The Formal Semantics and Evolution of the F* Verification System

October 11, 2018

Advisor: Cătălin Hriţcu ⟨catalin.hritcu@inria.fr⟩

Institition: INRIA Paris, Prosecco Team

Location: 2 rue Simone Iff, 75012 Paris, France

Language: English

Existing skills or strong desire to learn:

- programming language semantics, dependent types, type theory, effects, monads, mechanized metatheory
- 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);

Introduction

F* [2, 3, 17, 34] is a general-purpose functional programming language with effects aimed at program verification. It puts together the automation of an SMT-backed deductive verification tool with the expressive power of a proof assistant based on dependent types. After verification, F* programs can be extracted to efficient OCaml, F#, C or Wasm code [30].

F*'s type system includes dependent types, monadic effects, refinement types, and a weakest precondition calculus. Together, these features allow expressing precise and compact specifications for programs, including functional correctness and security properties. The F* type-checker allows to prove that programs meet their specifications using a combination of SMT solving and interactive tactic proofs [23].

F* is developed collaboratively by Microsoft Research, Inria Paris, and the community at large. It is open source[1] and implemented in a common subset of F# and F* itself. The main ongoing use case of F* is building a verified, drop-in replacement for the whole HTTPS stack in Project Everest [9], including the underlying cryptographic primitives [38].

While the F* verification system shows great promise in practice, its theory is interesting, difficult, and a large extent work in progress. Many challenging conceptual problems remain to be solved, many of which can directly inform the further evolution and design of the language:

1. Working out more of the **F* semantics and metatheory**
2. **Semantically deriving Dijkstra monads**, i.e., specification-level monads for program verification;
3. **Making F*'s effect system more flexible**, by supporting tractable forms of effect polymorphism and allowing some of the effects of a computation to be hidden if they do not impact the observable behavior;
4. **Relational reasoning in F***: devising scalable verification techniques for properties of multiple program executions (e.g., confidentiality, noninterference) or of multiple programs (e.g., program equivalence).

[1] `https://github.com/FStarLang/FStar`

An excellent Student or PostDoc could contribute to any of these topics, which are further detailed in the reminder of this document. This list is not exhaustive, and if you are interested you should get in contact and we can work together to find a topic that is in sync with your interests and expertise.

1 F* semantics and metatheory

The semantics and metatheory of F* are interesting and still work in progress. While our previous work formally investigated several subsets of F* [2, 3, 16, 34], to be tractable these subsets simplified away several F* features that are in fact interesting. We would like to extend the theory of F* to cover such features, with a particular focus on solving challenging conceptual issues that could directly inform the further evolution and design of the language.

One important issue that is easy to explain is that in F* the type `list nat` is not a subtype of `list int`, even though `nat` is a subtype of `int`. We have the intuition that such subtyping on datatypes could be safely allowed if we additionally prevent pattern matching on datatype parameters (since allowing these parameters to be projected out immediately leads to unsoundness: `https://github.com/FStarLang/FStar/issues/65`). However, validating that this is indeed sound requires working out the metatheory of an extensional type theory with inductive types, refinement types, and subtyping. One could potentially take inspiration in the new universe cumulative inductive types in Coq [36] and other recent work on subtyping for inductive types [1].

Another interesting issue to formally investigate is the treatment of F*'s propositional universe (`prop`) and its squashing of types [28]. While several encodings of `prop` were investigated, they all rely on axioms (e.g., for eliminating squash types) that need to be semantically justified to ensure the logical consistency of F*. Moreover, practical F* developments also rely on further axioms such as (not entirely standard variants of) functional and propositional extensionality, whose soundness also needs to be semantically justified. One way to approach this would be to prove consistency taking inspiration in established semantic model constructions for existing type theories [19] such as Martin-Löf type theory [13, 14]. This could shed more light on the connection between F* and established systems like Coq, Agda or the PRL family. Beyond `prop` and squashing, the F*-specific challenges include equality reflection, semantic termination, subtyping, etc.

2 Dijkstra monads, semantically

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 [34, 35] are the mechanism by which F⋆ efficiently computes verification conditions, generically for any effect. While the verification condition generation algorithm is generic, for each effect one needs to define monadic operations used to combine weakest preconditions (return, bind, and any effect-specific actions). Our recent work [3] shows that for a certain class of effects it is possible to derive these operations automatically and prove their correctness generically, starting from a monadic definition of the effect itself. The class of effects for which we can currently derive Dijksta monads is, however, restrictive: the monadic definition of the effect has to fit a small syntactic subset of F⋆ in order for our syntactic translation and generic logical relation proof to apply. While the class of supported effects includes state and exceptions, it excludes other important effects such as non-termination, IO, nondeterminism, and probabilities.

We are currently working to lift these limitations using insights from category theory. While the connection between monads and category theory is well established [25], 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 [20], which is, however, not general enough to apply to our setting, since it can only deal with variations of the state monad. Works on monadic predicate transformers [18] show the relevance of algebra structures on the type of propositions for a commutative monad, a datum that can reasonably be represented in F⋆. Leveraging on recent work by Rauch *et al.* [31] that treats the case of exceptions, we have recently developed a prototype framework encompassing both the algebra-based predicates transformer approach of Hasuo *et al.* [18] and our previous monad transformer approach to Djikstra monads [3]. This new framework is likely to provide a practical solution for reasoning about a wide class of monadic effects. The soundness of the framework will probably rely on a general semantic notion of Djikstra monad yet to be developed.

3 More flexible effect system

Another interesting research direction is making the F⋆ effect system more flexible. A first limitation we could try to lift is the lack of effect polymorphism, which means that higher-order functions such as `List.map` are duplicated many times, once for each effect their function argument might have. With the recent introduction in F⋆ of support for type classes and for resolving implicit arguments using tactics, one can hope to also support the overloading (i.e., ad-hoc polymorphism) of functions such as `map` by parameterizing them over a Dijkstra monad and verifying their code in a generic fashion using tactics to apply the properties of Dijkstra monads.

A second limitation is that at the moment, effects in F⋆ are syntactic: if any sub-computation triggers a certain effect then the whole computation is tainted with that effect. An ambitious project would be to relax this when the effect of a computation is unobservable to its context [11, 22, 27]. 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. Proving that we can soundly forget such non-observable effects proved to be hard already for state [6, 7, 37], many works targetting the case of hidden state [29, 33]. In the ideal case, we should be able to forget most effects, provided we prove in F⋆ that these effects do not matter (which often requires relational reasoning, which is a separate topic below).

In the longer term it would be nice to also be able to prove termination extrinsically and then hide the divergence effect. One first step towards this could be to make the purity and divergence effects less primitive in F⋆, by encoding partiality as a monad supporting fixpoints [12, 15, 24] and on top of this defining Dijkstra monads for total and partial correctness.

4 Relational reasoning in F⋆

By default, F⋆ reasons about the execution of each effectful computation intrinsically, when the computation is defined, by inferring a (unary) weakest precondition (WP) for it. In recent work [17], we showed that by exposing the representation of the effect via monadic reification we can also reason about effectful computations extrinsically, after the fact. This allows proving *relational properties*, describing multiple executions of one or more programs. We evaluated this idea by encoding a variety of relational program analyses, including information flow control, program equivalence and refinement at higher order, correctness of program optimizations and game-based cryptographic security.

While monadic reification worked reasonably well on simple examples, two serious challenges remain: (1) monadic reification seems very difficult to soundly combine with reasoning about monotonic state [2], which at the moment crucially relies on the effect being treated abstractly, and the programmer not having direct access to the heap in user-programs; and (2) monadic reification seems hard to scale to large verification efforts such as miTLS [9, 10], which currently relies on meta-level arguments involving parametricity that are not fully formal, but that are at least more modular. One way to approach problem (1) would be to figure out how to safely use reification to expose the update monad structure [4] underpinning monotonic state. The challenge here is that the most natural approach inspired by hybrid modal logics [5, 32] (in which one also "reifies" the modal logic specifications of monotonic state) inevitably leads to reification becoming a whole program (and typing judgement) transformation, which of course does not scale well. Another way to approach this could be to keep the monadic representation abstract but do the relational reasoning not at the level of expressions but at the level of WPs; which can work if the WPs involved fully specify the result of the computation. Finally, for approaching the modularity issue (2) we could try to combine relational reasoning by reification or on WPs with internalized reasoning about parametricity [8, 21, 26].

References

[1] A. Abel, A. Vezzosi, and T. Winterhalter. Normalization by evaluation for sized dependent types. *PACMPL*, 1(ICFP):33:1–33:30, 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] D. Ahman, C. Hriţcu, K. Maillard, G. Martínez, G. Plotkin, J. Protzenko, A. Rastogi, and N. Swamy. Dijkstra monads for free. *POPL*. 2017.

[4] D. Ahman and T. Uustalu. Update monads: Cointerpreting directed containers. *TYPES*, 2013.

[5] C. Areces and B. ten Cate. *Handbook of Modal Logic*, chapter Hybrid Logics, pages 821–868. Elsevier, 2007.

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

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

[8] J. Bernardy, P. Jansson, and R. Paterson. Parametricity and dependent types. In P. Hudak and S. Weirich, editors, *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*. 2010.

[9] 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.

[10] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, J. Pan, J. Protzenko, A. Rastogi, N. Swamy, S. Zanella Béguelin, and J. K. Zinzindohoue. Implementing and proving the TLS 1.3 record layer. *IEEE Security & Privacy*, 2017.

[11] L. Birkedal, G. Jaber, F. Sieczkowski, and J. Thamsborg. A kripke logical relation for effect-based program transformations. *Inf. Comput.*, 249:160–189, 2016.

[12] A. Bove and V. Capretta. Modelling general recursion in type theory. *Mathematical Structures in Computer Science*, 15(4):671–708, 2005.

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

[14] 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.

[15] F. Faissole and B. Spitters. Synthetic topology in homotopy type theory for probabilistic programming (extended abstract). accepted for PPS17 and CoqPL17, 2017.

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

[17] N. Grimm, K. Maillard, C. Fournet, C. Hriţcu, M. Maffei, J. Protzenko, T. Ramananandro, A. Rastogi, N. Swamy, and S. Zanella-Béguelin. A monadic framework for relational verification: Applied to information security, program equivalence, and optimizations. *CPP*, 2018.

[18] I. Hasuo. Generic weakest precondition semantics from monads enriched with order. *Theor. Comput. Sci.*, 604:2–29, 2015.

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

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

[21] N. R. Krishnaswami and D. Dreyer. Internalizing relational parametricity in the extensional calculus of constructions. In S. R. D. Rocca, editor, *Computer Science Logic 2013 (CSL 2013), CSL 2013, September 2-5, 2013, Torino, Italy*. 2013.

[22] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In J. Ferrante and P. Mager, editors, *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*. 1988.

[23] G. Martínez, D. Ahman, V. Dumitrescu, N. Giannarakis, C. Hawblitzel, C. Hriţcu, M. Narasimhamurthy, Z. Paraskevopoulou, C. Pit-Claudel, J. Protzenko, T. Ramananandro, A. Rastogi, and N. Swamy. Meta-F*: Proof automation with SMT, tactics, and metaprograms. arXiv:1803.06547, 2018.

[24] C. McBride. Turing-completeness totally free. In R. Hinze and J. Voigtländer, editors, *Mathematics of Program Construction - 12th International Conference, MPC 2015, Königswinter, Germany, June 29 - July 1, 2015. Proceedings*. 2015.

[25] E. Moggi. Computational lambda-calculus and monads. *LICS*. 1989.

[26] G. Moulin. *Internalizing Parametricity*. PhD thesis, Chalmers University of Technology, Gothenburg, Sweden, 2016.

[27] D. A. Naumann. Observational purity and encapsulation. *FASE*, 2005.

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

[29] F. Pottier. Hiding local state in direct style: A higher-order anti-frame rule. *LICS*. 2008.

[30] 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.

[31] C. Rauch, S. Goncharov, and L. Schröder. Generic hoare logic for order-enriched effects with exceptions. *WADT*, 2016.

[32] J. Reed. *A Hybrid Logical Framework*. PhD thesis, Carnegie Mellon University, 2009.

[33] 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.

[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] A. Timany and M. Sozeau. Cumulative inductive types in Coq. In H. Kirchner, editor, *3rd International Conference on Formal Structures for Computation and Deduction, FSCD 2018, July 9-12, 2018, Oxford, UK*. 2018.

[37] A. Timany, L. Stefanesco, M. Krogh-Jespersen, and L. Birkedal. A logical relation for monadic encapsulation of state: proving contextual equivalences in the presence of runST. *PACMPL*, 2(POPL):64:1–64:28, 2018.

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