr transforms the goal
- partial canonicalization: doesn’t do AC
VC will not contain the obligation, instead we get a \textit{goal} for it

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

Goal 1/1

\begin{align*}
\text{n : int} \\
\text{p : int} \\
\text{r : int} \\
\text{...} \\
\text{H0 : } \varphi_1 \\
\text{H1 : } \varphi_2 \\
\text{...} \\
\end{align*}

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

- canon\_semiring int\_cr transforms the goal
- \textit{partial} canonicalization: doesn’t do AC
- Proof only requires linear arithmetic
Splitting assertions

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

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

Goal 1/1
\[
\begin{align*}
n & : \text{int} \\
p & : \text{int} \\
r & : \text{int} \\
\ldots \\
\text{H0} & : \varphi_1 \\
\text{H1} & : \varphi_2 \\
\ldots \\
nf(L) & = nf(R)
\end{align*}
\]

- canon_semiring \text{int-cr} transforms the goal
- \text{partial} canonicalization: doesn’t do AC
- Proof only requires linear arithmetic
- Rest of proof automatic as before
Splitting assertions

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

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

\[
\begin{array}{|c|c|}
\hline
\text{Goal 1/1} & \text{Success Rate} \\
\hline
\text{n : int} & 0.5\% \\
\text{p : int} & 10\% \\
\text{smt} & 16.5\% \\
\text{smt6x} & 24\% \\
\text{smt25x} & 100\% \\
\text{smt100x} & 100\% \\
\text{tactics} & 100\% \\
\hline
\end{array}
\]

- canon_semiring int_cr transforms the goal
- \textit{partial} canonicalization: doesn’t do AC
- Proof only requires linear arithmetic
- Rest of proof automatic as before
Splitting assertions

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

\[ \forall n \ p \ r \ldots, \varphi_1 \rightarrow \psi_1 \wedge \varphi_2 = \ldots = \varphi_{\neg \not L} = \neg L = R \rightarrow \ldots \]

Goal 1/1

\begin{align*}
\text{n : int} & \\
\text{p : int} & \\
\text{r : int} & \\
\end{align*}

A small tactic (+ SMT) goes a long way

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

Beyond proving, Meta-F* enables constructing terms

let f (x y : int) : int = _ by (exact ('42))
Beyond proving, Meta-F* enables constructing terms

```fsharp
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

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

No more goals
Beyond proving, Meta-F* enables constructing terms

\[
\text{let } f \ (x \ y : \text{int}) : \text{int} = 42
\]

No more goals

- Metaprogramming goals are \textbf{relevant}.
- Proving goals are \textbf{irrelevant}, they have no operational meaning.
- SMT can only be called on irrelevant goals.
let parser t = seq byte → option (t * nat)
let serializer #t (p:parser t) = f:(t → seq byte){∀ x. p (f x) == Some (x, length (f x))}
type package t = { p : parser t ; s : serializer p }
let parser \( t = \text{seq byte} \rightarrow \text{option (t * nat)} \) \\
let serializer \( \#t \) (p:parser t) = f:t \rightarrow \text{seq byte}\{\forall x. \ p(f x) == \text{Some (x, length (f x))}\} \\
\textbf{type} package \( t = \{ \ p : \text{parser t} ; \ s : \text{serializer p} \ \} \) \\

\textbf{type} sample = \text{nlist 18 (u8 * u8)}
let parser t = seq byte → option (t ∗ nat)
let serializer #t (p:parser t) = f:(t → seq byte){∀ x. p (f x) == Some (x, length (f x))}
type package t = { p : parser t ; s : serializer p }

type sample = nlist 18 (u8 ∗ u8)

let ps_sample : package sample = _ by (gen_specs ('sample))
Metaprogramming parsers and serializers

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

type sample = nlist 18 (u8 * u8)

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

let p_low : parser_impl ps_sample.p = _ by gen_parser_impl
let s_low : serializer_impl ps_sample.s = _ by gen_serializer_impl
```
let parser t = seq byte → option (t * nat)
let serializer t (p:parser t) = f:(t → seq byte){∀ x. p (f x) == Some (x, length (f x))}
type package t = { p : parser t ; s : serializer p }

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

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

• All proofs done automatically by the tactic
• Generated parsers/serializers are in Low*, can be extracted to C
Part of TLS' handshake:

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

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

Part of TLS’ handshake:

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

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

(* + proof of mutual correspondance via SMT *)
Customizing implicit arguments

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

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

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

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

\[
\text{let } \text{diag (x:int) (#[same_as x] y : int) : int } \ast \text{ int } = (x, y) \\
\text{diag 42 } \equal{} (42, 42)
\]
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)
  diag 42 == (42, 42)
  diag 42 #50 == (42, 50)
  ```
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)
```

- `diag 42 == (42, 42)`
- `diag 42 #50 == (42, 50)`

- We combine this with some metaprogramming to implement typeclasses completely in **user space**.
Customizing implicit arguments

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

  \begin{verbatim}
  let diag (x:int) (#[same_as x] y : int) : int * int = (x, y)
  diag 42 == (42, 42)
  diag 42 #50 == (42, 50)
  \end{verbatim}

- We combine this with some metaprogramming to implement typeclasses completely in \texttt{user space}.

- Dictionary resolution, \texttt{tcresolve}, is a 20 line metaprogram
Native metaprograms

F*  helloworld.fst  F* compiler

F*

OCaml

ocamlc

Binary  helloworld.exe  fstar.exe

Extract, compile, and load

Run metaprogram natively, no interpretation needed

New primitive: 10x speed gain!

- Safe and fast compiler extensions!
Native metaprograms

- `helloworld.fst` -> `F*` -> `F*` compiler -> `mytac.fst` -> `F*` -> `OCaml` -> `ocamlc` -> `Binary` -> `helloworld.exe` or `fstar.exe`
Native metaprograms

- Extract, compile, and load

```
<table>
<thead>
<tr>
<th>F*</th>
<th>helloworld.fst</th>
<th>F* compiler</th>
<th>mytac.fst</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCaml</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ocamlc</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary</td>
<td>helloworld.exe</td>
<td>fstar.exe</td>
<td>mytac.cmxs</td>
</tr>
</tbody>
</table>
```

New primitive: 10x speed gain!
Can be done for tcresolve: user level typeclasses running at native speed.
Native metaprograms

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

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

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

- Mixing SMT and Tactics, use each for what they do best
  - Simplify proofs for the solver
  - No need for full decision procedures
- Meta-F* enables to extend F* in F* safely
  - Customize how terms are verified, typechecked, elaborated...
  - Native compilation allows fast extensions
- Using an effect for metaprograms increases reuse
Summary

- Mixing SMT and Tactics, use each for what they do best
  - Simplify proofs for the solver
  - No need for full decision procedures
- Meta-F* enables to extend F* in F* safely
  - Customize how terms are verified, typechecked, elaborated...
  - Native compilation allows fast extensions
- Using an effect for metaprograms increases reuse

In the paper:
- Details on implementation
- Trust argument and TCB
- Specifying metaprograms
- More examples
Summary

• Mixing SMT and Tactics, use each for what they do best
  - Simplify proofs for the solver
  - No need for full decision procedures
• Meta-F* enables to extend F* in F* safely
  - Customize how terms are verified, typechecked, elaborated...
  - Native compilation allows fast extensions
• Using an effect for metaprograms increases reuse

In the paper:
• Details on implementation
• Trust argument and TCB
• Specifying metaprograms
• More examples

Thank you!