F⋆: SMT-Based Verification for ML

Co-advisors: Karthikeyan Bhargavan and Cătălin Hriţcu

Institition: INRIA Paris-Rocquencourt, Prosecco Team

Location: 23 Avenue d'Italie, Paris, France

Language: English

Existing skills or strong desire to learn:

- functional programming (e.g. OCaml, F#, Standard ML, or Haskell),
- optional: formal verification in a proof assistant (e.g. Coq),
- optional: security, compiler construction, programming languages theory, automated reasoning

Research Context

F⋆ is a new ML-like programming language and verification tool for higher-order, effectful programs (Swamy et al., 2011, 2013a). Specifications are expressed using refinement types, extending ML types with logical predicates that have to hold for any element of the type. Refinement types can express pre- and post- conditions of functions as well as stateful invariants. F⋆ uses a weakest-precondition calculus (Swamy et al., 2013b) to translate refinement types into verification conditions, which are discharged automatically by an SMT solver (de Moura and Bjørner, 2008). F⋆ has been successfully used to verify nearly 50,000 lines of code, ranging from cryptographic protocol implementations (Swamy et al., 2013a) to web browser extensions (Guha et al., 2011), and from cloud-hosted web applications (Swamy et al., 2013a) to key parts of the F⋆ compiler itself (Strub et al., 2012).

F⋆ is now towards the end of a rational redesign, simplification, and reimplementation process. The new version of F⋆ that resulted from this is open source and cross-platform,[1] compiling to OCaml and F#, on which it is based. It also compiles securely to JavaScript (Fournet et al., 2013; Swamy et al., 2014), enabling safe interoperability with arbitrary, untrusted JavaScript libraries. This allows one to write and verify ML programs in F⋆ and deploy them to the web as JavaScript.

F⋆ is a collaborative effort between Microsoft Research, MSR-Inria, Inria, and IMDEA Software Institute. The current development team includes Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Pierre-Yves Strub, and Nikhil Swamy.

Mechanized Metatheory in F⋆

Doing mechanized metatheory (Aydemir et al., 2005) in proof assistants such as Coq is often tedious and time consuming. We believe that SMT-based tools like F⋆ have the potential to greatly reduce the manual effort needed for PL proofs. As a first step in this direction we use F⋆'s new ability to do "extrinsic proofs" (as opposed to intrinsically proving refinement types at definition time) to show progress and preservation for the simply-typed λ-calculus,[2] following the Software Foundations textbook (Pierce et al., 2013). Even without special support from the type-checker and editor, these proofs are small, readable, and apparently easy to maintain and extend, which is rarely the case with Coq proofs.

We would like to extend such simple proofs to System F-sub (Aydemir et al., 2005) and eventually to the metatheory of F⋆ itself (see below). Further improvements to F⋆ are, however, necessary before building such large proofs becomes feasible with reasonable effort. At the moment the F⋆ user has no easy way to obtain

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

[2] `https://github.com/FStarLang/FStar/blob/master/examples/metatheory/stlc.fst`

the typing context at a particular point in the proof. Displaying the current typing context wouldn't always be informative enough though, since in F⋆ types and pure definitions are translated to the SMT solver via a rather involved logical encoding, which we want to hide that from the user as much as possible. Moreover, the SMT solver can perform complex logical reasoning that sometimes also needs to be debugged during proof construction. We want to address these challenges and allow F⋆ users to build proofs in a more intuitive way. For this we will take inspiration in the Agda editor (Coquand et al., 2006) and in a previous effort that used the Dafny (first-order, general purpose) verification system for formalized metatheory (Amin, 2013; Amin et al., 2014).

Verify Metatheory of F⋆

While for some previous versions of F⋆ proofs existed (Strub et al., 2012), this is not the case for the newest version. In particular, the newest F⋆ version involves interesting termination and logical consistency arguments that we have not yet formalized. We plan to address this by studying a sequence of increasingly complex calculi starting from System F-sub and potentially ending with full-fledged certification of the F⋆ type-checker. One added benefit of this incremental approach is that it will help introducing newcomers to the (otherwise rather sophisticated) F⋆ type system and it might even lead to improvements in the design of F⋆ (e.g. teasing out orthogonal concepts more clearly). More ambitiously, we would like to certify F⋆ within F⋆ itself, taking inspiration in the previous self-certification work (Strub et al., 2012) as well as the "Coq in Coq" work (Barras and Werner, 1997).

Exploiting SMT Models for Precise Counterexamples

When SMT solvers such as Z3 (de Moura and Bjørner, 2008) fail to discharge a proof obligation they can often produce a model that demonstrates the falsity of the statement. The produced models are, however, partial and not always proper counterexamples to the original logical statement; for instance when Z3 times out it will often produce bogus models. Therefore a way to check the correctness of models before displaying them to the user is highly desirable. Moreover, the model has to be converted from logic back to F⋆ values. In the past, I have successfully devised such a conversion for a simple data processing language with semantic subtyping (Bierman et al., 2012).

Exploiting SMT models for producing precise F⋆ counterexamples seems, however, more difficult, since the logical encoding from F⋆ to the first-order logic of the SMT solver is more complex. Moreover, we will need to deal with the backtracking aspect of the F⋆ type-checker, which means that obtaining a counterexample on one branch of the (imaginary) type derivation is not enough for obtaining a counterexample for the whole derivation. Instead we can try to find a way to combine the counterexamples obtained on the various branches. If this turns out too hard, we can still resort to displaying the typing derivation to the user, with counterexamples for the sub-branches that failed.

Probabilistic Relational Verification in F⋆

By default, verification in F⋆ is not relational or probabilistic. While F7, a precursor of F⋆, was used to analyze security protocol implementations, such as TLS (Bhargavan et al., 2013), the verification style employed (Fournet et al., 2011) is often rather indirect. Direct support for relational and probabilistic reasoning (Barthe et al., 2012, 2015; Nanevski et al., 2013) within F⋆ could greatly simplify security protocol verification. A previous attempt at achieving this was only partially successful (Barthe et al., 2014), and more research is needed for a compelling solution. Because the relational typing rules are not at all syntax directed, devising an efficient and predictable type-checking algorithm that can deal with a large class of program equivalences that includes most practically relevant examples is a challenge. One thing to try is offloading more of the work to the SMT solver. We will illustrate the expressiveness of our solution by encoding existing IFC type systems, proofs of cryptographic lemmas, and eventually full fledged protocol implementations like miTLS. On the theory side, scaling the proofs up to all the features of F⋆ will be challenging. The proofs for the previous relation probabilistic systems are based on simple denotational semantics, and do not deal with features such as dynamically allocated higher-order state and higher-order polymorphism, which are the bread and butter of F⋆. We will try to attack this challenge

using step-indexed logical relations (Bizjak and Birkedal, 2015).

Systematically Testing the Soundness of the F⋆ Implementation

Producing a (self-)certified implementation is a large amount of work. In the meanwhile, we could investigate systematically *testing* the soundness of the F⋆ implementation. Manually testing that a type-checker rejects the programs it should can be very tedious, so we will instead use a new technique I recently devised, called *polarized mutation testing*. With this technique we would only write programs that pass type-checking and then change the refinement formulas to automatically produce lots of variants, so that each of these variants is either (1) ill-typed and should fail type-checking or (2) has a stronger specification (weaker pre-conditions, stronger post-conditions). From a well-typed program we can probably automatically generate hundreds of such polarized mutants and check whether they type-check or not. The ones that do type-check (usually much fewer) have to be manually categorized as (1) ill-typed in which case we found a soundness bug, or (2) well-typed, in which case we found a stronger spec for the program. I have used this idea with great success on testing and guiding the design of dynamic IFC mechanisms (the 39 bugs from Figure 21 in Hriţcu et al. (2014) are all automatically introduced this way), and testing the F⋆ implementation seems another good place to apply this idea. Polarized mutation testing is further explained in a separate document on testing.[3]

A Certifying Compiler from F⋆ to CompCert C

The trusted computing base (TCB) of F⋆ is very large, and includes the compilers and runtime systems of F# or OCaml, all of which are very complex (e.g., for F# the .NET framework). This is unsatisfactory, since F⋆ is targeted primarily at security-critical applications in which every bug potentially leads to a security violation. For instance one of the most promising applications of F⋆ is the analysis of security protocol implementations, such as SSL/TLS (Bhargavan et al., 2013), and, as the Heartbleed vulnerability has illustrated, there is no room for bugs in such pervasively used protocols. It is thus crucial that the correctness and security guarantees obtained by the F⋆ type-checker hold not only at the F⋆ source code, but are preserved all the way to the machine code level.

This topic is aimed at developing a trustworthy compiler for F⋆ producing efficient machine code that satisfies the high-level properties established by the F⋆ type-checker. We will achieve this with reasonable effort by building a *certifying compiler* from F⋆ to CompCert C. Each run of this compiler will produce not just a C program but also a certificate, attesting that this program enjoys the properties verified by the F⋆ type-checker. The certificate will be validated by an independent checker (e.g., the Coq kernel). If certificate validation succeeds we compile the C program using the CompCert C compiler to obtain the binary. Because CompCert is a verified compiler, it is guaranteed to preserve all trace properties of any C program with well-defined behavior, and thus also the ones established by typing and preserved by our certifying F⋆ compiler. To further reduce the TCB, our compiler will rely on region based memory management (Tofte et al., 2004). This will allow F⋆ code to run even without a garbage collector, which also has other advantages such as more predictable execution times, which eliminates one possible source of side-channel attacks on cryptography.

We believe this is achievable with a reasonable amount of effort; in contrast fully verifying an ML compiler is still a daunting task (Kumar et al., 2014) Moreover, checking the certificates will quickly find most bugs in the new compiler, while still allowing us to evolve F⋆ at a fast pace. Finally, We might be able to leverage some of the infrastructure (e.g., intermediate languages, certified garbage collector (McCreight et al., 2010), etc.) that will be produced by the new CertiCoq project.

References

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

N. Amin, K. R. M. Leino, and T. Rompf. Computing with an SMT solver. In *8th International Conference on Tests and Proofs*, pages 20–35, 2014.

[3] `http://prosecco.gforge.inria.fr/personal/hritcu/students/topics/2015/quick-chick.pdf`

B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The PoplMark challenge. In *18th International Conference on Theorem Proving in Higher Order Logics*, pages 50–65, 2005.

B. Barras and B. Werner. Coq in Coq. Technical report, 1997.

G. Barthe, B. Grégoire, and S. Z. Béguelin. Probabilistic relational Hoare logics for computer-aided security proofs. In *11th International Conference on Mathematics of Program Construction*, volume 7342 of *Lecture Notes in Computer Science*, pages 1–6. Springer, 2012.

G. Barthe, C. Fournet, B. Grégoire, P. Strub, N. Swamy, and S. Zanella Béguelin. Probabilistic relational verification for cryptographic implementations. In S. Jagannathan and P. Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 193–206. ACM, 2014.

G. Barthe, M. Gaboardi, E. J. Gallego Arias, J. Hsu, A. Roth, and P.-Y. Strub. Higher-order approximate relational refinement types for mechanism design and differential privacy. To appear in POPL, 2015.

K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P.-Y. Strub. Implementing TLS with verified cryptographic security. In *IEEE Symposium on Security & Privacy (Oakland)*, pages 445–462, 2013.

G. M. Bierman, A. D. Gordon, C. Hriţcu, and D. Langworthy. Semantic subtyping with an SMT solver. *Journal of Functional Programming (JFP)*, 22(1):31–105, Mar. 2012.

A. Bizjak and L. Birkedal. Step-indexed logical relations for probability. To appear in FOSSACS, 2015.

C. Coquand, M. Takeyama, and D. Synek. An Emacs interface for type directed support constructing proofs and programs. European Joint Conferences on Theory and Practice of Software, ENTCS, 2006.

L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

C. Fournet, M. Kohlweiss, and P. Strub. Modular code-based cryptographic verification. In Y. Chen, G. Danezis, and V. Shmatikov, editors, *Proceedings of the 18th ACM Conference on Computer and Communications Security*, pages 341–350. ACM, 2011.

C. Fournet, N. Swamy, J. Chen, P. Dagand, P. Strub, and B. Livshits. Fully abstract compilation to JavaScript. In *Symposium on Principles of Programming Languages, POPL*, pages 371–384. ACM, 2013.

A. Guha, M. Fredrikson, B. Livshits, and N. Swamy. Verified security for browser extensions. In *Symposium on Security and Privacy, S&P*, pages 115–130. IEEE Computer Society, 2011.

C. Hriţcu, L. Lampropoulos, A. Spector-Zabusky, A. A. de Amorim, M. Dénès, J. Hughes, B. C. Pierce, and D. Vytiniotis. Testing noninterference, quickly. arXiv:1409.0393; Submitted to Special Issue of Journal of Functional Programming for ICFP 2013, Sept. 2014.

R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: a verified implementation of ML. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, pages 179–192. ACM, 2014.

A. McCreight, T. Chevalier, and A. P. Tolmach. A certified framework for compiling and executing garbage-collected languages. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming*, pages 273–284. ACM, 2010.

A. Nanevski, A. Banerjee, and D. Garg. Dependent type theory for verification of information flow and access control policies. *ACM Transactions on Programming Languages and Systems*, 35(2):6, 2013.

B. C. Pierce, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjöberg, and B. Yorgey. *Software Foundations*. Electronic textbook, 2013.

P. Strub, N. Swamy, C. Fournet, and J. Chen. Self-certification: bootstrapping certified typecheckers in F* with Coq. In *Symposium on Principles of Programming Languages, POPL*, pages 571–584. ACM, 2012.

N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *International Conference on Functional Programming, ICFP*, pages 266–278, 2011.

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

N. Swamy, J. Weinberger, C. Schlesinger, J. Chen, and B. Livshits. Verifying higher-order programs with the Dijkstra monad. In *Conference on Programming Language Design and Implementation, PLDI*, pages 387–398. ACM, 2013b.

N. Swamy, C. Fournet, A. Rastogi, K. Bhargavan, J. Chen, P.-Y. Strub, and G. M. Bierman. Gradual typing embedded securely in JavaScript. In *Symposium on Principles of Programming Languages (POPL)*, pages 425–438, 2014.

M. Tofte, L. Birkedal, M. Elsman, and N. Hallenberg. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation*, 17(3):245–265, 2004.