F*: From Program Verification System to Proof Assistant

Last updated: December 27, 2015

Advisor: Cătălin Hriţcu ⟨catalin.hritcu@gmail.com⟩

Institition: INRIA Paris, Prosecco Team

Location: 2 rue Simone Iff, 75012 Paris, France

Language: English

Existing skills or strong desire to learn:

- functional programming (e.g. ML or Haskell);
- formal verification in a proof assistant (e.g. Coq) or in a program verification tool (e.g. Why3 or Dafny);
- others depending topic: F*, logic, type theory, semantics, mechanized metatheory, automated deduction, metaprogramming, effects, monads, secure compilation, gradual typing, model finding, property-based testing

Introduction

F* is a verification system for ML programs developed collaboratively by Inria and Microsoft Research. ML types are extended with logical predicates that can conveniently express precise specifications for programs (pre- and post- conditions of functions as well as stateful invariants), including functional correctness and security properties. The F* typechecker implements a weakest-precondition calculus [35] to produce first-order logic formulas that are automatically discharged using the Z3 SMT solver [13]. The original F* implementation [31] has been successfully used to verify nearly 50,000 lines of code, including cryptographic protocol implementations [31, 6], web browser extensions [19, 35], cloud-hosted web applications [31], and key parts of the F* compiler itself [30]. F* has also been used for formalizing the semantics of other languages, including JavaScript and a compiler from a subset of F* to JavaScript [18] and TS*, a secure subset of TypeScript [32]. Programs verified with F* can be extracted to F#, OCaml, and JavaScript and then efficiently executed and integrated into larger code bases.

While the old F* design [31] was a quite successful, it had reached its limits in terms of expressiveness. Over the past 1.5 years, we have completely redesigned and reimplemented F* [33], producing a new prototype tool that is open source and cross-platform,[1][2] and that is already surpassing old F* both in terms of external users and size of the verified code (about 55,000 lines). One of the main problems we aim to address in this redesign is that the user of old F* had very few escape hatches when SMT-based automation fails. In particular, users of old F* had often no way to tell whether the property they were trying to verify was true or not. This problem is not specific to old F*, but general to SMT-based type-checkers and verification systems like Liquid Haskell [38], Dafny [21], and Why3 [16] where often users are left shooting in the dark when verification fails, having to enter an extremely costly debug loop of adding assertions to narrow down the problems. Instead, we want a system that combines SMT-based automation with some of the power and expressiveness of proof assistants like Coq [37] and Lean [14], which enable users to prove arbitrarily complex properties manually. To address this our F* redesign has introduced full dependent types and tracking of effects, while isolating a core language of pure total functions that can be used to write specifications and proof terms. While this is a good foundation for proofs, more research is needed before F* can effectively assist the user in building complex proofs using a *seamless combination* of automatic SMT solving and constructive manual proofs.

This document is aimed at briefly presenting various topics from the vast F* project to which a student or young researcher could contribute (during a research internship, MSc or PhD thesis, PostDoc, etc). The list is not exhaustive, and if you are interested you should get in contact and we will together try to find a topic that is in sync with your interests and expertise.

1. **completing the design of F*'s proof language**; this is challenging because we are looking for a logic that is simultaneously: (a) flexible enough to allow freely mixing automatic and manual proofs, (b) expressive enough for practical verification, (c) sound, (d) simple to implement, (e) integrated with the rest of the F* language, in particular with the effectful part;
2. **strengthening the formal foundations of F***: (a) proving the soundness, normalization, and consistency for increasingly large fragments of F*, and (b) formalizing the complex SMT encoding used by F*;
3. **working towards the self-certification of F***, in particular (a) applying F* to mechanizing parts of its own metatheory and (b) to the lightweight verification of simple internal invariants of its own implementation, and (c) devising a certifying SMT backend;
4. **solving the engineering challenges of turning F* into a usable proof assistant**; most importantly devising an Mtac-inspired [39] dependently-typed tactic language that uses F* itself as the meta-language;
5. **Dijkstra monads "for free"**, i.e., deriving correct-by-design, specification-level monads for efficient automatic reasoning from expression-level monads;

[1] `https://www.fstar-lang.org/`
[2] `https://github.com/FStarLang/FStar`

6. **fully abstract extraction to ML**, showing that programs that are written in a combination of ML and F* are still secure, even if only the F* part is verified;

7. **fictional purity**, allowing some of the effects of a computation to be hidden if they do not impact the observable behavior of the computation;

8. **producing counterexamples** showing that a stated property is actually wrong using SMT-based model finding and property-based testing.

1 Design of F*'s proof language

The most important feature we want in new F* is freely mixing classical proofs performed automatically by the SMT with manually written constructive proofs. For this we added full dependent types and tracking of effects, including a semantic termination check, which allowed us to isolate a core language of pure total functions for writing specifications and proof terms. While this is a good foundation for devising constructive proofs, well-known paradoxes from type theory make it non-trivial to connect such proofs with the classical proofs produced using the SMT solver. In particular, impredicative polymorphism (as in System Fω) does not soundly compose with classical axioms and proof relevance [4]. Moreover, the size of inductive types has to be careful tracked since otherwise one can directly encode Russell's paradox.

In order to allow the seamless combination of automatic SMT solving and constructive manual proofs we are currently implementing a predicative system with an infinite hierarchy of universes like in Coq [37, 29], Lean [14], Agda [25], or Nuprl [11]. To recover the full expressive power of an impredicative system one alternative is introducing a new impredicative kind Prop and use a Coq-like elimination restriction to prevent programs from observing proofs [37, 20]. Instead we are considering a more light-weight alternative, using squash types [24] à la Nuprl to isolate the classical proofs produced by the SMT solver to a proof-irrelevant fragment. While these ides seem appealing, the formal details still need to be figured out, and a good student could help pushing this forward more quickly.

2 Formal foundations of F*

While the metatheory of F* is still work in progress, we have already formally studied two subsets of F* called μF*(micro-F*) and pF* (pico-F*) [33]. For μF*, we prove partial correctness for the specifications of effectful computations via a syntactic progress and preservation argument. While the complete proofs are on paper, we have recently formalized part of these proofs in F* itself [17]. μF* is already a complex language in its own right that already presents a non-trivial formalization challenge: the type system has around 100 rules and features dependent types; type operators; subtyping; semantic termination checking; a lattice of user-defined effects; predicate transformers; user-defined heap models, and higher-order state. For pF*, a much smaller pure fragment of μF*, the author of this proposal has additionally proved (on paper) weak normalization and logical consistency using logical relations.

Unfortunately, our current proofs do not include the troublesome features from the previous section. Extending the μF* and pF*proofs to predicative polymorphism (universes) and inductive types is thus of crucial importance for putting the whole F* design on more solid formal foundations, and could potentially impact the design process. Afterwards, we would like to gradually extend pF*and its consistency proof with all the features of μF*, and eventually all the features of F* itself. This could potentially also push the limits of the consistency proofs one can do using logical relations, or require us to search for alternative proof techniques like reducibility candidates or constructing full-fledged denotational models.

Another interesting formalization target is the SMT encoding used by F*. The soundness of F* crucially relies on an encoding of its dependently-typed higher-order logic (HOL) into simply-typed first-order logic (FOL). This encoding is complex and optimized to strike a pragmatic balance between completeness and efficiency. We believe that formalizing and proving the soundness and maybe partial completeness of this encoding could serve as inspiration for other SMT-based verification tools such as Dafny [2] and could be a first step towards bringing SMT support to proof assistants that do not yet have any, such as Coq [37] and Lean [14]. While HOL to FOL translations have been implemented and formalized in the past [23, 27, 8], we are not aware of any formalization of a *practical* translation of *dependently-typed* HOL to FOL.

Working out the metatheory of F* and of its logical encoding are also necessary steps towards self-certification, which is discussed below.

3 Towards the self-certification of F*

F* is implemented in F* and can bootstrap itself via F# or OCaml. Our long-term goal is to use F* to verify a core F* type-checker, while producing a Coq certificate [30] or some other kind of evidence that is independently checkable with a small TCB [12]. This would ensure that no wrong program or proof is accepted as valid, a strong guarantee that would eliminate the practical inconsistencies that regularly appear in F* and even in much more established proof assistants like Coq. While such a self-certification effort was already performed for old F* [30], doing the same for the much more expressive, current variant of F* is a big challenge. Moreover, for the old self-certification effort the metatheory was done in Coq and the verification effort was limited to producing Coq certificates. For new F* we want to work out most of the metatheory of F* in F* itself, as shown possible by the formalization of the calculus of constructions in Coq [5]. Finally, we do not want to trust the SMT solver or the logical encoding of F* for this certification process.

While this ambitious goal might be out of reach even for the time-frame of a PhD, there are useful intermediate goals we can aim for. First, we are already applying F* to mechanizing parts of its own metatheory [17], and we would like to scale these proofs up to larger F* fragments, by first eliminating conceptual and practical limitations in the F* proof assistant (as already discussed above). Second, we would like to apply F* to the lightweight verification of simple internal invariants of its own implementation, for instance showing that certain kinds of "Unexpected Errors" can indeed not happen. Finally, we would like to devise a certifying SMT backend that produces independently checkable evidence for each logical formula that F⋆ encodes and sends to the SMT solver. This would cover not just the SMT solver itself [3], but also the implementation of F⋆'s logical encoding.

4 Making F⋆ a usable proof assistant

Turning F⋆ into an usable proof assistant raises many engineering challenges we have to address, such as interactive proof editing, proof goal visualization, typed-directed programming, proof caching, SMT verification debugging, etc. One particularly interesting and important task is designing and implementing a tactic language for F⋆. We aim to take inspiration in the recently proposed Mtac language [39] for Coq and devise a dependently-typed tactic language for F⋆ that uses F⋆ itself as the meta-language. In fact, we believe that F⋆'s *efficient* support for primitive effects such as state and exceptions make it a better base for implementing such a tactic language. A new effect we would need to add to F⋆ in order to implement an Mtac-like tactic language is reflecting on the syntax of F⋆ expressions. This form of meta-programming is of course very useful in its own right [28], beyond just tactics, and we could try to not only verify meta-programs within F⋆, but also the program produced by meta-programming.

5 Dijkstra monads for free

Dijkstra monads [35, 34] are the mechanism by which F⋆ efficiently computes verification conditions, generically for all the effects of F⋆. While the verification condition generation algorithm is generic, for each effect one needs to define the operations used to combine weakest preconditions (return, bind, and a dozen others). This is rather subtle, because these operations are phrased in continuation-passing style and are subject to significant (meta-level) proof obligations in order to preserve the soundness of F⋆. Moreover, at the moment Dijkstra monads only work for the primitive effects of F⋆ (basically the effects of ML), and we have no good way of adding new *user-specified effects*. Finally, the weakest precondition calculus is currently the *only* way by which one can reason about impure code, which means that all proofs about impure code have to be done *intrinsically*, when the code is defined. Proving properties *extrinsically*, after the fact, currently only works for pure code.

We are currently working to lift these limitations by automatically deriving correct-by-design Dijkstra monads from expression-level monads (e.g., from a direct implementation of the state or error monad) via a continuation-passing style translation. We will use logical relations to prove once and for all, in F⋆, that this translation is correct, producing well-defined Dijkstra monads. Moreover, the produced monads will be arranged in a lattice structure, and the translation will produce the lifts for going up in the lattice (layering monads was previously studied by Filinski [15]). Finally, the translation will produce "reify" and "reflect" operations for translating between effectful computations and their pure monadic encoding, enabling extrinsic reasoning, which is crucial for instance for proving relational properties in a less ad-hoc way [6]. We will experimentally evaluate our design by deriving Dijkstra monads for the primitive effects of F⋆ as well as new effects such as non-determinism and probabilities.

6 Fully abstract extraction to ML

F⋆ components can be extracted to OCaml and F# using a backend based on Coq's extraction algorithm [22]. Basically F⋆ types are erased to ML types and type coercions are inserted to convince the ML type system that the program is well-typed. Provided this backend is correct, a complete F⋆ program should behave the same when extracted to ML, as it did in F⋆. In reality, the situation is more complicated, since F⋆ programs often have parts written directly in ML (e.g., libraries), and the unverified ML can easily break the invariants of the verified F⋆. The key to secure interoperability is enforcing that ML and F⋆ code interoperate via ML-typed interfaces. Efficiently decidable properties can be turned into contracts and checked dynamically, effectively extending the set of interfaces at which ML and Coq can safely interoperate. Formally, we will construct machine-checked proofs of *full abstraction* [1] with respect to the F⋆ semantics. This is much stronger than just compiler correctness and ensures that no ML attacker can do more harm to securely compiled programs than a program in F⋆ already could. The key to proving full abstraction is that F⋆ can express both pure and effectful computations, so it can faithfully express ML interfaces, while other proof assistants cannot [36].

7 Fictional purity

At the moment, effects in F⋆ are syntactic: if any subcomputation triggers a certain effect then the whole computation is tainted with that effect. However, we would like to be able to relax this when the effect of a computation is unobservable to its context. For example, consider computations that use state locally for memoization, or those that handle all exceptions that may be raised: it would be convenient to treat such computations as pure. While there are some limits to this (e.g., currently, termination in F⋆ has to be proven intrinsically), we should be able to forget most effects, provided we prove in F⋆ that these effects do not matter. We would like to use relational reasoning within F⋆ to do these proofs. One major semantic complication is hiding dynamic allocation, but recent work by Benton *et al.* [7] uses proof-relevant logical relations (i.e., setoids) for overcoming this.

8 Refuting properties

Providing useful feedback when semi-automatic verification fails is notoriously difficult. At the moment, F⋆ tags each part of a proof obligation with a unique label that identifies the source code location from where the part came. This way the models produced by Z3 on a verification failure are used to identify the code that could be responsible. While this works reasonably well, simply identifying the code that could be responsible for the failure is often not informative enough for understanding the problem. Producing concrete counterexamples falsifying the property would be very useful in such cases. Mapping Z3 models back to counterexamples is, however, challenging. For a start, we would need a way to reverse the complex logical encoding of F⋆ that is robust to wrong models (for F⋆'s logical encoding Z3 often produces wrong models and not only when it times out). Then, we would need to check that the produced counterexample is indeed correct, which could itself involve additional Z3 queries. Alternatively, we could use a different logical encoding that is not sound for proving but that is guaranteed to produce correct counterexamples [10]. Finally, even if we have a counterexample for a branch of a type derivation, the F⋆ type-checker can backtrack, so in the end we can end up with multiple counterexamples, one for each type-checking branch. So we either need to find a way to combine counterexamples or otherwise to display the failed typing derivations to the user.

A completely different way to approach this problem is to use property-based testing (PBT) [26] to find counterexamples. Previous experience [9] shows that PBT has complementary strengths and weaknesses compared to SMT or model finding techniques, so for the best outcomes one would run all these tools in parallel. A separate document describes integrating PBT in a proof assistant in detail.

References

[1] M. Abadi. Protection in programming-language translations. Research Report 154, SRC, 1998.

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

[3] M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Werner. A modular integration of SAT/SMT solvers to Coq through proof witnesses. *CPP*. 2011.

[4] F. Barbanera and S. Berardi. Proof-irrelevance out of excluded-middle and choice in the calculus of constructions. *Journal of Functional Programming*, 6(3):519–525, 1996.

[5] B. Barras. Coq en Coq, 1996.

[6] G. Barthe, C. Fournet, B. Grégoire, P. Strub, N. Swamy, and S. Z. Béguelin. Probabilistic relational verification for cryptographic implementations. *POPL*. 2014.

[7] N. Benton, M. Hofmann, and V. Nigam. Abstract effects and proof-relevant logical relations. *POPL*. 2014.

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

[9] J. C. Blanchette, L. Bulwahn, and T. Nipkow. Automatic proof and disproof in Isabelle/HOL. *FroCoS*. 2011.

[10] J. C. Blanchette and T. Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. *ITP*. 2010.

[11] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, NJ, 1986.

[12] J. Davis and M. O. Myreen. The reflective Milawa theorem prover is sound (down to the machine code that runs it). *J. Autom. Reasoning*, 55(2):117–183, 2015.

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

[14] L. M. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer. The Lean theorem prover (system description). *CADE*. 2015.

[15] A. Filinski. Representing layered monads. *POPL*. 1999.

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

[17] S. Forest and C. Hriţcu. Micro-F* in F*. Inria internship report, 2015.

[18] C. Fournet, N. Swamy, J. Chen, P.-E. Dagand, P.-Y. Strub, and B. Livshits. Fully abstract compilation to JavaScript. *POPL*, 2013.

[19] A. Guha, M. Fredrikson, B. Livshits, and N. Swamy. Verified security for browser extensions. *Oakland S&P*. 2011.

[20] B. Jacobs. The essence of Coq as a formal system. Online manuscript, 2013.

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

[22] P. Letouzey. Extraction in Coq: An overview. *CiE*. 2008.

[23] J. Meng and L. C. Paulson. Translating higher-order clauses to first-order clauses. *JAR*, 40(1):35–60, 2008.

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

[25] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, 2007.

[26] Z. Paraskevopoulou, C. Hriţcu, M. Dénès, L. Lampropoulos, and B. C. Pierce. Foundational property-based testing. *ITP*. 2015.

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

[28] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *CACM*, 55(6):121–130, 2012.

[29] M. Sozeau and N. Tabareau. Universe polymorphism in Coq. *ITP*. 2014.

[30] P. Strub, N. Swamy, C. Fournet, and J. Chen. Self-certification: bootstrapping certified typecheckers in F* with Coq. *POPL*. 2012.

[31] N. Swamy, J. Chen, C. Fournet, P. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. *JFP*, 23(4):402–451, 2013.

[32] N. Swamy, C. Fournet, A. Rastogi, K. Bhargavan, J. Chen, P.-Y. Strub, and G. M. Bierman. Gradual typing embedded securely in JavaScript. *POPL*, 2014.

[33] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, and J.-K. Zinzindohoue. Dependent types and multi-monadic effects in F*. Draft, 2015.

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

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

[36] É. Tanter and N. Tabareau. Lost in extraction, recovered: An approach to compensate for the loss of properties and type dependencies when extracting Coq components to OCaml. ML workshop, 2015.

[37] The Coq development team. *The Coq proof assistant*.

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

[39] B. Ziliani, D. Dreyer, N. R. Krishnaswami, A. Nanevski, and V. Vafeiadis. Mtac: a monad for typed tactic programming in Coq. *ICFP*. 2013.