F⋆: Reliable Automatic Proofs and Counterexamples using SMT

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);
- logic, type theory, automated deduction, model finding

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 [19] to produce first-order logic formulas that are automatically discharged using the Z3 SMT solver [7]. The original F⋆ implementation [16] has been successfully used to verify nearly 50,000 lines of code, including cryptographic protocol implementations [16, 4], web browser extensions [11, 19], cloud-hosted web applications [16], and key parts of the F⋆ compiler itself [15]. F⋆ has also been used for formalizing the semantics of other languages, including JavaScript and a compiler from a subset of F⋆ to JavaScript [10] and TS⋆, a secure subset of TypeScript [17]. 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 [16] 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⋆ [18], 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 [21], Dafny [12], and Why3 [9] 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 [20] and Lean [8], 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.

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 [2, 1] and could be a first step towards bringing SMT support to proof assistants that do not yet have any, such as Coq [20] and Lean [8]. While HOL to FOL translations have been implemented and formalized in the past [13, 14, 5], 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; in particular we do not want to trust the SMT solver or the logical encoding of F⋆ for this certification process. 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.

Finally, 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 in-

[1] https://www.fstar-lang.org/

[2] https://github.com/FStarLang/FStar

deed 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 [6]. 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.

References

[1] HaTT 2016: First international workshop hammers for type theories, 2016.

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

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

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

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

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

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

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

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

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

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

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

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

[16] 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.

[17] 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.

[18] 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.

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

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

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