# Developing Certified Program Verifiers with a Proof Assistant

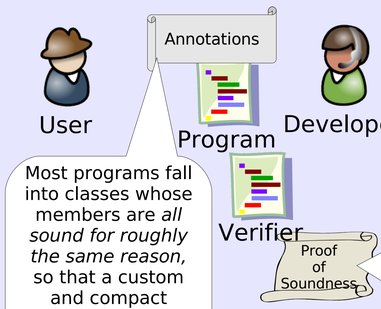
Adam Chlipala, University of California, Berkeley

## Why Certified Verifiers?

### Proof-Carrying Code


The expressivity of traditional Foundational PCC requires that these proofs can be very large and expensive to check!

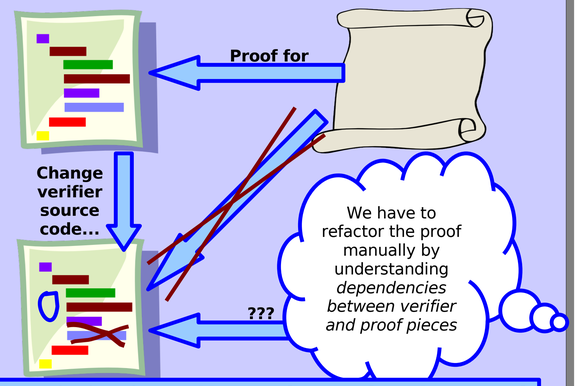
### Proof-Carrying *Verifiers*




Most programs fall into classes whose members are *all sound for roughly the same reason,* so that a custom and compact language of "hints" suffices to enable efficient checking.

*Certified verifiers* encapsulate knowledge of these classes as executable program verifiers with *proofs of soundness*. A verifier's soundness proof is checked *once*, when it is installed. It can be used subsequently to check many programs *without any further proof generation or checking*.

## OK, now how do we write these things?




*My solution here*: **Use a dependently-typed language to make these dependencies explicit**

## Coq and Extraction

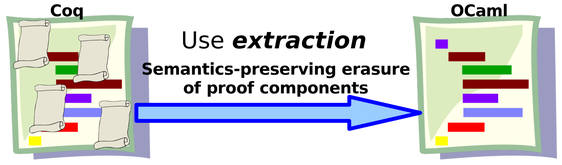
*Platform*: **The Coq Proof Assistant, which includes a rich dependently-typed functional programming language**

**Advantages for *Programming***

- Dependent types ensure total correctness.
- Includes an ML-style module system that can mix *program* and *proof* module elements.
- Easy OCaml integration

**Advantages for *Proving***

- Combine *goal-directed proof search* with standard programming.
- Take advantage of mechanisms for organizing and automating direct and tactic-based proofs.




*End result*: **Fast native code with a proof of correctness but no runtime manipulation of proofs**

## Implementation Technique: Monads for Composing Decision Procedures

```
check_inBounds : forall (abs : absState) (e : exp),
  [forall (conc : concState),
    compatible conc abs
    -> inBounds (eval conc e)]

typeof : forall (abs : absState) (e : exp),
  {{t : type | forall (conc : concState),
    compatible conc abs
    -> hasType (eval conc e) t}}

Definition do_write :
  forall (abs : absState)
    (dst src : exp),
    compatible conc abs
    -> {{abs' : abstractState |
      forall (conc : concState),
        compatible conc abs
        -> compatible
          (exec (Write dst src) conc)
          abs'}}.
  intros.
  refine (Hb <- check_inBounds abs dst;
          t : Ht <- typeof abs src;
          Success (setType abs dst t)).
  (* Tactics here to prove that Hb and Ht imply the
   * needed property for the argument to Success. *)
Qed.
```

`[P]` is an optional proof of proposition P.

`{{x : T | P(x)}}` is an optional value of type `T` that, if present, satisfies the predicate `P`.

All three procedures here have types that allow them to fail with no conclusion, which is important when dealing with undecidable problems!

Monadic notation leads to a failure return if either call fails; otherwise, type `t` and proofs `Hb` and `Ht` are bound in the body.

Using tactics and proof automation, we write a dependently-typed program without puzzling through any explicit proof terms! Better still, the *extracted* code doesn't mention concrete states at all, since they are only mentioned in *propositions* and *proofs*.

## Implementation Technique: Functors for Lowering Abstractions

It would be very costly to implement each new verifier from scratch. We would like a library of components that handle both program and proof pieces of verifiers, with explicit connections between the two aspects. The main technique I've chosen relies on using functors (parameterized modules) to transform one *verifier or abstraction* into a verifier or abstraction at a lower level.
For instance, we can compile a type system into an abstract interpretation:

```
Module Type TYPE_SYSTEM.
  Parameter type : Set.
  Parameter hasType : value -> type -> Prop.
  (* ...etc... *)

  Axiom soundness : (* Usual type soundness property *)
End TYPE_SYSTEM.

Module Type ABSTRACT_INTERPRETATION.
  Parameter absState : Set.
  Parameter compatible : concState -> absState -> Prop.
  (* ...etc... *)

  Axiom soundness :
    (* Usual abstract interpretation soundness property *)
End ABSTRACT_INTERPRETATION.

Module TypeChecker (Sys : TYPE_SYSTEM) : ABSTRACT_INTERPRETATION.
  (* Define a type-checking-based abstract interpretation.... *)
End TypeChecker.
```

**The box on the right shows the complete stack of abstraction levels from the implementation.**

## The Final Product

### What

A certified memory safety verifier for x86 machine code programs compiled by a fictitious certifying compiler for a language featuring algebraic datatypes

### Trusted Base

- 2000 lines of Coq formalizing bitvectors and x86 semantics
- OCaml code for parsing x86 binaries into ASTs
- Coq checking and extraction
- OCaml compiler

### Reusable Pieces

- 10,000-line Coq utility library
- 7000 lines of Coq for a library of functors spanning 8 levels of abstraction

### Verifier-Specific Code

One 600-line Coq file describing a type system declaratively

### Extracted Code

5000 lines of OCaml representing bitvectors with native words and mathematical integers with infinite-precision integers, thanks to a custom extraction optimization

## Memory Safety from the Ground Up

Weak Update Type System

Use a simplified type system based on partial type assignments to memory cells.

Simple Flags

Track dependencies between condition flags and registers/memory.

Stack Types

Augment type system with types to track stack and calling conventions.

Type System

Use Cartesian abstraction assigning types to registers.

Fixed Code

Enforce immutable code.

Reduction

Compile to simplified RISC.

Abstract Interpretation

Perform an exhaustive fixed-point calculation on a conservative approximation of the real semantics.

x86 Semantics