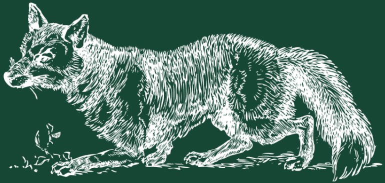
Verifier Of Lifted Pascal In Coq



VOLPIC

Charles Averill

Dallas Hackers' Association

February 2024

What am I doing?

- I am building a transpiler to convert Pascal code to Coq code
    - Pascal: imperative, low-level, memory-managed, simple types, released in 1970 (C++ but better)
    - Coq: functional, high-level, memory doesn't exist, polymorphic and dependent types, released in 1989 (the Universe's gift to Mathematicians)
- I am writing a theorem library to aid in the formal verification of Pascal programs
- I am writing a "Pascal virtual machine library" in OCaml for the extraction of Coq programs generated from Pascal programs

Why am I doing this?

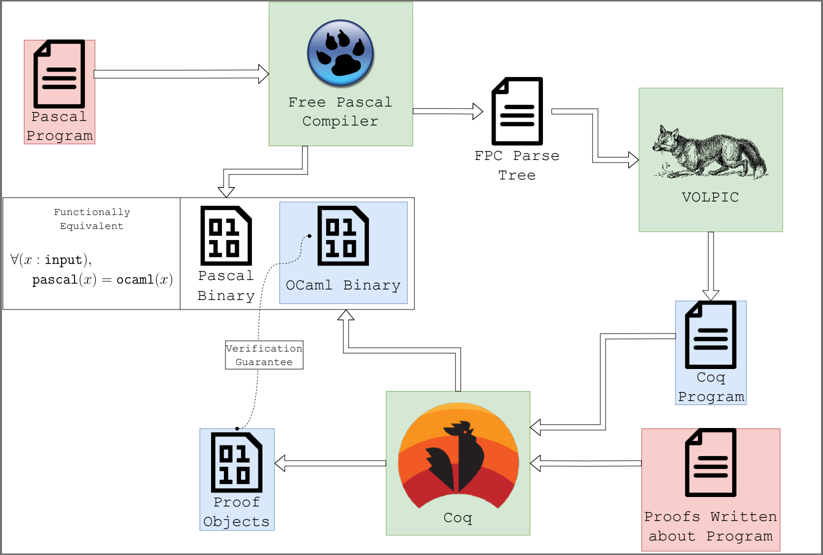
- Formal Verification provides the opportunity for developers to mathematically prove that their code is bug-free
- Lots of code is written in Pascal
    - Photoshop
    - Skype
    - FL Studio
    - A highly-cited DNA Sequence Assembler
    - Tons of DoD stuff we don't know about
    - TeX/Metafont
- I want to get a bug bounty check from Donald Knuth by verifying TeX and Metafont
- I have nothing to do from my graduation (Dec 18) to the day I leave the country (Feb 17)

How am I doing this?

The short answer: lots of painful engineering

1. **Process**: Utilize the Free Pascal Compiler (FPC) to provide a structured form of the program
2. **Lift**: Transpile structured program to Coq
3. **Verify**: Write proofs about lifted Coq program
4. **Extract**: Convert lifted Coq program into equivalent OCaml or Haskell code

Workflow


Pascal Program
Free Pascal Compiler
FPC Parse Tree
VOLPIC
Functionally Equivalent
$\forall(x : \texttt{input}),$
$\texttt{pascal}(x) = \texttt{ocaml}(x)$
Pascal Binary
OCaml Binary
Coq Program
Verification Guarantee
Coq
Proof Objects
Proofs Written about Program

Issues

Believe it or not, this task is complicated. Some issues I've run into:

- FPC parse tree output is more like a log file than a language, making it extremely difficult to write. Here's the state of my parser:

```
opam exec -- menhir -v lib/lang/parser.mly
Warning: 12 states have shift/reduce conflicts.
Warning: one state has reduce/reduce conflicts.
Warning: 17 shift/reduce conflicts were arbitrarily resolved.
Warning: 3 reduce/reduce conflicts were arbitrarily resolved.
Warning: 6 end-of-stream conflicts were arbitrarily resolved.
```

- Pascal is imperative and mutable, Coq is functional and immutable
- Dependent typing necessary to achieve language expressivity while maintaining mutability
- Pascal is way more complex than something like C, so there are a ton of fairly-complex language features to support

Lifter Structure

The lifter essentially does:

1. Call out to FPC to compile program and get parse tree
2. `parser.mly` parses the tree into an OCaml object for manipulation
3. `converter.ml` translates Pascal language concepts into Coq language concepts, generating a new OCaml object
4. `generator.ml` traverses the new object and prints out corresponding Coq code
   - I was initially very excited to write the generator, planned to hook into the Coq compiler at runtime and feed it ASTs that it converts to strings
   - Coq compiler API doesn't seem like it is built for that, had to resort to bare string manipulation `T_T`

FPC Contributions

- After writing parser I began to write test programs
- Thought project was dead when I realized that FPC parse tree output didn’t include key info such as string constants or struct access field names
- Remembered that I work on compilers all the time
- Wrote and merged FPC MR 567, commits cd9ed54d and bb2e2f83 to add the features I needed to the compiler

Dependent Typing

- Dependent typing is really neat now that I have the base knowledge to understand what's going on
- You're probably familiar with parametric polymorphism and maybe type constructors

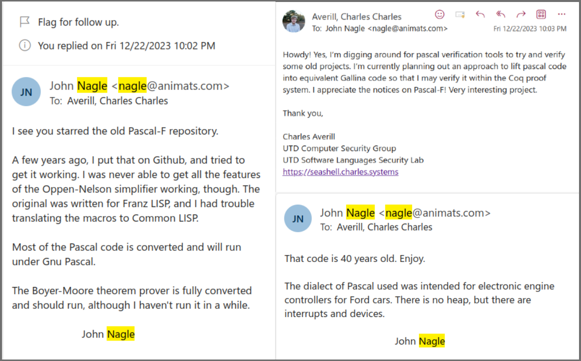
```
(* Type Constructors *)
Inductive list (T : Type) := nil | cons (h : T) (t : list T).

(* Parametric Polymorphism *)
Definition first {T : Type} (l : list T) : option T :=
    match l with
    | nil _ => None
    | cons _ h t => Some h end.

(* Dependent Types *)
Definition list_or_string (b : bool) :
        (match b with true => list int | false => string end) :=
    match b with true => [99;55;-500] | false => "Hello World" end.
```

I'm not the first

- I'm not the first person to attempt to verify Pascal code
- Donald Knuth considered formally verifying the TeX/Metafont compilers in tripman.tex, a "torture test" for TeX
- John Nagle (of Nagle's TCP Algorithm fame) worked on pasv, an early (pre-Coq) formal verification system specifically for Pascal


Flag for follow up.
You replied on Fri 12/22/2023 10:02 PM

JN John Nagle <nagle@animats.com>
To: Averill, Charles Charles

I see you starred the old Pascal-F repository.

A few years ago, I put that on Github, and tried to get it working. I was never able to get all the features of the Oppen-Nelson simplifier working, though. The original was written for Franz LISP, and I had trouble translating the macros to Common LISP.

Most of the Pascal code is converted and will run under Gnu Pascal.

The Boyer-Moore theorem prover is fully converted and should run, although I haven't run it in a while.

John Nagle

Averill, Charles Charles
To: John Nagle <nagle@animats.com>
Fri 12/22/2023 10:03 PM

Howdy! Yes, I'm digging around for pascal verification tools to try and verify some old projects. I'm currently planning out an approach to lift pascal code into equivalent Gallina code so that I may verify it within the Coq proof system. I appreciate the notices on Pascal-F! Very interesting project.

Thank you,

Charles Averill
UTD Computer Security Group
UTD Software Languages Security Lab
https://seashell.charles.systems

JN John Nagle <nagle@animats.com>
To: Averill, Charles Charles

That code is 40 years old. Enjoy.

The dialect of Pascal used was intended for electronic engine controllers for Ford cars. There is no heap, but there are interrupts and devices.

John Nagle

Demo

Future Plans

- Easiest: add more features
  - Better handling of user-defined data types
  - Support for record types
  - Figure out how I'll handle function calls
  - Implement more FPC functions
- Harder: write proofs of correctness for some unedited, lifted sample programs
  - Searches and sorts
  - Common array functions
  - More complex math functions
- Hardest: write proofs of correctness for TeX and MF
  - Only ∼4% of functions lift without error
  - Most failures to lift caused by unsupported language features such as special loop forms, array/struct assignments, etc.

Thank you!

- Source code: https://github.com/CharlesAverill/VOLPIC
- https://seashell.charles.systems/
  (or https://charlesaverill.github.io/ if it's down)
- Bluesky: @caverill.bsky.social