Automated Sketching and Repairing of Large Mechanised Proofs

Chengsong Tan*†, Alastair F. Donaldson*, John Wickerson*, and Jonathan Julián Huerta y Munive‡
*Imperial College London, UK. Email: {alastair.donaldson, j.wickerson}@imperial.ac.uk
†Kaihong, China. Email: tanchengsong@kaihong.com
‡Aalborg University, Denmark. Email: jjhymuni@cs.aau.dk

Formal verification with Interactive Theorem Proving (ITP) provides high assurance of software quality, but it is time-consuming and difficult to implement properly. This is perhaps the biggest reason why the industry has yet to see more widespread adoption of formally verified software, despite various success stories [1], [2], [3]. This extended abstract summarises ongoing extensions to a conference paper [4] on automation tools for large mechanised proofs. Our target workflow supports proof engineers who iteratively refine conjectures and use automation to confirm assertions, identify errors, and to minimise human effort in each cycle. When proofs span tens of thousands of subgoals, automating the mundane parts of this process can determine whether a verification project succeeds or stalls. We aim to address the automation challenges that hinder such workflows by developing tools that streamline interactions among the human expert, the proof assistant, and automation tools.

Motivating problem. Our work originates from a formal model of `CXL.cache` [5] in Isabelle/HOL, comprising 74 theory files and roughly 310k lines of code, where we proved the Single-Writer-Multiple-Reader (SWMR) property to establish cache coherence [6]. The proof requires finding an *inductive invariant* P—a conjunction of atomic formulas—preserved by all m transition rules of the system. This yields an $m \times n$ proof obligation matrix (fig. 1), where cell (i, j) asserts that rule i preserves conjunct j. We call each row a *rule lemma* (one rule against all conjuncts) and each column a *conjunction lemma* (one conjunct against all rules).

What we expected to be a moderate 70×2 proof grew well beyond our initial estimate. Some conjuncts were not preserved by certain rules, requiring us to strengthen P with additional conjuncts—which themselves were not preserved, necessitating further additions. This process expanded P to nearly 800 conjuncts, yielding a 68×796 matrix with over 54,000 proof obligations. Each iteration required regenerating and rechecking a large fraction of these proofs, which became infeasible with standard Isabelle tooling. This inspired us to develop **`super_sketch`**, **`super_fix`**, and the extensions described below.

First tools: `super_sketch` and `super_fix`. Rather than manually invoking Sledgehammer on each subgoal, we built **`super_sketch`** inspired by Haftmann's **`sketch`** tool [7], which generates Isar [8] proof skeletons without generative AI. Our **`super_sketch`** tool extends this by retrieving all proof goals after applying user-suggested methods and concurrently invoking Sledgehammer and user-supplied heuristics on them. For harder subgoals, it applies additional splitting and reduction steps before retrying. Using **`super_sketch`**, we could regenerate proofs for a rule lemma (covering over 700 conjuncts) in under 30 minutes, amounting to about one working day of machine time to iterate through the entire obligation matrix—with fewer than 100 failures out of over 50,000 goals.

To handle the remaining failures—and errors caused by upstream changes to definitions or the invariant—we developed **`super_fix`**, which emulates a proof engineer's debugging workflow: it sequentially processes a theory file, detects non-terminating proofs (via timeouts), misaligned proof obligations, and failed method applications, and attempts automatic repairs. A hybrid approach that combines scripting and these tools produced 68 theory files with 796 goals each. After **`super_sketch`**, 18 files contained unfinished proofs (13 with at most 2 errors; the remaining 5 with 14, 10, 6, 6, and 4 errors). A first pass of **`super_fix`** reduced this to 9 files with 1 error each; a second pass completed the proof entirely.

Ongoing extensions. We are extending our work along three directions. Firstly, we developed **`meta_sketch`**, a recursive generalisation of **`super_sketch`** driven by a small DSL that lets users tailor the proof-generation pipeline to their problem. The DSL accepts a hierarchical sequence of *splitters*—methods that decompose goals into subgoals—together with a concurrent portfolio of leaf-solving strategies with priorities. Its leaf engine employs a three-stage pipeline: racing, rescuing, and last-resort splitting. This is implemented via Isabelle/ML's `Future` constructs for "winner-takes-all" efficiency with prompt cancellation of losing branches.

The command **meta_sketch** encapsulates what previously required Python scripting and manual orchestration into a single, configurable command. An example usage of its previous version (named differently) appears in Figure 4

Secondly, we have integrated generative AI into the Isabelle proof assistant to explore and compare its sketching or **sorry**-removal abilities against our tools. Specifically, we have provided infrastructure for interacting with LLMs in Isabelle via a client-server architecture. That is, we have enabled Isabelle to call a Python-based server that can host local Hugging Face (HF) [9] models, query other LLM-based servers such as Ollama [10], or use other APIs from state-of-the-art models such as ChatGPT [11] or Google's Gemini [12]. We have leveraged the DeepIsaHOL project's libraries [13] to build a collection of Standard ML functions in charge of sending the required data to the LLM server. These library implementations result in two main commands for interacting with LLMs in Isabelle:

The command **llm_recommend** receives a line number pointing to the line immediately after the statement of a conjecture in a `.thy` file. As its name suggests, this function queries the LLM to suggest the next step in the proof. The command **llm_try_proof** iterates the same querying process to incrementally complete a proof string. Its implementation relies on a depth-first search (DFS) algorithm that returns when Isabelle confirms that a proof has been obtained or when a maximum number of proof (depth) steps is attained (Figure 3). At the moment, HF models can provide several suggestions, enabling the DFS to produce various candidate proofs. However, we only support one suggestion at a time for Ollama models. The proof depth is configured on the Isabelle side, while the breadth of the proof exploration comes from the model's (server) configuration file. The extension of these commands to a sorry-first approach [14] is ongoing work.

Thirdly, we have observed that most of the operations defined while working on improving **super_sketch** and **super_fix** are functions that map states in an ITP to strings $f : S \to \Sigma^*$. Namely, the interaction between a user and an ITP can be formalised as a sequence of state transitions $\sigma_i \to \sigma_{i+1}$ driven by strings $s_i \in \Sigma^*$, where a proof is successful if it reaches a state σ_m free of proof obligations and errors. Thus, we have explored the modular refactoring of our tools using these *fixer functions*. For instance, **try_sketch**, which is our reimplementation of **super_sketch** using fixer functions, decomposes a goal into subgoals, generates preliminary "fixer strings" with **sorry** placeholders, and concurrently invokes user-supplied fixer functions to replace them with valid proofs (see Figure 2). Similarly, **super_fix** leverages this model to sequentially traverse theory files, detect error states or **sorry** commands and apply fixer functions to repair them. An example of its usage appears in Figure 5. We are exploring the benefits of this re-implementation and have initially observed efficiency improvements when comparing **try_sketch** against **super_sketch**.

Finally, we have constantly evaluated our developments. For instance, we stress-tested Isabelle with our tools feeding them an obligation matrix comprising up to 54,128 subgoals (a 68×796 subset of the original obligation matrix consisting of up to 54,128 subgoals (a 68×796 subset of the original one). The experimental data demonstrates that Isabelle effectively handles large-batch processing, achieving an average throughput of approximately 0.90s per goal. However, the evaluation revealed a significant performance trade-off: while concurrent execution is necessary to avoid the failures of sequential processing, excessive parallelism introduces thread contention, memory pressure, and scheduler delays. This contention frequently inflates the latency of trivial methods, triggering a "timeout cascade" that over-approximates failure rates—estimated between 16% and 40% using a **sorry**-based metric. Future work aims to stabilise these dynamics through adaptive timeouts and refined concurrency scheduling. We have made our tools publicly available [15].

Further scoping of the problem. Given the newly developed capabilities of generative AI for theorem proving [14], we aim to identify the biggest pain points on the "automation wishlist" of the ITP community. To this end, we have designed a survey that, among other things, asks experts which parts of the proof engineering process they would most like to automate. We are considering selecting a few of the respondents to participate in a more in-depth interview to understand how their automation needs can be accommodated. The plan is then to conduct case studies, informed by the survey results, and to develop and assess tooling to alleviate these pain points, beginning with the case study above from our own experience. This is ongoing work, and the survey results will inform future directions of our case study.

Acknowledgements

This work was supported by EPSRC grant EP/R006865/1 and a Horizon MCSA 2022 Postdoctoral Fellowship (project number 101102608). The first author is also supported by the National Science and Technology Major Project of the Ministry of Science and Technology of China (Grant No. 2024ZD0803002).

References

[1] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. A. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: formal verification of an OS kernel." in *Proceedings of the 22nd ACM Symposium*

on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009, J. N. Matthews and T. E. Anderson, Eds. ACM, 2009, pp. 207–220. [Online]. Available: https://doi.org/10.1145/1629575.1629596

[2] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens, "CakeML: a verified implementation of ML," in *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, S. Jagannathan and P. Sewell, Eds. ACM, 2014, pp. 179–192. [Online]. Available: https://doi.org/10.1145/2535838.2535841

[3] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, no. 7, pp. 107–115, 2009. [Online]. Available: https://doi.org/10.1145/1538788.1538814

[4] C. Tan, A. F. Donaldson, J. J. Huerta y Munive, and J. Wickerson, "The burden of proof: Automated tooling for rapid iteration on large mechanised proofs," in *2025 IEEE/ACM 13th International Conference on Formal Methods in Software Engineering (FormaliSE)*, 2025, pp. 34–45, https://doi.org/10.1109/FormaliSE66629.2025.00010.

[5] CXL Consortium, "Compute Express Link Specification, Revision 3.1," 2023, accessed: 2025-02-03. [Online]. Available: https://computeexpresslink.org/wp-content/uploads/2024/02/CXL-3.1-Specification.pdf

[6] C. Tan, A. F. Donaldson, and J. Wickerson, "Formalising CXL cache coherence," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2025*. New York, NY, USA: ACM, 2025. [Online]. Available: https://doi.org/10.1145/3676641.3715999

[7] F. Haftmann, "The Sketch and Explore library," 2023, accessed: 2025-02-04. [Online]. Available: https://isabelle.in.tum.de/dist/library/HOL/HOL-ex/Sketch_and_Explore.html

[8] M. Wenzel and L. C. Paulson, "Isabelle/Isar," in *The Seventeen Provers of the World, Foreword by Dana S. Scott*, ser. Lecture Notes in Computer Science, F. Wiedijk, Ed. Springer, 2006, vol. 3600, pp. 41–49. [Online]. Available: https://doi.org/10.1007/11542384_8

[9] Hugging Face, "Hugging face – the ai community building the future." online: https://huggingface.co/; accessed 20-October-2025. [Online]. Available: https://huggingface.co/

[10] Ollama, "Ollama – chat & build with open models." online: https://ollama.com/; accessed 20-October-2025. [Online]. Available: https://ollama.com/

[11] OpenAI, "Openai api platform," online: https://openai.com/api/; accessed 25-October-2025. [Online]. Available: https://chatgpt.com/

[12] Google, "Gemini api," online: https://ai.google.dev/gemini-api/docs; accessed 25-October-2025. [Online]. Available: https://gemini.google.com/app

[13] J. J. Huerta y Munive, "DeepIsaHOL," Nov. 2023, https://github.com/yonoteam/DeepIsaHOL.

[14] D. Bryant, Jonathan Julián Huerta y Munive, C. Kaliszyk, and J. Urban, "Munkres' general topology autoformalized in Isabelle/HOL," 2026. [Online]. Available: https://arxiv.org/abs/2604.07455

[15] C. Tan and J. J. Huerta y Munive, "super_sketch_and_super_fix," Nov. 2025, https://github.com/ChengsongTan/super_sketch_and_super_fix.

I. Appendix: Figures

$$\begin{pmatrix} \ddots & & \iddots \\ & \begin{pmatrix} P\ \sigma \wedge \\ \text{guard}_{R_i}\ \sigma \wedge \\ \sigma \rightarrow_{R_i} \sigma' \\ \Longrightarrow \\ \phi_j\ \sigma' \end{pmatrix}_{(i,j)} & \\ \iddots & & \ddots \end{pmatrix}_{m \times n}$$

Fig. 1: Proof obligation matrix for the inductiveness of P. Each single cell represents a certain conjunct being preserved by a rule.

```
lemma "(∑i≤(n::nat). 2*i-1) = n²"
(*start state σ_φ after initial string s_0*)

try_sketch SORRYS[simp] (induct n)
proof(induct n)

  show goal0:"(∑i::nat≤0. 2*i-1)=0²"
    by simp
next
  show goal1:"(∑i≤Suc n. 2*i-1)=(Suc n)²"
    if "(∑i≤n. 2*i- 1) = n²"
    for n :: "nat"
    using that
    sorry
qed
(* intended final substring s_m turned into
σ_m once the induct step is solved *)
```

Fig. 2: Example of a proof produced by **try_sketch**.

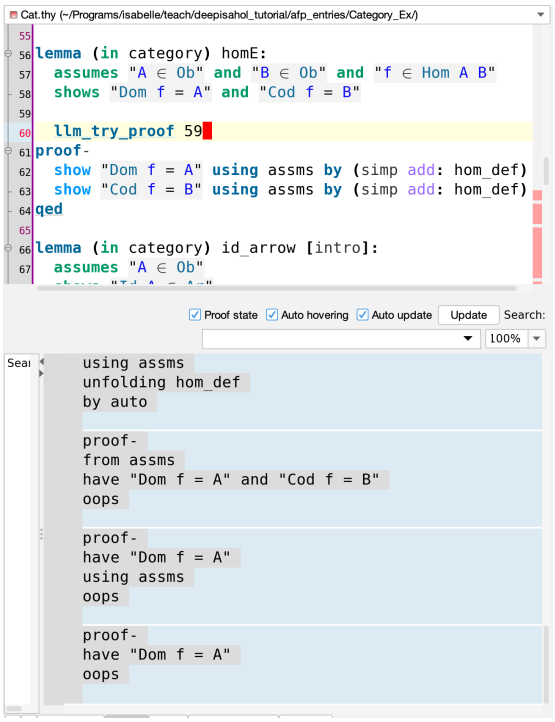
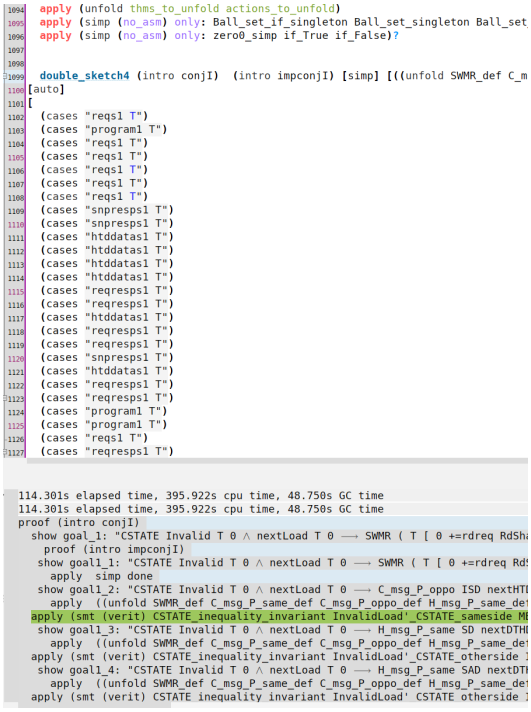
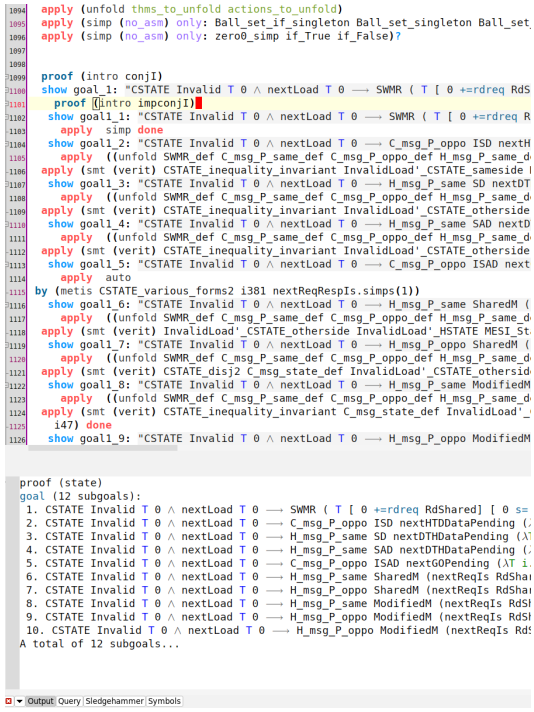


Fig. 3: Example of the command **llm_try_proof** interacting with a local LLM to produce a proof for the theorem homE in the AFP entry on Category Theory.



(a) **double_sketch** finished, with time information and proof suggestion



(b) **double_sketch** suggestion got accepted

Fig. 4: Side-by-side comparison of **double_sketch** command suggestion being adopted.

```
1 super_fix HAMMER[simp] "path/to/file"
```

Super_Fix_Demo.thy (~/Programs/isabelle/teach/deepisahol_tutorial/s...

```
1  theory Super_Fix_Demo
2    imports Complex_Main
3
4  begin
5
6
7  lemma real_sqrt_le_iff': "x ≥ 0 ⟹ y ≥ 0
8    ⟹ sqrt x ≤ y ⟷ x ≤ y ^ 2"
9    apply blast
10   done
11
12 lemma sqrt_le_itself: "1 ≤ x ⟹ sqrt x ≤ x"
13   sorry
14
15 lemma sqrt_real_nat_le:"sqrt (real n) ≤ real n"
16   apply simp
17   done
18
19 lemma semiring_factor_left:
20   fixes b:: "'a::semiring"
21   shows "a * b + a * c = a * (b + c)"
22   apply simp
23   done
24
25 lemma sin_cos_squared_add3:
26   fixes x:: "'a:: {banach,real_normed_field}"
27   shows "x * (sin t)² + x * (cos t)² = x"
28   apply simp
29   done
30
31 lemma sin_cos_squared_add4:
32   fixes x:: "'a:: {banach,real_normed_field}"
33   shows "x * (cos t)² + x * (sin t)² = x"
34   apply simp
35   done
36
37 lemma "φ ⟶ (ψ ⟶ φ)"
38   sorry
39
40 lemma "(φ ∨ φ) = (φ ∧ φ)"
41   sorry
42
43 lemma "¦¦a¦ - ¦b¦¦ ≤ ¦a - b¦" for a::real
44   apply rule
45   done
```

Super_Fix_Demo_Fixed.thy (~/Programs/isabelle/teach/deepisahol_tu...

```
1  theory Super_Fix_Demo_Fixed
2    imports Complex_Main
3
4  begin
5
6
7  lemma real_sqrt_le_iff': "x ≥ 0 ⟹ y ≥ 0
8    ⟹ sqrt x ≤ y ⟷ x ≤ y ^ 2"
9    by (smt (verit, ccfv_SIG) real_le_lsqrt sqrt_le
10
11
12 lemma sqrt_le_itself: "1 ≤ x ⟹ sqrt x ≤ x"
13   sorry
14
15 lemma sqrt_real_nat_le:"sqrt (real n) ≤ real n"
16   by (smt (verit, ccfv_SIG) less_Suc0 nat_1 nat_
17
18
19 lemma semiring_factor_left:
20   fixes b:: "'a::semiring"
21   shows "a * b + a * c = a * (b + c)"
22   by (smt (verit, del_insts) distrib_left)
23
24
25 lemma sin_cos_squared_add3:
26   fixes x:: "'a:: {banach,real_normed_field}"
27   shows "x * (sin t)² + x * (cos t)² = x"
28   by (smt (verit, del_insts) One_nat_def cos_of_
29
30
31 lemma sin_cos_squared_add4:
32   fixes x:: "'a:: {banach,real_normed_field}"
33   shows "x * (cos t)² + x * (sin t)² = x"
34   by (simp add: semiring_factor_left)
35
36 lemma "φ ⟶ (ψ ⟶ φ)"
37   by (smt (verit, del_insts))
38
39 lemma "(φ ∨ φ) = (φ ∧ φ)"
40   by (smt (verit, ccfv_threshold))
41
42 lemma "¦¦a¦ - ¦b¦¦ ≤ ¦a - b¦" for a::real
43   by (smt (verit))
44
45
```

Fig. 5: Example of a file fixed with **super_fix**. We show (1) at the top, the template command used to call the tool, (2) on the left, the file with errors, and (3) on the right, the fixed file.