Abstract. Proof by induction plays a central role in formal verification. However, its automation remains as a formidable challenge in Computer Science. To solve inductive problems, human engineers often have to provide auxiliary lemmas manually. We automate this laborious process with property-based conjecturing, a novel approach to generate auxiliary lemmas and use them to prove final goals. Our evaluation shows that our working prototype, PBC, achieved 40 percentage point improvement of success rates for problems at intermediate difficulty level.

1 Introduction

Consider the following definitions of add and even on natural numbers:

\begin{align*}
\text{add} \quad & 0 \ m = m \\
\text{add} \ (\text{Suc} \ n) \ m = \text{Suc} \ (\text{add} \ n \ m) \\
\text{even} \quad & 0 = \text{True} \\
\text{even} \ (\text{Suc} \ 0) = \text{False} \\
\text{even} \ (\text{Suc} \ (\text{Suc} \ n)) = \text{even} \ n
\end{align*}

Intuitively, the following statement holds: \text{even} \ (\text{add} \ n \ n).

However, if we apply structural induction on \(n\), the simplification based on the definitions of \text{add} and \text{even} gets stuck at \text{even} \ (\text{add} \ n \ n) \Rightarrow \text{even} \ (\text{Suc} \ (\text{add} \ n \ (\text{Suc} \ n))) when attacking the induction step. This is due to the definition of \text{add}, which does not allow us to operate on its second argument. Hence, if we want to prove this statement, we need to introduce auxiliary lemmas.

What lemmas should we introduce? Empirically, we know various mathematical structures share well-known algebraic properties such as associativity and commutativity. For example, our example problem uses \text{add}, which satisfies the following properties:

\begin{align*}
\text{add} \ n \ (\text{add} \ m \ k) &= \text{add} \ (\text{add} \ n \ m) \ k \quad \text{(add is associative)} \\
\text{add} \ n \ m &= \text{add} \ m \ n \quad \text{(add is commutative)}
\end{align*}
The commutative property of \texttt{add} allows us to operate on its second argument. Hence, if we prove this property, we can revert back to the original goal and finish its proof.

To automate this process, this paper introduces PBC, a tool that produces such property-based conjectures and attempts to prove them as well as the original proof goal in Isabelle/HOL [17]. For example, when applied to \texttt{even (add n n)}, PBC first proves 10 conjectures then proves the original goal using two of them as shown in Program 4 in Appendix.

We chose Isabelle/HOL to exploit its powerful proof tactics and counterexample finders; however, the underlying idea of property-based conjecturing is not specific to Isabelle/HOL: we can build similar systems for other provers if they are equipped with equivalent tools. We developed PBC under the following research hypothesis:

We can improve the proof automation of inductive problems by producing and proving conjectures based on fixed but general properties about relevant functions.

Our contributions are:

– the working prototype of a powerful inductive prover based on property-based conjecturing and newly developed default strategy (Section 2.1),
– the identification of useful properties (Section 2.2), and
– extensive evaluations of PBC to test our research hypothesis (Section 3).

2 System Description

![Workflow of PBC](image-url)
2.1 Overview

Figure 1 shows how PBC attacks inductive problems using property-based conjecturing. Given an induction problem, the tool first attempts to prove the goal using a default strategy, PBC.Strategy, written in the proof strategy language, PSL [15]. As shown in Program 1, PBC.Strategy combines Isabelle’s proof tactics, such as auto andclarsimp, and other sub-tools, such as smart induction [12,13] and Sledgehammer [19] to prove the goal completely. That is, PSL uses Sledgehammer as a sub-tool, even though Sledgehammer itself is a meta-tool that uses external provers and Isabelle’s tactics to prove given problems.

As PSL is a new meta-tool, we first explain the language constructs in Program 1. Ors is a combinator for deterministic choice, whereas Thens and PThenOne combine sub-strategies sequentially. Subgoal lets PSL focus on the first sub-goal, temporarily hiding other sub-goals from the scope, while IsSolved checks if all proof obligations are solved within the current scope. Auto, Clasrimp, and Fastforce correspond to Isabelle’s default tactics of the same name, while Hammer calls Sledgehammer [19] and Smart.Induct applies 5 promising candidates of proof by induction [12,13]. Essentially, this strategy applies increasingly expensive sub-strategies to solve proof goals using backtracking search.

If PBC.Strategy fails to prove the goal, it produces conjectures based on properties specified in advance, following the process explained in Section 2.2. Then, the tool attempts to refute the conjectures using Isabelle’s counter-example generators: Quickcheck [2] and Nitpick [1]. After filtering out refuted conjectures, PBC attempts to prove the remaining conjectures using the default strategy. While doing so, PBC registers proved conjectures as auxiliary lemmas, so that it can use them to prove other conjectures.

For example, PBC.Strategy finds the following proof script for the commutativity of add. To demonstrate how PBC.Strategy finds proofs using backtracking search, we highlighted the parts of Program 1 that were not backtracked but resulted in this script. We invite readers to compare these highlighted parts in Program 1 against the resulting script and to find out which proof tactic Sledgehammer used to prove the corresponding sub-goal.

```
lemma commutativity: "add var.1 var.2 = add var.2 var.1"
  apply ( induct_tac "var.1" )
  apply ( simp add : identity )
subgoal
  apply clarsimp
subgoal
  apply ( induct_tac "var.2" )
  apply auto
  done
  done
  done
```

Answer: Sledgehammer used the simp tactic with an auxiliary lemma about identity. Furthermore, IsSolved resulted in the done command in scripts.
Program 1  PBC_Strategy: PBC’s default strategy.

Ors [  
Thens [Auto, IsSolved],  
PThenOne [Smart_Induct, Thens [Auto, IsSolved]],  
Thens [Hammer, IsSolved],  
PThenOne [  
Smart_Induct,  
Ors  
[Thens [  
Repeat (  
Ors [  
Fastforce,  
Hammer,  
Thens [ Clarsimp, IsSolved ],  
Thens [  
Subgoal,  
Clarsimp,  
Repeat (  
Thens [ Subgoal,  
Ors [ Thens [Auto, IsSolved],  
Thens [ Smart_Induct, Auto, IsSolved ] ] ]  
IsSolved  
)  
IsSolved  
]  
]  
]  
]  
]  
]  
]

After processing the list of conjectures, PBC comes back to the original goal. This time, it attacks the goal, using proved conjectures as auxiliary lemmas. If PBC still fails to prove the original goal, it again attacks the remaining conjectures hoping that proved conjectures may help the strategy to prove remaining ones. By default, PBC gives up after the second round and shows proved conjectures and their proofs in Isabelle’s standard editor’s output pane, so that users can exploit them when attacking original goals manually.

The seamless integration of PBC into the Isabelle ecosystem lets users build PBC as an Isabelle theory using Isabelle’s standard build command without installing additional software. Furthermore, when PBC finds a proof for the original goal, our tool shows the final proof as well as proved conjectures with their proofs in the output pane as shown in Fig. 2. Users can copy and paste them with a single click to the right location of their proof scripts. The produced scripts are human readable, and Isabelle can check them without PBC.
Fig. 2: Screenshot of Isabelle/HOL with PBC. The upper pane shows the definition of a type and functions. The new command `prove_by_conjecturing` invokes PBC, which presents the proof script appearing in the lower pane.
Program 2 The Complete List of Property-Based Conjectures. We added the highlighted four conjectures after manually solving some benchmark problems. One can see that none of these conjectures are specific to particular problems.

associativity \[ f(f(x, y), z) = f(x, f(y, z)) \]
identity element \[ f(e, x) = x \text{ or } f(x, e) = x \text{ for some } e \]
commutativity \[ f(x, y) = f(y, x) \]
idempotent element \[ f(e, e) = e \text{ for some } e \]
distributivity \[ f(x, g(y, z)) = g(f(x, y), f(x, z)) \]
anti-distributivity \[ f(g(x, y)) = g(f(y, f(x)) \]

2.2 Property-Based Conjecturing

As mentioned in Section 2.1, our tool produces conjectures based on 16 properties specified in advance. 12 of them are either well-known algebraic properties, such as associative property, or relational properties, such as transitivity. Program 2 lists all the PBC properties. Note that we added the 4 highlighted properties based on the feedback from students who manually solved several benchmark problems. None of these properties are specific to particular functions.

To produce conjectures for such properties, PBC first collects functions appearing in the original proof goal. Then, it looks for the definitions of these functions and adds functions in these definitions into the list of functions for conjecturing. Then, PBC filters out functions defined within the standard library since the standard library already contains useful auxiliary lemmas for them. Finally, PBC creates conjectures by filling templates with these functions.

3 Evaluation

3.1 Benchmark and Environment

We evaluated our tool using Tons of Inductive Problems (TIP) [6], which is a benchmark consisting of 462 inductive problems. TIP consists of three main problem sets: 85 problems in Isaplanner, 50 in Prod, and 327 in TIP15. Isaplanner is the easiest, whereas Prod contains problems at the intermediate difficulty level, and TIP15 has difficult problems, such as Fermat’s Last Theorem.

The advantage of using TIP is that each problem is complete within a single file. That is, data types and functions are defined afresh within each problem
Property-Based Conjecturing for Automated Induction in Isabelle/HOL

file, instead of using the standard definition. For example, our running example problem from Section 1 is formalised as an independent Isabelle theory file in the Prod set in TIP. The functions, add and even, are defined afresh in this file, instead of using the default ones from the standard library. This allowed us to ignore manually developed lemmas for similar functions in the standard library. This way, by using TIP, we focused on PBC’s conjecturing capability to prove the final goal.

In this experiment, we set the following timeouts for the counter-example generators: one second for Quickcheck, two seconds for Nitpick. The timeout for Sledgehammer is more flexible: 10 seconds when attacking conjectures in $n$-th round where $n$ is an odd number, whereas 30 seconds when attacking conjectures in $n$-th round where $n$ is an even number or attacking the original goals.

However, when measuring the performance of PBC against TIP15 problems, we set the following short timeouts to process 327 problems using computational resources available to us: 5 seconds for Sledgehammer to prove produced conjectures, 10 seconds for Sledgehammer to attack the original goal. Furthermore, we use 15 minutes as the overall timeout for each problem in TIP15.

We ran our evaluations on consumer-grade laptops. Specifically, we used a Lenovo Thinkpad T490s, with Intel Core i7-8665U CPU and 16GB of RAM. We used Windows 10 Pro as our evaluation operating system.

### 3.2 Results

**Success rates for different difficulty levels.** Figure 3 shows the percentage of problems proved by each tool at each stage. We use an induction prover for Isabelle/HOL [14], TAP21, as our baseline prover. “Round0” represents the percentage of solved problems after the zeroth round of PBC, where PBC shows the percentage of solved problems after the second round for Isaplanner and Prod, but after the first round for TIP15 due to our limited computational resources.

The figure shows that PBC brought the largest improvement (40 percentage points) to the Prod category. On the other hand, we can prove 60% of problems in Isaplanner without producing conjectures, while PBC struggles at harder problems in the TIP15 category.

**Program 3 TAP2021** is the strategy used in the baseline prover introduced by Nagashima [14]. Since we added minor improvements to Smart_Induct, we represent their version of Smart_Induct as Old_Smart_Induct in this paper.

```plaintext
Ors [
  Auto_Solve,
  PThenOne [Old_Smart_Induct, Auto_Solve],
  PThenOne [Old_Smart_Induct,
    Thens [ Auto, RepeatN (Hammer), IsSolved ]
  ]
]
```
Proof completion rates and execution time. Fig. 3, Fig. 4, and Fig. 5 show the chances of solving a problem in each category relative to how long the program is run. For example, Fig. 5 illustrates that approximately 20% of the problems are solved within 5 minutes in the Prod category, and 60% of the problems are solved within 20 minutes of runtime. Beyond this time, the chances of producing a proof increase marginally, reaching 66% of problems after an hour.

Fig. 3: Proof completion rates.

Fig. 4: Success rates over time for Isaplanner.
Difficult for humans and PBC. Figure 7 shows the relation between the PBC’s performance and problem difficulty for humans based on some Prod problems. We estimated the difficulty of each problem by measuring how much time undergraduate maths students spent to prove these in Isabelle/HOL and their solution size. A linear relation is seen in the log-log plot, showing a polynomial increase.
of solving time against solution size. White circles stand for problems that PBC proved automatically, whereas black circles represent problems PBC failed to prove. The figure shows that PBC tends to solve problems that are easier to humans.

Fig. 7: Difficulty for humans for Prod.

Refuting and proving. Fig. 8 and Fig. 9 show how many conjectures PBC produced for each problem in Isaplanner and Prod and how it handled them, respectively. As shown in the figure, PBC did not produce any conjectures for some problems, since it proved these problems even before producing conjectures. Furthermore, the number of conjectures does not blow up in PBC, since PBC produces conjectures about commonly used properties only. Note that keeping the number of conjectures low is the main challenge in other conjecturing tools, as we discuss in Section 4. Moreover, these figures show that most conjectures are either proved or refuted for problems in Isaplanner and Prod, and only a few conjectures are left unsolved thanks to the strong default strategy and counter-example finders.

4 Related Work

Conjecturing. We have two schools of conjecturing to automate inductive theorem proving: top-down approach and bottom-up approach. Top-down approaches [3,4,16] create auxiliary lemmas from an ongoing proof attempt, whereas bottom-up approaches [5,9] produce lemmas from available functions and data types to enrich the background theory [10]. PBC falls into the latter category. While most bottom-up tools, such as HipSpec [5] and Hipster [9], produce conjectures
randomly, PBC makes conjectures based on a fixed set of templates. Furthermore, Hipster aims to *discover* new lemmas, PBC checks for *known* properties to keep the number of conjectures low. In this respect, RoughSpec [7] is similar
to PBC: it produces conjectures based on templates, which describe important properties. Contrary to PBC, RoughSpec supports only equations as templates and is a tool for Haskell rather than a proof assistant.

Inductive theorem proving. PBC is an automatic tool developed for an interactive theorem prover (ITP) based on a higher-order logic. Others have introduced proof by induction for automatic theorem provers (ATPs) \[11\][20][21][18]. ATPs are typically based on less expressive logics and use different proof calculi compared to LCF-style provers. Moreover, ATPs are built for performance, whereas LCF-style provers are designed for high assurance and easy user-interaction. Such differences make a straightforward comparison difficult; however, we argue that a stronger automation of inductive proofs in ITPs helps users reason data types and functions they introduce to tackle unique problems.

5 Discussion and Conclusion

Careful investigations into generated proofs reveal that PBC proves conjectures that are not used to attack the original goal as shown in Appendix. Although such conjectures may serve as auxiliary lemmas when users prove other problems in the future, the time spent to prove these conjectures certainly slows down the execution speed of PBC. Furthermore, PBC fails to prove difficult problems since they require conjectures specific to them. We expect that combining PBC with other top-down approaches would result in more powerful automation, which remains as our future work.

This paper presented our property-based conjecturing tool, PBC. To the best of our knowledge, PBC is the only tool that achieved high proof completion rates for the TIP benchmarks while producing human readable proofs that are native to a widely used ITP.

References

1. Blanchette, J.C., Nipkow, T.: Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In: Kaufmann, M., Paulson, L.C. (eds.) Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6172, pp. 131–146. Springer (2010). https://doi.org/10.1007/978-3-642-14052-5_11
2. Bulwahn, L.: The new quickcheck for Isabelle - random, exhaustive and symbolic testing under one roof. In: Hawblitzel, C., Miller, D. (eds.) Certified Programs and Proofs - Second International Conference, CPP 2012, Kyoto, Japan, December 13-15, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7679, pp. 92–108. Springer (2012). https://doi.org/10.1007/978-3-642-35308-6_10
3. Bundy, A., Basin, D.A., Hutter, D., Ireland, A.: Rippling - meta-level guidance for mathematical reasoning, Cambridge tracts in theoretical computer science, vol. 56. Cambridge University Press (2005)
4. Bundy, A., van Harmelen, F., Horn, C., Smaill, A.: The Oyster-Clam system. In: Stickel, M.E. (ed.) 10th International Conference on Automated Deduction, Kaiserslautern, FRG, July 24-27, 1990, Proceedings. Lecture Notes in Computer Science, vol. 449, pp. 647–648. Springer (1990). https://doi.org/10.1007/3-540-52885-7_123

5. Claessen, K., Johansson, M., Rosén, D., Smallbone, N.: HipSpec: Automating inductive proofs of program properties. In: Fleuriot, J.D., Höfner, P., McIver, A., Smaill, A. (eds.) ATx’12/WInG’12: Joint Proceedings of the Workshops on Automated Theory cXploration and on Invariant Generation, Manchester, UK, June 2012. EPiC Series in Computing, vol. 17, pp. 16–25. EasyChair (2012). https://doi.org/10.29007/3qwr

6. Claessen, K., Johansson, M., Rosén, D., Smallbone, N.: TIP: tons of inductive problems. In: Kerber, M., Carette, J., Kaliszyk, C., Rabe, F., Sorge, V. (eds.) Intelligent Computer Mathematics - International Conference, CICM 2015, Washington, DC, USA, July 13-17, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9150, pp. 333–337. Springer (2015). https://doi.org/10.1007/978-3-319-20615-8_23

7. Einarsdóttir, S.H., Smallbone, N., Johansson, M.: Template-based theory exploration: Discovering properties of functional programs by testing. In: Chitil, O. (ed.) IFL 2020: 32nd Symposium on Implementation and Application of Functional Languages, Virtual Event / Canterbury, UK, September 2-4, 2020. pp. 67–78. ACM (2020). https://doi.org/10.1145/3462172.3462192

8. Hajdú, M., Hozsová, P., Kovács, L., Voronkov, A.: Induction with recursive definitions in superposition. In: Formal Methods in Computer Aided Design, FMCAD 2021, New Haven, CT, USA, October 19-22, 2021. pp. 1–10. IEEE (2021). https://doi.org/10.34727/2021/ibsn.978-3-85448-046-4_34

9. Johansson, M.: Automated theory exploration for interactive theorem proving: - an introduction to the hipster system. In: Ayala-Rincón, M., Muñoz, C.A. (eds.) Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017. Proceedings. Lecture Notes in Computer Science, vol. 10499, pp. 1–11. Springer (2017). https://doi.org/10.1007/978-3-319-66107-0_1

10. Johansson, M.: Lemma discovery for induction - A survey. In: Kaliszyk, C., Brady, E.C., Kohlhase, A., Coen, C.S. (eds.) Intelligent Computer Mathematics - 12th International Conference, CICM 2019, Prague, Czech Republic, July 8-12, 2019. Proceedings. Lecture Notes in Computer Science, vol. 11617, pp. 125–139. Springer (2019). https://doi.org/10.1007/978-3-030-23250-4_9

11. Matt Kaufmann, Panagiotis Manolios, J.S.M. (ed.): Computer-Aided Reasoning ACL2 Case Studies. Advances in Formal Methods, Springer New York, NY (2000). https://doi.org/https://doi.org/10.1007/978-1-4757-3188-0

12. Nagashima, Y.: Smart induction for Isabelle/HOL (tool paper). In: 2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020. pp. 245–254. IEEE (2020). https://doi.org/10.34727/2020/ibsn.978-3-85448-042-6_32

13. Nagashima, Y.: Faster smarter proof by induction in Isabelle/HOL. In: Zhou, Z. (ed.) Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021.
Appendix

Program 4 Generated Proof Script for Our Running Example

```isabelle
lemma associativity_5382114:
  "add var_1 (add var_2 var_3) = add (add var_1 var_2) var_3"
  apply ( induct "var_1" arbitrary : var_2 ) apply auto done
lemma associativity_5382286:
  "add (add var_1 var_2) var_3 = add var_1 (add var_2 var_3)"
  apply ( induct "var_1" arbitrary : var_2 ) apply auto done
lemma identity_5382450: "add 0 var_1 = var_1" apply auto done
lemma identity_5382590: "add var_1 0 = var_1"
  apply ( induct_tac "var_1" ) apply auto done
lemma commutativity_5382730:
  "add var_1 var_2 = add var_2 var_1"
  apply ( induct_tac "var_1" )
  apply ( simp add : identity_5382590 )
  subgoal apply clarsimp subgoal apply ( induct_tac "var_2" )
  apply auto done done done
lemma idempotent_Element_5382882: "add 0 0 = 0" apply auto done
lemma swap_Unary_5383532: "add var_1 (S var_2) = add (S var_1) var_2"
  apply ( induct_tac "var_1" ) apply auto done
lemma composite_Commutativity_5383684:
  "add (add var_1 var_2) = add (add var_2 var_1)"
  apply ( simp add : commutativity_5382730 ) done
lemma composite_Commutativity_5383846:
  "S (add var_1 var_2) = S (add var_2 var_1)"
  apply ( simp add : commutativity_5382730 ) done
lemma composite_Commutativity_5383998:
  "even (add var_1 var_2) = even (add var_2 var_1)"
  apply ( simp add : commutativity_5382730 ) done
lemma original_goal_5347090: "even (add x x)"
  apply ( induct_tac "x" )
  apply fastforce apply ( metis Nat.distinct [1] Nat.inject
  even.simps(3) commutativity_5382730 add.elims ) done
```

Program 4 shows the output of PBC for our running example. The original goal is proved using commutativity_5382730, which is in turn proved using identity_5382590. 8 out of 10 proved conjectures are not used to prove the final goal; however, PBC outputs them, so that users may exploit them in future.