ML as a Tactic Language, Again

GUIDO MARTÍNEZ, CIFASIS-CONICET Rosario
DANEL AHMAN, Inria Paris
VICTOR DUMITRESCU, MSR-Inria Joint Centre
NICK GIANNARAKIS, Princeton University
CHRIS HAWBLITZEL, Microsoft Research
CĂTĂLIN HRIŢCU, Inria Paris
MONAL NARASIMHAMURTHY, University of Colorado Boulder
ZOE PARASKEVOPOULOU, Princeton University
CLÉMENT PIT-CLAUDEL, MIT CSAIL
JONATHAN PROTZENKO, Microsoft Research
TAHINA RAMANANANDRO, Microsoft Research
ASEEM RASTOGI, Microsoft Research
NIKHIL SWAMY, Microsoft Research

ML originated as the ***meta**language* for constructing proofs in the LCF theorem prover. Due to its many virtues, it became a rôle model for *object-level* languages as well, inspiring an entire family of strict functional programming languages.

A member of this family is F*, a verification-oriented dialect of ML including dependent types and a monadic effect system, the combination of which allows to express arbitrarily complex properties of (effectful!) programs within its type system itself. To prove these properties, F* relies on querying an SMT solver (usually Z3). For properties in well-supported SMT theories (e.g. linear arithmetic or uninterpreted functions) this approach works very well, allowing programmers to verify programs with relative ease. However, for poorly-supported theories (e.g. non-linear arithmetic), the SMT solver might require a prohibitive amount of resources or be wildly inpredictable. Furthermore, these rotten apples spoil the barrel, since combining even a basic proveable fact from a poorly-supported theory with a well-supported one might stump the solver.

Motivated by these issues, we have recently extended F* with a *tactics and metaprogramming engine*, which allows to attack proof obligations with user-defined procedures. Tactics are written in Meta-F*, which is simply the set of those F* programs of a particular Tac effect, defined within the existing type-and-effect system. As such, they readily interoperate with the rest of language and can *themselves* be verified (to a degree). Using tactics allows the programmer to massage proof obligations before querying the SMT solver (possibly simplifying them into the well-supported fragments) or even solving them completely.

Here, we present an overview of Meta-F*, focusing on its design and its execution models. In addition, we discuss the benefits of tactics and metaprogramming in F* through some idiomatic case studies.

1 INTRODUCTION

Scripting proofs using tactics and metaprogramming has a long tradition in interactive theorem provers, starting from Milner's Edinburgh LCF (Gordon et al. 1979). In this lineage, properties about *pure programs* are expressed and proven in expressive higher-order (and often also dependently typed) logics, and proofs are conducted using various programming languages, starting originally with ML.

Along a different axis, program verification tools like Dafny (Leino 2010), Why3 (Filliâtre and Paskevich 2013), Liquid Haskell (Vazou et al. 2014), etc., target pure *and effectful* programs, with side-effects ranging from divergence to concurrency, but provide relatively weak logics in which to specify properties (e.g., first-order logic with a few selected theories like linear arithmetic). The verification conditions computed by these tools are encoded to automated theorem provers such as SMT solvers, with few facilities for user interaction, beyond querying the SMT solver for facts it can

prove. The lack of fine-grained control over proofs and automation, especially when (inherently incomplete) SMT solving times out, is an often cited shortcoming of this second approach.

This paper describes a way to bridge the gap between these worlds in the context of F⋆, a programming language and verification tool combining user-defined effects and a full-spectrum, dependent type system (Ahman et al. 2017; Swamy et al. 2016). Verification in F⋆ has, to date, only relied on generating verification conditions that are discharged by SMT solving. In this work, we describe Meta-F⋆, a metaprogramming framework for F⋆ that unleashes the full power of F⋆'s logic, while still retaining its strong support for SMT-based proof automation. This combination enables new techniques for program verification, which we exemplify through three case studies.

Rather than defining a separate language for metaprogramming and tactics, a central design principle of our work is that Meta-F⋆ resides *within* F⋆. As such, metaprograms are typed, call-by-value, direct-style, higher-order functional programs, as envisioned by Milner. Further, like any other F⋆ program, Meta-F⋆ programs can, in principle, themselves be verified and metaprogrammed. Metaprograms are executed during type-checking and are isolated using F⋆'s user-extensible effect system from programs that execute at run-time.

2 A TASTE OF META-F⋆

It all begins with the notion of a "proofstate". Briefly, the proofstate corresponds to all internal state of F⋆ (and Meta-F⋆) that is relevant for executing metaprograms. In particular it includes the set of goals that are yet to be solved, the state of all unification metavariables, configuration options. It is exposed to the metaprograms as an abstract type, proofstate, for safety reasons.

Once the proofstate is introduced, we introduce our metaprogramming effect simply by invoking F⋆'s effect extension feature. For Meta-F⋆ we define a monad, named tac, combining divergence, exceptions and stateful computations over a proofstate. The definition of tac is straightforward and given in F⋆'s standard library:

```
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 #a (x:a) = λps → Success x ps
let t_bind #a #b (m:tac a) (f:a → tac b) : tac b =
    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 }
```

The new_effect declaration causes F⋆ to generate the TAC effect using tac as its *representation*, and then insert it into its environment. As with all F⋆ effects, TAC-computations are given *computation types* of the form TAC a wp, where a is the computation's result type and wp is a weakest-precondition transformer, in this case of type tacwp a = proofstate → (result a → prop) → prop. We define Tac a as TAC a (null_wp a), i.e., those Tac programs with trivial pre- and post-conditions.

The simplest actions stem from the tac monad definition: there are (get : unit → Tac proofstate), (fail : string → Tac α); and (catch : (unit → Tac α) → Tac (either error α)) all of which have the usual interpretation (catch resets the proofstate on failure). We emphasize that a "set" action is not provided (for safety reasons) and that the argument to catch must be thunked (since F⋆ is call-by-value, computations must be suspended in order to abstract over them).

The only thing separating Tac from other user-defined effects is the existence of Tac-specific primitive actions, which give access to the metaprogramming engine proper. We list here but a few:

```
val trivial : unit → Tac unit        val tc : term → Tac term        val dump : string → Tac unit
```

These are given an interpretation internally by Meta-F⋆: trivial calls into F⋆'s simplifier to solve goals when they are trivial propositions, failing otherwise; tc queries F⋆'s typechecker to infer the type of a term and returns it (with no effect on the proofstate); and dump outputs the current proofstate to the user in a prettified format.

With Tac as an effect, writing metaprograms is as straightforward as writing any other F⋆ code. As an example, here are two small combinators: repeat repeatedly calls its argument until failure returning the list of results; repeat_fold is similar, but folding over the results via some function.

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

As for all other F⋆ effects, metaprograms are written in applicative style, without explicit return, bind, or lift of computations (which are monadic under the hood). This also works across different effects: repeat_fold can combine the pure fold_left from F⋆'s list library with a metaprogram like repeat. Metaprograms are also type- and effect-inferred: while repeat_fold was not at all annotated, F⋆ infers the polymorphic type $(\beta \to \alpha \to \beta) \to \beta \to (\text{unit} \to \text{Tac}\ \alpha) \to \text{Tac}\ \alpha$ for it (in fact, F⋆ infers a richer specification).

There are many ways to invoke metaprograms. The syntax assert φ by tau is equivalent to an assertion of φ, except φ (and only φ) will be preprocessed by the tau tactic. When we want to apply a tactic to an entire VC, the syntax let e : C by tau = E provides the entire verification condition for e (i.e., the fact that E has type C) to tau. For generating terms, the syntax (_ by tau)) calls tau in order to generate a solution for the underscore, where its expected type is taken from the context (or by annotating: (_ by tau) <: ty). Finally, metaprograms can be used to generate top-level definitions via the %splice tau directive, which calls tau to generate the syntax of a list of definitions which will then be typechecked as if they were user-written.

Of course, metaprograms must also be *run*. For this, we again rely on F⋆'s effect extension system to *reify* the effectful metaprograms (TAC) into their pure proofstate-passing representations (tac). Having done that, we may apply this pure representation to a (suitable) initial proofstate and evaluate it according to F⋆'s operational semantics. The assumed actions are implemented as "primitive steps" (just like, say, (+)) and call into Meta-F⋆ to compute a proper value of type result.

It is here where the user can choose the evaluation method, with different tradeoffs:

- By default, tactics are *interpreted* in F⋆'s (previously existing) abstract machine for computing normal forms.
- Also, programmers can use *normalization-by-evaluation* to more efficiently interpret tactics.
- Finally, programmers can mark (certain) tactics as *native*, which will cause them to be extracted to OCaml code, compiled, and dynamically linked into the running F⋆ process. Afterwards, the tactic is simply *called*.

Roughly, the "setup" cost increases further down the list, as overall efficiency increases. However, not all metaprograms can be natively compiled (essentially those that use dynamic syntax inspection or reflection cannot).

3 CASE STUDIES

Separation logic + SMT for stateful programs. Despite being expressible in its logic, proofs using separation logic (Reynolds 2002) have been impractical in F⋆. A key difficulty is in solving for existentially bound heap frames at nearly every step of the proof, which is intractable with an SMT solver. With Meta-F⋆, we programmed a tactic sl_auto to verify stateful programs in a fragment of separation logic. The tactic only instantiates the heap frames, leaving the rest of the proof to the SMT solver, allowing for the mixture of separation logic with F⋆'s regular specifications and theories well-supported in SMT. Here is a small program automatically verified using this approach,

whose correctness is proven without any extra user input. The by (sl_auto ()) annotation indicates that the sl_auto tactic should be used to prove that the given definition has the type given in the let, i.e. that swap meets its specification.

```
let swap (r1 r2:ref int) : STATE unit (λ post m0 → ∃x y. m0 == r1 ↦ x • r2 ↦ y ∧ post () (r1 ↦ y • r2 ↦ x))
                           by (sl_auto ())
  = let x = !r1 in r1 := !r2; r2 := x
```

Non-linear arithmetic in proofs of cryptography. Non-linear arithmetic poses a serious challenge for predictable and scalable SMT proofs, yet it is crucially needed for the verification of cryptographic primitives (Bond et al. 2017; Zinzindohoué et al. 2017). With Meta-F⋆, we program a tactic for canonicalizing terms in commutative semirings, and apply it to prove the main lemmas in the correctness proof of an optimized implementation of the Poly1305 MAC algorithm (Bernstein 2005). Where previous proofs in tools like F⋆ and Dafny were painfully detailed and brittle (e.g., requiring dozens of explicit steps of rewriting of large polynomials to nudge the SMT solver towards a proof), in Meta-F⋆ the proof is reduced to a single call to our canon_semiring tactic, leaving the rest of the proof to the efficient, linear arithmetic theory in Z3 (de Moura and Bjørner 2008).

Below is a main lemma in the proof of correctness of OpenSSL's Poly1305 Message Authentication Code (MAC) (Bernstein 2005). A key step in the proof is the final assertion, a identity in non-linear arithmetic that is reduced by the tactic canon_semiring (in this case applied to the commutative semiring of integers int_cr) to a problem in linear arithmetic that can easily be discharged by an SMT solver, while the original assertion is not.

```
let poly_multiply (n p r h r0 r1 h0 h1 h2 s1 d0 d1 d2 h1 h2 hh : int) =
  let r14 = r1 / 4 in
  let h_r_expand = (h2 * (n * n) + h1 * n + h0) * ((r14 * 4) * n + r0) in
  let hh_expand = (h2 * r0) * (n * n) + (h0 * (r14 * 4) + h1 * r0 + h2 * (5 * r14)) * n
                + (h0 * r0 + h1 * (5 * r14)) in
  let b = (h2 * n + h1) * r14 in
  modulo_addition_lemma hh_expand p b; // (hh_expand + b * p) % p = hh_expand % p
  assert (h_r_expand == hh_expand + b * (n * n * 4 + (−5))) by (canon_semiring int_csr)
```

Metaprogramming the construction of verified effectful programs. Tactics in Meta-F⋆ are a special case of general metaprogramming that supports the inspection and construction of arbitrary program syntax. Our third case study uses Meta-F⋆ to implement a *correct-by-construction* transformation of F⋆ terms within F⋆ itself. Specifically, we generate a low-level formatter for network messages from a high-level, pure specification of the network format, with a proof of their equivalence. The generated low-level code is compatible with the rest of the F⋆ toolchain, allowing it to be extracted to efficient C code by F⋆'s C backend (Protzenko et al. 2017).

4 FULL VERSION

This is a short version of a longer report available at https://arxiv.org/abs/1803.06547. Meta-F⋆ is part of the latest F⋆ release and is under continued active development. The release, including several example metaprograms, is available at https://github.com/FStarLang/FStar/releases/tag/v0.9.6.0.

REFERENCES

D. Ahman, C. Hriţcu, K. Maillard, G. Martínez, G. Plotkin, J. Protzenko, A. Rastogi, and N. Swamy. Dijkstra monads for free. *POPL*. 2017.

D. J. Bernstein. The Poly1305-AES message-authentication code. In *Proceedings of Fast Software Encryption*, 2005.

B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Parno, A. Rane, S. Setty, and L. Thompson. Vale: Verifying high-performance cryptographic assembly code. In *Proceedings of the USENIX Security Symposium*, 2017.

L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. *TACAS*. 2008.

J.-C. Filliâtre and A. Paskevich. Why3 — where programs meet provers. *ESOP*. 2013.

M. J. C. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation.* Springer-Verlag, 1979.

K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. *LPAR*. 2010.

J. Protzenko, J.-K. Zinzindohoué, A. Rastogi, T. Ramananandro, P. Wang, S. Zanella-Béguelin, A. Delignat-Lavaud, C. Hriţcu, K. Bhargavan, C. Fournet, and N. Swamy. Verified low-level programming embedded in F*. *PACMPL*, 1(ICFP):17:1–17:29, 2017.

J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*. 2002.

N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoué, and S. Zanella-Béguelin. Dependent types and multi-monadic effects in F*. *POPL*. 2016.

N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. L. P. Jones. Refinement types for Haskell. *ICFP*, 2014.

J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche. HACL*: A verified modern cryptographic library. Cryptology ePrint Archive, Report 2017/536, 2017. http://eprint.iacr.org/2017/536.