VOLPIC: Verifying Lifted Pascal in Coq

Anonymous

ABSTRACT

VOLPIC is a new Coq-based framework for lifting and verifying Pascal code. VOLPIC provides a pipeline to convert Pascal code into native Coq, formally prove correctness specifications, then extract into modern, verified OCaml to further improve maintainability. This is demonstrated with a proof of partial correctness for a function lifted from Pascal code. Pascal is a critical target for machine-checked verification due to its use in many common userspace applications such as TEX and Photoshop, in addition to its use in military organizations.

1 PROBLEM AND MOTIVATION

A large hurdle for cybersecurity efforts is the slow adoption rate of state-of-the-art techniques at a large scale. Due to the frequently-changing landscape of attacks and defenses, it is difficult for programmers to continually update their development practices to match the state-of-the-art. As a result, an attractive approach known as *security retrofitting* has gained popularity, providing security to existing software at a significantly lower cost than attempting to modify development practices.

In some forms, formal verification is a viable technique for retrofitting security to existing code. Because proofs about code can exist outside of the codebase, verification can be independent of software development. However, few tools exist to formally verify Pascal code, and none provide the high level of assurance that Coq provides with its minimal trusted computing base. Designing tools to promote the verification of Pascal code will afford higher assurance for mission-critical systems containing legacy and new Pascal components.

Although Pascal has declined in popularity over time in both industrial and academic contexts, a large amount of legacy code written during the height of its use remains a large part of digital infrastructure today. In fact, Pascal is one of 10 higher-order programming languages permitted by the US Department of Defense for mission-critical software development[7].

2 BACKGROUND AND RELATED WORK

There are multiple programs that transpile code from other languages into Coq for the purpose of verification. For example, coq-of-ocaml [1] and hs-to-coq [6] transpile OCaml and Haskell to Coq, respectively. Transpilation is more challenging when the source language is dissimilar to Coq (such as low-level imperative languages like C and Pascal), making the lifting process more difficult and often error-prone.

Another common approach encodes the semantics of the source language in Coq (cf. [5]). A pitfall of semantic encoding is that a sizeable custom theorem library is required to feasibly build proofs that utilize the complex types and relations introduced in the language semantics. Additionally, the trusted computing base becomes much larger, as the encoded language semantics must match what the language's compiler actually computes, and the correctness of the lifter from source code to a Coq data structure becomes critical.

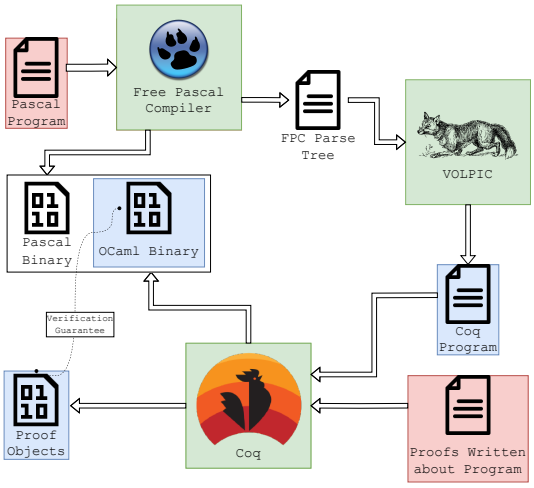

Figure 1: Lifting, verification, and extraction pipeline

A third approach involves writing native Coq code with the goal of extracting it to OCaml or Haskell. This is convenient if verification is a goal from the start of the project, but has its own drawbacks. Primarily, there are few languages that can be extracted from Coq source code, most of which are not frequently-used languages that can be maintained by many engineers. Furthermore, Coq's standard library may be unsuitable for general software development in certain application domains, complicating software written in this style.

There have been multiple attempts to verify Pascal code specifically. The Stanford Pascal Verifier [3] implements an *interactive program verification system* based on Hoare logic [2]. This system made significant progress towards general-purpose verification, but suffers a complex proof checker and a limited assertion language compared to modern alternatives. The Pascal-F Verifier [4] improves some of these issues, but requires verification to be a core aspect of the development process, and therefore requires significant Pascal code refactoring prior to verification.

3 APPROACH AND UNIQUENESS

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 corresponding components for these tasks.

3.1 Lifter

VOLPIC's lifter follows a common pattern for compilers: parse, translate, and generate:

Parsing is achieved almost entirely within *FPC*, the Free Pascal Compiler. FPC provides an option for compiler developers that dumps a parse tree in a structured log format similar to a type-annotated concrete syntax tree, or *CST*. Utilizing FPC for parsing provides the ability to write proofs about programs at any level of optimization supported by the compiler. Because it is primarily used to debug FPC, this format is generally unstable and lacks many features necessary to transpile programs. As such, we present a custom fork of FPC that provides a stable tree format with information that is missing from the main branch, namely string constant

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

Figure 2: 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_X")
    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_X"); VP_store) in
      ("VP_X" !-> VInteger (get_int VP_store "VP_X" - 1); VP_store)
   end
  with
  | None => (VP_store, true)
  | Some s => (s, VP_poison)
  end.
```

Figure 3: VOLPIC-lifted form of factorial calculator[1]

values and record field accesses, while retaining all modern language features. Parsing then continues in the VOLPIC codebase as the tree format is read into a concrete syntax tree.

Translation occurs between the FPC format's CST and an internal simplified representation AST to reduce the amount of logic required for code generation. This representation removes a large amount of Pascal-specific information, including but not limited to

- loop dichotomies (`for`, `foreach`, `do while`, etc., all become `while`);
- `Function-Procedure` dichotomies (procedures are functions with no return value, which are disallowed in Coq);
- redundant nodes of the parse tree; and
- inline computations, which expand to full expressions.

Generation from the simplified AST to Coq source code occurs via a recursive traversal of the AST, generating sub-terms along the way to be assembled into a complete definition. We found this approach preferable to using Coq's OCaml API to directly convert Pascal ASTs to Coq terms, since the Coq-OCaml API is a subject of rapid churn and lacks the machinery necessary to generate terms at will in a self-contained manner.

3.2 Verification and Notation Libraries

Lifting to native, standard-library-based Coq allows for standard goal manipulation tactics, such as `simpl`, `auto`, and `inversion`, to remain useful; however, VOLPIC also introduces a number of custom values and relationships . To reduce the load of verifying this code, we provide a theorem library that aims to simplify terms utilizing common Pascal data types such as strings. Additionally, given the verbose nature of code translated from an imperative paradigm, we provide a `vpex` tactic that extracts subterms (e.g., loop bodies) from termination assumptions, making it easier to write subproofs about the behavior of smaller parts of functions. `vpex` excels when a complex inductive hypothesis is encountered, as extraction prevents sensitive terms from being simplified beyond what the induction hypothesis can apply to. Finally, we provide a notation scope to minimize the representations of code while interacting with terms such as assignments and loops, as seen in Figure 3.

```
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 4: Correctness specification

3.3 Extraction Library

VOLPIC utilizes Coq's verified extraction facilities to convert lifted Pascal code from Coq into OCaml. In order to support FPC's standard features, we provide a small extraction library that aims to replicate common I/O and fundamental data structure manipulation functions.

3.4 Lifter Correctness

A significant consideration is whether the lifter and extractor maintain semantic equivalence between the original Pascal program, the intermediate Coq program, and the output OCaml program. Some forms of equivalence are intentionally not maintained between the three languages, including boundless looping, recursion, and runtime errors.

To model Pascal runtime errors, we pass along *poison values*, representing whether a runtime error has been encountered. When a function's poison value is true, evaluation immediately stops and the function returns early, indicating the error. Poison values provide a built-in precondition for correctness proofs that the code has evaluated without errors by requiring the value to be false.

In Coq, loops are purely recursive and require a termination condition provable by structural reduction. To accommodate this, we employ a standard approach: transform loops into anonymous recursive functions, and affix each recursive definition with a termination counter that decreases with each iteration. If the loop counter reaches zero, the loop terminates and the poison value is set to true.

This approach does not limit the scope of proofs we can write about lifted code, as we can universally quantify over all values of the loop counter to effectively remove it from consideration. This loop counter can then be manually removed from the extracted OCaml code; however we reserve this for future work.

As a proof of concept, we have proved the correctness of the iterative factorial calculator shown in Figure 3, lifted from the code shown in Figure 2. This code is then extracted from Coq into OCaml, providing a verified, high-level representation of the original program.

4 RESULTS AND CONTRIBUTIONS

VOLPIC is an effective solution for lifting and extracting Pascal source code into native, verifiable, and extractable Coq code. These approaches can be used to verify mission-critical code that previously would have relied on standard software testing or lower-assurance proof checkers. We show that the lifted form is verifiable with a machine-checked proof of correctness for an iterative factorial calculation function, using VOLPIC's verification libraries.

[1]Figure 3 has been simplified for brevity - redundant checks generated by the lifter such as `if false then e1 else e2` have been replaced with `e2`. These redundancies are trivially simplified with the `simpl` tactic, and therefore the changes do not impact the correctness of the function.

REFERENCES

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

[2] C. A. Hoare. 1978. An axiomatic basis for computer programming. *Programming Methodology* (1978), 89–100. https://doi.org/10.1007/978-1-4612-6315-9_9

[3] 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. 1979. *Stanford Pascal Verifier user manual.* Technical Report. https://apps.dtic.mil/sti/tr/pdf/ADA071900.pdf

[4] John Nagle. 1982. Pascal-F Verifier User's Manual. http://www.animats.com/papers/verifier/verifiermanual.pdf

[5] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach, and Brent Yorgey. 2024. *Programming Language Foundations.* Software Foundations, Vol. 2. Electronic textbook. http://softwarefoundations.cis.upenn.edu Version 6.5.

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

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