Thesis Outline

Bolded points are points I anticipate becoming *\sections* or *\paragraphs*. Hopefully that helps you.

1. **Introduction**
    - Motivation for verifying systems
    - Era of scale---enter proof engineering (first survey paper mention)
    - Looking back (Social Processes), development has come a long way, but maintenance is still hard! And this is a problem in practice!
    - But missed opportunity: automation doesn't understand that proofs evolve
    - So we build automation that does, and we call this *proof repair*. Proof repair shows that there *is* reason to believe that verifying a modified system should often, in practical use cases, be easier than verifying the original the first time around.
    - Or, in other words (thesis statement): Changes in programs, specifications, and proofs carry information that a tool can extract, generalize, and apply to fix other proofs broken by the same change. A tool that automates this can save work for proof engineers relative to reference manual repairs in practical use cases.
    - Key technical bit: differencing and program transformations, taking advantage of the rich and structured language proofs are written in.
    - We implement this in a tool suite for Coq, get some sweet results.
    - Pave path to the next era of verification
    - **Reading Guide**
        - How to read this thesis
        - Mapping of papers to chapters
        - Authorship statements for included paper materials, to credit coauthors
        - Expected reader background & where to find more info
2. **Motivating Proof Repair**
    - Before we talk more about proof repair, it helps to know what it's like to develop and maintain proofs to begin with, and what happens under the hood when you do that. This chapter gives you that context, then explains the high-level approach to proof repair that builds on that.
    - **Proof Development**
        - Cartoon version of development: program, spec, proof
        - Proof assistants: short overview of foundations & different options (survey paper), then say focus on Coq
        - Slightly less brief overview of Coq and its foundations and automation and so on (including proof terms), going through a running example of proof development in Coq
    - **Proof Maintenance**
        - Problem is when something changes---change something in running example
        - There are a lot of development processes people use to make proofs less likely to break to begin with (survey paper)
        - But still, even with these, the reality: This happens all the time (REPLICA)
        - And in fact not just after developing a proof, but during development too (REPLICA)
        - And breaks proofs even for experts (REPLICA)
        - And it's an extra big problem when you have a large development and the changes are outside of your control
        - Hence Social Processes

        - Why automation breaks, even with good development processes
        - Hence proof repair---smarter automation
    - **Proof Repair**
        - Name inspired by program repair, but quite different as we'll soon see.
        - Recall thesis: Changes in programs, specifications, and proofs carry information that a tool can extract, generalize, and apply to fix other proofs broken by the same change. A tool that automates this can save work for proof engineers relative to reference manual repairs in practical use cases.
        - Proof repair accomplishes this using a combination of *differencing* and *program transformations*.
        - Differencing extracts the information from the change in program, specification, or proof.
        - The transformations then generalize that information to a more general fix for other proofs broken by the same change.
        - The details of applying the fix vary by the kind of fix, as we'll soon see.
        - Crucially, all of this happens over the proof terms in this rich language we saw in the Development section. This is kind of the key insight that makes it all work.
        - This is great because this language gives us so much information and certainty. This helps us with two of the biggest challenges from program repair. (generals related work)
        - But it's also challenging because this language is so unforgiving. Plus, in the end, we need these tactic proofs, not just proof terms. So we can't just reuse program repair tools. (generals related work)
        - So next two chapters will show two tools in our tool suite that work this way, how they handle these challenges, and how they save work.
3. **Proof Repair by Example** (PUMPKIN PATCH)
    - The first tool (PUMPKIN PATCH) focuses on changes in programs and specifications, though these changes are limited in scope as we'll see later.
    - What this tool does is, when programs and specifications change and this breaks a lot of proofs, it lets the proof engineer fix just one of those proofs. It then generalizes the example patch into something that can fix other proofs broken by the same change.
    - So in other words, the information from those changes is carried in the difference between the old and new version of the example patched proof.
    - PUMPKIN PATCH generalizes that information.
    - Application can be automated in some cases at the end, or it can be manual.
    - The work saved is shown retroactively on case studies replaying changes from large proof devleopments in Git. Results for this tool are preliminary compared to what we'll see later, since this was the first prototype.
    - **Motivating Example** (PUMPKIN PATCH intro & automation, reimagined)
    - **Approach** (parts of PUMPKIN PATCH Motivating the Core, plus more)
        - Like I mentioned earlier, this works using differencing and program transformations. And of course all of this happens over proof terms.
        - Here's the system diagram.
        - Here, differencing thus looks at the difference between versions of the example patched proof for this information, and finds something called a patch candidate---which is localized to the context of the example, but not enough to fix other proofs broken by the change.
        - Then, program transformations generalize that candidate to a reusable proof patch, something that can fix other proofs broken by the same change.
        - Application works with hint databases or is manual.

- **Differencing** (parts of PUMPKIN PATCH Motivating the Core, Inside the Core, Testing Boundaries, Future Work)
  - How differencing works in detail
  - Limitations and whether they're addressed in later tools yet or not
- **Transformation** (parts of PUMPKIN PATCH Motivating the Core, Inside the Core, Testing Boundaries, Future Work)
  - How the four transformations work in detail
  - Limitations and whether they're addressed in later tools yet or not
- **Implementation** (parts of PUMPKIN PATCH Inside the Core, plus more)
  - **Tool Details**
  - **Workflow Integration**
- **Results** (PUMPKIN PATCH Case Studies, key technical results)
- **Conclusion**
  - Rehashing thesis and how we do it
  - What we haven't accomplished yet at this point (parts of PUMPKIN PATCH future work), segue into next chapter

4. **Proof Repair Across Type Equivalences** (PUMPKIN Pi + DEVOID)
   - This extension to the suite adds support for a broad class of changes in datatypes, handling a large class of practical repair scenarios.
   - What this tool (PUMPKIN Pi) does is, when datatypes change and this breaks a lot of proofs, it generalizes the change in datatype itself (possibly with some user input) so that it can automatically fix proofs broken by the change in datatype.
   - So in other words, the information from those changes is carried in the difference between the old and new version of the changed datatype, possibly with some user input.
   - PUMPKIN Pi generalizes that information and applies it automatically.
   - The work saved is shown on a lot of case studies (see Table from PUMPKIN Pi).
   - **Motivating Example** (PUMPKIN Pi motivating example)
   - **Approach** (parts of PUMPKIN Pi intro, problem definition, plus more)
     - Like I mentioned earlier, this also works using differencing and program transformations. And of course all of this happens over proof terms.
     - Here's the system diagram.
     - Here, differencing thus looks at the difference between versions of the changed datatype, and finds something called a type equivalence. I'll explain that with examples. Sometimes differencing is automatic, and sometimes it's manual.
     - Then, program transformation ports proofs across the equivalence directly. So they take care of application.
   - **Differencing** (DEVOID 3.1 and 4.1, with some more general things from PUMPKIN Pi and more)
     - How differencing works in detail
     - Limitations and whether they're addressed in other tools yet or not
   - **Transformation** (parts of PUMPKIN Pi Transformation, with DEVOID 3.2 and 4.2 as examples, plus some of the beautiful Carlo theory to explain why we go from equivalences to configurations and what that really *means*)
     - How the transformation works in detail
     - Limitations and whether they're addressed in other tools yet or not
   - **Implementation** (parts of PUMPKIN Pi and DEVOID implementation)

        - **Tool Details**
        - **Workflow Integration** (PUMPKIN Pi Decompiler and Implementation)
    - **Results** (PUMPKIN Pi Case Studies, key technical results)
    - **Conclusion**
        - Rehashing thesis and how we do it
        - What we got here beyond what we had in PUMPKIN PATCH, segue into next chapter
5. **Related Work** (from various papers, incomplete, will figure out later, may be redundant with earlier stuff)
    - **Programs**
        - **Program Refactoring & Repair** (anything not covered before)
        - **Ornaments**
        - **Programming by Example**
        - **Differencing & Incremental Computation**
    - **Proofs**
        - **Proof Reuse**
        - **Proof Refactoring & Repair**
        - **Proof Design**
        - **Proof Automation**
        - **Transport**
        - **Parametricity**
        - **Refinement**
6. **Conclusions & Future Work**
    - Reflect on thesis statement and explain how we got it exactly now that you know everything
    - But I want to spend the resst of this thesis talking about the next era of verification so I can write out a bunch of ideas for students who might want to work with me
    - **Proof Engineering for All** (Future Work from many papers, plus research statement, DARPA thoughts, plus more, but trimmed down a lot)
        - What I want in the long run, how this all fits in, is a world of proof engineering for all. From research statement, three rings (four including experts in the center).
        - And what we have so far with my thesis is a world where it's easier for experts and a bit easier for practitioners, but there's still a lot left to go building on it.
        - So here are 12 short future project summaries that reach each of these tiers, building that world. Super please contact me if any of these seem fun to you.
        - **Experts** (unifying theme: lateral reach), some examples:
            - Supporting other proof assistants (thoughts from PUMPKIN Pi on Isabelle/HOL, future work from PUMPKIN PATCH)
            - Difficult changes: version updates, isolating large changes (PUMPKIN PATCH), relations more general than equivalences (PUMPKIN Pi)
            - ML for decompiler (PUMPKIN Pi, REPLICA) : more for diverse proof styles (PUMPKIN PATCH). Note that this is a WIP, but sketch out project, challenges, future ideas, expectations, evaluation a bit
        - **Practitioners** (unifying theme: usability), some examples:
            - Lots more automation (more search procedures for automatic configuration, e-graphs from PUMPKIN Pi, custom unification heuristics)
            - IDE & CI integration, HCI for repair, repair challenge, user studies ideas (PUMPKIN PATCH, REPLICA, panel w/ Benjamin Pierce, QED at large)

        - ?? (look for more in survey paper, email, DARPA TAs)
    - **Software Engineers** (unifying theme: mixed methods verification, or the 2030 vision from Twitter thread), some examples:s
        - A continuum from testing to verification, tools to help with that
        - Tool-assisted development to follow good design principles for verificattion (James Wilcox conversation, final REPLICA takeaway)
        - Analysis to infer specs (TA1)
    - **Domain Experts** (unifying theme: collaboration, new abtstractions for new domains), some examples:
        - Fairification & other ML correctness properties
        - ?? (look for more in survey paper, email, DARPA TAs)
        - ?? (look for more in survey paper, email, DARPA Tas)

Dan: It might be good either in 1 or in both 3 and 4 to lay out the key /results/ -- you have _design_ of the transformation, the highly non-trivial _implementation_ in Coq in a way that doesn't extend the TCB, and you have substantial case studies to _evaluate_.You have a bit of formalism and metatheory, but that is more for exposition than for specific results (am I right?) -- you are guided by the theory to build tools that actually work in this unforgiving domain -- "hacks won't work".

Dan: For the future work, I think the "experts" and "practitioners" sections have content that definitely fits naturally. The rest is a bit broader well beyond the focus of the thesis statement and work itself. That's fine if the writing flows -- maybe just explicitly acknowledging that or something will help it flow.