Verifying non-terminating programs with IO in F*

Cezar-Constantin Andrici[1], Théo Winterhalter[1],
Cătălin Hriţcu[1], and Exequiel Rivas[2]

[1] MPI-SP, Bochum, Germany [2] Tallinn University of Technology, Estonia

Abstract

We present a work-in-progress definition of an F* effect for writing, specifying, and verifying functional correctness for higher-order programs with input-output. We then extend this effect to support potentially non-terminating programs. Crucially we do not stop at partial correctness where the specification only concludes about terminating runs, but instead rely on infinite traces to also reason about infinite runs. We aim to use this to verify meaningful properties of infinite loops, such as the main loop of a web server. As F* does not natively support coinduction, we encode specifications about infinite traces using an impredicative encoding of an infinite loop at the level of specifications.

Statically verifying functional correctness of programs with input-output (IO) is difficult and typically involves talking about *traces*, *i.e.* the stream of IO events that occur during a specific run of the program. Ideally, one would like to be able to write the program, specify it, and verify it in the same language. Furthermore, one would hope to be able to leverage a certain level of automation to alleviate the proof burden. In this spectrum, the proof-oriented programming language F* [SHK+16] is a good candidate in that it makes verification of programs practical thanks to a collection of features that range from SMT-based automation to user-defined effects with specifications. We take advantage of this to define[1] a new effect in F* that supports verification of trace properties of IO programs. Beyond the definition of such an effect, we strive to make it practical by tuning it to benefit as much as possible from F*'s automation capabilities. In the short term, the goal is to verify a simple web server.

Web servers are an instance of programs with IO for which running indefinitely is the expected behaviour. When verifying such programs we have to give a meaningful specification to an infinite loop. In this setting, partial correctness specifications, which only conclude about terminating runs, are not enough. Extending our effect to take into account infinite runs requires us to considerably complicate the specifications so that they can conclude in both terminating and non-terminating cases. Thankfully, in practice, such as in the case of a web server, most parts of the program are in fact terminating and typically only the main loop is non-terminating. We take advantage of this and of F*'s sub-effecting mechanism and introduce two effects: one for terminating IO computations; and one which also allows non-termination. Terminating programs can then be verified in the simpler terminating setting where automation is more efficient, and they can later be lifted to the more general effect to be used in the main loop.

Terminating code. Following *Dijkstra Monads for All* [MAA+19], we first define an F* effect for terminating IO computations by defining a computational monad, a specification monad, and a monad morphism between the two. For our computational monad we use a free monad variant:[2]

```
type free (sg : signature) (a : Type) : Type =
| Ret : a -> free sg a
| Call : ac:sg.act -> x:sg.arg ac -> k:(sg.res x -> free sg a) -> free sg a
```

The computational monad is parametrised by a signature `sg` which contains a type of actions `ac : sg.act` that take arguments `x : sg.arg ac` and return results of type `sg.res x` which may depend on the argument `x`. These actions correspond to the IO operations such as opening files, reading from, writing to and closing them. For instance, a program reading a string from a file descriptor `fd` and returning its length would be encoded as:

[1] Available at https://github.com/andricicezar/fstar-io/tree/hope-submission.

[2] A constructor of our free monad is omitted here. We explain it in *Partial Dijkstra Monads for All* [WAH+22].

```
Call Read fd (fun s -> Ret (length s))
```

We use the `Call` constructor to invoke the `Read` action which takes file descriptor `fd` as argument and *continues* with resulting string `s` whose length we return using `Ret`.

Thankfully, in F* we can rely on the user-defined effect mechanism [RMF+21] and—instead of writing the above—we can write the direct-style code below:

```
let file_length (fd : file_descr) : IO nat (requires is_open fd) (ensures p) =
  length (read fd)
```

This example also showcases the proof-oriented aspect of F*. Indeed, not only are we specifying that the program returns a natural number while using the `IO` effect, but we also state that it should only be called on an *open* file descriptor. In this case, we present the effect specification using Hoare-style pre- and post-conditions. The pre-condition is a predicate that must be verified by the history of the program, *i.e.* the list of events that happened before it was run. `is_open fd` will for instance check that an `Open` event occurred for `fd` and that it was not followed by a `Close` event. The post-condition is a predicate on the same history, but also on the list of newly emitted events and on the final value. For instance, we could take `p` to state that there exists a string `s` whose length is the result `r` and such that the only emitted states that a read was performed on file `fd` resulting in `s`:

```
let p = fun hist tr r -> exists s. tr == [ ERead fd s ] /\ r == length s
```

In practice, we found out that it is much easier to write predicates if the history of events is kept backwards (from recent to old). This also proved to work well with the SMT and we were able to use this effect to verify a simple terminating web server.

Non-terminating code. In order to support non-termination in our effect, we extend our free monad with an extra constructor corresponding to unbounded iteration[3], from which we can define the following operation in our free monad:

```
iter (f : b -> free sg (either b a)) (i : b) : free sg a
```

The idea is that `iter f i` first applies the loop body `f` to `i` and when this returns `Inl j`, the loop continues with `iter f j`; when it returns `Inr x` the whole loop returns `x : a`.

To specify `iter`, we take inspiration from related work entitled *Dijkstra Monads for Ever* (DM4Ever) [SZ21], which uses interaction trees [XZH+20], a coinductive data-type in Coq to represent their potentially infinite programs and their runs. They define `iter` in Coq using the following co-fixed-point:

```
iter f i = match f i with Inl j -> tau ; iter f j | Inr x -> Ret x
```

They need to add a silent step—called `tau`—in order to ensure the co-fixed-point is productive. A program which loops silently is thus an infinite sequence of `tau`.

We reproduce essentially the same specifications as them, with two important differences: (i) instead of just proving various program logic rules such as loop invariants, we provide a specification combinator `w_iter` to compute the specification `w_iter w i` of `iter f i` from a specification `w` of its body `f`; (ii) we do not attempt to have an unfolding equation for `iter` and instead only have it for `w_iter`.

F* does not support coinduction natively, but we can nevertheless easily represent streams of events as sequences indexed by natural numbers: `stream a = nat -> a`. More interestingly, we define `w_iter f i` using an impredicative encoding of a co-fixed-point which reflects the equation above at the level of specifications:

```
w_iter w i = match w i with Inl j -> w_tau ; w_iter w j | Inr x -> w_ret x
```

[3] This works as a purely syntactic counterpart for a dagger operator ($-^\dagger$) in the spirit of monads with iterative structure [GMR16, GSRP17].

Ongoing and future work. Several challenges remain to make this work practical. We list here what we are currently doing as well as what we plan to do regarding this work.

- As a case study, we are verifying a simple stateless web server that serves files over HTTP. We managed to verify safety properties about the terminating computations relatively easily by taking advantage of the SMT solver. We showed safety properties about the loop body in the terminating effect, and we plan to make use of sub-effecting to lift it to the non-terminating effect.
- We are working to make non-termination verification practical in F* by overcoming automation challenges. Mainly, our specifications have to distinguish between finite and infinite runs of a program, which can lead to exponential blow-up in the generated verification conditions.
- So far, we have only tested our work verifying properties of finite prefixes of infinite traces. It remains to see how we fare on more global properties of infinite runs such as liveness properties.
- We would like to extend our monad to support more effects such as state and exceptions in order to verify a realistic web server.
- This work is an important component of another direction we are working on. Verifying a web server in F* is difficult, but then it is extracted to another language such as OCaml and linked with OCaml code. Thus, all the security guarantees are lost. Our goal is to create a mechanism through which a programmer is able to integrate a statically verified component with an unverified piece of code, and still be able to prove safety properties about the whole program — thus obtaining secure F*-ML interoperability [And20, And21].

Related work. We verify monadic computations represented by a free monad parameterized by the signature of the actions using Hoare-style specifications.

In the same style, by using Hoare-style specification, the FreeSpec framework [LR20] for Coq, Ynot library [MMW11] for Coq and Penninckx *et al* [PJP15, PTJ19] present different ways to verify partial correctness specification for IO computations — while we verify full correctness specification. The FreeSpec framework uses almost the same computational monad as us, but allows choosing the internal state for verification, *e.g.* the trace in our case. The Ynot library uses a similar setting to us for static verification in that they use properties that are defined over traces of events. Penninckx *et al* present an approach to verifying IO programs using Petri nets and separation logic, implemented in VeriFast [JSP+11].

Interaction trees [XZH+20, SZ21] were shown to be a great fit to verify non-terminating impure computations in Coq. Interaction trees are not obvious to implement without coinductive types, yet our work enables verification of non-terminating IO computations without the need of coinductive types. Interaction trees were used to verify an HTTP Key-Value Server that is part of CertiKOS [ZHK+21]. The web server is written in C and the trace properties were verified in Coq. Coq requires to apply tactics manually to prove verification goals. Our work targets to simplify this kind of use-case by taking advantage of the SMT automation.

References

[And20] Cezar-Constantin Andrici. Hybrid enforcement of IO trace properties. *ICFP Student Research Competition*, 2020.

[And21] Cezar-Constantin Andrici. Secure F*-ML interoperability for IO programs. Master's thesis, 2021. Advised by Ştefan Ciobâcă and Cătălin Hriţcu.

[GMR16] Sergey Goncharov, Stefan Milius, and Christoph Rauch. Complete Elgot monads and coalgebraic resumptions. *Electronic Notes in Theoretical Computer Science*, 325:147–168, 2016. The Thirty-second Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXII).

[GSRP17] Sergey Goncharov, Lutz Schröder, Christoph Rauch, and Maciej Piróg. Unifying guarded and unguarded iteration. In *Proceedings of the 20th International Conference on Foundations of Software Science and Computation Structures - Volume 10203*, page 517–533, Berlin, Heidelberg, 2017. Springer-Verlag.

[JSP+11] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and java. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, volume 6617 of *Lecture Notes in Computer Science*, pages 41–55. Springer, 2011.

[LR20] Thomas Letan and Yann Régis-Gianas. Freespec: specifying, verifying, and executing impure computations in Coq. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 32–46. ACM, 2020.

[MAA+19] Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Cătălin Hriţcu, Exequiel Rivas, and Éric Tanter. Dijkstra monads for all. *PACMPL*, 3(ICFP):104:1–104:29, 2019.

[MMW11] Gregory Malecha, Greg Morrisett, and Ryan Wisnesky. Trace-based verification of imperative programs with I/O. *J. Symb. Comput.*, 46(2):95–118, 2011.

[PJP15] Willem Penninckx, Bart Jacobs, and Frank Piessens. Sound, modular and compositional verification of the input/output behavior of programs. In Jan Vitek, editor, *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 158–182. Springer, 2015.

[PTJ19] Willem Penninckx, Amin Timany, and Bart Jacobs. Abstract I/O specification. *CoRR*, abs/1901.10541, 2019.

[RMF+21] Aseem Rastogi, Guido Martínez, Aymeric Fromherz, Tahina Ramananandro, and Nikhil Swamy. Programming and proving with indexed effects, July 2021. In submission.

[SHK+16] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, et al. Dependent types and multi-monadic effects in F*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 256–270, 2016.

[SZ21] Lucas Silver and Steve Zdancewic. Dijkstra monads forever: termination-sensitive specifications for interaction trees. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–28, 2021.

[WAH+22] Théo Winterhalter, Cezar-Constantin Andrici, Cătălin Hriţcu, Kenji Maillard, Guido Martínez, and Exequiel Rivas. Partial dijkstra monads for all. TYPES, 2022.

[XZH+20] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.*, 4(POPL):51:1–51:32, 2020.

[ZHK+21] Hengchu Zhang, Wolf Honoré, Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. Verifying an HTTP key-value server with interaction trees and VST. In Liron Cohen and Cezary Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference)*, volume 193 of *LIPIcs*, pages 32:1–32:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.