Towards formally secure compilation of verified F⋆ programs against unverified ML contexts

(Extended Abstract)

Cezar-Constantin Andrici[1] Danel Ahman[2] Cătălin Hriţcu[1] Guido Martínez[3]
Abigail Pribisova[1,4] Exequiel Rivas[5] Théo Winterhalter[6]

[1]MPI-SP, Bochum, Germany [2]University of Tartu, Estonia [3]Microsoft Research, Redmond, WA, USA
[4]MPI-SWS, Saarbrücken, Germany [5]Tallinn University of Technology, Estonia [6]Inria Saclay, France

We are working towards a formally secure compilation framework to compile verified F⋆ programs to an ML language. The framework, itself written and verified in F⋆, aims to compile verified F⋆ programs and link them with unverified ML programs *securely*, so that any unverified code *cannot* inadvertently or maliciously break the internal invariants of the verified code. In previous work on this project we have built formally secure compilation frameworks [4, 5] between shallowly embedded subsets of F⋆—i.e., we used shallow embeddings not only for the verified code (which is standard in F⋆) but also for the unverified code. To compile further to OCaml, we relied on F⋆'s extraction mechanism, which is, however, unverified. In this extended abstract, we present ongoing work on extending our framework with a formally secure compilation step to an ML language deeply embedded in F⋆, thus achieving end-to-end security guarantees.

This is challenging, since compiling to a deeply embedded ML language generally involves *quotation*, which is a meta-program taking a shallowly embedded program and returning a deep embedding of it. If compilation uses quotation, then end-to-end verification would involve a lot of complex meta theory, including verifying quotation—proof that was started for Rocq, but never finished.[1] For our goal of building an end-to-end secure compiler for F⋆ this approach is a non-starter because there is not enough formalized meta theory for F⋆ and such a formalization would be very challenging, since F⋆ has many verification features that go beyond standard type theory and also a much larger core than Rocq.

1 Relational quotation

Instead of using quotation, we propose *relational quotation*, a technique inspired from relational compilation [12, 13] and work on reflection via canonical structures [8]. Relational quotation involves a user-defined unary relation on arbitrary F⋆ values that carves out the subset of the host language that represents the shallow embedding. A derivation of the relation stands both as a quotation of a program, and *as a proof that it is the quotation of that program*, which eliminates the necessity of a formalized meta theory.

[1]Verifying quoting in MetaRocq: https://github.com/MetaRocq/metarocq/blob/coq-8.16/quotation/theories/README.md

For intuition, here we present a simplified relation C for Booleans, defined as an inductive relation:

```
type C : #a:Type → a → Type =
| Cfalse : C false
| Ctrue : C true
| CIf : #a:Type → #cond:bool → #b1:a → #b2:a →
      C cond → C b1 → C b2 →
      C (if cond then b1 else b2)
```

To see how this simple relation C works, let us take the simple program p1 and manually quote it by defining p1C.

```
let p1 = if true then false else true
let p1C : C p1 = CIf Ctrue Cfalse Ctrue
```

With this relation, we see clearly the quotation of p1 (i.e., CIf Ctrue Cfalse Ctrue). We can obtain a deep embedding by recursively defining a projection on the derivation of the relation.

The difference between standard quotation and relational quotation is that the latter *constructs* a program of type α that gets unified with the program we are quoting. In other words, the derivation constructs a program and F⋆'s typing ensures that it is a correct derivation of the program we are quoting—thus, eliminating the necessity for additional meta-theoretic proofs to verify quotation.

Automation. Above we obtained C p1 manually, but this was just for intuition. On the one hand, we can write a meta program that automatically tries to build a derivation. On the other hand, based on our experiments in F⋆, however, we did not have to write our own meta program, but **we just defined C as a type class and used type class resolution!** So we can actually quote program p1 by writing let p1C : C p1 = solve, where solve is the F⋆ tactic that calls type class resolution. It will be interesting to see if type class resolution scales with adding more features to the language.

2 Secure Compilation

We define compilation as the meta program that takes a program of type α and returns the deep embedding together with a proof. We are proving the same compilation criterion as in our previous work [4, 5], Robust Relational Hyperproperty

Preservation (RrHP), the strongest criterion of Abate et al. [1], and also stronger than full abstraction. RrHP ensures that compilation preserves observational equivalence, non-interference and trace properties. We think such a strong criterion should hold for our compiler to an ML language after adding more interesting effects (such as state and Input-Output) because F* itself is designed in the tradition of the ML family of languages.

Compilation first tries to derive the quotation of the program, this way checking if the program is part of our shallow embedding, and if it succeeds, it constructs the deeply embedded ML program and the proof. The deeply embedded program is untyped because the shallow embeddings may use dependent types. The proof, however, states that the deeply embedded program can be securely linked with arbitrary syntactically typed contexts.

We prove RrHP using one more relation, a cross-language logical relation between shallowly embedded and deeply embedded programs. This logical relation is asymmetric, since it relates shallowly embedded programs to deeply embedded ones, however, its definition is still quite standard though. The proof that a shallowly embedded program and its deep embedding are in the logical relation is done recursively on the derivation of the relation used to do quotation.

The current status is that we produce proofs for compiling simply typed pure F* programs and that are safe to link them with syntactically typed pure ML programs.

3 Towards end-to-end proofs

To be able to connect with our previous work, and be able to compile verified effectful programs, we are working on the following extensions.

Refinement types. Refinements are a core feature of F* and they allow one to have functions with pre- and post-conditions. The challenge with quoting refined programs is that the relation used to do the quotation is bottom-up, it *constructs* a program of type α, and then it unifies the new program with the one we are quoting. This means that when we are quoting a function with a pre- and a post-condition, the relation constructs the function and it also constructs the proof that the post-condition is satisfied. Intuitively, constructing the proof should be guided by the proof done for the quoted program, however, F* does not save the proofs done for refinements, thus, we do not know how to do it (e.g., if one has an x refined with p x, there is no way to project out the proof of p x). This means that one has to reprove that the quoted program satisfies the refinements. Hopefully, this can be done by the SMT. Myreen and Owens [12] face the same problem when synthesising CakeML programs from HOL functions using relational compilation.

Monadic shallow embeddings. In our previous work, we represented verified stateful and IO-performing programs using shallow embeddings that use monads. While we did not explore compiling such programs yet, previous work on relational compilation shows that such a thing is possible. Most notably, the work of Abrahamsson et al. [2] shows how to synthesize CakeML programs from HOL functions that embed references, exceptions, and IO operations.

Dependent types. We think this technique could be scaled to quotation of dependently typed programs. For that, we would probably need a mutually recursive definition between a relation for types and one for expressions. A general definition would require first-class universe polymorphism (as in Agda) which F* does not have. Since our goal is to link with ML programs, maybe one can limit the definition to a limited set of universes.

While working on these extensions, one would have to also deal with two **challenges** of relational quotation (inherited from relational compilation) about completness: (1) one is able to relate only what is expressible by a user defined relation (e.g., pattern matching is difficult to do in a general way), (2) finding the derivation is done using tactics that may loop or fail.

4 Related work

To our knowledge, there is no existing formal secure compiler for a proof-oriented languages (e.g., Rocq, Dafny, HOL), but there are verified compilers for Rocq and HOL that we discuss next.

Verified compilation of Rocq. Forster et al. [7] present a verified compiler for Gallina to Malfunction (an intermediate untyped language of OCaml). Their compiler is based on the existing MetaRocq [14] project that formalizes Rocq in itself. They define compilation as a series of steps, one of them being quotation, which is trusted. They prove a realizability relation that guarantees that the extracted untyped code operationally behaves as the initial Rocq program. They compile Rocq without any special step for shallowly embedded code using monads, something we would like to do together with a proof of security.

Similar to Forster et al. [7], there is CertiCoq [3] and ConCert [6, 10], two compilers for Rocq that target C, Elm, Rust, WebAssembly and blockchain languages. Their compilers are partially verified.

Œuf [11] formalizes a subset of Gallina and provides a verified compiler to Assembly code. They translate-validate quotation to have an end-to-end correctness proof. To be able to translate-validate, they require a computational denotation, which restricts what they can quote.

Relational compilation. Traditionally, compilation is defined as a function from source programs to target programs. Relational compilation treats compilation as a relation, which now frees one to search for a target program so that ∃t. s R t. The technique we present, relational quotation, is a variant

of relational compilation specialized for the specific task of doing quotation.

Our work is inspired by Pit-Claudel et al. [13] and Myreen and Owens [12] who present how to do relational compilation for Gallina and Isabelle HOL with support for programs shallowly embedded using monads. Their focus is on proving correct compilation, while we focus on proving secure compilation.

Pit-Claudel et al. [13] present a framework for relational compilation designed for performance-critical applications. The framework allows one to build a relational compiler in an extensible and composable way. They show the versatility of the framework by compiling a subset of Gallina to a low-level imperative language.

CakeML. Abrahamsson et al. [2] show how starting from a shallowly embedded program in Isabelle HOL one can synthesize an equivalent CakeML program. CakeML [9] is a subset of Standard ML that has a verified compiler that can target ARMv6, ARMv8, x86-x64, MIPS-64, RISC-V and Silver RSA architectures. They have an end-to-end proof of correctness from the original program written in HOL to the low level code that targets a specific architecture. The synthesis they do, however, is a process that happens outside of HOL: it is implemented in Standard ML and it produces a deeply embedded program and a proof of correctness in HOL. In our case, everything happens in F⋆. Even if CakeML does not have a secure compiler, it will be interesting to see if we can target one of the untyped intermediate languages they have to reuse correct compilation.

References

[1] Carmine Abate, Roberto Blanco, Deepak Garg, Catalin Hritcu, Marco Patrignani, and Jérémy Thibault. 2019. Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation. In *32nd IEEE Computer Security Foundations Symposium (CSF)*. IEEE, 256–271. doi:10.1109/CSF.2019.00025

[2] Oskar Abrahamsson, Son Ho, Hrutvik Kanabar, Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Yong Kiam Tan. 2020. Proof-Producing Synthesis of CakeML from Monadic HOL Functions. *Journal of Automated Reasoning (JAR)* (2020). https://rdcu.be/b4FrU

[3] Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, and Matthew Weaver. 2017. CertiCoq: A verified compiler for Coq. In *3rd Workshop on Coq for Programming Languages (CoqPL)*. https://popl17.sigplan.org/details/main/9/CertiCoq-A-verified-compiler-for-Coq

[4] Cezar-Constantin Andrici, Danel Ahman, Cătălin Hriţcu, Ruxandra Icleanu, Guido Martínez, Exequiel Rivas, and Théo Winterhalter. 2025. SecRef*: Securely Sharing Mutable References between Verified and Unverified Code in F*. *Proc. ACM Program. Lang.* 9, ICFP, Article 253 (Aug. 2025), 31 pages. doi:10.1145/3747522

[5] Cezar-Constantin Andrici, Ştefan Ciobâcă, Catalin Hritcu, Guido Martínez, Exequiel Rivas, Éric Tanter, and Théo Winterhalter. 2024. Securing Verified IO Programs Against Unverified Code in F⋆. *Proc. ACM Program. Lang.* 8, POPL (2024), 2226–2259. doi:10.1145/3632916

[6] Danil Annenkov, Mikkel Milo, Jakob Botsch Nielsen, and Bas Spitters. 2022. Extracting functional programs from Coq, in Coq. *Journal of Functional Programming* 32 (2022), e11. doi:10.1017/S0956796822000077

[7] Yannick Forster, Matthieu Sozeau, and Nicolas Tabareau. 2024. Verified Extraction from Coq to OCaml. *Proc. ACM Program. Lang.* 8, PLDI (2024), 52–75. doi:10.1145/3656379

[8] Georges Gonthier, Beta Ziliani, Aleksandar Nanevski, and Derek Dreyer. 2013. How to make ad hoc proof automation less ad hoc. *Journal of Functional Programming* 23, 4 (2013), 357–401.

[9] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*. ACM, 179–192. doi:10.1145/2535838.2535841

[10] Wolfgang Meier, Martin Jensen, Jean Pichon-Pharabod, and Bas Spitters. 2025. CertiCoq-Wasm: A Verified WebAssembly Backend for CertiCoq. In *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Denver, CO, USA) *(CPP '25)*. Association for Computing Machinery, New York, NY, USA, 127–139. doi:10.1145/3703595.3705879

[11] Eric Mullen, Stuart Pernsteiner, James R. Wilcox, Zachary Tatlock, and Dan Grossman. 2018. Œuf: minimizing the Coq extraction TCB. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Los Angeles, CA, USA) *(CPP 2018)*. Association for Computing Machinery, New York, NY, USA, 172–185. doi:10.1145/3167089

[12] Magnus O. Myreen and Scott Owens. 2014. Proof-producing Translation of Higher-order logic into Pure and Stateful ML. *Journal of Functional Programming (JFP)* 24, 2-3 (May 2014), 284–315. doi:10.1017/S0956796813000282

[13] Clément Pit-Claudel, Jade Philipoom, Dustin Jamner, Andres Erbsen, and Adam Chlipala. 2022. Relational compilation for performance-critical applications: extensible proof-producing translation of functional models into low-level code. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 918–933. doi:10.1145/3519939.3523706

[14] Matthieu Sozeau, Yannick Forster, Meven Lennon-Bertrand, Jakob Nielsen, Nicolas Tabareau, and Théo Winterhalter. 2025. Correct and Complete Type Checking and Certified Erasure for Coq, in Coq. *J. ACM* 72, 1, Article 8 (Jan. 2025), 74 pages. doi:10.1145/3706056