A Framework for Certified Program Analysis and Its Applications to Mobile-Code Safety

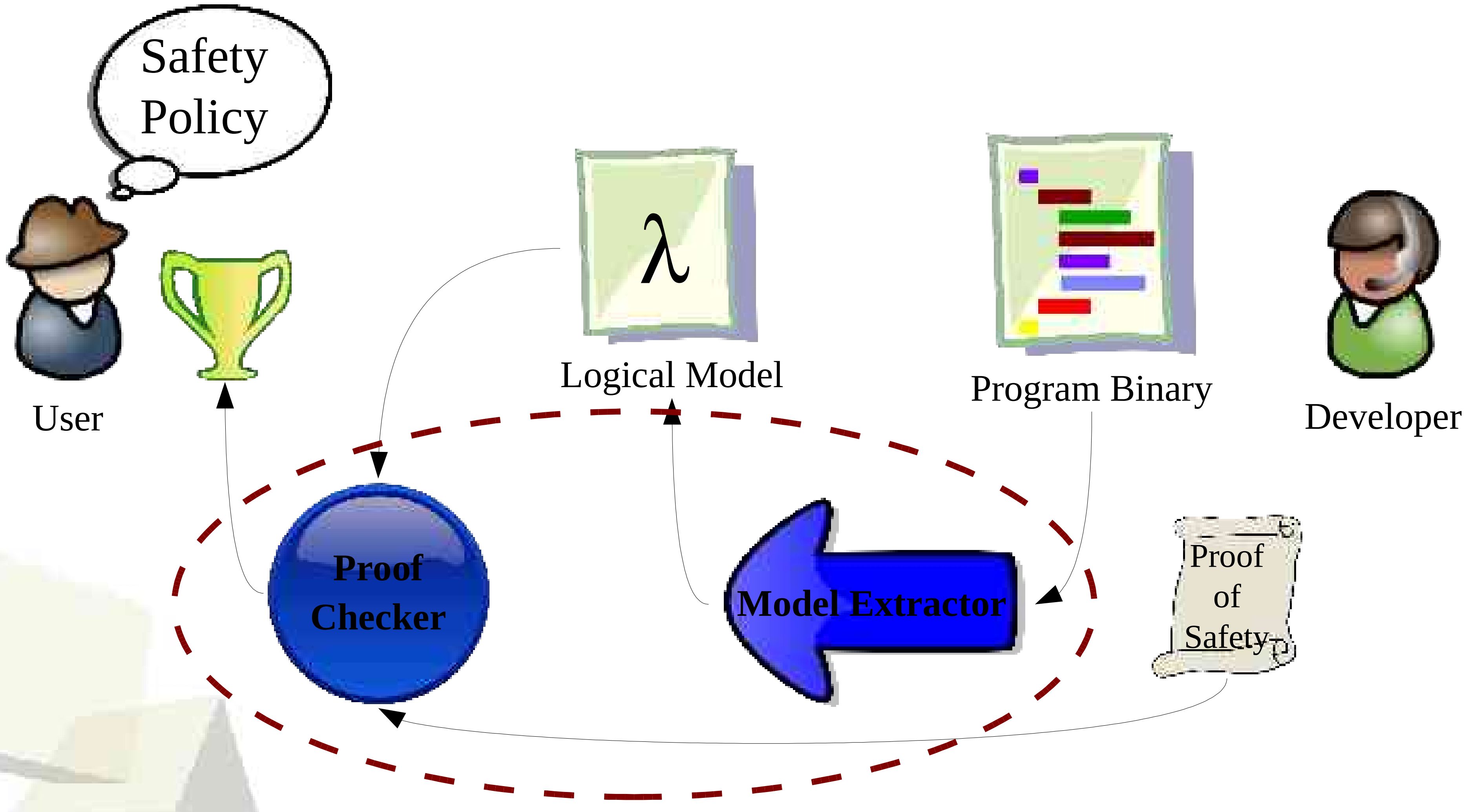
Adam Chlipala
Bor-Yuh Evan Chang
George C. Necula

UC Berkeley

Trustworthy Program Analysis

- We depend on program analysis results for assurances of program security and reliability.
  - Verifying properties of software from untrusted sources
- How can you be sure that your analysis guarantees what it's supposed to guarantee?
  - And how can the users of your analysis be sure, if they don't want to trust you?
- *Proof-carrying code* (PCC) provides a general way of achieving such guarantees.
- We suggest some techniques for applying PCC to program analysis implementations...
  - ...and then apply the results to construct a more efficient PCC system!

Proof-Carrying Code



Reusable Trusted Code Base

Certifying Program Analyses

Traditional PCC	PCC for Certified Analyses
Prove memory safety and other low-level properties	Prove abstract interpretation soundness and other high-level correctness properties
Input programs fall into classes to which the same proof technique applies, thanks to the use of certifying compilers, etc.	Most analyses require new proof ideas
Must analyze every detail of the input program to prevent it from circumventing the safety policy	Program analyses usually decompose naturally into soundness-critical and non-soundness-critical parts

Our Strategy

- We want to verify the correctness of program analyses written in general purpose programming languages that support imperative state.
- Verify implementations at the source level
- Generate models in constructive type theory
  - These models can be almost identical to the original source code in most cases.
  - Model extraction takes advantage of common ways of decomposing analyses into *find* and *check* pieces.

Models of ML Programs

Precondition: $n \geq 0$

```
let rec double (n : int) : int =
   if n = 0 then
      0
   else
      2 + double (n - 1)
```

Postcondition: $result = 2n$

$$\forall n, n \geq 0 \Rightarrow$$
$$(n = 0 \Rightarrow 0 = 2n)$$
$$\wedge (n \neq 0 \Rightarrow \forall r, r = 2(n-1) \Rightarrow 2(2+r) = 2n)$$

Verification Condition

```
Fixpoint double (n : nat) : nat :=
   match n with
     O => O
   | S(n) => S (S (double n))
   end.
```

Coq Definition

Keeping a Close Correspondence

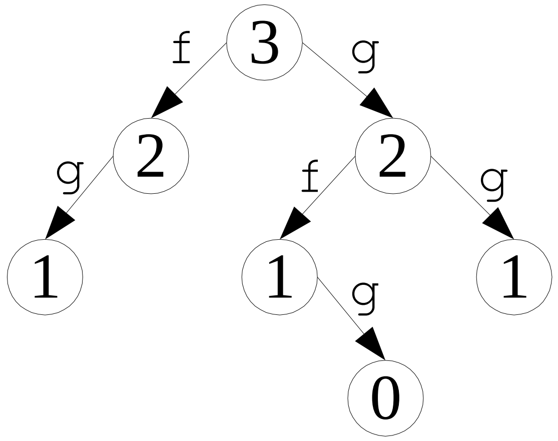
- We want to produce models that are as readable and close to the original program as possible...
  - ...because the theorems we need to prove about them are too hard for automated deduction tools.
- There are two main problems:
  - **Recursion.** Coq doesn't allow unrestricted recursive definitions, since they would threaten consistency.
  - **Mutable state.** Coq's functions are pure, so they can't support mutation without pervasive modification.
- Base on knowledge of our domain, we are able to approximate both of these in a nice way.
  - Relevant question: Do we really care whether analyses terminate and how they use imperativity?

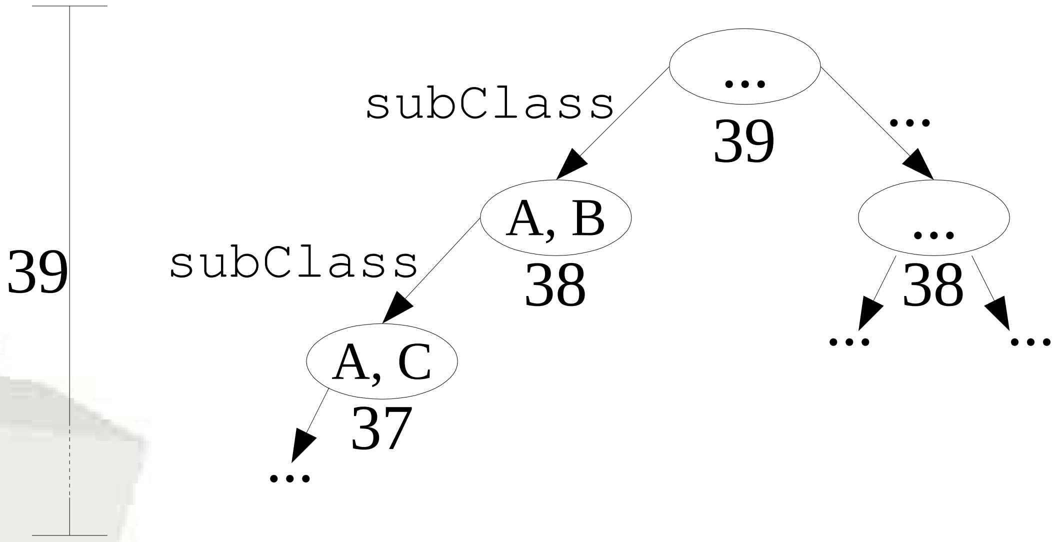
Recursion Example

```
let rec subClass (c1 : class, c2 : class) : bool =
    c1 = c2 ||
    (match super c1 with
        None -> false
      | Some sup -> subClass (sup, c2))
```

Producing Primitive Recursion

```
let rec f (n : nat) : t1 = ... f n ... g n ...
    and g (n : nat) : t2 = ... f n ... g n ...
```





Producing Primitive Recursion

- Add an extra argument to each function whose termination isn't clear.
  - This is a natural number giving *an upper bound on the remaining call stack depth*.
- Decrement this argument at each recursive call.
- For any terminating execution of the whole program, we can use the *real* call stack depth as the initial value for this parameter in the model.
- Non-terminating executions of program analyzers have no soundness consequences, so we don't need to worry about them!

Recursion Example Solution

```
let rec subClass (n : nat, c1 : class, c2 : class)
    : bool option =
    if c1 = c2 then
        Some true
    else match super c1 with
        None -> Some false
      | Some sup ->
          if n = 0 then
              None
          else
              subClass (n-1, sup, c2)
```

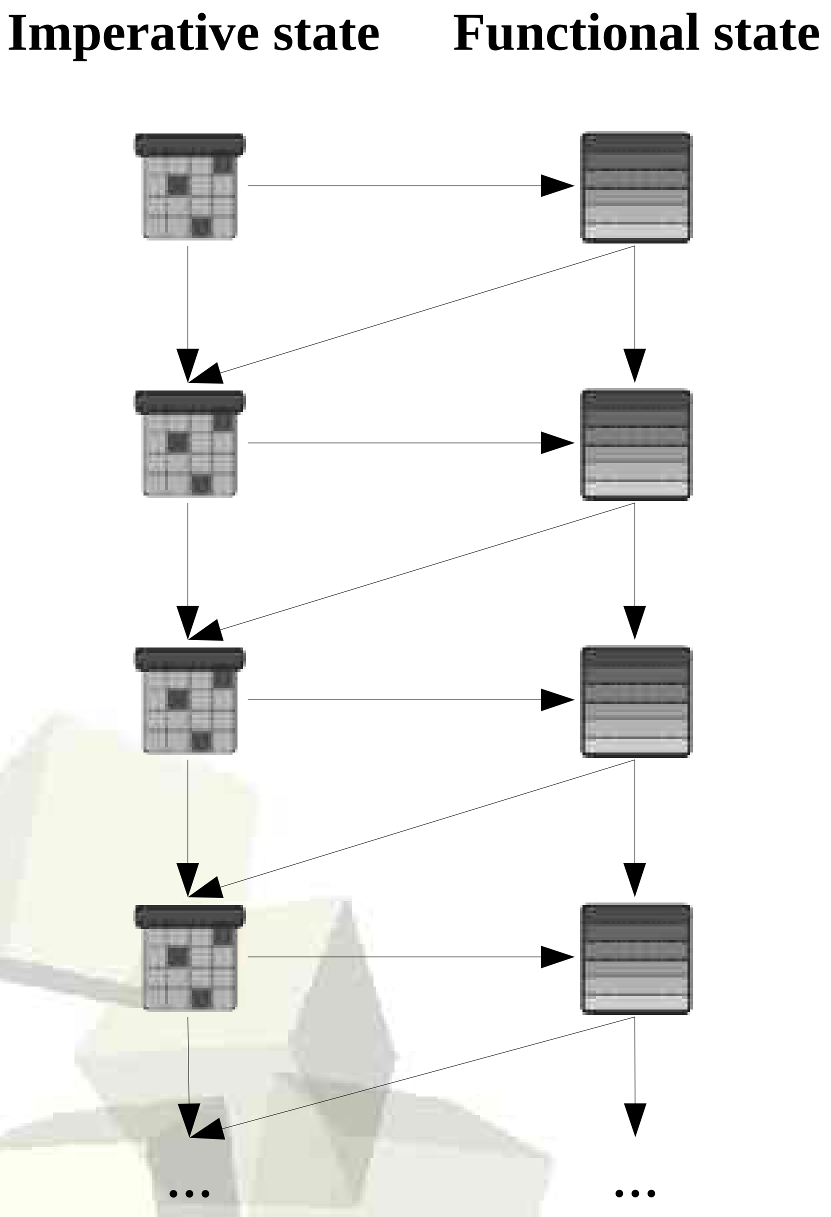
Mutation Example

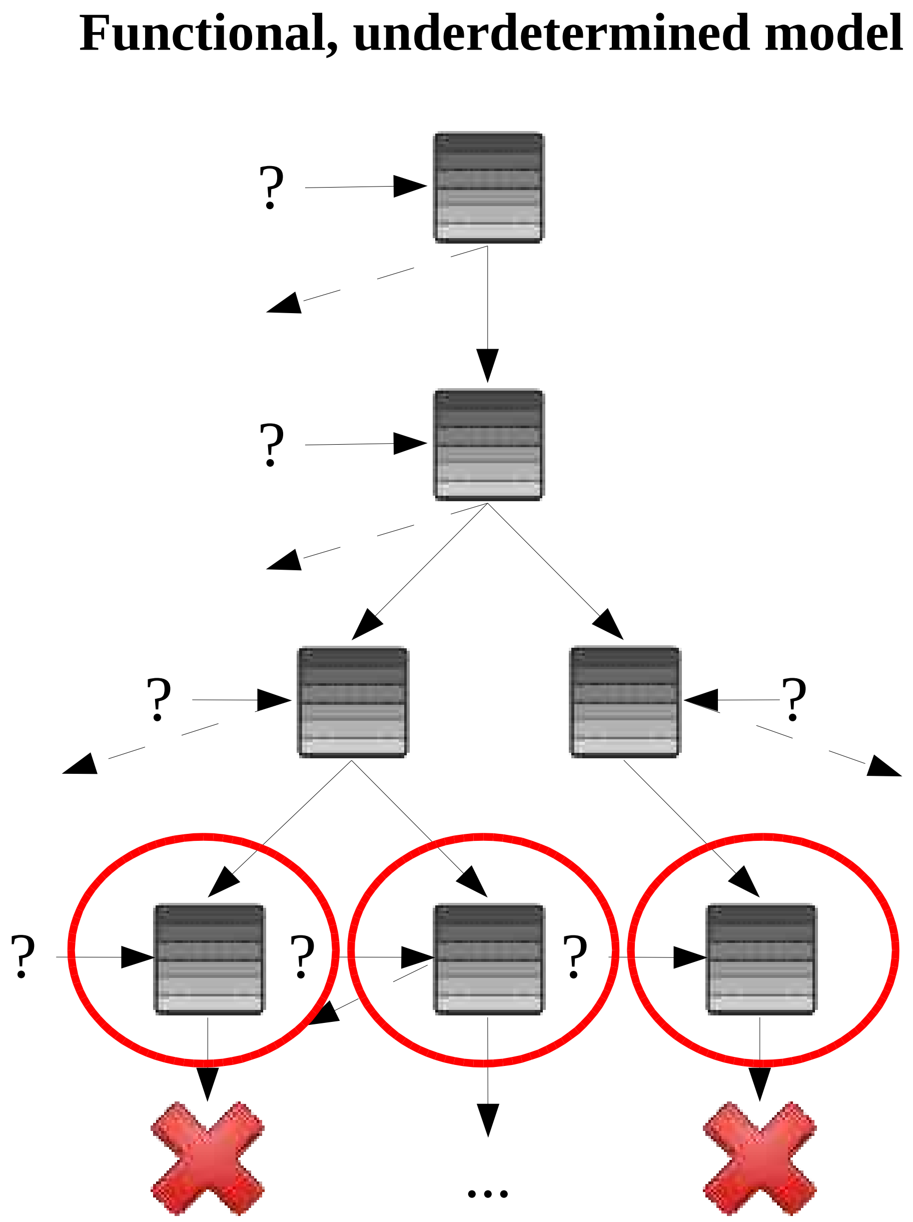
```
let checkProgram (p : program) : bool =
    let fixedPoint = findFixedPoint p in
    checkFixedPoint p fixedPoint
```

Mutation Example Solution I

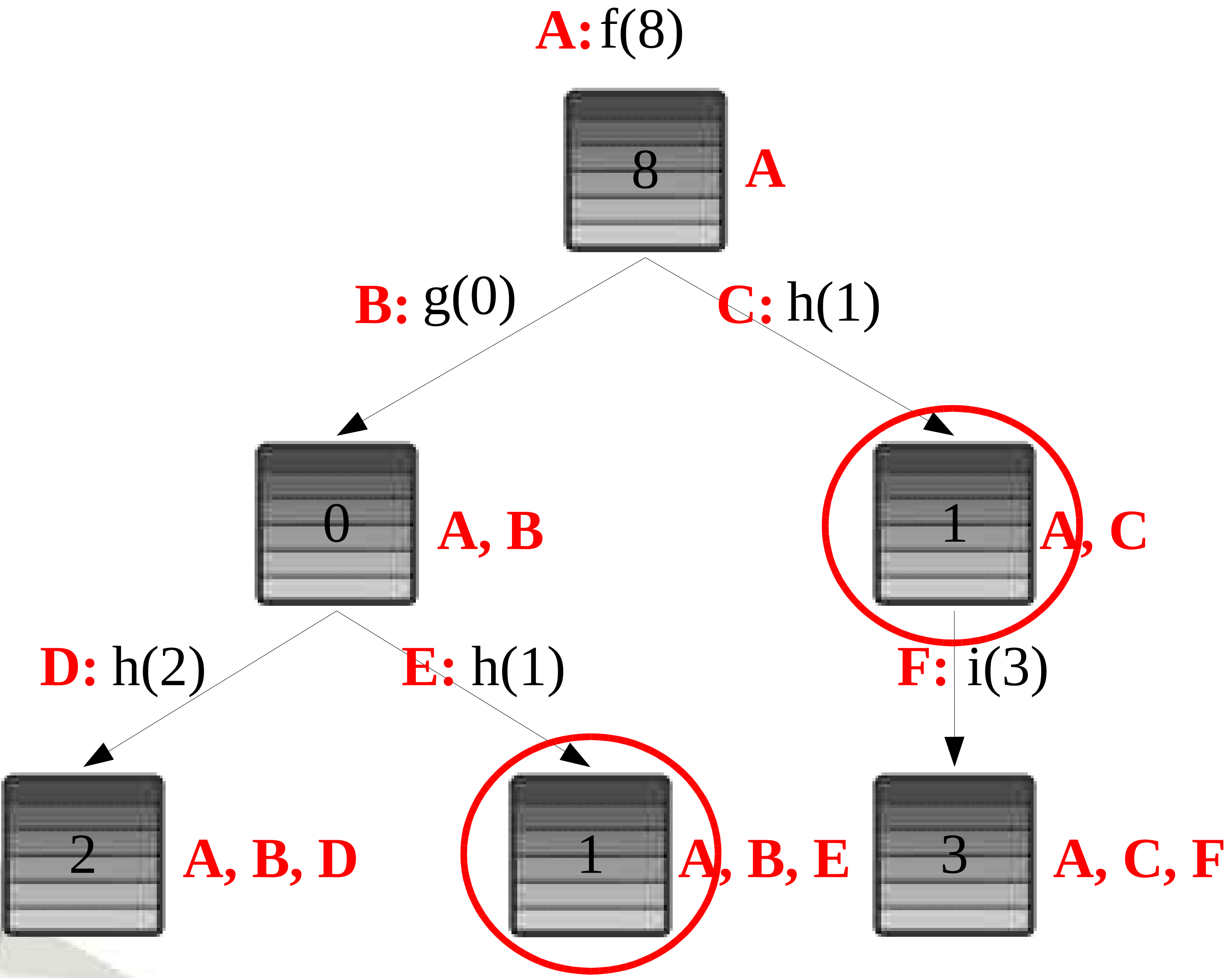
```
let checkProgram (s : state) (p : program)
    : bool * state =
    let (fixedPoint, s) = findFixedPoint s p in
    checkFixedPoint s p fixedPoint
```

Underdetermining Mutation





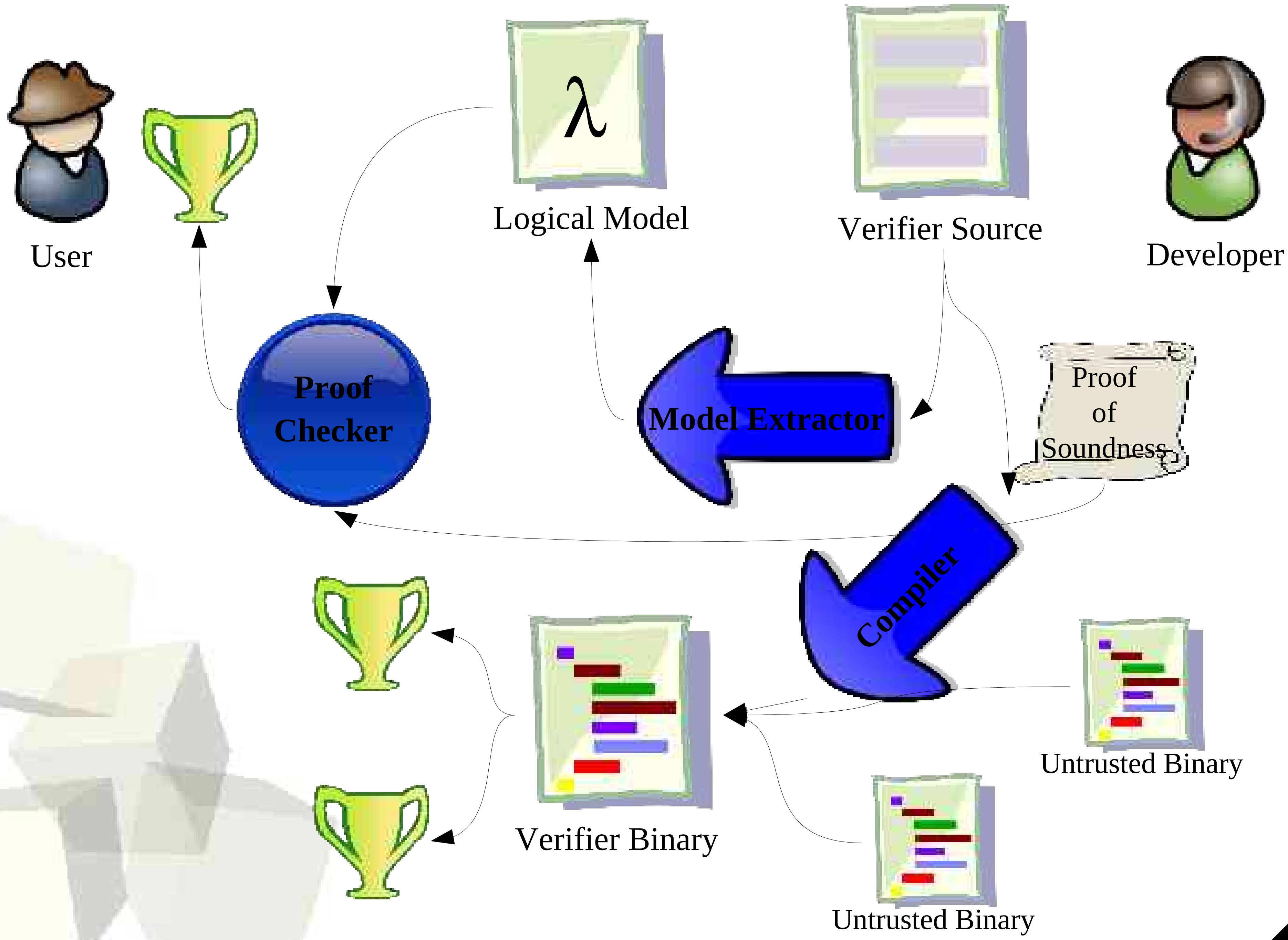
Modeling It Functionally



Mutation Example Solution II

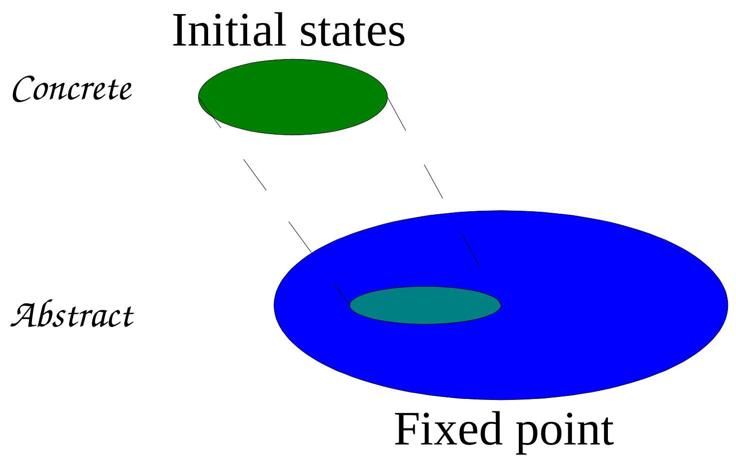
```
let checkProgram (s : stack) (p : program) : bool =
    let fixedPoint = findFixedPoint (call1 s) p in
    checkFixedPoint (call2 s) p fixedPoint
```

Application: Certified Verifiers

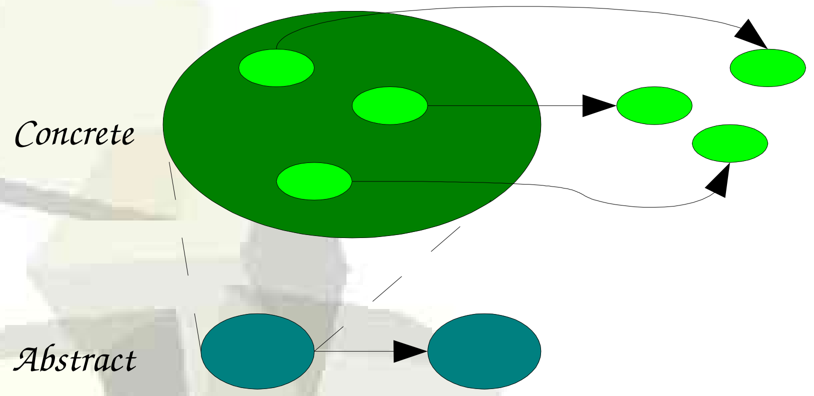


Soundness Proof Obligations

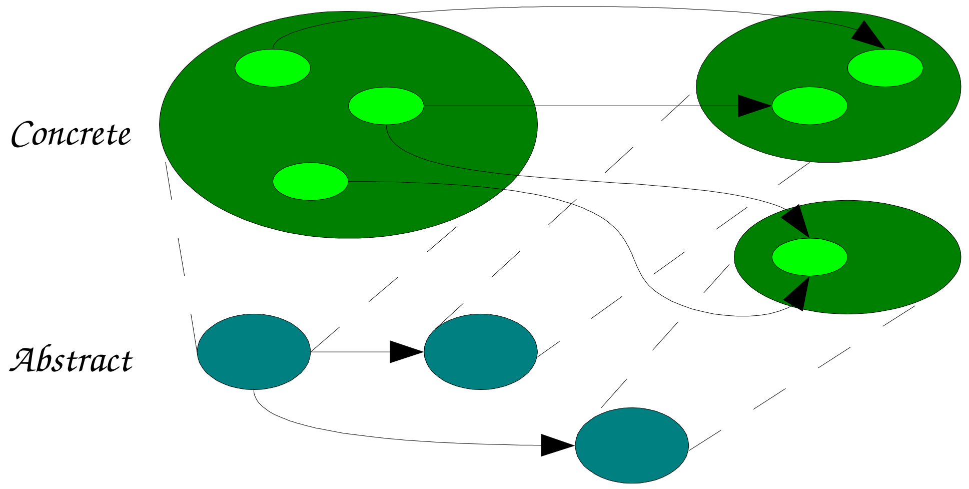
Initialization



Progress



Preservation



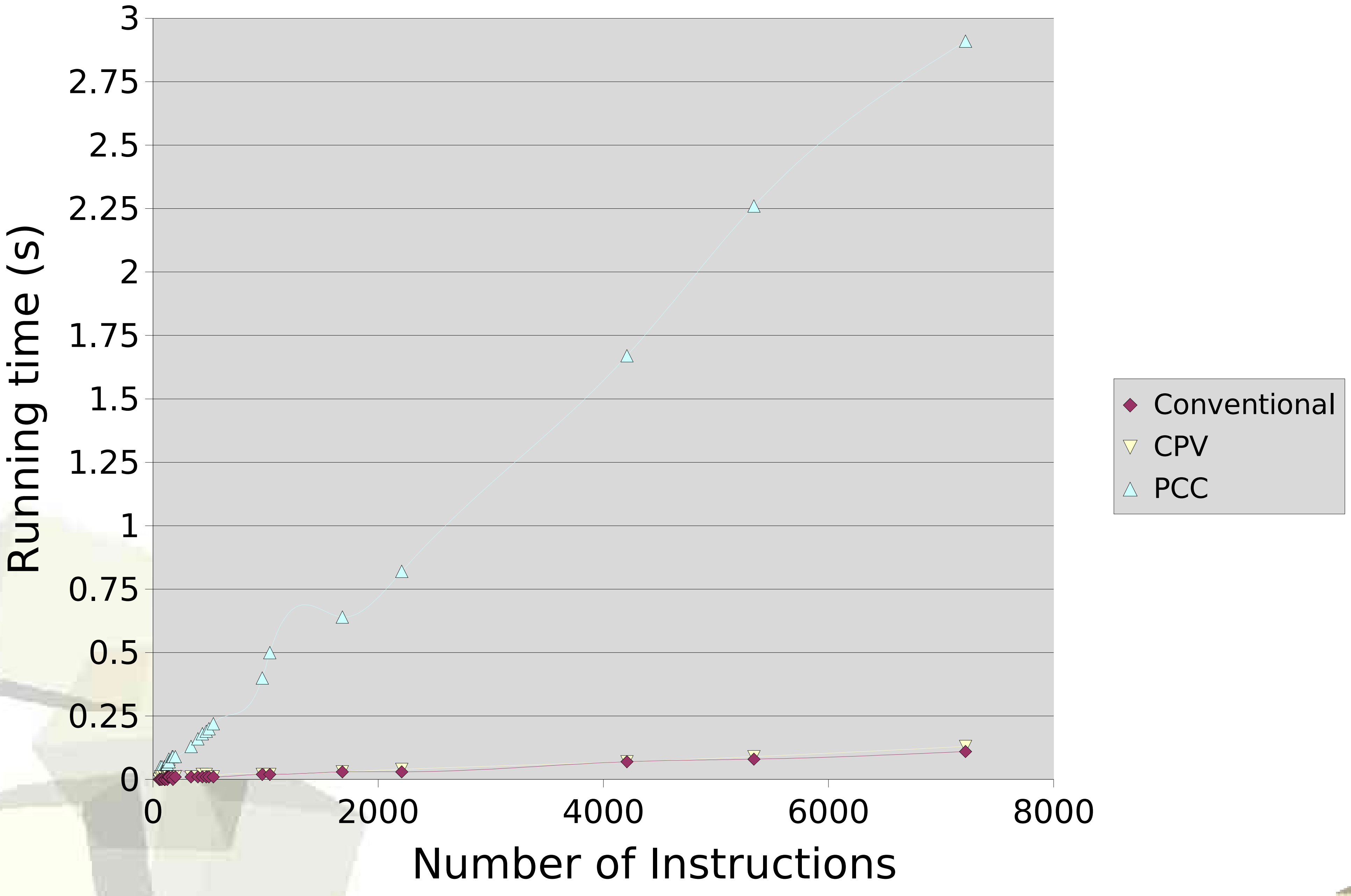
Implementation

- A 3000-line model extraction tool for a subset of OCaml
  - Hooks into the standard OCaml compiler
  - Produces Coq theorem statements, which we prove interactively
- About 2000 lines of OCaml for the core abstract interpretation framework
  - Including decoding assembly code, finding a fixed point with the provided abstract interpretation, etc.

Case Study: Typed Assembly Language

- We've written and proved sound a verifier for x86 assembly programs compiled from TALx86.
  - Includes continuation, universal, existential, recursive, product, sum, stack, and array types.
- Our implementation uses the provided TAL type system and compilers unchanged.
- We're able to verify memory safety of all of the test cases included with the distribution...
- ...about as efficiently as the non-certified type checker can.

Running Times of TAL Verifiers



Related Work

- Rhodium [Lerner et al.]
  - Works well for traditional compiler optimization problems, but isn't expressive enough for verification
- Foundational proof checkers [Wu et al.]
  - Proves goals using a trusted Prolog interpreter
  - Still forces everything to fit into one logic at run-time, which brings the usual performance penalty
- Extracting trustworthy verifiers from logical developments [Bertot, Cachera et al., Klein et al., ...]
  - No published performance figures

Future Work

- Decreasing the amount of trusted code
  - Reason about compiled verifiers instead of their source code
  - Perhaps using translation validation?
- Exploring ways to get some of the same benefits with techniques based on program extraction
  - Extraction has many nice theoretical and practical properties...
  - ...but we need an “optimizing extractor” to maintain the performance levels we've shown in this work.
    - E.g., data structure representation

Conclusion

- Certified program analyses provide strong soundness guarantees without sacrificing efficiency.
- We've used this idea to implement a Proof-Carrying Code-style system with performance comparable to an uncertified verifier.