Meta-F*: Proof Automation with
SMT, Tactics, and Metaprograms

Guido Martínez[1,2], Danel Ahman[3], Victor Dumitrescu[4], Nick Giannarakis[5],
Chris Hawblitzel[6], Cătălin Hriţcu[2], Monal Narasimhamurthy[7],
Zoe Paraskevopoulou[5], Clément Pit-Claudel[8], Jonathan Protzenko[6],
Tahina Ramananandro[6], Aseem Rastogi[6], and Nikhil Swamy[6]

[1]CIFASIS-CONICET   [2]Inria Paris   [3]University of Ljubljana
[4]MSR-Inria Joint Centre   [5]Princeton University   [6]Microsoft Research
[7]University of Colorado Boulder   [8]MIT CSAIL

Abstract. We introduce Meta-F*, a tactics and metaprogramming framework for the F* program verifier. The main novelty of Meta-F* is allowing the use of tactics and metaprogramming to discharge assertions not solvable by SMT, or to just simplify them into well-behaved SMT fragments. Plus, Meta-F* can be used to generate verified code automatically.

Meta-F* is implemented as an F* *effect*, which, given the powerful effect system of F*, heavily increases code reuse and even enables the lightweight verification of metaprograms. Metaprograms can be either interpreted, or compiled to efficient native code that can be dynamically loaded into the F* type-checker and can interoperate with interpreted code. Evaluation on realistic case studies shows that Meta-F* provides substantial gains in proof development, efficiency, and robustness.

1   Introduction

Scripting proofs using tactics and metaprogramming has a long tradition in interactive theorem provers (ITPs), starting with Milner's Edinburgh LCF [38]. In this lineage, properties of *pure* programs are specified in expressive higher-order (and often dependently typed) logics, and proofs are conducted using various imperative programming languages, starting originally with ML.

Along a different axis, program verifiers like Dafny [48], VCC [25], Why3 [34], and Liquid Haskell [60] target both 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). They work primarily by computing verification conditions (VCs) from programs, usually relying on annotations such as pre- and postconditions, and encoding them to automated theorem provers (ATPs) such as SMT solvers, often providing excellent automation.

These two sub-fields have influenced one another, though the situation is somewhat asymmetric. On the one hand, most interactive provers have gained support for exploiting SMT solvers or other ATPs, providing push-button automation for certain kinds of assertions [28, 32, 44, 45, 55]. On the other hand,

recognizing the importance of interactive proofs, Why3 [34] interfaces with ITPs like Coq. However, working over proof obligations translated from Why3 requires users to be familiar not only with both these systems, but also with the specifics of the translation. And beyond Why3 and the tools based on it [27], no other SMT-based program verifiers have full-fledged support for interactive proving, leading to several downsides:

Limits to expressiveness The expressiveness of program verifiers can be limited by the ATP used. When dealing with theories that are undecidable and difficult to automate (e.g., non-linear arithmetic or separation logic), proofs in ATP-based systems may become impossible or, at best, extremely tedious.

Boilerplate To work around this lack of automation, programmers have to construct detailed proofs by hand, often repeating many tedious yet error-prone steps, so as to provide hints to the underlying solver to discover the proof. In contrast, ITPs with metaprogramming facilities excel at expressing domain-specific automation to complete such tedious proofs.

Implicit proof context In most program verifiers, the logical context of a proof is implicit in the program text and depends on the control flow and pre- and postconditions of preceding computations. Unlike in interactive proof assistants, programmers have no explicit access, neither visual nor programmatic, to this context, making proof structuring and exploration extremely difficult.

In direct response to these drawbacks, we seek a system that successfully combines the convenience of an automated program verifier for the common case, while seamlessly transitioning to an interactive proving experience for those parts of a proof that are hard to automate. Towards this end, we propose Meta-F*, a tactics and metaprogramming framework for the F* [1, 59] program verifier.

Highlights and Contributions of Meta-F*

F* has historically been more deeply rooted as an SMT-based program verifier. Until now, F* discharged VCs exclusively by calling an SMT solver (usually Z3 [29]), providing good automation for many common program verification tasks, but also exhibiting the drawbacks discussed above.

Meta-F* is a framework that allows F* users to manipulate VCs using *tactics*. More generally, it supports *metaprogramming*, allowing programmers to script the construction of programs, by manipulating their syntax and customizing the way they are type-checked. This allows programmers to (1) implement custom procedures for manipulating VCs; (2) eliminate boilerplate in proofs and programs; and (3) to inspect the proof state visually and to manipulate it programmatically, addressing the drawbacks discussed above. SMT still plays a central role in Meta-F*: a typical usage involves implementing tactics to transform VCs, so as to bring them into theories well-supported by SMT, without needing to (re)implement full decision procedures. Further, the generality of Meta-F* allows

implementing non-trivial language extensions (e.g., typeclass resolution) entirely as metaprogramming libraries, without changes to the F* type-checker.

The technical **contributions** of our work include the following:

"Meta-" is just an effect (§3.1) Meta-F* is implemented using F*'s extensible effect system, which keeps programs and metaprograms properly isolated. Being first-class F* programs, metaprograms are typed, call-by-value, direct-style, higher-order functional programs, much like the original ML. Further, metaprograms can be themselves verified (to a degree, see §3.4) and metaprogrammed.

Reconciling tactics with VC generation (§4.2) In program verifiers one often guides the solver towards the proof by supplying intermediate assertions. Meta-F* retains this style, but additionally allows assertions to be solved by tactics. To this end, a contribution of our work is extracting from a VC a proof state encompassing all relevant hypotheses, even if implicit in the program text.

Executing metaprograms efficiently (§5) Metaprograms are executed during type-checking. As a baseline, they can be interpreted using F*'s existing (but slow) abstract machine for term normalization, or a faster normalizer based on normalization by evaluation (NbE) [13, 19]. For much faster execution speed, metaprograms can be run natively. This is achieved combining the existing extraction mechanism of F* to OCaml with a new framework for safely extending the F* type-checker with such native code.

Examples (§2) and evaluation (§6) We evaluate Meta-F* on several case studies. First, we present a functional correctness proof for the Poly1305 message authentication code (MAC) [14], using a novel combination of proofs by reflection for dealing with non-linear arithmetic and SMT solving for linear arithmetic. We measure a clear gain in proof robustness: SMT-only proofs succeed only rarely (for reasonable timeouts), whereas our tactic+SMT proof is concise, never fails, and is faster. Next, we demonstrate an improvement in expressiveness, by developing a small library for proofs of heap-manipulating programs in separation logic, which was previously out-of-scope for F*. Finally, we illustrate the ability to automatically construct verified effectful programs, by introducing a library for metaprogramming verified low-level parsers and serializers with applications to network programming, where verification is accelerated by processing the VC with tactics, and by programmatically tweaking the SMT context.

We conclude that tactics and metaprogramming can be prosperously combined with VC generation and SMT solving to build verified programs with better, more scalable, and more robust automation.

2   Meta-F* by Example

F* is a general-purpose programming language aimed at program verification. It puts together the automation of an SMT-backed deductive verification tool

with the expressive power of a language with full-spectrum dependent types. Briefly, it is a functional, higher-order, effectful, dependently typed language, with syntax loosely based on OCaml. F* supports refinement types and Hoare-style specifications, computing VCs of computations via a type-level weakest precondition (WP) calculus known as a *Dijkstra monad* [58]. F*'s effect system is also user-extensible [1]. Using it, one can model or embed imperative programming in styles ranging from ML to C [56] to assembly [36]. After verification, F* programs can be extracted to efficient OCaml or F# code. A first-order fragment of F*, called Low*, can also be extracted to C via the KreMLin compiler [56].

This paper introduces Meta-F*, a metaprogramming framework for F* that allows users to safely customize and extend F* in many ways. For instance, Meta-F* can be used to preprocess or solve proof obligations; synthesize F* expressions; generate top-level definitions; and resolve implicit arguments in user-defined ways, enabling non-trivial extensions. This paper primarily discusses the first two features. Technically, none of these features deeply increase the expressive power of F*, since one could manually program in F* terms that can now be metaprogrammed. However, as we will see shortly, manually programming terms and their proofs can be so prohibitively costly as to be practically infeasible.

Meta-F* is similar to other tactic frameworks, such as Coq's [30] or Lean's [31], in presenting a set of goals to the programmer, providing commands to break them down, allowing to inspect and build abstract syntax, etc. In this paper, we mostly detail the characteristics where Meta-F* *differs* from other engines.

This section presents Meta-F* informally, displaying its usage through case studies. We present any necessary F* background as needed.

2.1   Tactics for Individual Assertions and Partial Canonicalization

Non-linear arithmetic reasoning is crucially needed for the verification of optimized, low-level cryptographic primitives [21, 66], an important use case for F* [16] and other verification frameworks, including those that rely on SMT solving alone (e.g., Dafny [48]) as well as those that rely exclusively on tactic-based proofs (e.g., FiatCrypto [33]). While both styles have demonstrated significant successes, we make a case for a middle ground, leveraging the SMT solver for the parts of a VC where it is effective, and using tactics only where it is not.

We focus on Poly1305 [14], a widely-used cryptographic MAC that computes a series of integer multiplications and additions modulo a large prime number $p = 2^{130} - 5$. Implementations of the Poly1305 multiplication and mod operations are carefully hand-optimized to represent 130-bit numbers in terms of smaller 32-bit or 64-bit registers, using clever tricks; proving their correctness requires reasoning about long sequences of additions and multiplications.

Previously: Guiding SMT solvers by manually applying lemmas Prior proofs of correctness of Poly1305 and other cryptographic primitives using SMT-based program verifiers, including F* [66] and Dafny [21], use a combination of SMT automation and manual application of lemmas. On the plus side, SMT

solvers are excellent at linear arithmetic, so these proofs delegate all associativity-commutativity (AC) reasoning about addition to SMT. Non-linear arithmetic in SMT solvers, even just AC-rewriting and distributivity, are, however, inefficient and unreliable—so much so that the prior efforts above (and other works too [41, 42]) just turn off support for non-linear arithmetic in the solver, in order not to degrade verification performance across the board, due to poor interaction of theories. Instead, users need to explicitly invoke lemmas.[1]

For instance, here is a statement and proof of a lemma about Poly1305 in F⋆. The property and its proof do not really matter; the lines marked "*(∗argh! ∗)*" do. In this particular proof, working around the inability of the solver to reason effectively about non-linear arithmetic, the programmer has spelled out basic facts about distributivity of multiplication and addition, by calling the library lemma distributivity_add_right, in order to guide the solver towards the proof. (Below, p44 and p88 represent 2^{44} and 2^{88} respectively)

```
let lemma_carry_limb_unrolled (a0 a1 a2 : nat) : Lemma (ensures (
    a0 % p44 + p44 * ((a1 + a0 / p44) % p44) + p88 * (a2 + ((a1 + a0 / p44) / p44))
    == a0 + p44 * a1 + p88 * a2)) =
  let z = a0 % p44 + p44 * ((a1 + a0 / p44) % p44)
          + p88 * (a2 + ((a1 + a0 / p44) / p44)) in
  distributivity_add_right p88 a2 ((a1 + a0 / p44) / p44); (* argh! *)
  pow2_plus 44 44;
  lemma_div_mod (a1 + a0 / p44) p44;
  distributivity_add_right p44 ((a1 + a0 / p44) % p44)
          (p44 * ((a1 + a0 / p44) / p44)); (* argh! *)
  assert (p44 * ((a1 + a0 / p44) % p44) + p88 * ((a1 + a0 / p44) / p44)
          == p44 * (a1 + a0 / p44) );
  distributivity_add_right p44 a1 (a0 / p44); (* argh! *)
  lemma_div_mod a0 p44
```

Even at this relatively small scale, needing to explicitly instantiate the distributivity lemma is verbose and error prone. Even worse, the user is blind while doing so: the program text does not display the current set of available facts nor the final goal. If the user makes a mistake in some lemma call, she might simply get a useless hint instead of an error; then, the lemma fails to verify usually without providing much insight on why.

Given enough time, the solver can sometimes find a proof without the additional hints, but this is usually rare and dependent on context, and almost never robust. In this particular example we find by varying Z3's random seed that, in an isolated setting, the lemma is proven automatically about 32% of the time. The numbers are much worse for more complex proofs, and where the context contains many facts, making this style quickly spiral out of control. For example, a proof of one of the main lemmas in Poly1305, poly_multiply, requires 41 steps

---

[1] Lemma (requires pre) (ensures post) is F⋆ notation for the type of a computation proving pre ⟹ post—we omit pre when it is trivial. In F⋆'s standard library, math lemmas are proved using SMT with little or no interactions between problematic theory combinations. These lemmas can then be explicitly invoked in larger contexts, and are deleted during extraction.

of rewriting for associativity-commutativity of multiplication, and distributivity of addition and multiplication—making the proof too long to show here.

SMT and tactics in Meta-F* The listing below shows the statement and proof of poly_multiply in Meta-F*, of which the lemma above was previously only a small part. Again, the specific property proven is not particularly relevant to our discussion. But, this time, the proof contains just two steps.

```
let poly_multiply (n p r h r0 r1 h0 h1 h2 s1 d0 d1 d2 h1 h2 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 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;
  assert (h_r_expand == hh_expand + b * (n * n * 4 + (−5)))
      by (canon_semiring int_csr) (* Proof of this step by Meta-F* tactic *)
```

First, we call a single lemma about modular addition from F*'s standard library. Then, we assert an equality annotated with a tactic (assert..by). Instead of encoding the assertion as-is to the SMT solver, it is preprocessed by the canon_semiring tactic. The tactic is presented with the asserted equality as its goal, in an environment containing not only all variables in scope but also hypotheses for the precondition of poly_multiply and the postcondition of the modulo_addition_lemma call (otherwise, the assertion could not be proven). The tactic will then canonicalize the sides of the equality, but notably only "up to" linear arithmetic conversions. Rather than fully canonicalizing the terms, the tactic just rewrites them into a sum-of-products canonical form, leaving all the remaining work to the SMT solver, which can then easily and robustly discharge the goal using linear arithmetic only.

This tactic works over terms in the commutative semiring of integers (int_csr) using proof-by-reflection [15, 23, 37, 39]. Internally, it is composed of a simpler, also proof-by-reflection based tactic canon_monoid that works over monoids, which is then "stacked" on itself to build canon_semiring. The basic idea of proof-by-reflection is to reduce most of the proof burden to mechanical computation, obtaining much more efficient proofs compared to repeatedly applying lemmas. For canon_monoid, we begin with a type for monoids, a small AST representing monoid values, and a denotation for expressions back into the monoid type.

```
type monoid (a:Type) = { unit : a; mult : (a → a → a); (* + monoid laws ... *) }

type exp (a:Type) : Type =
  | Unit : exp a | Var : a → exp a | Mult : exp a → exp a → exp a
(* Note on syntax: #a below denotes that a is an implicit argument *)
```

```
let rec denote (#a:Type) (m:monoid a) (e:exp a) : a =
  match e with
  | Unit → m.unit | Var x → x | Mult x y → m.mult (denote m x) (denote m y)
```

To canonicalize an exp, it is first converted to a list of operands (flatten) and then reflected back to the monoid (mldenote). The process is proven correct, in the particular case of equalities, by the monoid_reflect lemma.

```
val flatten : #a:Type → exp a → list a
val mldenote : #a:Type → monoid a → list a → a
let monoid_reflect (#a:Type) (m:monoid a) (e₁ e₂ : exp a)
              : Lemma (requires (mldenote m (flatten e₁) == mldenote m (flatten e₂)))
                      (ensures (denote m e₁ == denote m e₂)) = ...
```

At this stage, if the goal is $t_1 == t_2$, we require two monoidal expressions e_1 and e_2 such that $t_1 ==$ denote m e_1 and $t_2 ==$ denote m e_2. They are constructed by the tactic canon_monoid by inspecting the *syntax* of the goal, using Meta-F*'s reflection capabilities (detailed ahead in §3.3). We have no way to prove once and for all that the expressions built by canon_monoid correctly denote the terms, but this fact can be proven automatically at each application of the tactic, by simple unification. The tactic then applies the lemma monoid_reflect m $e_1 e_2$, and the goal is changed to mldenote m (flatten e_1) == mldenote m (flatten e_2). Finally, by normalization, each side will be canonicalized by running flatten and mldenote.

The canon_semiring tactic follows a similar approach, and is similar to existing reflective tactics for other proof assistants [12, 39], except that it only canonicalizes up to linear arithmetic, as explained above. The full VC for poly_multiply contains many other facts, e.g., that p is non-zero so the division is well-defined and that the postcondition does indeed hold. These obligations remain in a "skeleton" VC that is also easily proven by Z3. This proof is much easier for the programmer to write and much more robust, as detailed ahead in §6.1. The proof of Poly1305's other main lemma, poly_reduce, is also similarly well automated.

Tactic proofs without SMT Of course, one can verify poly_multiply in Coq, following the same conceptual proof used in Meta-F*, but relying on tactics only. Our proof (see Fig. 1 in the appendix) is 27 lines long, two of which involve the use of Coq's ring tactic (similar to our canon_semiring tactic) and omega tactic for solving formulas in Presburger arithmetic. The remaining 25 lines of proof include steps to destruct the propositional structure of terms, rewrite by equalities, enriching the context to enable automatic modulo rewriting (Coq does not fully automatically recognize equality modulo p as an equivalence relation compatible with the arithmetic operators). While a mature proof assistant like Coq has libraries and tools to make this kind of proof manipulation palatable, it can still be verbose.

In contrast, in Meta-F* all of these mundane parts of a proof are quickly dispatched to an SMT solver, which decides linear arithmetic efficiently, beyond the quantifier-free Presburger fragment supported by tactics like omega, handles congruence closure natively, etc.

2.2 Tactics for Entire VCs and Separation Logic

A different way to invoke Meta-F* is over an entire VC. While the exact shape
of VCs is hard to predict, users with some experience can write tactics that
find and solve particular sub-assertions within a VC, or simply massage them
into shapes better suited for the SMT solver. We illustrate the idea on proofs of
heap-manipulating programs.

One verification method that has eluded F* until now is separation logic,
the main reason being that the pervasive "frame rule" requires instantiating
existentially quantified heap variables, a challenge for SMT solvers. Manually
specifying frames would also be extremely tedious for users. With Meta-F*, one
can do better. We have written a (proof-of-concept) embedding of separation
logic and a tactic (sl_auto) that performs heap frame inference automatically.

The approach we follow consists of designing the WP specifications for prim-
itive stateful actions so as to make their footprint syntactically evident. The
tactic then descends through VCs until it finds an existential for heaps aris-
ing from the frame rule. Then, by solving an equality between heap expressions
(which requires canonicalization, for which we use a variant of canon_monoid
targeting *commutative* monoids) the tactic finds the frames and instantiates
the existentials. Notably, as opposed to other tactic frameworks for separation
logic [7, 46, 50, 52], this is *all* our tactic does before dispatching to the SMT
solver, which can now be effective over the instantiated VC.

We now provide some detail on the framework. Below, 'emp' represents the
empty heap, '$\bullet$' is the separating conjunction and '$r \mapsto v$' is the heaplet with the
single reference r set to value v. Our development distinguishes between a "heap"
and its "memory" for technical reasons, but we will treat the two as equivalent
here. Further, defined is a predicate discriminating valid heaps (as in [53]), i.e.,
those built from separating conjunctions of *actually* disjoint heaps.

We first define the type of WPs and show the WP for the frame rule:

```
let pre = memory → prop (∗ predicate on initial heaps ∗)
let post a = a → memory → prop (∗ predicate on result values and final heaps ∗)
let wp a = post a → pre (∗ transformer from postconditions to preconditions ∗)

let frame_post (#a:Type) (p:post a) (m₀:memory) : post a =
  λx m₁ → defined (m₀ • m₁) ∧ p x (m₀ • m₁)
let frame_wp (#a:Type) (wp:wp a) (post:post a) (m:memory) =
  ∃m₀ m₁. defined (m₀ • m₁) ∧ m == (m₀ • m₁) ∧ wp (frame_post post m₁) m₀
```

Intuitively, frame_post p m_0 behaves as the postcondition p "framed" by m_0, i.e.,
frame_post p m_0 x m_1 holds when the two heaps m_0 and m_1 are disjoint and p
holds over the result value x and the conjoined heaps. Then, frame_wp wp takes
a postcondition p and initial heap m, and requires that m can be split into disjoint
subheaps m_0 (the footprint) and m_1 (the frame), such that the postcondition p,
when properly framed, holds over the footprint.

In order to provide specifications for primitive actions we start in small-
footprint style. For instance, below is the WP for reading a reference:

```
let read_wp (#a:Type) (r:ref a) = λpost m₀ → ∃x. m₀ == r ↦ x ∧ post x m₀
```

We then insert framing wrappers around such small-footprint WPs when exposing the corresponding stateful actions to the programmer, e.g.,

```
val (!) : #a:Type → r:ref a → STATE a (λ p m → frame_wp (read_wp r) p m)
```

To verify code written in such style, we annotate the corresponding programs to have their VCs processed by sl_auto. For instance, for the swap function below, the tactic successfully finds the frames for the four occurrences of the frame rule and greatly reduces the solver's work. Even in this simple example, not performing such instantiation would cause the solver to fail.

```
let swap_wp (r₁ r₂ : ref int) =
    λp m → ∃x y. m == (r₁ ↦ x • r₂ ↦ y) ∧ p () (r₁ ↦ y • r₂ ↦ x)

let swap (r₁ r₂ : ref int) : ST unit (swap_wp r₁ r₂) by (sl_auto ()) =
    let x = !r₁ in let y = !r₂ in r₁ := y; r₂ := x
```

The sl_auto tactic: (1) uses syntax inspection to unfold and traverse the goal until it reaches a frame_wp—say, the one for $!r_2$; (2) inspects frame_wp's first explicit argument (here read_wp r_2) to compute the references the current command requires (here r_2); (3) uses unification variables to build a memory expression describing the required framing of input memory (here $r_2 \mapsto ?u_1 \bullet ?u_2$) and instantiates the existentials of frame_wp with these unification variables; (4) builds a goal that equates this memory expression with frame_wp's third argument (here $r_1 \mapsto x \bullet r_2 \mapsto y$); and (5) uses a commutative monoids tactic (similar to §2.1) with the heap algebra (emp, •) to canonicalize the equality and sort the heaplets. Next, it can solve for the unification variables component-wise, instantiating $?u_1$ to y and $?u_2$ to $r_1 \mapsto x$, and then proceed to the next frame_wp.

In general, once the frames are instantiated, the SMT solver can efficiently prove the remaining assertions. In particular, all obligations about heap definedness are left to the solver. Thus, with relatively little effort, Meta-F⋆ brings a widely used yet previously out-of-scope program logic (i.e., separation logic) into F⋆. To the best of our knowledge, the ability to *script* separation logic into an SMT-based program verifier, without any primitive support, is unique.

2.3   Metaprogramming Verified Low-level Parsers and Serializers

Above, we used Meta-F⋆ to manipulate VCs for existing code. Here, we focus instead on generating the verified code automatically. We loosely refer to the previous setting as using "tactics", and to this one as "metaprogramming". In most ITPs, tactics and metaprogramming are not distinguished; however in a verifier of effectful programs like F⋆, where some proofs are not materialized at all (§4.1), proving VCs of existing terms is distinct from generating new terms.

Metaprogramming in F⋆ involves programmatically generating a (potentially effectful) term (e.g., by constructing its syntax and instructing F⋆ how to type-check it) and processing any VCs that arise via tactics. When applicable (e.g., when working in a domain-specific language), metaprogramming verified code can substantially reduce, or even eliminate, the burden of manual proofs.

We illustrate this by automating the generation of parsers and serializers from a type definition. Of course, this is a routine task in many mainstream metaprogramming frameworks (e.g., Template Haskell, camlp4, etc). The novelty here is that we produce imperative parsers and serializers extracted to C, with proofs that they are memory safe, functionally correct, and mutually inverse. This section is slightly simplified, more detail can be found in §A.

We proceed in several stages. First, we program a library of pure, high-level parser and serializer combinators, proven to be (partial) mutual inverses of each other. A parser for a type t is represented as a function possibly returning a t along with the amount of bytes of input consumed. The type of serializers is indexed by a p:parser t and contains a refinement stating that the parser is an inverse of the serializer. A package consists of a parser and its associated serializer.

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

Basic combinators in the library include constructs for parsing and serializing base values and pairs, with the following signatures:

```
p_u8 : parse u8
s_u8 : serializer p_u8
p_pair : parser t1 → parser t2 → parser (t1 ∗ t2)
s_pair : serializer p1 → serializer p2 → serializer (p_pair p1 p2)
```

Next, we define a library of low-level combinators that parse from and serialize to mutable arrays of bytes; these are coded in the Low⋆ subset of F⋆ that can be extracted to efficient C code. We prove our low-level combinators correct with respect to their high-level counterparts, and hence also mutually inverse. The type parser_impl (p:parser t) is the type of an imperative function that reads from an array of bytes and returns a t, as characterized by the specificational parser p. Conversely, serializer_impl (s:serializer p) writes into an array of bytes, as characterized by the specificational serializer s.

Given such a library, we would like to build verified, mutually inverse, low-level parsers and serializers for specific data formats. The task is mechanical, yet overwhelmingly tedious by hand, with many auxiliary proof obligations of a predictable structure: this is a perfect metaprogramming task.

Deriving specifications from a type definition Consider the following F⋆ type, representing lists of 18 pairs of bytes.

```
type sample = nlist 18 (u8 ∗ u8)
```

The first component of our metaprogram is gen_specs, which from a type definition generates parser and serializer specifications.

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

The syntax _ by τ is the way to call Meta-F⋆ for code generation. Meta-F⋆ will run the metaprogram τ and, if successful, replace the underscore by the result. In this case, the gen_specs (`sample) reflects on the syntax of the sample type (§3.3) and produces the pair of specificational parser and serializer below:

```
Mk (p_nlist 18 (p_u8 `and_then` p_u8)) (s_nlist 18 (s_u8 `and_then` s_u8))
```

Deriving low-level implementations that match specifications From this pair of specifications, we can generate a Low* implementation via gen_parser_impl:

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

which is a metaprogram producing the following implementation:

```
parse_nlist_impl 18ul (parse_u8_impl `and_then` parse_u8_impl)
serialize_nlist_impl 18ul (serialize_u8_impl `and_then` serialize_u8_impl)
```

using low-level parser and serializer combinators. These combinators are proved memory-safe and functionally correct with respect to the specification combinators, once and for all within the library.

For simple types like the ones above, the generated code is fairly simple. However, for more complex types, using the combinator library comes with non-trivial proof obligations. For example, even for a simple enumeration, type color = Red | Green, the parser specification is as follows:

```
parse_synth (parse_bounded_u8 2)
            (λ x2 → mk_if_t (x2 = 0uy) (λ _ → Red) (λ _ → Green))
            (λ x → match x with | Green → 1uy | Red → 0uy)
```

We represent Red with 0uy and Green with 1uy. The parser first parses a "bounded" byte, with only two values. The parse_synth combinator then expects functions between the bounded byte and the datatype being parsed (color), which must be proved to be in bijection. This proof is conceptually easy, but for large enumerations nested deep within the structure of other types, this is cumbersome. Interestingly, the SMT solver isn't particularly efficient at doing this proof either when faced with a large context of facts; since the proof is inherently computational, a proof that destructs the inductive type into its cases and then normalizes is much more natural. With our metaprogram, we can produce the term and then discharge any proof obligations that arise with a tactic, automating the whole process efficiently. We also explore simply tweaking the SMT context, again by a tactic, with good results. A quantitative evaluation is provided in §6.2.

The generated parsers and serializers are correct by construction, memory-safe, and mutually inverse. On extraction, the combinators are inlined to yield Low* code that can be translated by the KreMLin compiler to efficient C code.

3   The Design of Meta-F*

Having seen some of the use cases for Meta-F*, we now turn to its design. As usual in proof-assistants, Meta-F* tactics work over a set of goals and apply primitive actions to them, possibly solving some goals and generating new goals in the process. We instead focus the most on the aspects where Meta-F* differs from other tactic engines. We first describe how metaprograms are modelled as

an effect (§3.1). We then describe the runtime model for Meta-F⋆ (§3.2). We then detail some of Meta-F⋆'s syntax inspection and building capabilities (§3.3). Finally, the tactic effect is also paired with a WP calculus, which enables some (lightweight) verification of metaprograms (§3.4).

3.1   An Effect for Metaprogramming

Meta-F⋆ tactics are, at their core, programs that transform the "proof state", i.e. the set of remaining goals to be solved. As in Lean [31] and Idris [24], we define a monad combining exceptions and stateful computations over a proof state, along with actions that can access internal components such as the type-checker. For this we first introduce abstract types for the proof state, goals, terms, environments, etc., together with some functions to access them, some of them shown below.

```
type proofstate                          val goals_of : proofstate → list goal
type goal                                val goal_env : goal → env
type term                                val goal_type : goal → term
type env                                 val goal_solution : goal → term
```

Based on this, we can readily define our metaprogramming monad: tac. The monad combines F⋆'s existing effect for potential divergence (Div), with a monad for exceptions and stateful computations over a proofstate. The definition of tac, shown below, is straightforward and given in F⋆'s standard library. Then, we use F⋆'s effect extension capabilities (see [1] for details) to elevate the tac monad and its actions to an effect, which we dub TAC.

```
type error = exn * proofstate (* error and proofstate at the time of failure *)
type result a = | Success : a → proofstate → result a | Failed : error → result a
let tac a = proofstate → Div (result a)
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 → ... (* omitted, yet simple *)
let get () : tac proofstate = λps → Success ps ps
let raise #a (e:exn) : tac a = λps → Failed (e, ps)
new_effect { TAC with repr = tac ; return = t_return ; bind = t_bind
                              ; get = get ; raise = raise }
```

The new_effect declaration introduces the TAC effect using tac as its *representation*. Until §3.4 we only use the derived form Tac a, where a is the metaprogram's result type and the pre- and postconditions are trivial. The *computation type* Tac a is distinct from its underlying monadic representation type tac a— users cannot directly access the proof state aside from primitive actions provided by our framework. The simplest actions stem from the tac monad definition: get : unit → Tac proofstate returns the current proof state and raise: exn → Tac α fails with the given exception (we use greek letters α, β, ... to abbreviate universally quantified type variables). Failures can be handled using catch : (unit → Tac α) → Tac (either exn α), which resets the state on failure, including that of unification metavariables. We emphasize two points here. First, there is no "set" action. This is to forbid metaprograms from arbitrarily replacing their proof state, which

would be unsound. Instead, the proof state can only be modified by primitive actions that are guaranteed to preserve soundness (see §4.1). Second, the argument to catch must be thunked, since in F⋆ impure un-suspended computations are evaluated before they are passed into functions.

The only aspect differentiating Tac from other user-defined effects is the existence of effect-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

All of these are given an interpretation internally by Meta-F⋆. For instance, trivial calls into F⋆'s logical simplifier to check whether the current goal is a trivial proposition and discharges it if so, failing otherwise. The tc primitive queries the type-checker to infer the type of a given term in the current environment (F⋆ types are simply terms, hence the codomain of tc is also term). This does not change the proof state; its only purpose is to return useful information to the calling metaprograms. Finally, dump outputs the current proof state to the user in a pretty-printed format, in support of user interaction.

Having introduced the Tac effect and some basic actions, writing metaprograms is as straightforward as writing any other F⋆ code. For instance, here are two metaprogram combinators. The first one repeatedly calls its argument until it fails, returning a list of all the successfully-returned values. The second one behaves similarly, but folds the results with some provided folding function.

```
let rec repeat (τ : unit → Tac α) : Tac (list α) =
  match catch τ with
  | Inl _ → []
  | Inr x → x :: repeat τ

let repeat_fold f e τ = fold_left f e (repeat τ )
```

These two small combinators illustrate a few key points of Meta-F⋆. As for all other F⋆ effects, metaprograms are written in applicative style, without explicit return, bind, or lift of computations (which are inserted under the hood). This also works across different effects: repeat_fold can seamlessly 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.

It should be noted that, if lacking an effect extension feature, one could embed metaprograms simply via the (properly abstracted) tac monad instead of the Tac effect. It is just more convenient to use an effect, given we are working within an effectful program verifier already. In what follows, with the exception of §3.4 where we infer and prove specifications for metaprograms, there is little reliance on metaprograms being embedded as an effect; so, our ideas could be applied in other settings.

3.2   Executing Metaprograms

Running Meta-F* metaprograms requires three steps. First, they are *reified* into their underlying `tac` representation, as state-passing functions [1]. (User code cannot reify `Tac` computations: only F* is able to do so.) Second, the resulting term is applied to an initial proof state, and then evaluated according to F*'s dynamic semantics, for instance using F*'s existing normalizer. For intensive applications, such as proofs by reflection, we provide faster alternatives (§5). In order to perform this second step, the proof state must be *embedded* as a term, i.e., as abstract syntax. Here is where its abstraction pays off: since metaprograms cannot interact with a proof state except through our limited interface, it need not be *deeply* embedded as syntax. By simply wrapping the internal proofstate into a new kind of term, and making the primitives aware of this wrapping, we can readily run the metaprogram that safely carries its "alien" proof state.

   The third step is interpreting the primitives. They are realized by functions of similar types implemented within the F* type-checker, but over an internal `tac` monad and the concrete definitions for `term`, `proofstate`, etc. Hence, there is a translation involved on every call and return, switching between embedded representations and their concrete variants. Take `dump`, for example, with type `string → Tac unit`. When interpreting a call to it, the interpreter must *unembed* the argument (which is an F* term) into a concrete string to pass to the implementation of `dump`. The situation is symmetric for the return value of the call, which must be *embedded* as terms. Fortunately, for the initial and final proof states the embedding and unembedding is very cheap, as explained above.

3.3   Syntax Inspection, Generation, and Quotation

If metaprograms are to be reusable over different kinds of goals, they must be able to reflect on the goals they are invoked to solve. Like any metaprogramming system, Meta-F* offers a way to inspect and construct the syntax of F* terms. Our representation of terms as an inductive type, and the variants of quotations, are inspired by the ones in Idris [24] and Lean [31].

Inspecting syntax   Internally, F* uses a locally-nameless representation with explicit, delayed substitutions. Hiding this complexity, we expose a simplified view [63] of terms to metaprograms, shielding them from some of the internal bureaucracy. Below we present a few constructors from the `term_view` type:

```
val inspect : term → Tac term_view    type term_view =
val pack : term_view → term             | Tv_BVar : v:dbvar → term_view
                                        | Tv_Var : v:name → term_view
                                        | Tv_FVar : v:qname → term_view
                                        | Tv_Abs : bv:binder → body:term → term_view
                                        | Tv_App : hd:term → a:(term ∗ aqual)
                                                                     → term_view
                                     ...
```

The `term_view` type provides the "one-level-deep" structure of a term: metaprograms must call `inspect` to reveal the structure of the term, one constructor at

a time. The view exposes three kinds of variables: bound variables, Tv_BVar; named local variables Tv_Var; and top-level fully qualified names, Tv_FVar. Bound variables and local variables are distinguished since the internal abstract syntax is locally nameless. For metaprogramming, it is usually simpler to use a fully-named representation, so we provide inspect and pack functions that open and close binders appropriately to maintain this invariant. Since opening binders requires freshness, inspect has effect Tac.[2]

As generating large pieces of syntax via the view easily becomes tedious, we provide some ways of *quoting* terms in Meta-F⋆:

Static quotations A static quotation `e is just a shorthand for statically calling the F⋆ parser to convert e into the abstract syntax of F⋆ terms above. For instance, `(1 + 2) is equivalent to the following,

pack (Tv_App(pack (Tv_FVar "Prims.op_Addition"), [pack (Tv_Const (C_Int 1), ...)]))

Dynamic quotations A second form of quotation is dquote: #a:Type → a → Tac term, an effectful operation that is interpreted by F⋆'s normalizer during metaprogram evaluation. It returns the syntax of its argument at the time dquote e is evaluated. Evaluating dquote e substitutes all the free variables in e with their current values in the execution environment, suspends further evaluation, and returns the abstract syntax of the resulting term. For instance, evaluating $(\lambda x \rightarrow$ dquote $(x + 1))$ 16 produces the abstract syntax of $16 + 1$.

Anti-quotations Static quotations are useful for building big chunks of syntax concisely, but they are of limited use if we cannot combine them with existing bits of syntax. Subterms of a quotation are allowed to "escape" and be substituted by arbitrary expressions. We use the syntax `#t to denote an antiquoted t, where t must be an expression of type term in order for the quotation to be well-typed. For example, `(1 + `#e) creates syntax for an addition where one operand is the integer constant 1 and the other is the term represented by e.

Unquotation Finally, we provide an effectful operation, unquote: #a:Type → t:term → Tac a, which takes a term representation t and an expected type for it a (usually inferred from the context), and calls the F⋆ type-checker to check and elaborate the term representation into a well-typed term.

3.4 Specifying and Verifying Metaprograms

Since we model metaprograms as a particular kind of effectful program within F⋆, which is a program verifier, a natural question to ask is whether F⋆ can specify and verify metaprograms. The answer is "yes, to a degree".

To do so, we must use the WP calculus for the TAC effect: TAC-computations are given computation types of the form TAC a wp, where a is the computation's

---

[2] We also provide functions inspect_ln, pack_ln which stay in a locally-nameless representation and are thus pure, total functions.

result type and wp is a weakest-precondition transformer of type tacwp a = proofstate → (result a → prop) → prop. However, since WPs tend to not be very intuitive, we first define two variants of the TAC effect: TacH in "Hoare-style" with pre- and postconditions and Tac (which we have seen before), which only specifies the return type, but uses trivial pre- and postconditions. The requires and ensures keywords below simply aid readability of pre- and postconditions—they are identity functions.

effect TacH (a:Type) (pre : proofstate → prop) (post : proofstate → result a → prop) =
      TAC a (λ ps post' → pre ps ∧ (∀ r. post ps r ⟹ post' r))
effect Tac (a:Type) = TacH a (requires (λ _ → ⊤)) (ensures (λ _ _ → ⊤))

Previously, we only showed the simple type for the raise primitive, namely exn → Tac α. In fact, in full detail and Hoare style, its type/specification is:

raise : e:exn→ TacH α (requires (λ _ → ⊤)) (ensures (λ ps r → r == Failed (e, ps)))

expressing that the primitive has no precondition, always fails with the provided exception, and does not modify the proof state. From the specifications of the primitives, and the automatically obtained Dijkstra monad, F⋆ can already prove interesting properties about metaprograms. We show a few simple examples.

The following metaprogram is accepted by F⋆ as it can conclude, from the type of raise, that the assertion is unreachable, and hence raise_flow can have a trivial precondition (as Tac unit implies).

let raise_flow () : Tac unit = raise SomeExn; assert ⊥

For cur_goal_safe below, F⋆ verifies that (given the precondition) the pattern match is exhaustive. The postcondition is also asserting that the metaprogram always succeeds without affecting the proof state, returning some unspecified goal. Calls to cur_goal_safe must ensure that the goal list is not empty, or F⋆ will statically reject the code.

let cur_goal_safe () : TacH goal (requires (λ ps → ¬(goals_of ps == [])))
                                  (ensures (λ ps r → ∃g. r == Success g ps)) =
    match goals_of (get ()) with | g :: _ → g

Finally, the divide combinator below "splits" the goals of a proof state in two at a given index n, and focuses a different metaprogram on each. It includes a runtime check that the given n is non-negative, and raises an exception in the TAC effect otherwise. Afterwards, the call to the (pure) List.splitAt function requires that its first argument be statically known to be non-negative, a fact which F⋆ can easily prove given the specification for raise and from the effect definition, which defines the control flow.

let divide (n:int) (tl : unit → Tac α) (tr : unit → Tac β) : Tac (α ∗ β) =
    if n < 0 then raise NegativeN;
    let gsl, gsr = List.splitAt n (goals ()) in ...

This enables a style of "lightweight" verification of metaprograms, where expressive invariants about their state and control-flow can be encoded. The programmer can exploit dynamic checks (n < 0) and exceptions (raise) or static ones (preconditions), or a mixture of both, as needed.

Due to type abstraction though, the specifications of most primitives cannot provide complete detail about their behavior, and deeper specifications (such as ensuring a tactic will correctly solve a goal) cannot currently be proven, nor even stated—to do so would require, at least, an internalization of the typing judgment of F*. While this is an exciting possibility [6], we have for now only focused on verifying basic safety properties of metaprograms, which helps users detect errors early, and whose proofs the SMT can handle well. Although in principle, one can also write tactics to discharge the proof obligations of metaprograms.

4   Meta-F*, Formally

We now describe the trust assumptions for Meta-F* (§4.1) and then how we reconcile tactics within a program verifier, where the exact shape of VCs is not given, nor known a priori by the user (§4.2).

4.1   Correctness and Trusted Computing Base (TCB)

As in any proof assistant, tactics and metaprogramming would be rather useless if they allowed to "prove" invalid judgments—care must be taken to ensure soundness. We begin with a taste of the specifics of F*'s static semantics, which influence the trust model for Meta-F*, and then provide more detail on the TCB.

Proof irrelevance in F* The following two rules for introducing and eliminating refinement types are key in F*, as they form the basis of its proof irrelevance.

$$
\begin{array}{cc}
\text{T-Refine} & \text{V-Refine} \\
\dfrac{\Gamma \vdash e \,:\, t \quad \Gamma \models \phi[e/x]}{\Gamma \vdash e \,:\, x{:}t\{\phi\}} & \dfrac{\Gamma \vdash e \,:\, x{:}t\{\phi\}}{\Gamma \models \phi[e/x]}
\end{array}
$$

The $\vDash$ symbol represents F*'s *validity judgment* [1] which, at a high-level, defines a proof-irrelevant, classical, higher-order logic. These validity hypotheses are usually collected by the type-checker, and then encoded to the SMT solver in bulk. Crucially, the irrelevance of validity is what permits efficient interaction with SMT solvers, since reconstructing F* terms from SMT proofs is unneeded.

As evidenced in the rules, validity and typing are mutually recursive, and therefore Meta-F* must also construct validity derivations. In the implementation, we model these validity goals as holes with a "squash" type [8, 54], where squash $\phi = \_{:}\mathsf{unit}\{\phi\}$, i.e., a refinement of unit. Concretely, we model $\Gamma \vDash \phi$ as $\Gamma \vdash \mathsf{?u} : \mathsf{squash}\ \phi$ using a unification variable. Meta-F* does not construct deep solutions to squashed goals: if they are proven valid, the variable ?u is simply solved by the unit value '()'. At any point, any such irrelevant goal can be sent to the SMT solver. Relevant goals, on the other hand, cannot be sent to SMT.

Scripting the typing judgment A consequence of validity proofs not being materialized is that type-checking is undecidable in F⋆. For instance: does the unit value () solve the hole $\Gamma \vdash ?u : \mathsf{squash}\ \phi$? Well, only if ϕ holds—a condition which no type-checker can effectively decide. This implies that the type-checker cannot, in general, rely on proof terms to reconstruct a proof. Hence, the primitives are designed to provide access to the typing judgment of F⋆ directly, instead of building syntax for proof terms. One can think of F⋆'s type-checker as implementing one particular algorithmic heuristic of the typing and validity judgments—a heuristic which happens to work well in practice. For convenience, this default type-checking heuristic is also available to metaprograms: this is in fact precisely what the exact primitive does. Having programmatic access to the typing judgment also provides the flexibility to tweak VC generation as needed, instead of leaving it to the default behavior of F⋆. For instance, the refine_intro primitive implements T-REFINE. When applied, it produces two new goals, including that the refinement actually holds. At that point, a metaprogram can run any arbitrary tactic on it, instead of letting the F⋆ type-checker collect the obligation and send it to the SMT solver in bulk with others.

Trust There are two common approaches for the correctness of tactic engines: (1) the *de Bruijn criterion* [9], which requires constructing full proofs (or proof terms) and checking them at the end, hence reducing trust to an independent proof-checker; and (2) the LCF style, which applies backwards reasoning while constructing validation functions at every step, reducing trust to primitive, forward-style implementations of the system's inference rules.

As we wish to make use of SMT solvers within F⋆, the first approach is not easy. Reconstructing the proofs SMT solvers produce, if any, back into a proper derivation remains a significant challenge (even despite recent progress, e.g. [20, 32]). Further, the logical encoding from F⋆ to SMT, along with the solver itself, are already part of F⋆'s TCB: shielding Meta-F⋆ from them would not significantly increase safety of the combined system.

Instead, we roughly follow the LCF approach and implement F⋆'s typing rules as the basic user-facing metaprogramming actions. However, instead of implementing the rules in forward-style and using them to validate (untrusted) backwards-style tactics, we implement them directly in backwards-style. That is, they run by breaking down goals into subgoals, instead of combining proven facts into new proven facts. Using LCF style makes the primitives part of the TCB. However, given the primitives are sound, any combination of them also is, and any user-provided metaprogram must be safe due to the abstraction imposed by the Tac effect, as discussed next.

Correct evolutions of the proof state For soundness, it is imperative that tactics do not arbitrarily drop goals from the proof state, and only discharge them when they are solved, or when they can be solved by other goals tracked in the proof state. For a concrete example, consider the following program:

```
let f : int → int = _ by (intro (); exact (`42))
```

Here, Meta-F⋆ will create an initial proof state with a single goal of the form $[\emptyset \vdash ?u_1 : \mathsf{int} \to \mathsf{int}]$ and begin executing the metaprogram. When applying the intro primitive, the proof state transitions as shown below.

$$[\emptyset \vdash ?u_1 : \mathsf{int} \to \mathsf{int}] \rightsquigarrow [\mathsf{x{:}int} \vdash ?u_2 : \mathsf{int}]$$

Here, a solution to the original goal has not yet been built, since it *depends* on the solution to the goal on the right hand side. When it is solved with, say, 42, we can solve our original goal with $\lambda \mathsf{x} \to 42$. To formalize these dependencies, we say that a proof state ϕ *correctly evolves (via f) to* ψ, denoted $\phi \preceq_f \psi$, when there is a generic transformation f, called a *validation*, from solutions to all of ψ's goals into correct solutions for ϕ's goals. When ϕ has n goals and ψ has m goals, the validation f is a function from term^m into term^n. Validations may be composed, providing the transitivity of correct evolution, and if a proof state ϕ correctly evolves (in any amount of steps) into a state with no more goals, then we have fully defined solutions to all of ϕ's goals. We emphasize that validations are not constructed explicitly during the execution of metaprograms. Instead we exploit unification metavariables to instantiate the solutions automatically.

Note that validations may construct solutions for more than one goal, i.e., their codomain is not a single term. This is required in Meta-F⋆, where primitive steps may not only decompose goals into subgoals, but actually combine goals as well. Currently, the only primitive providing this behavior is join, which finds a maximal common prefix of the environment of two irrelevant goals, reverts the "extra" binders in both goals and builds their conjunction. Combining goals using join is especially useful for sending multiple goals to the SMT solver in a single call. When there are common obligations within two goals, joining them before calling the SMT solver can result in a significantly faster proof.

We check that every primitive action respects the $\preceq$ preorder. This relies on them modeling F⋆'s typing rules. For example, and unsurprisingly, the following rule for typing abstractions is what justifies the intro primitive:

T-Fun
$$\frac{\Gamma, x : t \vdash e : t'}{\Gamma \vdash \lambda(x : t).e \ : \ (x : t) \to t'}$$

Then, for the proof state evolution above, the validation function f is the (mathematical, meta-level) function taking a term of type int (the solution for $?u_2$) and building syntax for its abstraction over x. Further, the intro primitive respects the correct-evolution preorder, by the very typing rule (T-Fun) from which it is defined. In this manner, every typing rule induces a syntax-building metaprogramming step. Our primitives come from this dual interpretation of typing rules, which ensures that logical consistency is preserved.

Since the $\preceq$ relation is a preorder, and every metaprogramming primitive we provide the user evolves the proof state according $\preceq$, it is trivially the case that the final proof state returned by a (successful) computation is a correct evolution of the initial one. That means that when the metaprogram terminates, one has indeed broken down the proof obligation correctly, and is left with a (hopefully)

simpler set of obligations to fulfill. Finally, with the evolution of the proof state being always governed by the $\preceq$ preorder, Tac provides an interesting example of monotonic state [2].

4.2   Extracting Individual Assertions

As discussed, the logical context of a goal processed by a tactic is not always syntactically evident in the program. And, as shown in the List.splitAt call in divide from §3.4, some obligations crucially depend on the control-flow of the program. Hence, the proof state must crucially include these assumptions if proving the assertion is to succeed. Below, we describe how Meta-F* finds proper contexts in which to prove the assertions, including control-flow information. Notably, this process is defined over logical formulae and does not depend at all on F*'s WP calculus or VC generator: we believe it should be applicable to any VC generator.

   As seen in §2.1, the basic mechanism by which Meta-F* attaches a tactic to a specific sub-goal is assert ϕ by τ. Our encoding of this expression is built similarly to F*'s existing assert construct, which is simply sugar for a pure function _assert of type ϕ:prop $\to$ Lemma (requires ϕ) (ensures ϕ), which essentially introduces a cut in the generated VC. That is, the term (assert ϕ; e) roughly produces the verification condition $\phi \wedge (\phi \Longrightarrow \mathrm{VC}_e)$, requiring a proof of ϕ at this point, and assuming ϕ in the continuation. For Meta-F*, we aim to keep this style while allowing asserted formulae to be decorated with user-provided tactics that are tasked with proving or pre-processing them. We do this in three steps.

   First, we define the following "phantom" predicate:

let with_tactic (ϕ : prop) (τ : unit $\to$ Tac unit) $= \phi$

Here ϕ `with_tactic` τ simply associates the tactic τ with ϕ, and is equivalent to ϕ by its definition. Next, we implement the assert_by_tactic lemma, and desugar assert ϕ by τ into assert_by_tactic ϕ τ. This lemma is trivially provable by F*.

let assert_by_tactic (ϕ : prop) (τ : unit $\to$ Tac unit)
                                      : Lemma (requires (ϕ `with_tactic` τ)) (ensures ϕ) = ()

   Given this specification, the term (assert ϕ by τ; e) roughly produces the verification condition ϕ `with_tactic` $\tau \wedge (\phi \Longrightarrow \mathrm{VC}_e)$, with a tagged left sub-goal, and ϕ as an hypothesis in the right one. Importantly, F* keeps the with_tactic marker uninterpreted until the VC needs to be discharged. At that point, it may contain several subformulae annotated with tactics. For example, suppose the VC is the one given in (0) below, in which we distinguish an ambient context of variables and hypotheses Δ (initially empty when processing a VC):

(0) $\Delta \models X \Longrightarrow (\forall$ (x:t). R `with_tactic` $\tau_1 \wedge (R \Longrightarrow S))$

Finally, in order to run the τ_1 tactic on R, we first need to "split it out". To do so, we must include all logical information "visible" for τ_1, i.e., the set of premises of the implications traversed and the binders introduced by quantifiers. In F*, these two cases in fact coincide since implication is simply a (non-dependent) universal quantification. As for any program verifier, these hypotheses include

the control flow information, postconditions, and any other logical fact that is known to be valid at the program point where the corresponding assert R by τ_1 was called. All of them are collected into Δ as we traverse the term. In this case, the VC for just the sub-goal for R is:

(1) $\Delta,\ \_:X, x{:}t \models R$

Afterwards, we can remove this obligation from the original VC. We do so by replacing it with $\top$, leaving a "skeleton" VC with all remaining facts.

(2) $\Delta \models X \implies (\forall\ (x{:}t).\ \top \wedge (R \implies S))$

The validity of (1) and (2) implies that of (0). The process also recursively descends into R and S, in case there are more with_tactic markers in them. Then, the "split out" tactics are run (e.g., τ_1 on (1)) to break them down further or solve them, and all remaining goals, plus the skeleton, are dispatched to SMT.

Note that while the *obligation* to prove R, in (1), is preprocessed by the tactic τ_1, the *assumption* R for the continuation of the code, in (2), is left as-is. This is crucial for tactics such as the canonicalizer from §2.1: if the skeleton VC (2) contained an assumption for the canonicalized equality it would not help the SMT solver show the uncanonicalized postcondition.

However, not all nodes marked with with_tactic are proof obligations. Suppose X in the previous VC was given as (Y `with_tactic` τ_2). In this case, one certainly does not want to attempt to prove Y, since it is an hypothesis. While it would be *sound* to prove it and replace it by $\top$, it is useless at best, and usually irreparably affects the system. Consider asserting the tautology $(\bot\,$`with_tactic`$\,\tau) \implies \bot$; replacing it with a pair of goals for $\bot$ and $\top \implies \bot$ renders it unprovable.

Hence, F⋆ splits such obligations only in strictly-positive positions. On all others, F⋆ simply drops the with_tactic marker, e.g., by just unfolding the definition of with_tactic. For regular uses of the assert..by construct, however, all occurrences are strictly-positive. It is only when (expert) users use the with_tactic marker directly that the above discussion might become relevant.

Formally, the soundness of this whole approach is given by the following metatheorem of F⋆, justifying the splitting out of a node into a separate formula, and by the correct evolution of the proof state detailed above. The proof of Theorem 1 is straightforward, and included in §B. We expect an analogous property to hold in other verifiers as well (in particular, it holds for first-order logic).

Theorem 1. *Let E be a context with $\Gamma \vdash E : prop \Rightarrow prop$, and ϕ a squashed proposition such that $\Gamma \vdash \phi : prop$. Then the following holds:*

$$\frac{\Gamma \vDash E[\top] \quad \Gamma, \gamma(E) \vDash \phi}{\Gamma \vDash E[\phi]}$$

where $\gamma(E)$ is the set of binders E introduces. If E is strictly-positive, then the reverse implication holds as well.

5   Executing Metaprograms Efficiently

F* provides three complementary mechanisms for running metaprograms. The first two, F*'s call-by-name (CBN) interpreter and a (newly implemented) call-by-value (CBV) NbE-based evaluator, support strong reduction—henceforth we refer to these as "normalizers". In addition, we design and implement a new *native plugin* mechanism that allows both normalizers to interface with Meta-F* programs extracted to OCaml, reusing F*'s existing extraction pipeline for this purpose. Below we provide a brief overview of the three mechanisms.

5.1   CBN and CBV Strong Reductions

As described in §3.1, metaprograms, once reified, are simply F* terms of type proofstate $\to$ Div (result a). As such, they can be reduced using F*'s existing computation machinery, a CBN interpreter for strong reductions based on the Krivine abstract machine (KAM) [26, 47]. Although complete and highly configurable, F*'s KAM interpreter is slow, designed primarily for converting types during dependent type-checking and higher-order unification.

Shifting focus to long-running metaprograms, such as tactics for proofs by reflection, we implemented an NbE-based strong-reduction evaluator for F* computations. The evaluator is implemented in F* and extracted to OCaml (as is the rest of F*), thereby inheriting CBV from OCaml. It is similar to Boespflug et al.'s 2011 NbE-based strong-reduction for Coq, although we do not implement their low-level, OCaml-specific tag-elimination optimizations—nevertheless, it is already vastly more efficient than the KAM-based interpreter.

5.2   Native Plugins & Multi-language Interoperability

Since Meta-F* programs are just F* programs, they can also be extracted to OCaml and compiled to run at native speed. Having a native version means one can directly call the metaprogram from the type-checker, which is more efficient than interpreting them. However, compilation has an overhead, and it would not be convenient to compile every single invocation. Instead, Meta-F* enables users to compile individual metaprograms (usually those which are expected to be computation-intensive, e.g. canon_semiring), and dynamically link their native versions into F*. The normalizers are extended with such native versions, which are invoked the same way as Meta-F* primitives. Users can then compile metaprograms as desired, while still quickly scripting their higher-level logic in the interpreter.

This requires (for higher-order metaprograms) a form of multi-language interoperability, converting between representations of terms used in the normalizers and in native code. We designed a small multi-language calculus, with ML-style polymorphism, to model the interaction between normalizers and native plugins, with conversions between the term representations. We outline it in §C.

Beyond the notable efficiency gains of running compiled code vs. interpreting it, native metaprograms also require fewer embeddings. Once compiled, metaprograms work over the internal, *concrete* types for proofstate, term, etc., instead of over their F* representations (though still treating them abstractly). Hence, compiled metaprograms can call primitives without needing to embed their arguments or unembed their results. Further, they can call each other directly as well. Indeed, operationally there is little difference between a primitive and a compiled metaprogram used as a plugin.

Native plugins, however, are not a replacement for the normalizers, for several reasons. First, there is an overhead in compilation which might not be justified by the execution speed-up. Second, extraction to OCaml erases types and proofs. As a result, the F* *interface* of the native plugins can only contain types that can also be expressed in OCaml, thereby excluding full-dependent types—internally, however, they can be dependently typed. Third, being OCaml programs, native plugins do not support reducing open terms, which is often required for dependent type-checking. However, when the programs treat their open arguments parametrically, relying on parametric polymorphism, the normalizers can pass such arguments *as-is*, thereby recovering open reductions in some cases. This allows us to use native datastructure implementations (e.g. List), which is much faster than using the normalizers, even for open terms. We discuss this briefly in §C.

6   Experimental evaluation

We now present an experimental evaluation of Meta-F*. First, we provide benchmarks comparing our reflective canonicalizer from 2.1 to calling the SMT solver directly without any canonicalization. Then, we return to the parsers and serializers from §2.3 and show how, for VCs that arise, a domain-specific tactic is much more tractable than a SMT-only proof.

6.1   A Reflective Tactic for Partial Canonicalization

In 2.1, we have described the canon_semiring tactic that rewrites semiring expressions into sums of products. Differently from existing reflective tactics, we do not canonicalize expressions fully. Instead, we turn them into a shape that is close enough to their normal form for the SMT solver to be able to conclude their equality. This approach pays off well in practice. The table below compares the success rates of proofs for the poly_multiply lemma from 2.1. To test the robustness of each alternative, we run the tests 150 times while varying the SMT solver's random seed. For the smt$i$x we ask the solver to prove the lemma without any help from tactics, where i represents the resource limit (`rlimit`) multiplier given to the solver. This `rlimit` is memory-allocation based and independent of the particular system or current load. For the interp and native lines we make use of the canon_semiring tactic, running it using F*'s KAM normalizer and as a compiled plugin respectively—both run with the `rlimit` multiplier set to 1.

For each setup, we display the success rate of verification and the average (wall) time in seconds together with the standard deviation. The total verification time is then split into SMT solving and tactic evaluation.

The experiments clearly demonstrate a significant increase in robustness: proofs exploiting canonicalization have not been observed to fail, even with consistently lower `rlimit`s. We recall that the tactic variants still leave goals

	Rate	Total	SMT	Meta-F*
smt1x	0%	3.966 ± 0.037	3.484 ± 0.034	
smt2x	1%	4.287 ± 0.046	3.805 ± 0.044	
smt3x	7%	4.582 ± 0.065	4.101 ± 0.063	
smt6x	10%	6.043 ± 0.161	5.489 ± 0.157	
smt12x	24%	7.154 ± 0.252	6.641 ± 0.250	
smt24x	40%	10.144 ± 0.647	9.649 ± 0.647	
smt48x	53%	20.393 ± 2.602	19.844 ± 2.601	
interp	100%	3.973 ± 0.051	2.008 ± 0.027	1.440 ± 0.020
native	100%	2.787 ± 0.031	2.011 ± 0.023	0.297 ± 0.005

for SMT solving, namely, the skeleton for the original VC and the canonicalized equality left by the tactic, easily dischargeable by the SMT solver through much more well-behaved linear reasoning. The last column shows that native compilation speeds up this tactic's execution by about 5x.

6.2   Combining SMT and Tactics for the Parser Generator

In §2.3, we presented a library of parser and serializer combinators and a metaprogramming approach to automate the construction of verified, mutually inverse, low-level parsers and formatter from a description of a given type. Metaprogramming helps inasmuch as it relieves the programmer from directly programming against our combinator library. Further, tactics are used to process and discharge proof obligations that arise when using our library.

We present three strategies for discharging auxiliary proofs, including proofs of bijectivity, that arise when constructing parsers and formatters for enumerated types. First, we used F*'s default strategy to present all of these proofs directly to the SMT solver. Second, we programmed a ∼100 line tactic to discharge these proofs without relying on the SMT solver at all. Finally, we used a hybrid approach where a simple, 5-line tactic is used to prune the context of the proof removing redundant facts before presenting the resulting goals to the SMT solver.

The table alongside shows the total time in seconds for verifying metaprogrammed low-level parsers and formatters for enumerations of different sizes. In short, the hybrid approach scales the best;

#cases	SMT only	Tactic only	Hybrid
4	178	17.3	6.6
7	468	38.3	9.8
10	690	63.0	19.4

the tactic-only approach is somewhat slower; while the SMT-only approach scales poorly and is an order of magnitude slower. Our hybrid approach is very simple. With some more work, a more sophisticated hybrid strategy could be more performant still, relying on tactic-based normalization proofs for fragments of the VC best handled computationally (where the SMT solver spends most of its time), while using SMT only for integer arithmetic, congruence closure etc.

However, with Meta-F*'s ability to manipulate proof contexts programmatically, our simple context-pruning tactic provides a big payoff at a small cost.

7   Related Work

Many SMT-based program verifiers [10, 11, 22, 35, 49], lacking a way to dynamically inspect the proof state, depend on user hints to succeed. This style of proving has been used in Dafny [3, 5, 48], Liquid Haskell [61, 62] and F* itself [59]. In contrast, Why3 [34] allows VCs to be discharged using ITPs such as Coq, Isabelle/HOL, and PVS, but this requires using these external tools to discharged machine-generated proof goals after an additional embedding. In recent concurrent work, support for *effectful* reflection proofs was added to Why3 [51], and it would be interesting to investigate if this could also be done in Meta-F*. Grov and Tumas [40] present Tacny, a tactic framework for Dafny, which is, however, limited in that it only transforms source code, with the program verifier unchanged. In contrast, Meta-F* combines the benefits of an SMT-based program verifier and those of tactic proofs within a single language.

Moving away from SMT-based verifiers, ITPs have long relied on separate languages for proof scripting, starting with Edinburgh LCF [38] and ML, and continuing with HOL, Isabelle and Coq, which are either extensible via ML, or have dedicated tactic languages [6, 30, 57, 64]. Meta-F* builds instead on a recent idea in the space of dependently typed ITPs [24, 31, 43, 65] of reusing the object-language as the meta-language. This idea first appeared in Mtac, a Coq-based tactics framework for Coq [43, 65], and has many generic benefits including reusing the standard library, IDE support, and type checker of the proof assistant. Mtac can additionally check the partial correctness of tactics, which is also sometimes possible in Meta-F* but still rather limited (§3.4). Meta-F*'s design is instead more closely inspired by the metaprogramming frameworks of Idris [24] and Lean [31], which provide a deep embedding of terms that metaprograms can inspect and construct at will without dependent types getting in the way. However, F*'s effects, its weakest precondition calculus, and its use of SMT solvers distinguish Meta-F* from these other frameworks, presenting both challenges and opportunities, as discussed in this paper.

Finally, ITPs are seeing increasing use of "hammers" such as Sledgehammer [17, 18, 55] in Isabelle/HOL, and similar tools for HOL Light and HOL4 [44], Mizar [45], and recently even Coq [28]. While hammers are similar to Meta-F* tactics in as much they allow combining interactive and automated theorem proving, there are also significant differences. Hammers encode ITP goals to an automated prover, *heuristically* selecting the provided context, and, if a proof is found, attempt to reconstruct it internally in the ITP. Meta-F* on the other hand uses F*'s existing SMT encoding, which presents the (by default) full program context, and does not require to reconstruct a proof on success, since F* natively supports SMT proofs. F*'s SMT encoding is targeted at soundness and at efficient, scalable, and reproducible SMT solver behavior, while hammers often give up many of these and then regain them by proof reconstruction.

8   Conclusions

A key challenge in program verification is to balance tradeoffs among automation and expressiveness. Whereas tactic-based interactive provers support highly expressive logics, the tactic author is responsible for all the automation. Conversely, SMT-based program verifiers provide good, scalable automation for comparatively weaker logics, but offer little recourse when verification fails. A design that allows picking the right tool, at the granularity of each verification sub-task, is a worthy area that we feel has been underexplored. Meta-F* aims to fill this gap: by using hand-written tactics along with SMT-automation, we have written proofs that were previously impractical in F*, and (to the best of our knowledge) in other SMT-based program verifiers.

A   Metaprogramming verified parsers and serializers

We consider parser and serializer specifications as ghost code that "operates" on finite sequences of bytes of unbounded lengths. For a given type t, a serializer for t marshals any data of type t into a finite sequence of bytes; and a parser for t reads from a sequence of bytes, checks whether it corresponds to a valid data of type t, and if so, returns that parsed data and the number of bytes consumed.

   The serializer specification always succeeds; the parser specification succeeds if and only if the input data is valid with respect to the parser. On those specifications, we aim to prove that the parser and the serializer are partial inverses of each other. We encode these requirements as refinements on the types of parser and serializer specifications:

```
type byte = FStar.UInt8.t
type parser t = (input: seq byte) → GTot (option (t ∗ (x: nat { x < length input } )))
let serialize_then_parse_eq_id #t (p: parser t) (s: (t → GTot (seq byte))) =
  (∀ (x: t) . let y = s x in p y == Some (x, length y))
let parse_then_serialize_eq_id #t (p: parser t) (s: (t → GTot (seq byte))) =
  (∀ (x: seq byte) . match p x with | Some (y, len) → s y == slice x 0 len | _ → ⊤) })
type serializer #t (p: parser t) = (s: (t → GTot (seq byte))
  { serialize_then_parse_eq_id p s ∧ parse_then_serialize_eq_id p s })
```

   Whereas those specifications are ghost code, we would like to generate implementations that can be extracted to C. To this end, we generate *stateful* implementations written in the Low* set of F* [56], operating on buffers, which are Low* mutable data structures representing C arrays. There, instead of a sequence of bytes, the parser implementation is given an input buffer and its length; and the serializer implementation is given an output buffer (along with its length) onto which it is to serialize the data. This means that, given a piece of data to serialize and a destination buffer, a serializer implementation can succeed only if the serialized data fits into the destination buffer; if so, then the serializer will return the number of bytes written.

   We bake the correctness of the implementations with respect to their specifications at the level of their types:

```
type buffer8 = LowStar.Buffer.buffer FStar.UInt8.t
type u32 = FStar.UInt32.t

let parser32_postcond #t (p: parser t) (input: buffer8) (l: u32 {l == len input})
  (h: mem) (res: option (t ∗ u32)) (h': mem) =
  modifies loc_none h h' ∧ (∗ memory safety ∗)
  match p (as_seq h input), res with (∗ functional correctness ∗)
  | Some (x, ln), Some (x', ln') → x' == x ∧ FStar.UInt32.v ln' == ln
  | None, None → ⊤| _ → ⊥

type parser32 #t (p: parser t) = (input: buffer8) → (l: u32 { l == len input } )
                                          → Stack (option (t ∗ u32))
  (requires (λ h → live h input)) (ensures (parser32_postcond p input))
```

Lemma poly_multiply
  (n p r h r0 r1 h0 h1 h2 s1 d0 d1 d2 hh : Z)
  (H: 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 mod 4 = 0 ∧
          d0 = h0 ∗ r0 + h1 ∗ s1 ∧ d1 = h0 ∗ r1 + h1 ∗ r0 + h2 ∗ s1 ∧
          d2 = h2 ∗ r0 ∧ hh = d2 ∗ (n ∗ n) + d1 ∗ n + d0)
 : ((h ∗ r) mod p = hh mod p).
Proof.
  repeat match goal with H : ?A ∧ ?B | _ => destruct H end.
  subst.
  match goal with | ?A mod p = ?B mod p => change (eqm p A B) end.
  match goal with K : r1 mod 4 = 0 | _ =>
   apply Z_div_exact_full₂ in K; try omega;
   revert K;
   generalize (r1 / 4);
   intro u;
   intros; subst
  end.
  pose (b := (h2 ∗ n + h1) ∗ u).
  generalize (Zopp_eqm p).
  generalize (Zplus_eqm p).
  generalize (Zminus_eqm p).
  generalize (Zmult_eqm p).
  generalize (eqm_setoid p).
  set (e := eqm p).
  intros.
  match goal with | e _ ?R =>
    generalize (modulo_addition_lemma R p b)
  end.
  fold e. intro K. setoid_rewrite < K. clear K.
  assert (p = 4 ∗ (n ∗ n) 5) as J by omega.
  replace (b ∗ p) with (b ∗ (4 ∗ (n ∗ n) 5)) by congruence.
  unfold b.
  apply eq_implies_eqm.
  ring.
Qed.

Fig. 1: Coq proof of poly_multiply

```
let serializer32_postcond #t (p: parser t) (s: serializer p) (output: buffer8)
  (l: u32 {l == len output}) (x: t) (h: mem) (res: option u32) (h': mem) =
  live h output ∧ live h' output ∧ modifies (loc_buffer b) h h' ∧ (∗ memory safety ∗)
  let ln = length (s x) in
  match res with (∗ functional correctness ∗)
  | Some ln' →
    FStar.UInt32.v ln' == ln ∧ ln ≤ FStar.UInt32.v l ∧
    let b' = sub b 0ul ln' in modifies (loc_buffer b') h h' ∧ as_seq h' b' == s x
  | None → ln > FStar.UInt32.v l

type serializer32 #t (#p: parser t) (s: serializer p) =
  (output: buffer8) → (l: u32 { l == len output } ) → (x: t) → Stack (option u32)
  (requires (λ h → live h output)) (ensures (serializer32_postcond s output l x))
```

Generating the specification from a F⋆ datatype Instead of mandating the
user to write the parser and serializer specification themselves, we write a Meta-
F⋆ tactic, `gen_specs`, to generate both the parser and the serializer specifications
directly from the F⋆ type t given by the user. These tactics operate by syntax
inspection on t itself, and generates the parser and serializer specifications using
our combinators.

Contrary to Coq's Ltac, Meta-F⋆ tactics can also inspect definitions of induc-
tive types. From there, we made `gen_specs` also generate a parser and serializer
specification out of an enumeration type defined as a F⋆ inductive type whose
constructors have no arguments: for such a type with $n \leq 256$ constructors, the
corresponding parser will associate a unique 8-bit integer from 0 to $n-1$ to each
constructor. For instance, if t is defined as a F⋆ enumeration type:

As before, by virtue of the correctness of serializers being baked in their
type, the correctness of the serializer generated by `gen_specs` does not appear
as such as a verification condition when F⋆ is type-checking the generated term.
In the example above, the only proof obligations generated are those to correctly
type-check the rewriting functions themselves passed to `serialize_synth` (e.g.
enum constructors are rewritten to integers less than 4), and the precondition
of `serialize_synth` (i.e. the fact that the two rewriting functions are inverse of
each other.) These can be discharged by the SMT solver, but they can also be
discharged automatically by our tactics directly by case analysis on the inductive
type, or by bounded integer enumeration.

Binders For instance, we specified `and_then`, a combinator for dependent pars-
ing:[3]

```
let and_then #t1 (p1: parser t1) #t2 (p2: (t1 → Tot (parser t2))) : Tot (parser t2) =
  λinput → match p1 input with | None → None | Some (x1, cons1) →
    let input2 = slice input cons1 (length input) in
    match p2 x1 input2 with
```

---

[3] This parsing combinator is based on yet unpublished work by Tej Chajed. Building
serializers for parsers derived from `and_then` is explicitly out of the scope of this
paper.

| None → None | Some (x2, cons2) → Some (x2, cons1 + cons2)

Then, we implemented it in Low⋆ (`parse_and_then_impl`), and manually proved it correct. From there, the user can write a simple parser specification for a parser that parses an integer encoded in one or two bytes depending on its value (where `parse_ret x` is a parser that returns `x` without reading its byte input:)

```
let example : parser_spec FStar.Int16.t =
  and_then parse_u8
        (λ lo → if lo <^ 128uy then parse_ret (cast_u8_to_i16 lo)
                  else parse_synth parse_u8 (λ hi → cast_u8_to_i16 (lo %^ 128uy)
                                                      +^ cast_u8_to_i16 hi))
```

Then, if the user writes:

```
let example_impl : parser_impl example = _ by gen_parser_impl
```

then `gen_parser32` inspects the shape of the *goal*, which is `parser32 example`, and so generates the following Low⋆ implementation:

```
parse32_and_then parse32_u8 (λ lo →
  parse32_ifthenelse (lo <^ 128uy) (λ _ → parse32_ret (cast_u8_to_i16 lo)) (λ _ →
parse32_synth parse32_u8 (λ hi → cast_u8_to_i16 (lo %^ 128uy) +^ cast_u8_to_i16 hi)))
```

from where KreMLin [56] then inlines the corresponding implementation combinators to produce C code.

First, `gen_parser32` inspects the shape of the goal to determine the type of the data to be parsed and the parser specification; then, it calls a recursive tactic `gen_parser32'` that inspects the shape of the parser specification and actually builds the implementation.

```
module T = FStar.Tactics
let rec gen_parser32' (env: T.env) (t: T.term) (p: T.term) : T.Tac T.term =
  let (hd, tl) = app_head_tail p in
  if hd `T.term_eq` (`(parse_ret)) then T.mk_app (`(parse32_ret)) tl else
  if hd `T.term_eq` (`(parse_u8)) then (`(parse32_u8)) else
  if hd `T.term_eq` (`(and_then)) then match tl with
  | [(t, _); (p, _); (t', _); (p', _)] →
    begin match T.inspect p' with
    | T.Tv_Abs bx body →
      let p32 = gen_parser32' env k t p in
      let env' = T.push_binder env bx in
      let body' = gen_parser32' env' k' t' body in
      let p32' = T.pack (T.Tv_Abs bx body') in
      (`(parse32_and_then #(`#t) #(`#p) (`#p32) (`#t') (`#p') (`#p32')))
    | _ → ...
    end
  | _ → ...
  else
  ...
```

```
let gen_parser32 () : T.Tac unit =
  let (hd, tl) = app_head_tail (T.cur_goal ()) in
  if hd `T.term_eq` (`parser32)
  then match tl with
  | [(t, _); (p, _)] →
    let env = T.cur_env () in
    let p32 = gen_parser32' env t p in
    T.exact_guard p32;
  | _ → ...
```

When `gen_parser32` (resp. `gen_serializer32`) builds the parser (resp. se-rializer) implementation, it may generate verification conditions due to specific preconditions required by certain combinators (such as the precondition of the `serialize_synth` rewriting serializer combinator: the rewriting functions being inverse of each other). Contrary to Coq, there are no proof objects, so those preconditions need to be proven again even if they had been proven earlier by the user at the specification level. Nevertheless, those preconditions can be auto-matically solved by our tactics, or they can directly be sent to the SMT solver.

However, by virtue of the correctness of the implementation combinators be-ing baked in their type, typechecking the resulting term generated by `gen_parser32` triggers no verification condition other than those preconditions required by the specific combinators, and unification constraints, the latter solved by reflexivity. In particular, the memory safety and correctness of the implementations gener-ated by `gen_parser32` do not appear as such as verification conditions when F* is type-checking the terms generated by `gen_parser32`.

A.1   Verified meta-programming of program transformations

`compile_bind` expects, in `bind f1 f2`, that `f2` be an abstraction, and so needs to manipulate binders and recursively compile `f1` and the body of `f2`. This will, in the latter case, make some bound variables appear free in terms being compiled.

If the head term t_0 is a free variable not corresponding to any combinator, then `compile_fvar` unfolds it, compiles the unfolded term, and inserts a coercion.[4] In most cases, a free variable to be unfolded will correspond to a function defini-tion, so that unfolding will yield a β-redex, which `compile_fvar` reduces by calling a normalization tactic that we provide as part of Meta-F*. This needs an *en-vironment* (e: env) tracking the bound variables encountered by nested calls to `compile_bind`.

We show below the implementation of `compile_bind`, which is representative of most features of Meta-F* that we use for our `compile` tactic.

(∗ extract the head symbol and the arguments of an application ∗)
```
let rec app_head_rev_tail (t: term) : Tac (term ∗ list argv) =
```

---

[4] We defined an explicit coercion combinator, `coerce_sz`, such that `coerce_sz f1 f1_sz f2` is well-typed and is a `m_sz f2` as soon as `f1 () == f2 ()` as m specifications.

```
   match inspect t with | Tv_App u v → let (x, l) = app_head_rev_tail u in (x, v :: l) | _ →
(t, [])
let app_head_tail (t: term) : Tac (term * list argv) = let (x, l) = app_head_rev_tail t in (x, rev l)

let compile_bind (e: env) (ty: term) (t: term) (compile: env → term → term → Tac term) : Tac term =
   let (t₀, ar) = app_head_tail t in
   guard (t₀ = quote bind); (* fail if false *)
   match ar with
   | [ (τ₁, _); (τ₂, _); (f1, _); (f2, _); ] →
     begin match inspect f2 with
     | Tv_Abs v f2_body →
       let f1' = compile e τ₁ f1 in
       let e' = push_binder e v in
       let f2_body' = compile e' τ₂ f2_body in
       let f2' = pack (Tv_Abs v f2_body') in
       (`(bind_sz #(`#τ₁) #(`#τ₂) (`#f1) (`#f1') (`#f2) (`#f2')))
     | _ → fail "compile_bind:␣Not␣an␣abstraction"
     end
   | _ → fail "compile_bind:␣4␣arguments␣expected"
```

Putting everything together Finally, the following wrapper combinator Φ puts everything together, taking a specification f and its size and buffer implementations, and yielding a stateful function, fitting in the Low* subset of F* amenable to extraction to C, that first computes the expected size of the output, then allocates one single buffer using the C malloc, then uses it to write the output of f:

```
let Φ τ (f: m τ) (f_sz: m_sz f) (f_st: m_st f) : ST (option (τ * buffer U8.t)) (requires (λ _ →
⊤))
   (ensures (λ h res h' → let (r, log) = f () in
     if length log > 2³¹ − 1 then res == None else
     Some? res ∧ (let (Some (r', b)) = res in r == b ∧ fresh b h h' ∧ as_seq h b == log)
   )) = match f_sz with | None → None | Some (_, sz') →
       let b = rcreate_mm root 42uy sz in (* then allocate a fresh buffer *)
       let (r, _) = f_st b in (* then write the output into the buffer *)
       Some (r, b)
```

Related Work Our approach of generating verified low-level formatters is related to Amin and Rompf's [4] work on LMS-Verify. They use metaprogramming facilities in Scala to generate C code together with proof annotations to be checked by Frama-C [27], an SMT-based C verifier. In contrast, we generate correct-by-construction imperative Low* code, which is verified by F* before being translated to C by the KreMLin compiler. LMS-Verify has also been used to generate efficient and safe HTTP parsers, a larger-scale effort than the network message formatters than we have currently done.

B   Proof of Theorem 1

This proof is based on the formal definition of EMF*, a recent formalization of
an F* subset [1]. We use the same notation and rule names from there, but the
proof is nevertheless quite direct with just minimum familiarity with EMF*. We
use E to represent F* contexts:

$$
\begin{aligned}
E = {} & \cdot \\
& |\; tE && |\; Et \\
& |\; \lambda(x : t).E && |\; \lambda(x : E).t \\
& |\; (x : t) \rightarrow E && |\; (x : E) \rightarrow t \\
& |\; x : t\{E\} && |\; x : E\{t\} \\
& \; \ldots
\end{aligned}
$$

Contexts present a set of bound variables to their hole. We denote those
variables by $\gamma(E)$:

$$
\begin{aligned}
\gamma(\cdot) &= \emptyset \\
\gamma(\lambda(x : t).E) &= \{x\} \cup \gamma(E) \\
\gamma(\lambda(x : E).t) &= \gamma(E) \\
\ldots
\end{aligned}
$$

We say a context E has type $t_1 \Rightarrow t_2$ for Γ (noted by $\Gamma \vdash E : t_1 \Rightarrow t_2$) when
for all e such that $\Gamma, \gamma(E) \vdash e : t_1$ we also have $\Gamma \vdash E[e] : t_2$.

B.1   Soundness of splitting the proof obligation

The proof follows mainly from the following lemma:

Lemma 1.
$$
\frac{\Gamma, \gamma(E) \vDash t_1 = t_2}{\Gamma \vDash E[t_1] = E[t_2]}
$$

Proof. The proof follows from applying functional extensionality and using a
carefully crafted abstraction.

Let $S = \lambda f.E[f(\gamma(E))]$ (where $f(\gamma(E))$ represents f applied to every variable
in $\gamma(E)$, sequentially). Also, $\lambda\gamma(E).t$ represents abstracting E over the variables
in $\gamma(E)$, and similarly for $\forall\gamma(E).t$.

Then,

$$
\cfrac{
  \cfrac{
    \cfrac{
      \cfrac{\Gamma, \gamma(E) \vDash t_1 = t_2}{\Gamma \vDash \forall\gamma(E).t_1 = t_2} \text{ V-}\forall\text{i}
    }{\Gamma \vDash (\lambda\gamma(E).t_1) = (\lambda\gamma(E).t_2)} \text{ V-Ext}
  }{\Gamma \vDash S(\lambda\gamma(E).t_1) = S(\lambda\gamma(E).t_2)} \text{ V-EqP}
}{\Gamma \vDash E[t_1] = E[t_2]} \text{ Reduction}
$$

Note that the particular set of contexts E does not influence the proof.

Having such lemma, the theorem is proven as follows:

$$\cfrac{\Gamma \vDash E[\top] \quad \cfrac{\cfrac{\cfrac{\Gamma, \gamma(E) \vDash \phi}{\Gamma, \gamma(E) \vDash \phi \iff \top} \text{ Trivial}}{\Gamma, \gamma(E) \vDash \phi = \top} \text{ PropExt}}{\Gamma \vDash E[\phi] = E[\top]} \text{ Lemma}}{\Gamma \vDash E[\phi]}$$

B.2   Partial completeness of splitting the proof obligation

While the previous theorem allows to soundly split any subformulae within a VC, we have seen that some of them constraint the system. Here we prove that for a particular set of contexts, splitting does not make the judgments stronger, and via Theorem 1, then they are equivalent.

We define a particular shape of "positive" contexts P. We don't attempt to follow all of EMF*'s syntax, just the parts we commonly see as VCs in practice.

$$\begin{aligned} P = \ &\cdot \\ | \ &\phi \wedge P \quad \quad | \ P \wedge \phi \\ | \ &\forall(x:t).P \end{aligned}$$

In EMF*, an implication $a \Rightarrow b$ is just sugar for $\forall(\_ : a).b$, so they are considered as well.

Theorem 2. *Take Γ, E, and ϕ such that: (1) ϕ is squashed, (2) $\Gamma \vdash E : prop \to prop$ and (3) $\Gamma \vdash \phi : prop$. Then the following holds*

$$\cfrac{\Gamma \vDash P[\phi]}{\Gamma, \gamma(P) \vDash \phi \quad \quad \Gamma \vDash P[\top]}$$

Proof. By induction on the structure of P, keeping Γ and ϕ universally quantified.

- For $P = \cdot$, trivial
- For $P = \psi \wedge P'$, our hypothesis is $\Gamma \vDash \psi \wedge P'[\phi]$. Hence, $\Gamma \vDash \psi$ and $\Gamma \vDash P'[\phi]$. By the IH, we get $\Gamma, \gamma(P') \vDash \phi$ (note that $\gamma(P) = \gamma(P')$) and $\Gamma \vDash P'[\top]$. By weakening and conjunction, we can prove $\Gamma, \gamma(P') \vDash \psi \wedge \phi$ and $\Gamma \vDash \psi \wedge P'[\top]$, and conclude.
- For $P = P' \wedge \psi$, the reasoning is analogous.
- For $P = \forall(x:t).P'$, our hypothesis is $\Gamma \vDash \forall(x:t).P'[\phi]$. Hence, we get $\Gamma, x : t \vDash P'[\phi]$ by eliminating the quantifier. From the IH, we get $\Gamma, x : t, \gamma(P') \vDash \phi$ and $\Gamma, x : t \vDash E[\top]$. From this last one, we can use V-$\forall$i to obtain $\Gamma \vDash \forall(x:t).E[\top]$, and conclude (noting that $\gamma(P) = x : t, \gamma(P')$).

C   Modelling native plugins

This appendix contains the formal definitions of our multi-language interoperability model. We formalize the semantics of source and target language as well as the translation between the two. We convey the essential ideas in the text below, and present full details in Figures 4–18 that follow the discussion below.

Part 1: Modeling Native Plugins with Simple Types

Reflecting our requirement that plugins are only supported at ML-typeable interfaces, we start with a source language that is an intrinsically typed (Church style), standard, simply typed lambda calculus with pairs and sums. Later in the section, we will add ML-style rank 1 polymorphism to it. Conversely, reflecting the type-less native representation of compiled OCaml code, our target language is the untyped lamdba calculus. For clarity, we markup the the syntax using colors, using *blue* for the source language and *red* for the target. Aside from the standard forms, we have two new expression forms to account for the "alien" expressions of one language appearing in the other: $[e]_\tau$, which *embeds* a τ-typed source language expression into the target language, and $\{e\}_\tau$, which *unembeds* a target language expression to the source language at type τ.

We explain the reductions through a small example. Consider a source language term $(\lambda x{:}\mathbb{Z}.x)\ 0$, where we want to compile the identity function to native code and then perform the application. The first step is to translate the function to the target, by systematically erasing the types, and unembed it in the source. In our example, the source language term then becomes $\{\lambda x.x\}_{\mathbb{Z}\to\mathbb{Z}}\ 0$.

Source semantics and unembedding (Figure 2) At a beta-reduction step (e.g. rule S-APP below), the source semantics uses a meta-function $force(v)$ to examine the head of the application v. In addition to the usual values, the term $\{v\}_\tau$ is a source value *iff* v is a target value. If the head is such a value, $force()$ invokes the unembedding coercion $\mathcal{C}_\tau(v)$ to coerce it to a suitable source value. For structural types, this amounts to lazily coercing the head constructor of the term (see below). For function types, a source-level closure is allocated, which first *embeds* its argument in the target, reduces a target application (using rule S-ALIEN), and then *unembeds* the result back in the source.

In our example, reduction proceeds by rule S-APP by first applying the unembedding coercion $\mathcal{C}_{\mathbb{Z}\to\mathbb{Z}}()$ to $\lambda x.x$ to get $\lambda x{:}\mathbb{Z}.\{(\lambda x.x)\ [x]_\mathbb{Z}\}_\mathbb{Z}$, and then applying the substitution to get $\{(\lambda x.x)\ [0]_\mathbb{Z}\}_\mathbb{Z}$.

Target semantics and embeddings (Figure 3) The target semantics is standard CBV. The only new rule is T-ALIEN, which first reduces the foreign source term using the source semantics, and then proceeds to *embed* the resulting value into the target language, using the embedding coercion $\mathcal{C}_\tau(v)$, as shown below. In our implementation, this rule is realized via a callback to the F⋆'s normalizer.

$$\frac{e \to^* v}{\{e\}_\tau \to \{v\}_\tau} \text{ S-Alien} \qquad\qquad \frac{force(v) = \lambda x{:}\tau.e}{v\ e_2 \to e[e_2/x]} \text{ S-App}$$

$$force(v) = \begin{cases} \mathcal{C}_\tau(v) & \text{if } v = \{v\}_\tau \\ v & \text{otherwise} \end{cases}$$

$$\mathcal{C}_{\text{unit}}(()) = ()$$
$$\mathcal{C}_{\tau_1 \times \tau_2}((v_1,v_2)) = (\{v_1\}_{\tau_1},\{v_2\}_{\tau_2})$$
$$\mathcal{C}_{\tau_1 \to \tau_2}(e) = \lambda x{:}\tau_1.\{e\ [x]_{\tau_1}\}_{\tau_2}$$

Fig. 2: Source semantics

$$\frac{e \to^* v}{[e]_\tau \to \mathcal{C}_\tau(v)} \text{ T-Alien} \qquad\qquad \frac{}{(\lambda x.e)\ v \to e[v/x]} \text{ T-App}$$

$$\mathcal{C}_{\tau_1 \to \tau_2}(e) = \lambda x.[e\ \{x\}_{\tau_1}]_{\tau_2}$$
$$\mathcal{C}_{\tau_1 \times \tau_2}((e_1,e_2)) = ([e_1]_{\tau_1},[e_2]_{\tau_2})$$
$$\mathcal{C}_{\tau'}(\{v\}_\tau) = v \text{ if } \tau \equiv \tau'$$

Fig. 3: Target semantics

The last case in the definition of the coercion cancels superfluous embeddings, in case the term inside is an unembedding. Continuing with our example, $\{(\lambda x.x)\ [0]_\mathbb{Z}\}_\mathbb{Z}$ reduces using S-Alien, which reduces the application in its premise by first reducing $[0]_\mathbb{Z}$ to 0 (using rule T-Alien, with $\mathcal{C}_\mathbb{Z}(0) = 0$), and then using rule T-App to get 0. The rule S-Alien then returns $\{0\}_\mathbb{Z}$ to the $F^\star$ normalizer, which applies the unembedding coercion to get 0.

Part 2: Adding ML-style Polymorphism

The key observation in adding ML-style polymorphism is as follows. Since we only consider target terms compiled from well-typed source terms, by virtue of parametricity, a compiled polymorphic function must treat its argument abstractly. As a result, the normalizers can pass such arguments *as-is* without applying any embedding coercions—passing an open term is simply a subcase without further difficulty. Thus, we can leverage polymorphism to support such limited but useful open reductions.

In the formal model, we add an *opaque* construct to the target language, written as $\langle e \rangle$, which denotes an embedding of e at some abstract type A (as mentioned above, coercion to opaque is a no-op in the implementation). The coercion functions are extended with rules to introduce and eliminate opaque

terms (the δ superscript carries type substitutions as a technical device), e.g.:

$$\mathcal{C}_A^\delta(e) = \langle e \rangle \qquad \mathcal{C}_A^\delta(\langle e \rangle) = e \qquad \mathcal{C}_{\tau_1 \times \tau_2}^\delta((e_1, e_2)) = (\mathcal{P}_{\tau_1}^\delta(e_1), \mathcal{P}_{\tau_2}^\delta(e_2))$$

where $\mathcal{P}_\tau^\delta(e)$ is $\langle e \rangle$ when $\tau = A$, or a usual type-directed embedding otherwise. The source semantics is extended with a rule for type applications, while target semantics, being untyped, remains as-is.

Consider an application of the polymorphic identity function to a variable y: $(\Lambda A.\lambda x{:}A.x)\ \mathbb{Z}\ y$. As before, we would like to compile the function to native code and then reduce the applications—except this time it is an open term. Translation of the function and its embedding in the source yields $\{\lambda x.x\}_{\forall A.A \to A}^{\cdot}\ \mathbb{Z}\ y$. We now apply the unembedding coercion $\mathcal{C}_{\forall A.A \to A}^{\cdot}(\lambda x.x)$ which returns a source type abstraction $\Lambda A.\mathcal{C}_{A \to A}^{\cdot}(\lambda x.x)$. Reducing the arrow coercion next, we get $\Lambda A.\lambda x{:}A.\{(\lambda x.x)\ \mathcal{P}_A^{\cdot}(x)\}_A^{\cdot}$, where $\mathcal{P}_A^{\cdot}(x) = \langle x \rangle$, as mentioned above. This is the key idea—when the original application reaches the beta-reduction with argument y (after the type application is reduced), y is simply substituted in this opaque construct, which is then returned back to the F* normalizer as $\{\langle y \rangle\}_A^{[A \mapsto \mathbb{Z}]}$. The normalizer can then apply the identity unembedding and get y.

This ability to reduce open terms relying on polymorphism allows us to evaluate expressions like $\{\textsf{List.map}\}_{\forall A.\forall B.(A \to B) \to \textsf{list}A \to \textsf{list}B}^{\cdot}\ \mathbb{Z}\ \mathbb{Z}\ \textsf{f}\ [0; ...; 10^6]$, with f a free variable, using native datastructure implementations (List in this case), which is much faster than using the normalizers.

Part 3: Full Details on our Multi-language Interoperability Model

$$
\begin{array}{lll}
\text{Types} & \tau & ::= \; \texttt{unit} \mid A \mid \tau_1 {\to} \tau_2 \mid \tau_1 {\times} \tau_2 \mid \tau_1 {+} \tau_2 \\[1.2em]
\text{Type Schemes} & \sigma & ::= \; \forall A.\sigma \mid \tau \\[1.2em]
\text{Environments} & \Gamma & ::= \; \cdot \mid \Gamma, x : \tau \\
& \Delta & ::= \; \cdot \mid \Delta, A \\[1.2em]
\text{Substitutions} & \delta & ::= \; \cdot \mid \delta[A {\mapsto} \tau] \\[1.2em]
\text{Expressions} & e & ::= \; () \mid x \mid \lambda x{:}\tau.e \mid e_1 \; e_2 \\
& & \quad \mid \varLambda A.e \mid e \; \tau \\
& & \quad \mid (e_1, e_2) \mid e.1 \mid e.2 \\
& & \quad \mid \texttt{inl} \; e \mid \texttt{inr} \; e \\
& & \quad \mid \texttt{case}(e, x : \tau_1.e_1, x : \tau_2.e_2) \\
& & \quad \mid \{e\}_\tau^\delta
\end{array}
$$

Fig. 4: Source Language

$$
\begin{array}{lll}
\text{Expressions} & e & ::= \; () \mid x \mid \lambda x.e \mid e_1 \; e_2 \\
& & \quad \mid (e_1, e_2) \mid e.1 \mid e.2 \\
& & \quad \mid \texttt{inl} \; e \mid \texttt{inr} \; e \\
& & \quad \mid \texttt{case}(e, x.e_1, x.e_2) \\
& & \quad \mid [e]_\tau^\delta \mid \langle e \rangle \\[1.2em]
\text{Environments} & \Gamma & ::= \; \cdot \mid \Gamma, x
\end{array}
$$

Fig. 5: Target Language

$$\boxed{\Delta \vdash \tau \ \mathtt{wf}}$$

$$\frac{}{\Delta, A \vdash A \ \mathtt{wf}} \ \text{TyVar} \qquad \frac{\Delta, A \vdash e \ \mathtt{wf}}{\Delta \vdash \forall A.\tau \ \mathtt{wf}} \ \text{Scheme} \qquad \frac{\forall \ x, \ \Delta \vdash \Gamma(x) \ \mathtt{wf}}{\Delta \vdash \Gamma \ \mathtt{wf}} \ \text{Env}$$
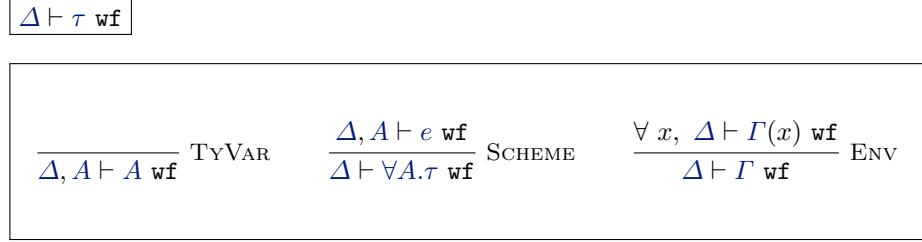
Fig. 6: Type Well-formedness

$$\boxed{\Delta \vdash \tau \ \mathtt{wf}}$$

$$\frac{}{\cdot \vdash \cdot \ \mathtt{wf}} \ \text{Empty} \qquad \frac{\Delta \vdash \delta \ \mathtt{wf} \qquad \Delta \vdash \tau \ \mathtt{wf}}{\Delta, A \vdash \delta[A \mapsto \tau] \ \mathtt{wf}} \ \text{Cons}$$
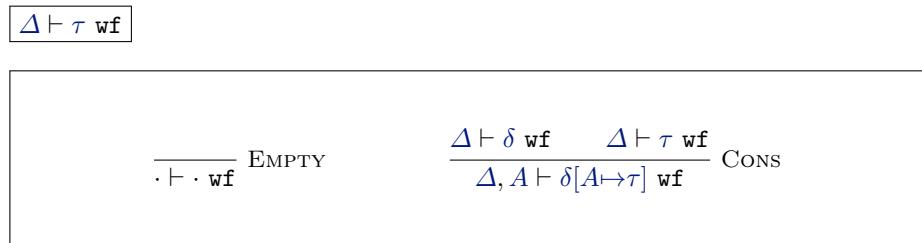
Fig. 7: Type Substitution Well-formedness

$$\boxed{\Delta; \ \Gamma \vdash e : \tau}$$

$$\frac{\Delta; \ \Gamma \vdash e \ \mathtt{ok}}{\Delta; \ \Gamma \vdash \{e\}_\tau^\delta : \delta\tau} \ \text{Alien}$$
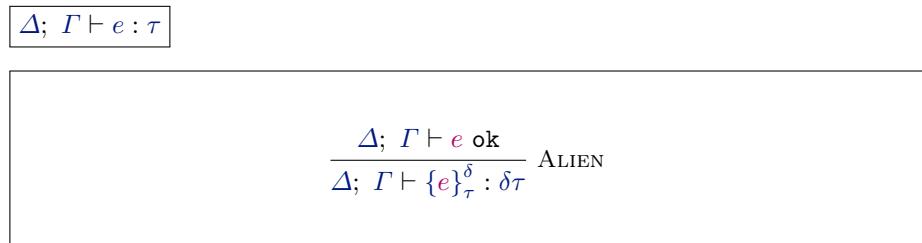
Fig. 8: Source Typing

$\boxed{e \;\rightarrow\; e'}$

$$\frac{e \;\rightarrow\; e'}{E[e] \;\rightarrow\; E[e']} \; \text{CTX} \qquad\qquad \frac{force(v) = \lambda x{:}\tau.e}{v\; e_2 \;\rightarrow\; e[e_2/x]} \; \text{APP}$$

$$\frac{force(v) = \Lambda A.e'}{v\; \tau \;\rightarrow\; e'[\tau/A]} \; \text{TAPP} \qquad \frac{force(v) = \texttt{inl}\; e'}{\texttt{case}(v, x : \tau_1.e_1, x : \tau_2.e_2) \;\rightarrow\; e_1[e'/x]} \; \text{CASEL}$$

$$\frac{force(v) = \texttt{inr}\; e'}{\texttt{case}(v, x : \tau_1.e_1, x : \tau_2.e_2) \;\rightarrow\; e_2[e'/x]} \; \text{CASER}$$

$$\frac{force(v) = (e_1, e_2)}{v.i \;\rightarrow\; e_i} \; \text{PROJ} \qquad\qquad \frac{e \;\rightarrow^*\; v}{\{e\}_\tau^\delta \;\rightarrow\; \{v\}_\tau^\delta} \; \text{ALIEN}$$

where

$$force(v) = \begin{cases} \mathcal{C}_\tau^\delta(v) & \text{if } e = \{v\}_\tau^\delta \text{ and } e \;\rightarrow^*\; v \\ v & \text{otherwise} \end{cases}$$

Fig. 9: Evaluation in the Source Language

$\boxed{e \;\rightarrow\; e'}$

$$\frac{e \;\rightarrow\; e'}{E[e] \;\rightarrow\; E[e']} \; \text{CTX} \qquad\qquad \frac{}{(\lambda x.e)\; v \;\rightarrow\; e[v/x]} \; \text{APP}$$

$$\frac{}{\texttt{case}(\texttt{inl}\; v, x.e_1, x.e_2) \;\rightarrow\; e_1[v/x]} \; \text{CASEL}$$

$$\frac{}{\texttt{case}(\texttt{inr}\; v, x.e_1, x.e_2) \;\rightarrow\; e_2[v/x]} \; \text{CASEL} \qquad \frac{}{(v_1, v_2).i \;\rightarrow\; v_i} \; \text{PROJ}$$

$$\frac{e \;\rightarrow^*\; v}{[e]_\tau^\delta \;\rightarrow\; \mathcal{C}_\tau^\delta(v)} \; \text{ALIEN}$$

Fig. 10: Evaluation in the Target Language

$$() [\tau/A] = ()$$
$$\lambda x{:}\tau'.e[\tau/A] = \lambda x{:}\tau'[\tau/A].e[\tau/A]$$
$$\vdots$$
$$\{e\}^{\delta}_{\tau'}[\tau/A] = \{e[\tau/A]\}^{\delta[A\mapsto\tau]}_{\tau'}$$

Fig. 11: Type Substitution in Source Terms

$$() [\tau/A] = ()$$
$$\lambda x.e[\tau/A] = \lambda x.e[\tau/A]$$
$$\vdots$$
$$[e]^{\delta}_{\tau'}[\tau/A] = [e[\tau/A]]^{\delta[A\mapsto\tau]}_{\tau'}$$
$$\langle v \rangle = \langle v[\tau/A] \rangle$$

Fig. 12: Type Substitution in Target Terms

$\boxed{e \rightsquigarrow e}$
Γ is a global parameter

$$\frac{}{() \rightsquigarrow ()} \ \text{Unit} \qquad \frac{}{x \rightsquigarrow [x]^{\cdot}_{\Gamma(x)}} \ \text{Var} \qquad \frac{}{\{e\}^{\delta}_{\tau} \rightsquigarrow e} \ \text{Box}$$

$$\frac{e_i \rightsquigarrow e_i}{(e_1,e_2) \rightsquigarrow (e_1,e_2)} \ \text{Pair} \qquad \frac{e \rightsquigarrow e}{e.i \rightsquigarrow e.i} \ \text{Proj} \qquad \frac{e \rightsquigarrow e}{\text{inl } e \rightsquigarrow \text{inl } e} \ \text{Inl}$$

$$\frac{e \rightsquigarrow e}{\text{inr } e \rightsquigarrow \text{inr } e} \ \text{Inr}$$

$$\frac{e \rightsquigarrow e \qquad e_i[\{x\}^{\cdot}_{\tau_i}/x] \rightsquigarrow e_i}{\text{case}(e, x : \tau_1.e_1, x : \tau_2.e_2) \rightsquigarrow \text{case}(e, x.e_1, x.e_2)} \ \text{Case}$$

$$\frac{e[\{x\}^{\cdot}_{\tau}/x] \rightsquigarrow e}{\lambda x{:}\tau.e \rightsquigarrow \lambda x.e} \ \text{Abs} \qquad \frac{e_i \rightsquigarrow e_i}{e_1 \ e_2 \rightsquigarrow e_1 \ e_2} \ \text{App} \qquad \frac{e \rightsquigarrow e}{\Lambda A.e \rightsquigarrow e} \ \text{TAbs}$$

$$\frac{e \rightsquigarrow e}{e \ \tau \rightsquigarrow e} \ \text{TApp}$$

Fig. 13: Translation

$$\mathcal{C}^\delta_{\mathtt{unit}}(()) = ()$$
$$\mathcal{C}^\delta_{\tau_1 \to \tau_2}(\lambda x.e) = \lambda x{:}\delta\tau_1.\{e[\mathcal{P}^\delta_{\tau_1}(x)/x]\}^\delta_{\tau_2}$$
$$\mathcal{C}^\delta_{\tau_1 \times \tau_2}((v_1,v_2)) = (\{v_1\}^\delta_{\tau_1},\{v_2\}^\delta_{\tau_2})$$
$$\mathcal{C}^\delta_{\tau_1 + \tau_2}(\mathtt{inl}\ v) = \mathtt{inl}\ \{v\}^\delta_{\tau_1}$$
$$\mathcal{C}^\delta_{\tau_1 + \tau_2}(\mathtt{inr}\ v) = \mathtt{inr}\ \{v\}^\delta_{\tau_2}$$
$$\mathcal{C}^\delta_A(\langle e\rangle) = e$$

$$\mathcal{C}^\delta_{\forall A.\sigma}(v) = \varLambda A.\mathcal{C}^\delta_\sigma(v)$$

Fig. 14: Target to Source Coercion

$$\mathcal{C}^\delta_{\mathtt{unit}}(()) = ()$$
$$\mathcal{C}^\delta_{\tau_1 \to \tau_2}(\lambda x{:}\tau.e) = \lambda x.\mathcal{P}^\delta_{\tau_2}(e[\{x\}^\delta_{\tau_1}/x])$$
$$\mathcal{C}^\delta_{\tau_1 \times \tau_2}((e_1,e_2)) = (\mathcal{P}^\delta_{\tau_1}(e_1),\mathcal{P}^\delta_{\tau_2}(e_2))$$
$$\mathcal{C}^\delta_{\tau_1 + \tau_2}(\mathtt{inl}\ e) = \mathtt{inl}\ \mathcal{P}^\delta_{\tau_1}(e)$$
$$\mathcal{C}^\delta_{\tau_1 + \tau_2}(\mathtt{inr}\ e) = \mathtt{inr}\ \mathcal{P}^\delta_{\tau_2}(e)$$
$$\mathcal{C}^\delta_A(e) = \langle e\rangle$$
$$\mathcal{C}^\delta_{\tau'}(\{e\}^\delta_\tau) = e \qquad \text{if } \tau \equiv \tau'$$

Fig. 15: Source to Target Coercion

$$\mathcal{P}^\delta_\tau(e) = \begin{cases} \langle e\rangle & \text{if } \tau = A \\ [e]^\delta_\tau & \text{otherwise} \end{cases}$$

Fig. 16: Protect meta-function

$$
\begin{aligned}
\texttt{unit} \equiv \texttt{unit} &\overset{\textbf{def}}{=} \top \\
A \equiv B &\overset{\textbf{def}}{=} \top \\
(\tau_1,\tau_2) \equiv (\tau_1',\tau_2') &\overset{\textbf{def}}{=} \tau_1 \equiv \tau_1' \wedge \tau_2 \equiv \tau_2' \\
\tau_1 {\to} \tau_2 \equiv \tau_1' {\to} \tau_2' &\overset{\textbf{def}}{=} \tau_1 \equiv \tau_1' \wedge \tau_2 \equiv \tau_2' \\
\tau_1 {+} \tau_2 \equiv \tau_1' {+} \tau_2' &\overset{\textbf{def}}{=} \tau_1 \equiv \tau_1' \wedge \tau_2 \equiv \tau_2' \\
\tau \equiv \tau' &\overset{\textbf{def}}{=} \bot \qquad \text{otherwise}
\end{aligned}
$$

Fig. 17: Type equivalence

$\boxed{e \rightsquigarrow e}$

$$
\frac{}{() \rightsquigarrow ()} \ \textsc{Unit} \qquad\qquad \frac{}{x \rightsquigarrow [x]_{\Gamma(x)}^{\cdot}} \ \textsc{Var} \qquad\qquad \frac{}{\{e\}_\tau^\delta \rightsquigarrow e} \ \textsc{Box}
$$

$$
\frac{e_i \rightsquigarrow e_i}{(e_1,e_2) \rightsquigarrow (e_1,e_2)} \ \textsc{Pair} \qquad \frac{e \rightsquigarrow e}{e.i \rightsquigarrow e.i} \ \textsc{Proj} \qquad \frac{e \rightsquigarrow e}{\texttt{inl}\ e \rightsquigarrow \texttt{inl}\ e} \ \textsc{Inl}
$$

$$
\frac{e \rightsquigarrow e}{\texttt{inr}\ e \rightsquigarrow \texttt{inr}\ e} \ \textsc{Inr}
$$

$$
\frac{e \rightsquigarrow e \qquad e_i[\{x\}_{\tau_i}^{\cdot}/x] \rightsquigarrow e_i}{\texttt{case}(e, x:\tau_1.e_1, x:\tau_2.e_2) \rightsquigarrow \texttt{case}(e, x.e_1, x.e_2)} \ \textsc{Case}
$$

$$
\frac{e[\{x\}_\tau^{\cdot}/x] \rightsquigarrow e}{\lambda x{:}\tau.e \rightsquigarrow \lambda x.e} \ \textsc{Abs} \qquad \frac{e_i \rightsquigarrow e_i}{e_1\ e_2 \rightsquigarrow e_1\ e_2} \ \textsc{App} \qquad \frac{e \rightsquigarrow e}{\Lambda A.e \rightsquigarrow e} \ \textsc{TAbs}
$$

$$
\frac{e \rightsquigarrow e}{e\ \tau \rightsquigarrow e} \ \textsc{TApp}
$$

Fig. 18: Translation

Bibliography

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

[2] D. Ahman, C. Fournet, C. Hriţcu, K. Maillard, A. Rastogi, and N. Swamy. Recalling a witness: Foundations and applications of monotonic state. *PACMPL*, 2(POPL):65:1–65:30, 2018.

[3] N. Amin. How to write your next POPL paper in Dafny. Microsoft Research talk, 2013.

[4] N. Amin and T. Rompf. LMS-Verify: Abstraction without regret for verified systems programming. *POPL*. 2017.

[5] N. Amin, K. R. M. Leino, and T. Rompf. Computing with an SMT solver. *TAP*, 2014.

[6] A. Anand, S. Boulier, C. Cohen, M. Sozeau, and N. Tabareau. Towards certified meta-programming with typed Template-Coq. In *ITP*. 2018.

[7] A. W. Appel. Tactics for separation logic. Early Draft, 2006.

[8] S. Awodey and A. Bauer. Propositions as [Types]. *J. Log. and Comput.*, 14 (4):447–471, 2004.

[9] H. Barendregt and H. Geuvers. Handbook of automated reasoning. chapter Proof-assistants Using Dependent Type Systems, pages 1149–1238. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, 2001.

[10] M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. *FMCO*. 2005.

[11] M. Barnett, R. DeLine, M. Fähndrich, B. Jacobs, K. R. M. Leino, W. Schulte, and H. Venter. The Spec# programming system: Challenges and directions. *VSTTE*. 2005.

[12] B. Barras, B. Grégoire, A. Mahboubi, and L. Théry. Coq reference manual; chapter 25: The ring and field tactic families. Available at `https://coq.inria.fr/refman/ring.html`.

[13] U. Berger and H. Schwichtenberg. An inverse of the evaluation functional for typed lambda-calculus. *LICS*. 1991.

[14] D. J. Bernstein. The Poly1305-AES message-authentication code. *FSE*. 2005.

[15] F. Besson. Fast reflexive arithmetic tactics the linear case and beyond. *TYPES*. 2006.

[16] K. Bhargavan, B. Bond, A. Delignat-Lavaud, C. Fournet, C. Hawblitzel, C. Hriţcu, S. Ishtiaq, M. Kohlweiss, R. Leino, J. Lorch, K. Maillard, J. Pang, B. Parno, J. Protzenko, T. Ramananandro, A. Rane, A. Rastogi, N. Swamy, L. Thompson, P. Wang, S. Zanella-Béguelin, and J.-K. Zinzindohoué. Everest: Towards a verified, drop-in replacement of HTTPS. *SNAPL*, 2017.

[17] J. C. Blanchette and A. Popescu. Mechanizing the metatheory of Sledgehammer. *FroCoS*. 2013.

[18] J. C. Blanchette, S. Böhme, and L. C. Paulson. Extending Sledgehammer with SMT solvers. *JAR*, 51(1):109–128, 2013.

[19] M. Boespflug, M. Dénès, and B. Grégoire. Full reduction at full throttle. *CPP*. 2011.

[20] S. Böhme and T. Weber. Fast LCF-style proof reconstruction for Z3. *ITP*. 2010.

[21] B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Parno, A. Rane, S. T. V. Setty, and L. Thompson. Vale: Verifying high-performance cryptographic assembly code. *USENIX Security*. 2017.

[22] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *STTT*, 7(3):212–232, 2005.

[23] A. Chaieb and T. Nipkow. Proof synthesis and reflection for linear arithmetic. *J. Autom. Reasoning*, 41(1):33–59, 2008.

[24] D. R. Christiansen and E. Brady. Elaborator reflection: extending Idris in Idris. *ICFP*. 2016.

[25] E. Cohen, M. Moskal, W. Schulte, and S. Tobies. Local verification of global invariants in concurrent programs. *CAV*. 2010.

[26] P. Crégut. Strongly reducing variants of the Krivine abstract machine. *HOSC*, 20(3):209–230, 2007.

[27] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C - A software analysis perspective. *SEFM*. 2012.

[28] L. Czajka and C. Kaliszyk. Hammer for Coq: Automation for dependent type theory. Submitted to JAR, 2017.

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

[30] D. Delahaye. A tactic language for the system Coq. *LPAR*. 2000.

[31] G. Ebner, S. Ullrich, J. Roesch, J. Avigad, and L. de Moura. A metaprogramming framework for formal verification. *PACMPL*, 1(ICFP):34:1–34:29, 2017.

[32] B. Ekici, A. Mebsout, C. Tinelli, C. Keller, G. Katz, A. Reynolds, and C. W. Barrett. SMTCoq: A plug-in for integrating SMT solvers into Coq. *CAV*, 2017.

[33] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala. Simple high-level code for cryptographic arithmetic - with proofs, without compromises. *IEEE S&P*, 2019.

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

[35] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. PLDI 2002: Extended static checking for Java. *SIGPLAN Notices*, 48(4S):22–33, 2013.

[36] A. Fromherz, N. Giannarakis, C. Hawblitzel, B. Parno, A. Rastogi, and N. Swamy. A verified, efficient embedding of a verifiable assembly language. *PACMPL*, (POPL), 2019.

[37] G. Gonthier. Formal proof–the four-color theorem. *Notices of the AMS*, 55 (11):1382–1393, 2008.

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

[39] B. Grégoire and A. Mahboubi. Proving equalities in a commutative ring done right in Coq. *TPHOLs*. 2005.

[40] G. Grov and V. Tumas. Tactics for the Dafny program verifier. *TACAS*. 2016.

[41] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad Apps: End-to-end security via automated full-system verification. *OSDI*. 2014.

[42] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. T. V. Setty, and B. Zill. Ironfleet: proving safety and liveness of practical distributed systems. *CACM*, 60(7):83–92, 2017.

[43] J. Kaiser, B. Ziliani, R. Krebbers, Y. Régis-Gianas, and D. Dreyer. Mtac2: typed tactics for backward reasoning in Coq. *PACMPL*, 2(ICFP):78:1–78:31, 2018.

[44] C. Kaliszyk and J. Urban. Learning-assisted automated reasoning with Flyspeck. *JAR*, 53(2):173–213, 2014.

[45] C. Kaliszyk and J. Urban. MizAR 40 for Mizar 40. *JAR*, 55(3):245–256, 2015.

[46] R. Krebbers, A. Timany, and L. Birkedal. Interactive proofs in higher-order concurrent separation logic. *POPL*. 2017.

[47] J.-L. Krivine. A call-by-name lambda-calculus machine. *Higher Order Symbol. Comput.*, 20(3):199–207, 2007.

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

[49] K. R. M. Leino and G. Nelson. An extended static checker for Modula-3. *CC*. 1998.

[50] A. McCreight. Practical tactics for separation logic. *TPHOLs*. 2009.

[51] G. Melquiond and R. Rieu-Helft. A Why3 framework for reflection proofs and its application to GMP's algorithms. *IJCAR*. 2018.

[52] A. Nanevski, J. G. Morrisett, and L. Birkedal. Hoare type theory, polymorphism and separation. *JFP*, 18(5-6):865–911, 2008.

[53] A. Nanevski, V. Vafeiadis, and J. Berdine. Structuring the verification of heap-manipulating programs. *POPL*. 2010.

[54] A. Nogin. Quotient types: A modular approach. *TPHOLs*. 2002.

[55] L. C. Paulson and J. C. Blanchette. Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. *IWIL*. 2010.

[56] 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.

[57] A. Stampoulis and Z. Shao. VeriML: typed computation of logical terms inside a language with effects. *ICFP*. 2010.

[58] N. Swamy, J. Weinberger, C. Schlesinger, J. Chen, and B. Livshits. Verifying higher-order programs with the Dijkstra monad. *PLDI*, 2013.

[59] 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.

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

[61] N. Vazou, L. Lampropoulos, and J. Polakow. A tale of two provers: verifying monoidal string matching in Liquid Haskell and Coq. *Haskell Symposium*. 2017.

[62] N. Vazou, A. Tondwalkar, V. Choudhury, R. G. Scott, R. R. Newton, P. Wadler, and R. Jhala. Refinement reflection: complete verification with SMT. *PACMPL*, 2(POPL):53:1–53:31, 2018.

[63] P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. *POPL*. 1987.

[64] M. Wenzel. The Isabelle/Isar reference manual. Available at `http://isabelle.in.tum.de/doc/isar-ref.pdf`, 2017.

[65] B. Ziliani, D. Dreyer, N. R. Krishnaswami, A. Nanevski, and V. Vafeiadis. Mtac: A monad for typed tactic programming in Coq. *JFP*, 25, 2015.

[66] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche. HACL*: A verified modern cryptographic library. *CCS*. 2017.