VOLPIC: Verifying Lifted Pascal in Coq

Charles Averill

Advisor: Kevin Hamlen

The University of Texas at Dallas

https://seashell.charles.systems/

charles@utdallas.edu

Motivation

- Lots of legacy **mission-critical code** is written in Pascal
- Pascal is one of 10 higher-order languages allowed by US DoD for mission-critical software development [5]
- **Formal verification** of this code is **currently not possible** without significant code refactoring

Developing a framework for formally verifying Pascal code without significant refactoring will encourage developers of mission-critical systems to enhance the security of their projects, improving the overall safety of these systems.

Background

- **Coq** is a formal verification platform comprised of a **functional programming language** and a **machine-checked proof environment**
- Code in other languages can be formally verified by **transpiling** [1, 4] it to Coq or by **encoding language semantics** in Coq
- Transpiling is easier when source and target languages are similar (e.g. OCaml/Haskell to Coq) but difficult when they are different (e.g. an imperative language to Coq)
- Transpiled Coq code can be **extracted** to equivalent OCaml or Haskell code for **real-world usage** via Coq's verified extraction facilities

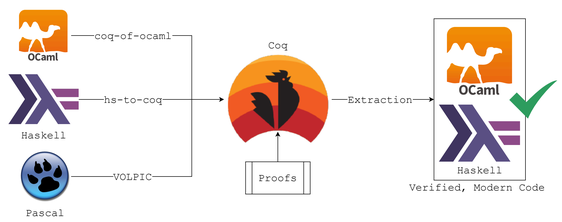

Figure 1. Transpilation pipelines

- Semantics encoding is more convenient for imperative languages, but requires more machinery and verification guarantee is indirect

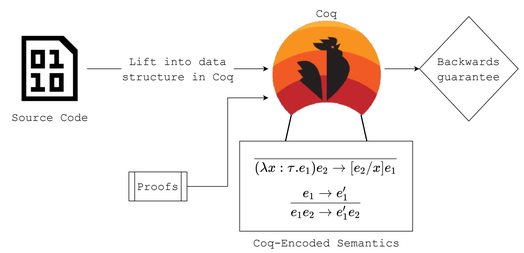

Figure 2. Encoded semantics pipeline

- **Previous attempts** at verifying Pascal code exist [2, 3], but they **do not allow for retroactive verification** and do not use modern proof checkers

Lifting, Verification, and Extraction Pipeline

VOLPIC (Verifier Of Lifted Pascal In Coq) aims to provide a **seamless pipeline** for **lifting, verification, and extraction of Pascal code**, and is composed of three components corresponding to these tasks.

- Lifter uses a standard parse-translate-generate compiler architecture to transpile Pascal code into **loosely-equivalent** native Coq code
- Verification library provides theorems and tactics for working with the constructs that frequently appear in lifted code
- Extraction facilities tune Coq's code extractor to generate equivalent OCaml code, allowing for legacy code to be **automatically converted** to a modern, high-level development language

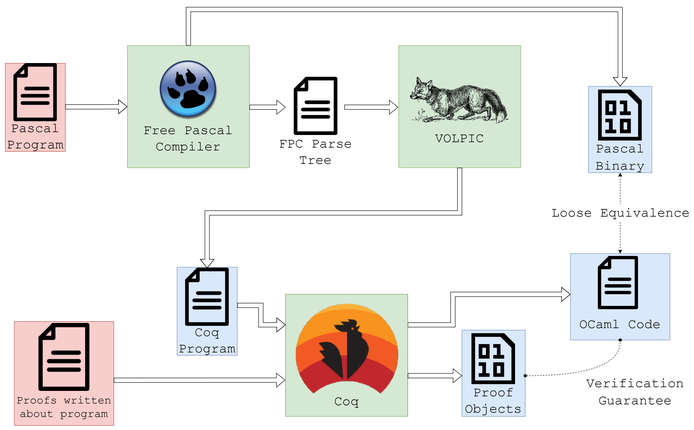

Figure 3. VOLPIC Pipeline

VOLPIC is able to **lift code at multiple levels of optimization supported by the Free Pascal Compiler (FPC)** because its lifter utilizes FPC to generate a parsed form of the input program.

To ensure a stable lifting interface, we provide a fork of FPC that extends the parse tree output with important program information, such as string constants, field names for struct accesses, and function input parameter names.

Because VOLPIC lifts into native Coq code, the **existing standard library** of theorems and tactics **remains useful** for proving properties of lifted code. Additionally, we provide a number of theorems and tactics that help with manipulation of intermediate terms and defining loop invariants.

We **intentionally do not maintain** some forms of **semantic equivalence** between the original Pascal, lifted Coq, and extracted OCaml programs. These intentional differences involve modeling runtime errors (such as out-of-bounds data structure accesses), ensuring termination conditions for loops in Coq, and allowing for the access of generically-typed terms by providing type-equivalence proofs. Although some of the differences do change the functionality of the original code at a high level, correctness specifications can effectively remove these differences from consideration.

Proof of Correctness Example

We provide an example of the utility of VOLPIC with a **proof of correctness** for an implementation of an **iterative factorial calculator**. In its lifted form, the structure of the original loop is still visible thanks to VOLPIC's notation library.

Because this structure persists through lifting, our proof of correctness must rely on an appropriate **loop invariant**, as is common for proofs of imperative code. Our loop invariant is a direct generalization of the goal statement, where $result$ represents the intermediate result value:

$$\begin{aligned} &\text{goal:} && \texttt{factorial}(n) = \texttt{Z\_fact}(n) \\ &\text{invariant:} && result' = result * \texttt{Z\_fact}(n) \wedge n' = n - 1 \end{aligned}$$

Figure 4. Goal and invariant for `factorial` correctness

```
function factorial(n : integer) : integer;
begin
    result := 1;
    while 1 <= n do
    begin
        result := result * n;
        n := n - 1;
    end;
end;
```

Figure 5. Iterative factorial calculator written in Pascal

```
Definition factorial (VP_store: store) loop_limit :=
  let VP_poison := false in
  let (VP_store,VP_poison) :=
        (("VP_result" !-> VInteger 1; VP_store),VP_poison) in
  match
   while (fun VP_store => 1 <=? get_int VP_store "VP_N")
    with VP_store upto loop_limit begin fun VP_store =>
      let VP_store := ("VP_result" !-> VInteger (get_int
          VP_store "VP_result" * get_int VP_store "VP_N");
          VP_store) in
      ("VP_N" !-> VInteger (get_int VP_store "VP_N" - 1);
          VP_store)
   end
  with
  | None => (VP_store, true)
  | Some s => (s, VP_poison)
  end.
```

Figure 6. VOLPIC-lifted form of factorial calculator

```
Definition correct f input expected :=
    forall loop_limit output,
        (output, false) = f input loop_limit -> output "VP_result" = expected.

Definition Z_fact (n : Z) := Z.of_nat (fact (Z.to_nat n)).

Theorem factorial_correct : forall n,
    correct factorial ("VP_X" !-> VInteger n; fresh_store) (VInteger (Z_fact n)).
```

Figure 7. Correctness specification

Reflection and Future Work

VOLPIC provides developers with the tools necessary to **retroactively formally verify the security of Pascal programs**. We are currently extending the capabilities of the lifter to support generically-typed terms, which will allow the lifting of arbitrary data structures and user-defined structures. Following these implementations, we aim to prove the utility of VOLPIC by formally verifying a mission-critical piece of Pascal software.

References

[1] Guillaume Claret. Coq of ocaml, 2014. URL `https://watch.ocaml.org/w/xb34CNYKTwC51ahyaYLQuN`.

[2] David C. Luckham, Steven M. German, Friedrich W. von Henke, Richard A. Karp, P. W. Milne, Derek C. Oppen, Wolfgang Polak, and William L. Scherlis. Stanford pascal verifier user manual. Technical report, 1979. URL `https://apps.dtic.mil/sti/tr/pdf/ADA071900.pdf`.

[3] John Nagle. Pascal-f verifier user's manual, 1982. URL `http://www.animats.com/papers/verifier/verifiermanual.pdf`.

[4] Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. Total haskell is reasonable coq. *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, Jan 2018. doi: 10.1145/3167092.

[5] William Jr. Ward. *Transition to Ada*, chapter A: DoD Directive 3405.1, page A9. US Army Corps of Engineers, 1994. URL `https://apps.dtic.mil/sti/tr/pdf/ADA286419.pdf`.