Infinity, The Woodpecker, and the Trace that Eludes

Scalable Binary Verification via Co-Induction

Charles Averill

The University of Texas at Dallas
Dartmouth College

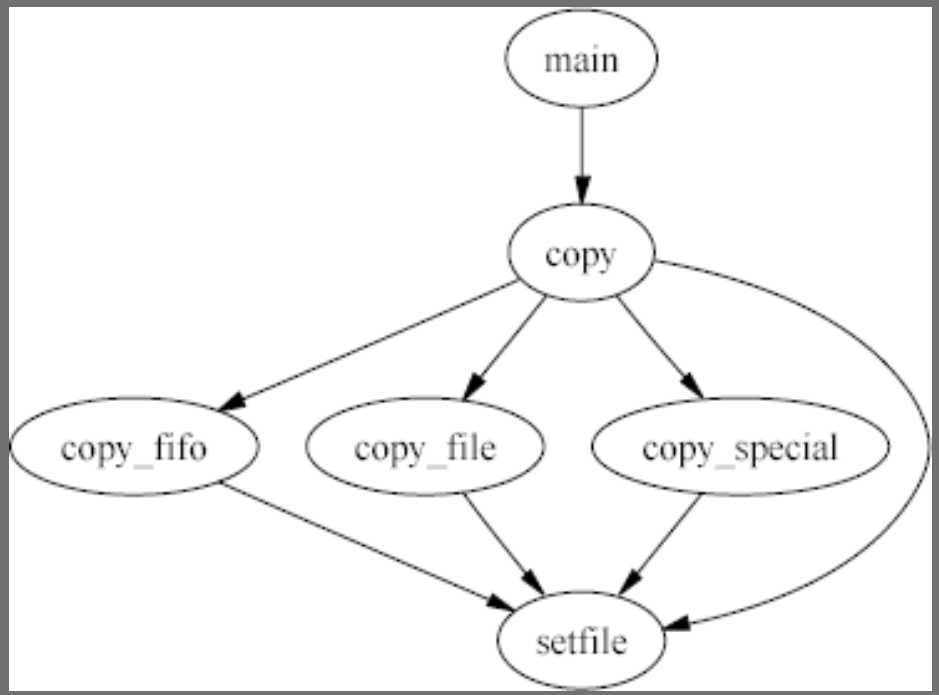
February 2026

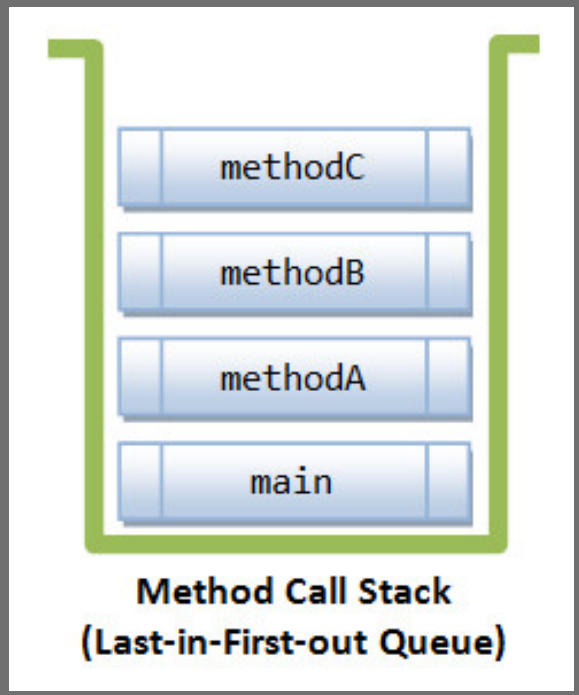
Formal Binary Verification Refresher

- Testing, auditing insufficient for critical systems (see Shellshock)
- Sound alternative? Machine-checked proofs of safety + correctness
- Top-down approach: write code in Rocq/Lean/Agda/Idris..., extract to OCaml/Haskell/C..., compile and run
  - Verified compiler (CompCert, CakeML)? What about libc?
  - No verified compiler? Can you trust transformations?
- Bottom-up approach: lift binary program into ITP, encode ISA semantics, proof logic, get a backwards guarantee back
  - I don't care what compiler you used (if you even used one)
  - Handles hand-written assembly (see musl memcpy.S)
- The future holds much of both approaches

We have ~~a million~~ a Problem

- Machine code is hard: bit math, IO, ISA instruction support, highly-optimized code, ...
- Our focus today: **function calls**
- Good software is usually modular - proofs should be too!





Duh?

- Is this really a "problem?" Don't verification environments (e.g., VST, Bedrock2) already support function calls? They do, for languages in which functions and calls are *first-class* notions!
- **No first-class notion of calls** in machine code, only "rich(er) gotos"

```
(* 4f8 => 0xcedff0ef (jal ra,1e4 <chacha20_quarter>) *)
Compute lift_riscv ChaCha20 0x4f8.
(* = Move R_RA 0x4fc $; Jmp 0x1e4 *)
```

- Gotos and **syntactic** Hoare logic do not go together — now we need a CFG-driven approach
- Used by lower-level tools (e.g., Bedrock, ~ Boogie, and Picinæ!)
- Verification via **invariant sets**, maps from labels to state properties
- Example

Syntactic Hoare Logic

$$\{P\}\ \texttt{SKIP}\ \{P\} \qquad \frac{\{P\}\ c\ \{Q\} \qquad P' \longrightarrow P}{\{P'\}\ c\ \{Q\}}$$

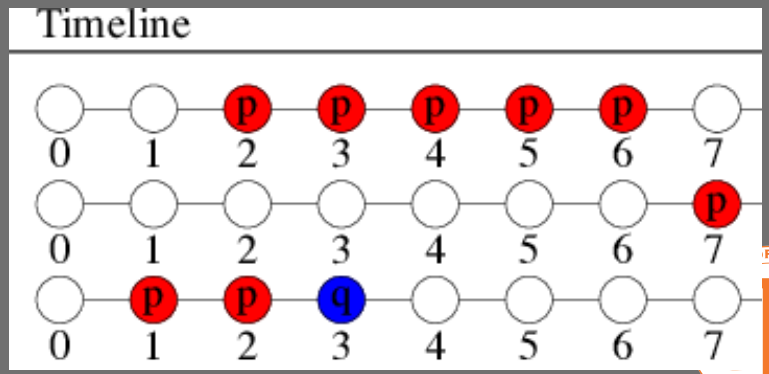
$$\frac{\{P \wedge \langle b \rangle\}\ c_1\ \{Q\} \qquad \{P \wedge \neg\langle b \rangle\}\ c_2\ \{Q\}}{\{P \wedge \langle\langle b \rangle\rangle\}\ \texttt{IF b THEN } c_1 \texttt{ ELSE } c_2\ \{Q\}}$$

$$\frac{\{P \wedge \langle b \rangle\}\ c\ \{P\} \qquad P \longrightarrow \langle\langle b \rangle\rangle}{\{P\}\ \texttt{WHILE b DO } c\ \{P \wedge \neg\langle b \rangle\}} \qquad \frac{\{P\}\ c_1\ \{Q\} \qquad \{Q\}\ c_2\ \{R\}}{\{P\}\ c_1 ;; c_2\ \{R\}}$$

Picinæ and Linear Temporal Logic

- I help maintain Picinæ, framework for symbolic binary verification
- Binaries have arbitrary jumps $\rightarrow$ we need a CFG-driven program logic
- Naïve approach: "a program is correct if invariants are preserved by every CFG edge"
- But this precludes "xyz eventually happens" (liveness), or reasoning about infinite loops (e.g. OS kernels, web servers)
- LTL gives us the "eventually" operator, allowing for "a program is correct if it *eventually* reaches a true invariant"

PICINAE



Back to the Problem

- Historically, Picinæ didn't support function calls
- Verifying large programs not really possible
- How much re-architecting do we have to do to get modularity?
- Well, what architecting happened in the first place?

Partial Correctness
"All invariants are satisfied for all non-terminated traces/program executions"

Invariants satisfied for trace {
- Currently at true invariant
- **Eventually** get to a true invariant

Program Execution
"All pairs in a trace are valid steps"

Small-Step
`eval_exp`
`exec_stmt`

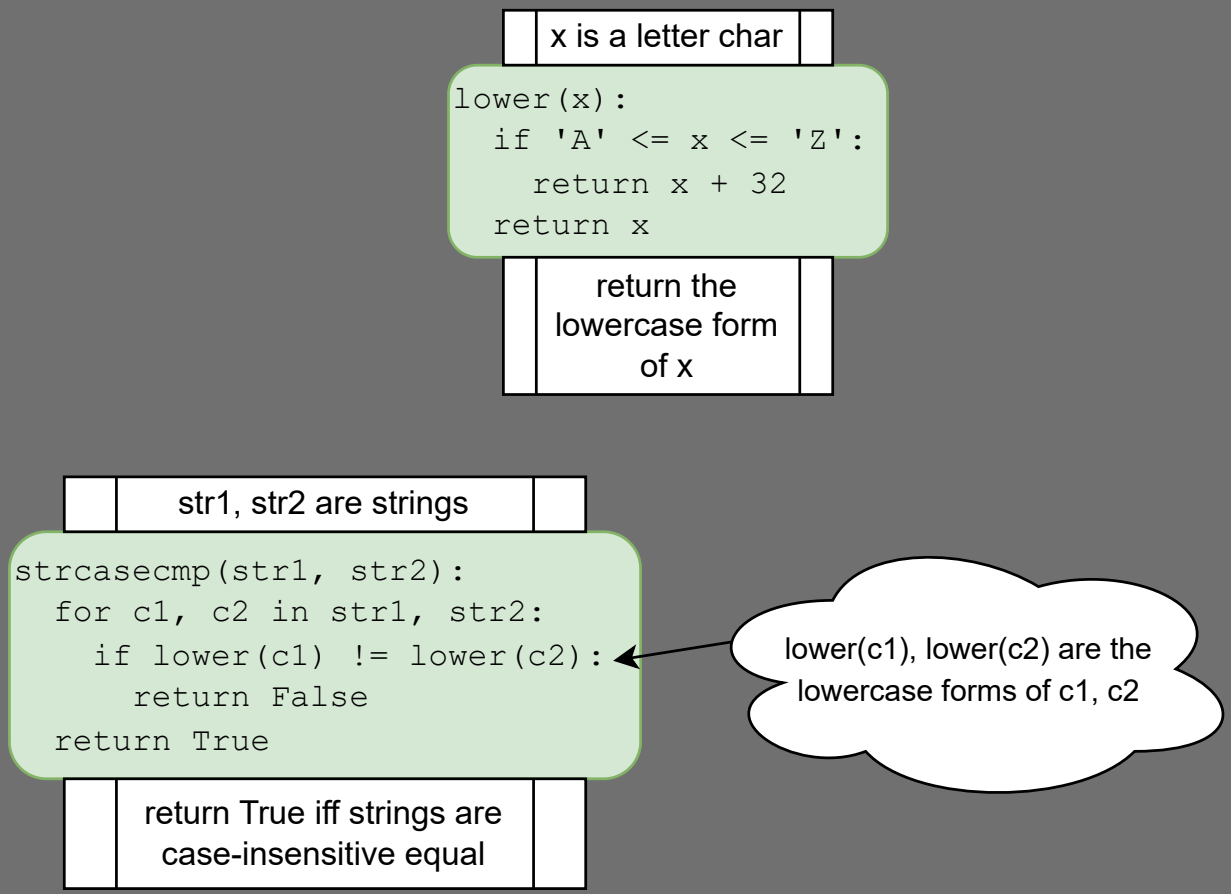
Picinæ Components

- Normal small-step semantics for expressions (`eval_exp`) and statements (`exec_stmt`)
- Program execution or **trace** (`exec_prog`) $\equiv$ a list of states where each neighboring pair is a valid small step
- The **next invariant** (`nextinv`) is satisfied (true when reached) if
  - Program is at a true invariant point
  - Program will eventually get to a true invariant point
- A terminating program is correct if all invariants are satisfied for all appropriate traces

Partial correctness (`satisfies_all`) is our top-level theorem, i.e., what we want to prove about *each function*.

Modularity

- **Goal**: define separate invariant sets, different `satisfies_all` proofs for caller/callee, and use callee `satisfies_all` result as a lemma

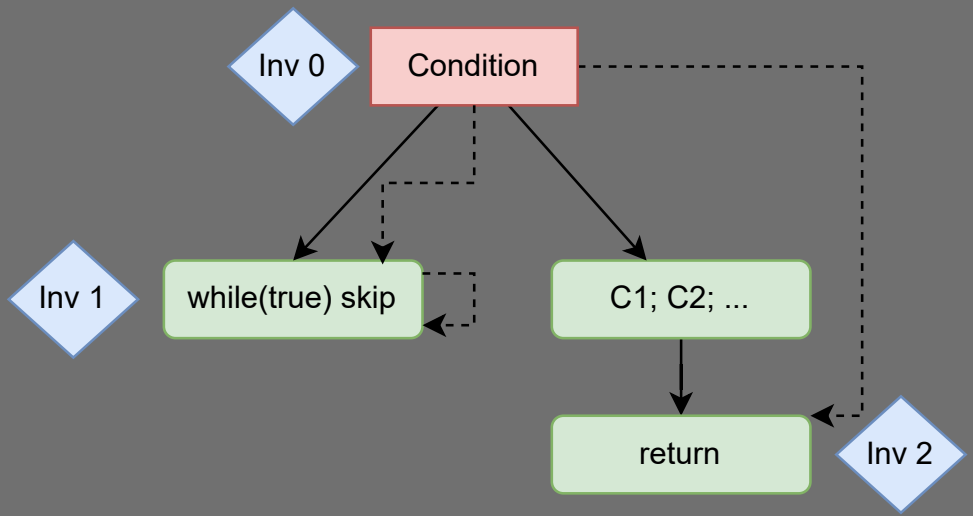


The Obvious(?) Approach

- Binary is flat list of bytes. Naïve proof considers programs this way with a monolithic invariant set. Any abstraction we build *must* be equivalent to this approach on some level
- Let's consider the distinct invariant sets + `satisfies_all` approach and try to show that it is equivalent to the naïve approach
- Something weird happens! What happens if you write a caller proof that does not include callee invariants?
- If the callee contains a potentially-infinite loop, any *inductive* proof of the callee must place an invariant there
- Remove the invariant? The proof could never have been constructed - contradiction!

Huh?

- This is very strange. We wrote an inductive proof of `satisfies_all` for the callee
- `satisfies_all` talks about invariants that are "true when reached..."
- If we remove invariants blindly, we allow infinite loops to invalidate our callee proof results
- But this seems to imply that we could call a function and end up with an infinite, *inductively-defined* trace upon returning to the call site



Co-Induction

- Can we just adjust our formalism to allow an infinite number of steps in callees? It never actually can happen, so it shouldn't be unsound...
- Introducing Co-Induction:
  - Co-Inductive proofs/data structures are *infinite* in size
  - Co-Recursive functions are *non-terminating*
- What if we make `nextinv a` CoInductive rather than an Inductive?