Semantic Foundations for F⋆

Introduction

F⋆ [23] 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⋆ type-checker implements a weakest-precondition calculus [24] to produce first-order logic formulas that are automatically discharged using the Z3 SMT solver [7]. The original F⋆ implementation [21] has been successfully used to verify nearly 50,000 lines of code, including cryptographic protocol implementations [21, 2], web browser extensions [13, 24], cloud-hosted web applications [21], and key parts of the F⋆ system itself [20]. F⋆ has also been used for formalizing the semantics of other languages, including JavaScript and a compiler from a subset of F⋆ to JavaScript [12] and TS⋆, a secure subset of TypeScript [22]. 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 [21] was a quite successful, it had reached its limits in terms of expressiveness. Over the past 2 years, we have completely redesigned and reimplemented F⋆ [23], 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 [26], Dafny [16], and Why3 [11] 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 [25] 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 the redesigned F⋆ verification system shows great promise in practice, its theory is more difficult and interesting than before and to a large extent work in progress. Many challenging problems remain for which operational techniques seem insufficient, and for which deeper semantic techniques seem required. We will follow three research directions:

1. Providing a solid formal foundation for the use of **F⋆ as a logic** by constructing semantic **models** and proving **consistency**;

2. **Dijkstra monads "for free"**, i.e., deriving correct-by-design, specification-level monads for efficient automatic reasoning from expression-level monads;

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

1 F⋆ as a logic, models and consistency

While the metatheory of F⋆ is still work in progress, we have formally studied two subsets of F⋆ [23]. For pF⋆(pico-F⋆), a small pure fragment of F⋆, the author of this topic proposal has proved weak normalization and logical consistency using logical relations. We would like to gradually extend pF⋆and its consistency proof to the pure fragment of F⋆ itself. However, a consistency proof for a language as complex as F⋆ seems out of reach for plain logical relations and will require more advanced proof techniques. We would thus like to build a consistency proof for F⋆ taking inspiration in the established semantic model constructions for existing type theories [14] such as Martin-Löf type theory [6, 5]. We hope that this will lead to a more abstract and concise characterization of the language, compared to the syntactic presentation. This should also lead to new insights into the connection between F⋆ and established systems like Coq.

2 Dijkstra monads for free

One of the key distinguishing features of F⋆ compared to proof assistants like Coq is the first-class treatment of effects in a way that enables efficient automatic reasoning. Dijkstra monads [24, 23] 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 indirect (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.

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

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

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 want to provide a more abstract definition of Dijkstra monads and a generic proof that shows once and for all that our 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 [9, 10]). 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 [2].

Category theory can provide interesting insights into the derivation of Dijkstra monads. While the connection between monads and category theory is well established [17], the categorical understanding of Dijkstra monads is still at a very early stage. The only attempt at categorically explaining Dijkstra monads is a recent work by Jacobs [15], which is, however, not general enough to apply to our setting, since it can only deal with variations of the state monad. We have recently set up a tentative categorical definition of an abstract translation using relative monads [1], which can apparently be instantiated to obtain both a definition of our concrete continuation-passing style translation and a definition of Dijkstra monads. This idea still needs thorough study.

3 Hiding effects and 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.* [4, 3] uses proof-relevant logical relations (i.e., setoids) for overcoming this. Other previous models of hidden state [18, 19] could also be helpful, but that work targeted languages without control of effects à la F⋆.

References

[1] T. Altenkirch, J. Chapman, and T. Uustalu. Monads need not be endofunctors. *Logical Methods in Computer Science*, 11(1), 2015.

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

[3] N. Benton, M. Hofmann, and V. Nigam. Proof-relevant logical relations for name generation. *TLCA*. 2013.

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

[5] S. Castellan. Dependent type theory as the initial category with families. Internship Report, 2014.

[6] P. Clairambault and P. Dybjer. The biequivalence of locally cartesian closed categories and martin-löf type theories. *Mathematical Structures in Computer Science*, 24(6), 2014.

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

[10] A. Filinski. Monads in action. In M. V. Hermenegildo and J. Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*. 2010.

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

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

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

[14] B. Jacobs. *Categorical logic and type theory*, volume 141. Elsevier, 1999.

[15] B. Jacobs. Dijkstra and hoare monads in monadic computation. *Theor. Comput. Sci.*, 604:30–45, 2015.

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

[17] E. Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*. 1989.

[18] F. Pottier. Hiding local state in direct style: A higher-order anti-frame rule. In *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24-27 June 2008, Pittsburgh, PA, USA*. 2008.

[19] J. Schwinghammer, L. Birkedal, F. Pottier, B. Reus, K. Støvring, and H. Yang. A step-indexed kripke model of hidden state. *Mathematical Structures in Computer Science*, 23(1):1–54, 2013.

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

[21] 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.

[22] 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.

[23] 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.

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

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

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