Modular Development of Certified Program Verifiers with a Proof Assistant

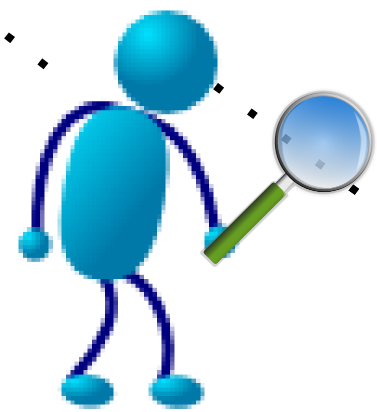
Adam Chlipala
University of California, Berkeley
ICFP 2006

Who Watches the Watcher?

Program Verifier Verifier

- Type-checker for stylized verifier language?
- Result checker on witnesses outputted by verifiers?
- Interactive proof assistant?

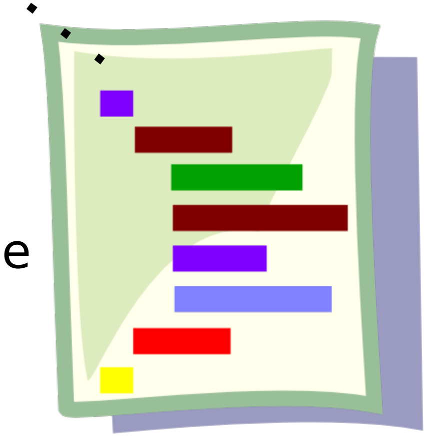
Program Verifier



- Java Bytecode Verifier
- Extended Static Checking
- Typed Assembly Language
- Proof-Carrying Code
- Model Checking

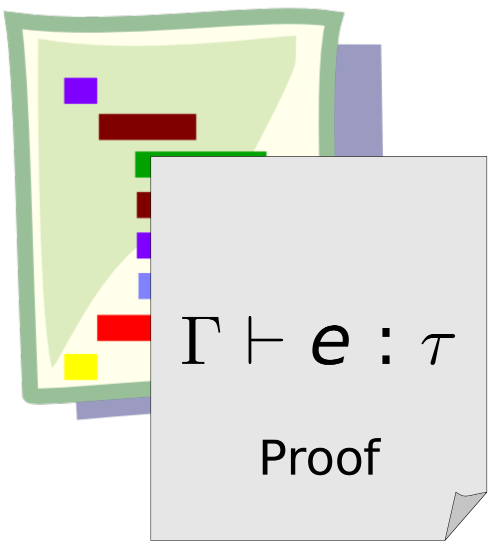
Might want to ensure:

- Memory safety
- Resource usage bounds
- Total correctness



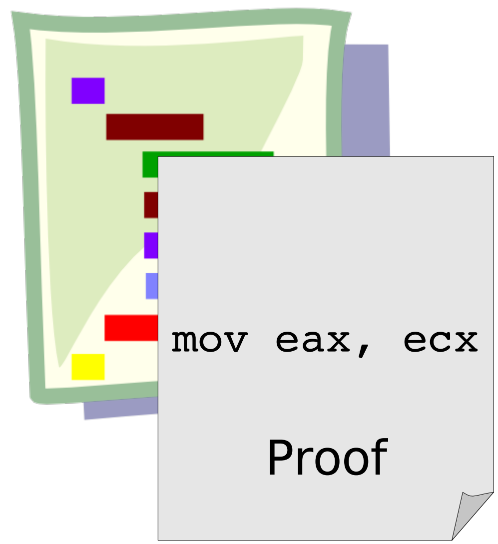
Untrusted Program

But Why?



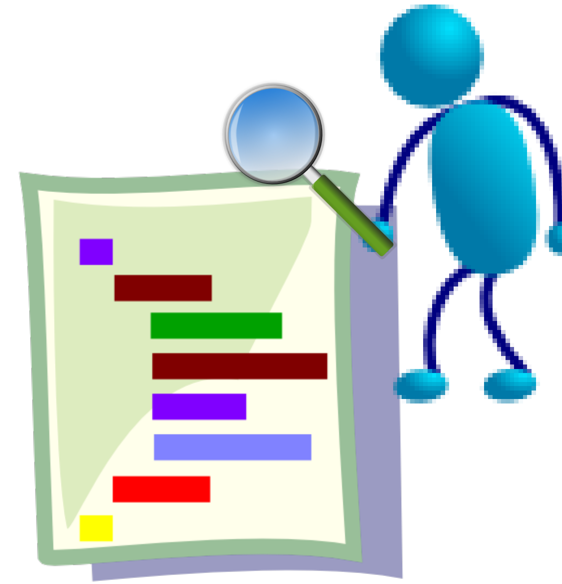
Proof-Carrying Code

- Compact proofs in a language specialized to one safety mechanism (e.g., a type system)
- Every new safety mechanism requires trusting a new body of code



Foundational Proof-Carrying Code

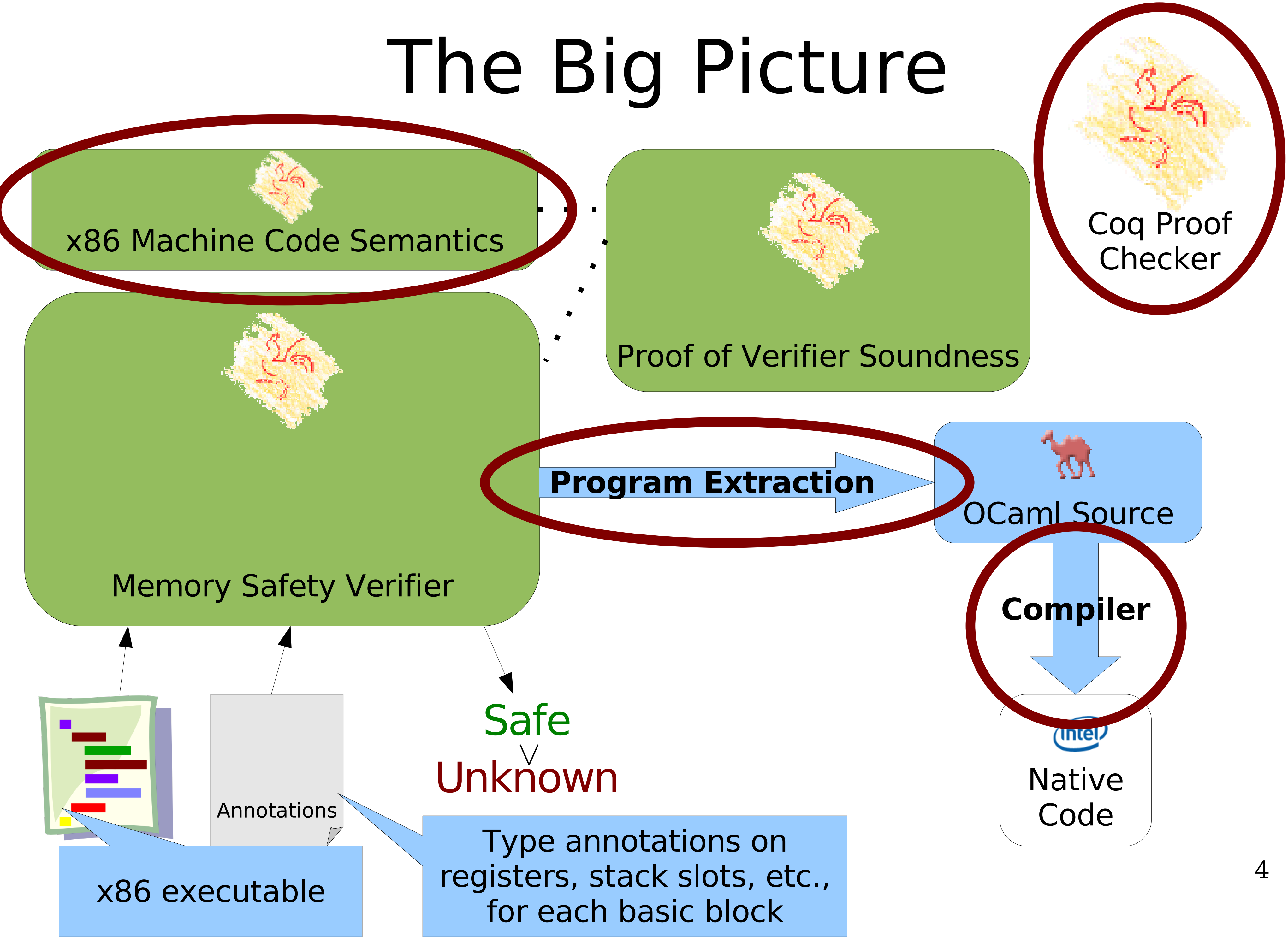
- Proofs about the real machine semantics, written in a very general language
- Proofs are much **larger**, making them expensive to check and transmit



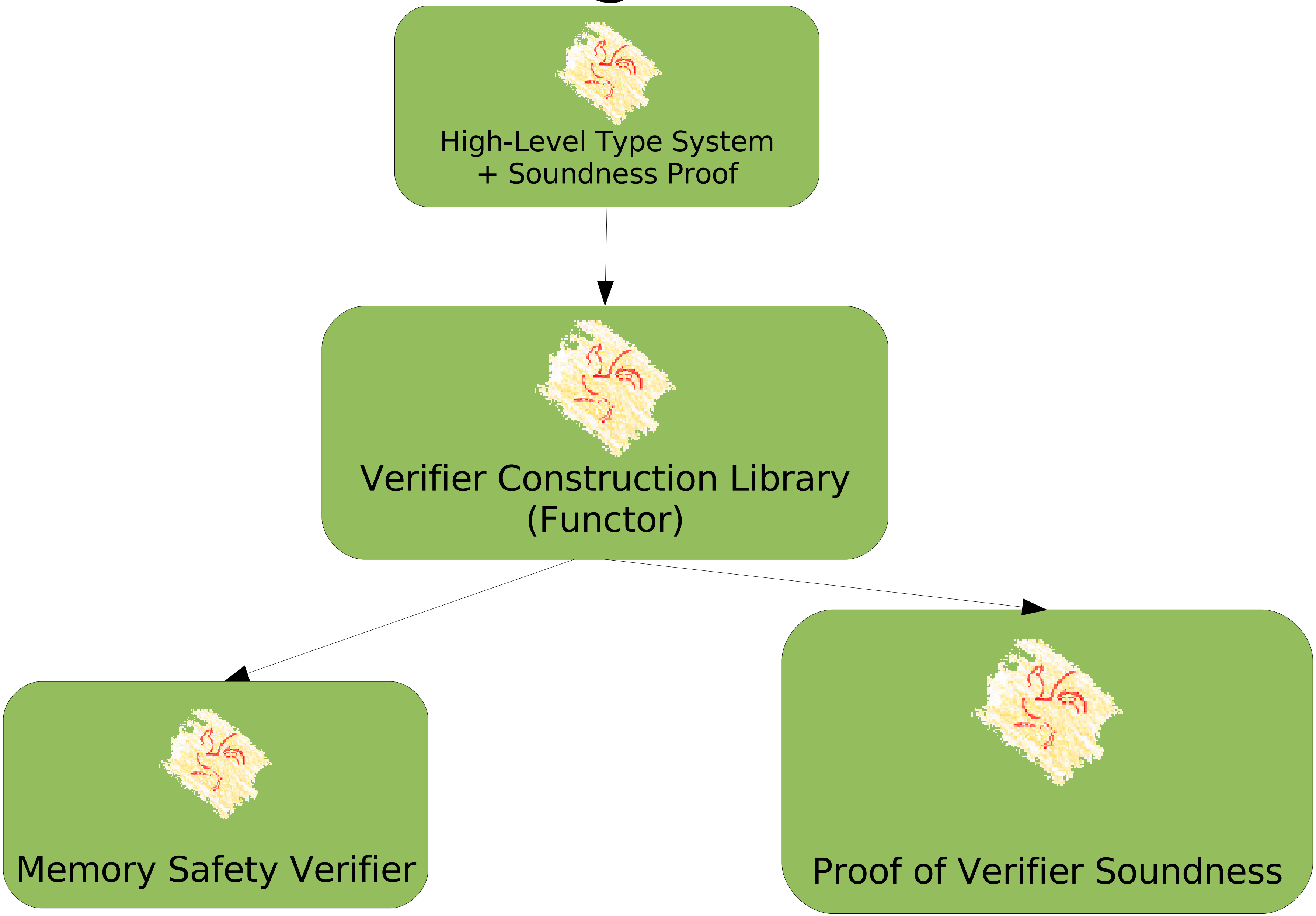
Certified Program Verifiers

- Allow custom executable verifiers that can be **reused**
- Require that every verifier be **proved sound**
- **No proofs** generated or checked at runtime

The Big Picture



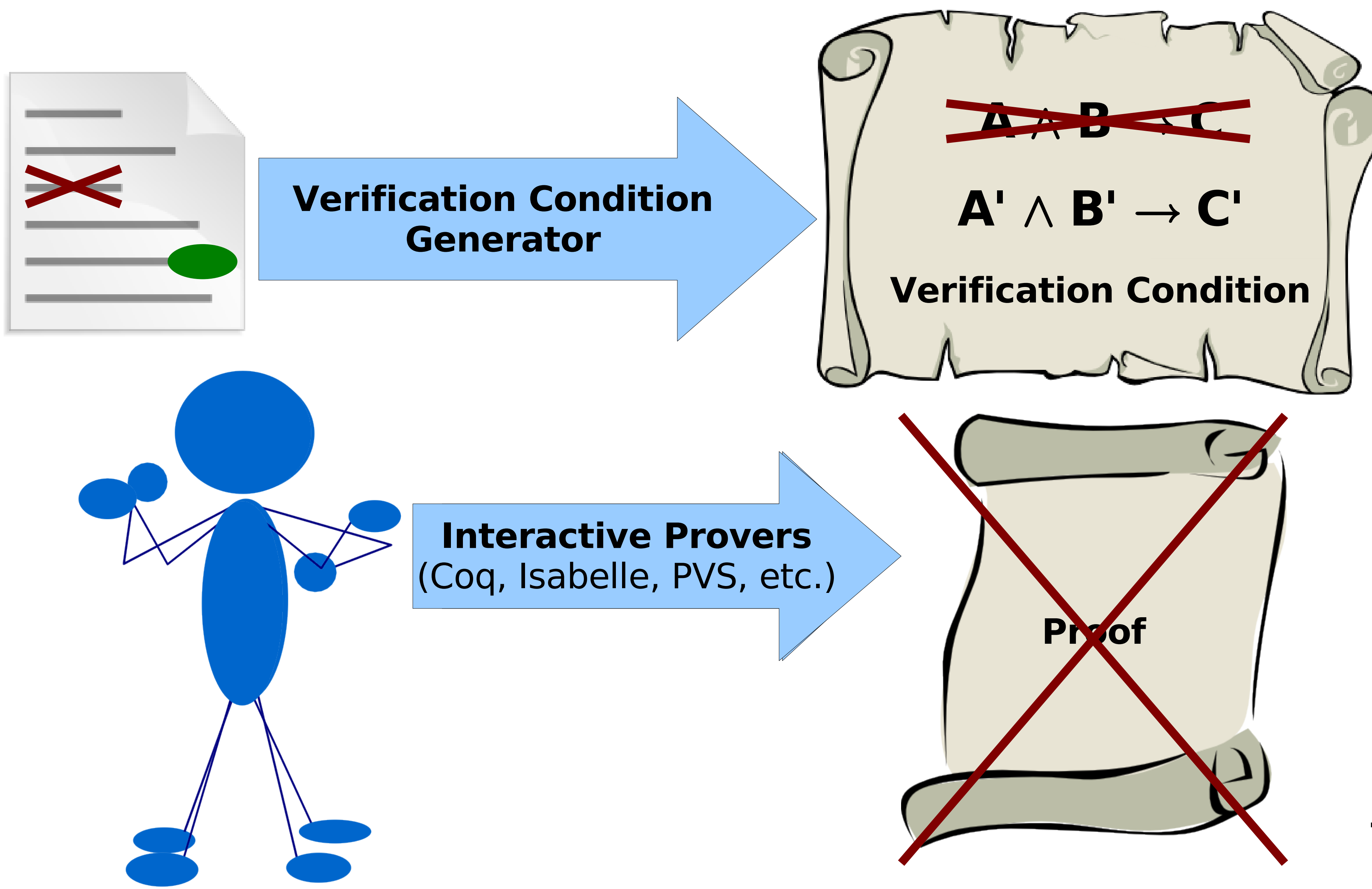
The Big Picture



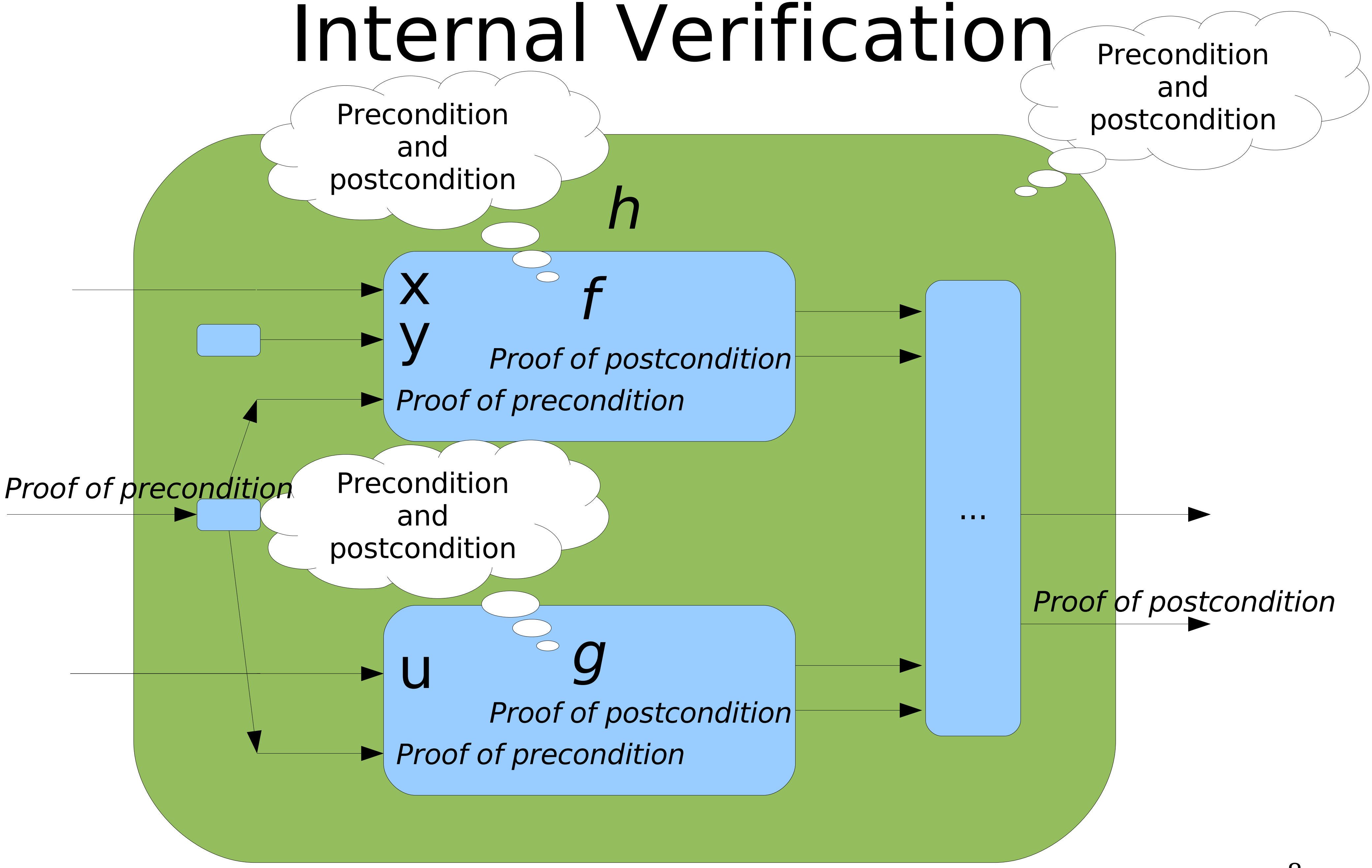
Outline

- Programming with dependent types
  - ...using a proof assistant
- A library for constructing certified verifiers
- Implementation

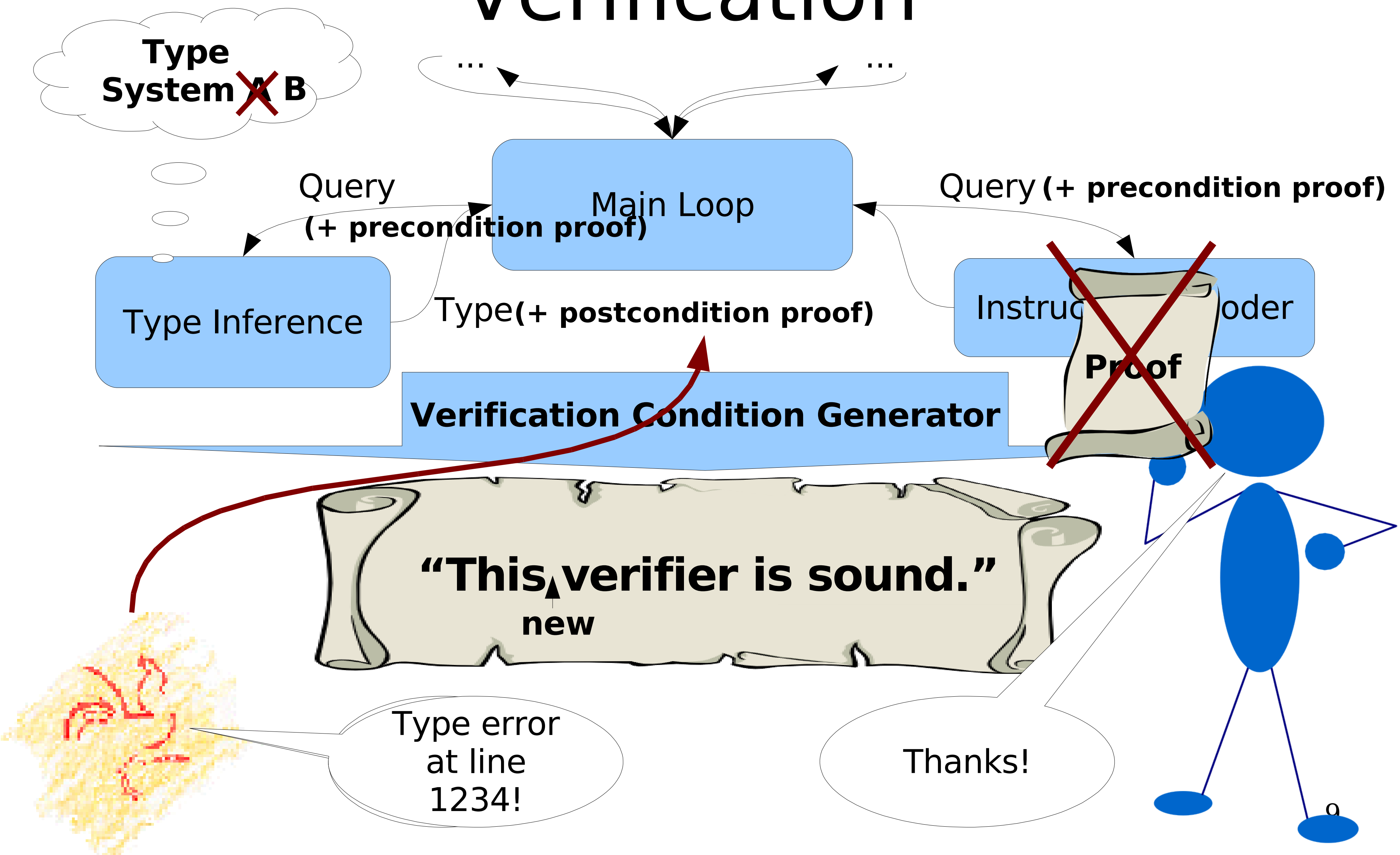
Classical Program Verification



Internal Verification

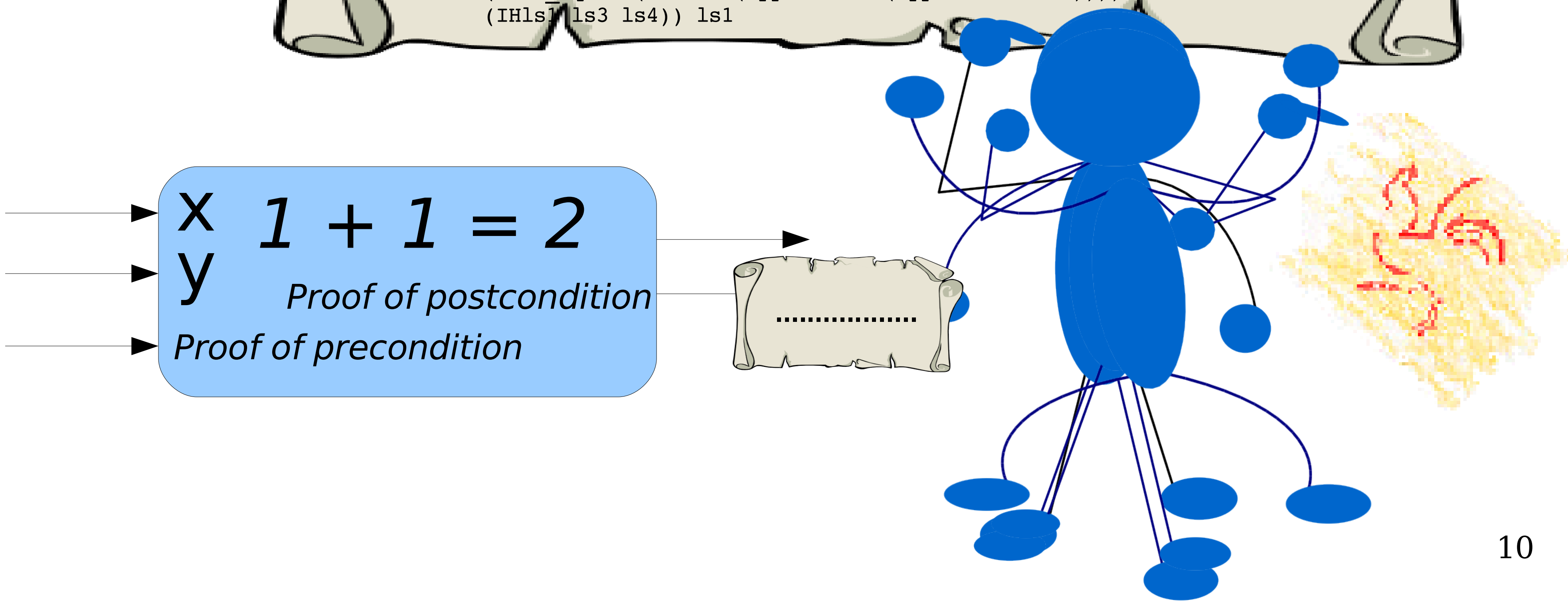


Benefits vs. Classical Verification



Dealing with Proof Terms

```
fun ls1 : list =>
list_ind
  (fun ls2 : list =>
   forall ls3 ls4 : list,
   append (append ls2 ls3) ls4 = append ls2 (append ls3 ls4))
  (fun ls2 ls3 : list => refl_equal (append ls2 ls3))
  (fun (n : nat) (ls2 : list)
     (IHls1 : forall ls3 ls4 : list,
              append (append ls2 ls3) ls4 = append ls2 (append ls3 ls4))
     (ls3 ls4 : list) =>
   eq_ind_r (fun l : list => cons n l = cons n (append ls2 (append ls3 ls4)))
     (refl_equal (cons n (append ls2 (append ls3 ls4))))
     (IHls1 ls3 ls4)) ls1
```



Mixing Programming with Tactics

```
Definition isEven : forall n, [even(n)].
  refine (fix is    ven (n : nat)
      : [even(n)     =
    match n retur     en(n)
      O -> Ye
    | S O ->
    | S (S n
      proof <- i
      Yes);
  auto.
Qed.
```

The type of an optional proof of a

Step 1. Declare the fu

Step 3. Gene   roof part" of the i   tactics.

Generate a proof obligation

Monadic notation: fail if the recursive call fails; otherwise, bind `proof` in the body.

t" of the   ion.

A Verification Stack

τ ::= int | τ ptr
Phantom state: Map from addresses to types
Phantom state: Map from flags to correlation with registers/memory

dangerous operations

τ ::= int | τ ptr
Phantom state: Map from flags to correlation with registers/memory

τ ::= int | τ ptr

semantics

τ ::= int | τ ptr | stackptr(n) | retptr | callee_save(r)

$\mathcal{D}$ ::= register $\rightarrow \tau$

step : state $\times$ **RISC instruction** $\rightarrow$ state

Assumption: Code is immutable

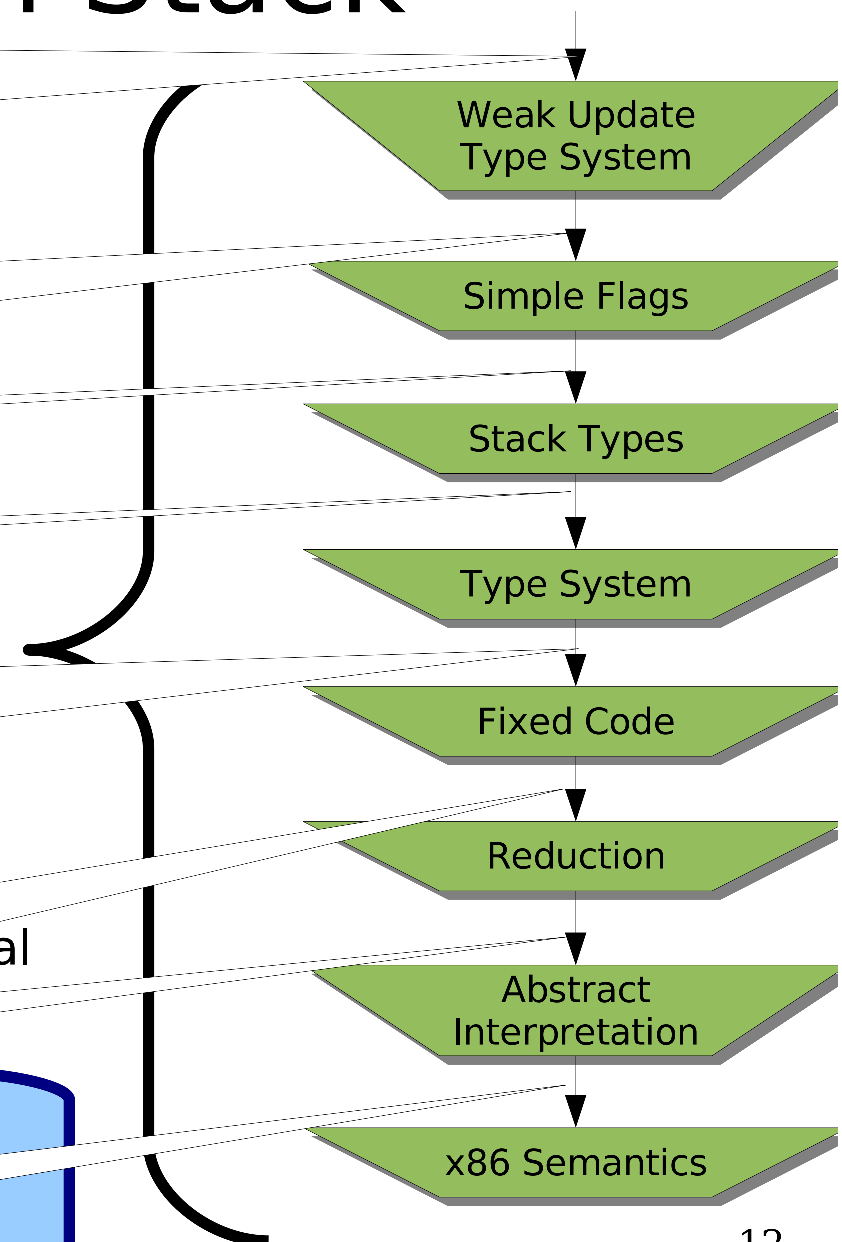
$\mathcal{D}$ ::= register $\rightarrow \tau$

step : state $\times$ **RISC instruction** $\rightarrow$ state

$\mathcal{D}$ ::= register $\rightarrow \tau$

step : state $\times$ **x86 instruction** $\rightarrow$ state

Generic fixed point computation procedure



Implementation

Component	**Lines of code**
Verification stack	7k (Coq)
Bitvectors and fixed-precision arithmetic	1k (Coq)
x86 semantics	1k (Coq)
Utility library	10k (Coq)
x86 binary parser	1500 (OCaml)
New extraction optimizations	1k (OCaml)
Algebraic datatype verifier	600 (Coq)
Total *trusted*	~5k

Sample Code: Type Language

```
Inductive ty : Set :=
  | Constant : int32 -> ty
  | Product : product -> ty
  | Sum : ty -> ty -> ty
  | Var : var -> ty
  | Recursive : var -> ty -> ty

with product : Set :=
  | PNil : product
  | PCons : ty -> product -> product.
```

Sample Code: Subtype Checker

```
Definition subTy : forall (t1 t2 : ty),
  poption (forall ctx v,
    hasTy ctx v t1 -> hasTy ctx v t2).
  refine (fix subTy (t1 t2 : ty) {struct t2}
    : poption (forall ctx v,
      hasTy ctx v t1 -> hasTy ctx v t2) :=
    match (t1, t2) with
      | (Constant n1, Constant n2) =>
        pfEq <- int32_eq n1 n2;
        Yes
      | (Product (PCons (Constant n) (PCons t PNil)),
          Sum t1 t2) =>
        if int32_eq n 0 && ty_eq t t1 then Yes
        else if int32_eq n 1 && ty_eq t t2 then Yes
        else No
      | (Recursive x body, t2) =>
        pfSub <- subTy
          (subst x (Recursive x body) body) t2;
        Yes
      | ...
    end); ....
Qed.
```

Related Work

- CompCert certified C compiler project [Leroy et al.]
- Foundational proof checkers with small witnesses [Wu et al.]
- Lots of work on building bytecode verifiers with proof assistants

Conclusion

- Today's technology makes constructing certified verifiers with dependent types feasible
- Good mixture of soundness guarantees, ease of engineering, and runtime efficiency

Code and documentation on the web at:
`http://proofos.sourceforge.net/`