Proof Robustness in ACL2

Eric Smith

Kestrel Institute and Kestrel Technology

HCSS: High Confidence Software and Systems

May 3, 2021

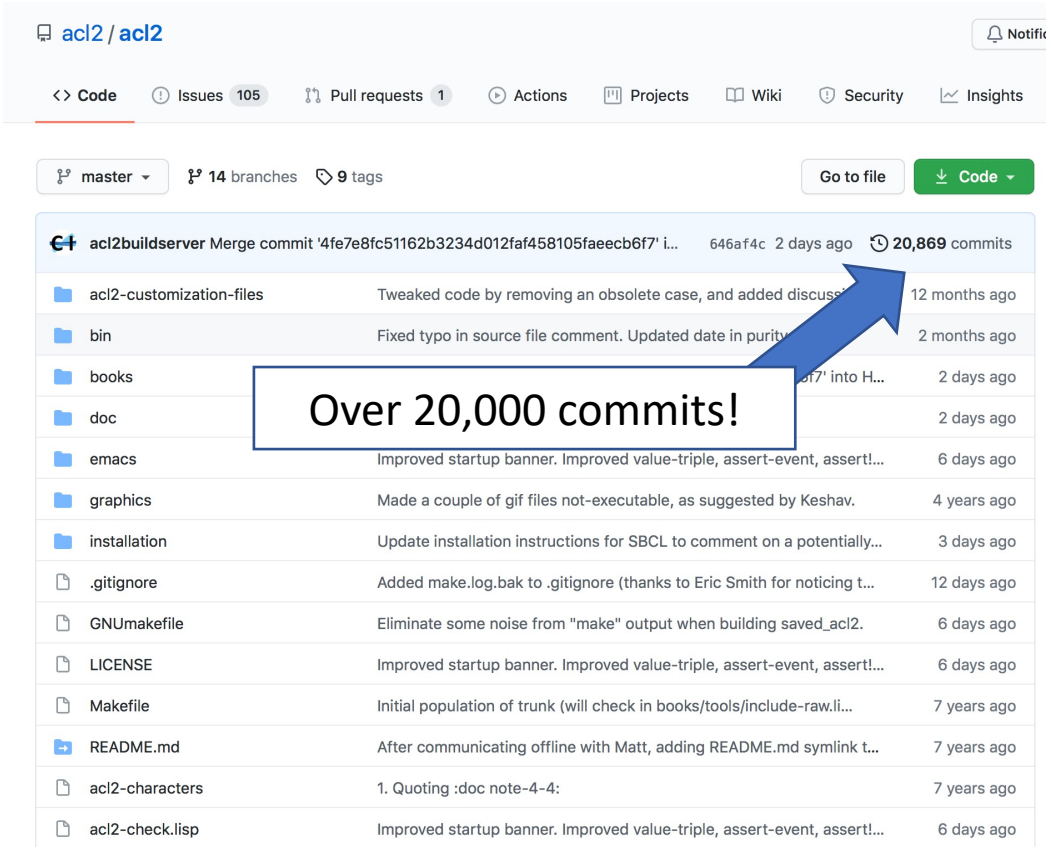
Who am I?

- Researcher at Kestrel Institute and Kestrel Technology
- 20 years of experience with ACL2
- Developed and maintained ACL2 proofs at:
  - UT Austin
  - AMD
  - Rockwell Collins
  - Stanford
  - Kestrel

- Opinions are my own
- Ideas and best practices taken from the broader ACL2 community

The proofs we maintain

- The ACL2 Community Books (github.com/acl2/acl2)
  - Developed over the past 30+ years
  - Specifications, proofs, libraries, tools, workshop material, etc.
  - Over 9,400 files
  - Over 90,000 function definitions
  - Over 170,000 theorems
  - Over 30,000 documentation topics
  - Very active: 10-20 new commits per day
- Kestrel's proprietary ACL2 libraries
  - Developed over the past 9+ years
  - Correct-by-construction software, proofs about Java and x86 programs, Flex theorem prover, etc.
  - Axe Toolkit (kestrel.edu/axe)
  - (Much of this is being open sourced, slowly)
  - Over 12,000 function definitions
  - Over 17,000 theorems



These proofs have great value, so we should try to keep them working.

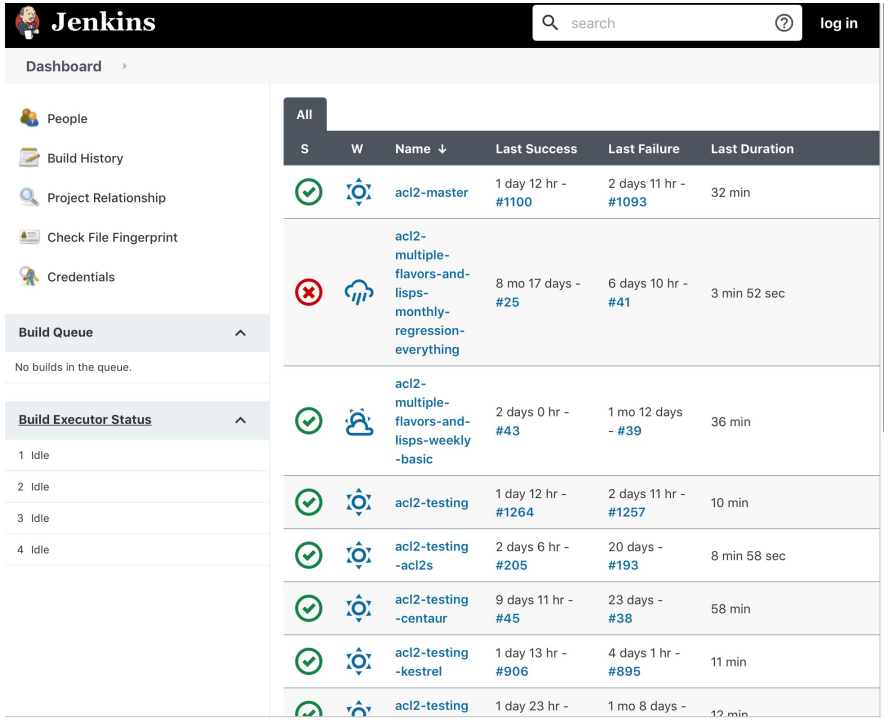
- The ACL2 libraries represent *person centuries* of human effort.
- Want to be able to build on this stuff
  - Use, extend, adapt, modify, fix
  - Start new projects without having to rebuild everything
  - Verify new versions of hardware / software
- Don't let "proof rot" happen to you!
  - Proofs depend on deep supporting hierarchies
  - Keeping a live proof working is usually easy
  - Reviving a dead proof after years of changes can be very hard
- **Regression suites let us test proposed changes**
  - To libraries, tools, or ACL2 itself
  - If new rules break existing proofs, you want to know about it!
  - Maybe your new rules are too expensive
  - … or don't respect preferred normal forms
  - Much of the value in a ACL2 development is in sets of rewrite rules
- Basic principle: We should always be able to re-play all of our proofs from scratch.

Changes that can break proofs

- Changes to definitions
  - Formal models
  - Specifications
  - Hardware or software artifacts (deeply or shallowly embedded)
  - Common changes: bugs fixed; features added; concepts generalized; functions renamed; arguments added, removed, or reordered; code moved or refactored, dependencies reduced
- Changes to reasoning libraries
  - Theories of lists, arithmetic, sets, bags, bit vectors, etc.
  - Common changes: New rules added that can change the course of proofs, rules replaced with more general versions, rules turned on or off
- Changes to supporting proof tools
  - Connections to SMT solvers
  - Custom theorem provers (e.g., my Axe prover)
  - Common changes: Interfaces, translations, rule sets
- Changes in ACL2 itself
  - Proof procedures, built-in utilities, system and interface tools
  - Conservative approach: Prefer backward compatibility, abort changes that break too many proofs (or slow things down too much), discuss breaking changes with the community

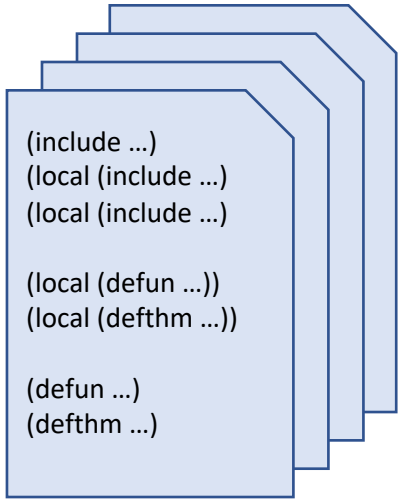
Infrastructure supporting proof robustness

- Scripts that provide single commands to build large sets of proofs
- Automated regression testing (http://leeroy2.defthm.com/)
  - Changes must pass regression testing before being merged into master
  - Multiple testing branches, any of which can get pushed to master when it passes
- File-level dependency tracking
  - Building everything takes a few hours, even with ~100 cores
  - But, ACL2 produces certificates for certified books
  - So, only build the stuff that depends on whatever has changed
- Key Idea: Detect failures quickly
  - Easier to debug the reason for failures



“Lightweight” Library Development

- Limit what is included by a file
  - Limits the effect of changes
  - A change to a library you don’t include can’t break your proofs!
  - “The best include-book is no include-book.”
- Also limit what is *exported*
  - Don’t make me use the arithmetic library you happened to use.
  - Don’t make me take your string library just to get your file-io library
  - Helps avoid name clashes, clashing normal forms, etc.
  - Key ACL2 construct: (local (include-book …))
  - “The second best include-book is a local include-book.”
- Embodied in libraries we are contributing the ACL2 Community Books
  - Arithmetic-light, lists-light, strings-light, file-io-light, etc.
  - Each book exports a minimal set of definitions and rules
  - “Just give me rules about expt”



Best Practices for Robust Proofs

- Developed over time by the ACL2 community
- Background: ACL2 proofs are done by supplying “hints” to the prover
  - “Turn this rule on”
  - “Expand this term”
  - “Instantiate this existing theorem”
  - “Split into these cases”
  - “Use this induction scheme”
- Some of these are more brittle than others

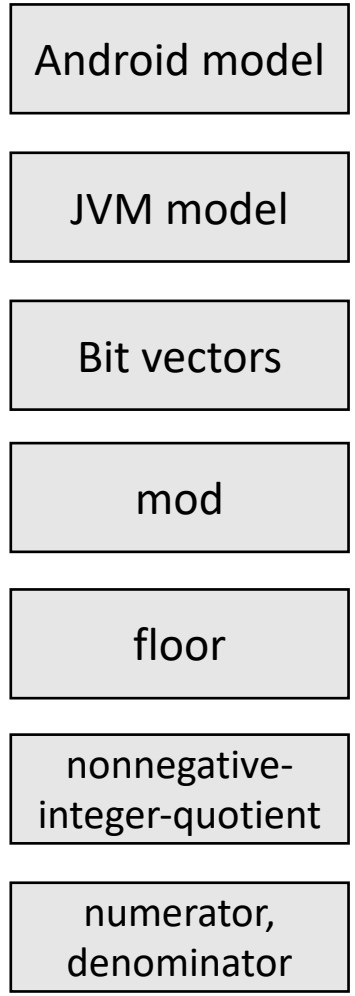
```
(defthm expt-of-*
  (equal (expt (* r1 r2) i)
         (* (expt r1 i)
            (expt r2 i)))
  :hints (("Goal" :in-theory (enable expt))))

(defthm *-of-floor-of-same-when-multiple
  (implies (and (equal 0 (mod y x))
                (rationalp y)
                (rationalp x))
           (equal (* x (floor y x))
                  y))
  :hints (("Goal" :cases ((equal 0 x)))))
```

Brittle Hints to Avoid

- Hints that depend on the detailed structure of the proof
  - :hints (("Subgoal 2.3.4" …*hints*…))
  - Means apply *hints* to the fourth subgoal arising from the third subgoal of the second main subgoal.
  - But subgoal numbering is not robust to prover or library changes
  - Better approach: Just attach the *hints* to "Goal", if possible
  - Even better: avoid the need for the hint
- Hints that mention particular pieces of syntax:
  - :cases hints
  - :use hints (consider making a rewrite rule instead)
  - Names may change, arguments may be added, removed, or re-ordered
- When your proof works, try to remove the hints.

An Ideal Development Style

- Functions developed in layers
  - Example: Bit vectors defined in terms of mod, expt, etc
- Proofs only require opening one layer of functions
  - Use rewrite rules about callees
  - Ideally, all hints are “enable” hints, or perhaps guidance on induction schemes
- Keep most function definition rules disabled
- Avoid nested inductions
  - Better to pull out the inner formula into a separate theorem (after generalizing it)
  - A sign that you are missing a nice fact / rule
- Avoid “Proof Builder” instructions
  - Alternate hint mechanism useful for exploring a proof
  - Can be brittle “Go to the third argument of the fifth hypothesis”)
  - Best not to leave them in your file
  - Instead, try to use conventional hints
- Prefer rules over hints
  - A hint would have to be given for every similar proof
  - A rule would rewrite that pattern every time it appears, from now on
  - **Much of the value of an ACL2 development is in the sets of rewrite rules developed**
  - Writing strong rewrite rules is an art…



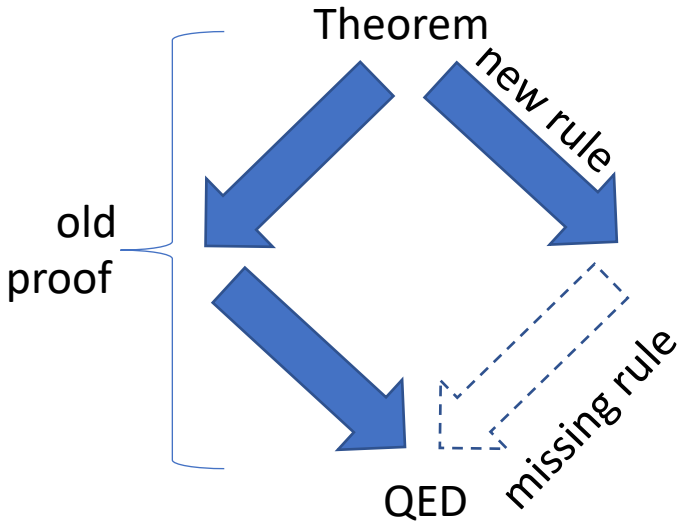
Robustness in *Generated* Proofs

- Many ACL2 proofs are generated by tools
  - Example: Kestrel's APT toolkit (kestrel.edu/apt)
  - Example: FTY library for typed programming (sum types, product types, etc)
  - Such macro libraries can save a lot of work
- But generated proofs must work in any context
  - No matter what rules are around
  - No matter what prover heuristics decide
- Different robustness principles apply than for hand-written proofs
- Best practices for robustness (also help in debugging failures)
  - Tightly control the theory (set of enabled definitions and rules), often turning off everything except a small set of desired rules
  - Make induction schemes explicit (don't rely on ACL2's heuristics)
  - Consider making instantiations explicit
  - Disable proofs techniques that won't be needed: generalization, destructor elimination
  - Break proof down into atomic steps

How to fix broken proofs

- Easy case: A name change, with a nice error message
- Hard case: Proof now takes a different path, reason for failure non-obvious

- Fix it ASAP
  - May be obvious: “I added one rule and this proof broke, so …”
- Sometimes additional rules can complete an alternate proof path
- If prover never finishes:
  - Debug rewrite loops (cw-gstack)
  - Re-run proofs in verbose mode (:set-gag-mode nil) to see what’s happening
  - Extreme case: Look at stack trace (set-debugger-enable, re-run and interrupt)
- Look for brittle hints
- Compare to a working version
  - Check out second copy, synced to old commit
  - Compare the proofs (emacs: M-x compare-windows)
  - Compare Induction scheme used, definitions expanded, rules used
  - If rules fail to fire, use “:monitor”
  - Advanced technique: grab a failing subgoal and submit it to the old session
- Rarely: Just re-do the proof
- *Very* rarely: Give up and remove the failing proof from the regression suite



Conclusion

- Keeping old proofs working has value
- … and is usually not too hard if you follow these principles.