A Proof Strategy Language and Proof Script Generation for Isabelle

Yutaka Nagashima and Ramana Kumar
Data61, CSIRO / NICTA** / UNSW

Abstract. Interactive theorem provers, like Isabelle, include various automatic tools for finding proofs under certain conditions. However, for each conjecture, knowing which automation to use, and how to tweak its parameters, is currently labour intensive. We have developed a language, PSL, designed to capture high level proof strategies. PSL offloads the construction of human-readable fast-to-replay proof scripts to automatic search, making use of search-time information about each conjecture. Our preliminary evaluations show that PSL reduces the labour cost of interactive theorem proving.

1 Introduction

Formal verification using interactive theorem provers (ITP) is a promising subfield of formal methods with successful artefacts [8]. However, when verifying real world systems with ITPs, one often needs to discharge proof obligations that should not really require human intuition but still beyond the scope of existing proof automation tools. Typically, engineers have to manually specialise and combine basic proof automation tools, called tactics [10], to discharge these proof obligations. This process requires expertise in the ITP, and it is time consuming, raising the cost of formal verification.

To address this problem, we introduce a language, PSL, which allows users to write proof strategies for a particular ITP, Isabelle [9]. A proof strategy describes in abstract how to attack proof obligations: by writing a strategy based on their intuitions about a conjecture, users can specify a larger search space than with existing proof automation tools. Furthermore, given a strategy, PSL’s runtime system attempts to generate efficient proof scripts by figuring out the appropriate specialisation and combination of tactics for the conjecture without direct user interaction. Thus, PSL does not only reduce the labour cost of theorem proving, but it also keeps proof scripts interactive and maintainable by reducing the execution time of subsequent proof checking.

Our main contributions are twofold: firstly we present the language and its runtime system for Isabelle2015 [1] through examples, secondly we present PSL’s default strategy, try_hard, which we use as a general purpose proof automation

** NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.
tool. Our preliminary evaluations show that try\texttt{hard} is more powerful than existing proof automation tools for Isabelle.

## 2 Background and Syntax

Interactive theorem proving can be seen as the exploration of a search tree. Nodes of the tree represent proof states. Edges represent applications of tactics, which transform the proof state. The shape of the tree is not known in advance; the behaviour of tactics are, in general, not completely predictable since they rely on information stored in background proof contexts. The goal is to find a node representing a solved state: one in which the proof is complete. The tree may be infinitely wide and deep, because there are endless variations of tactics that may be tried at any point. The goal for a \texttt{PSL} proof strategy is to direct an automated search of this tree to find a solved state; \texttt{PSL} will reconstruct an efficient path to this solved state as a human-readable proof script.

In Isabelle, proof states are represented by values of type \texttt{thm}. Tactics are functions of type \texttt{thm \rightarrow \{thm\}}, where \texttt{\{\cdot\}} denotes a (possibly infinite) lazy sequence; hence tactics in general admit non-deterministic choice via backtracking \cite{10}. This means the search tree grows wider not just when choosing between multiple tactics but also when individual tactics return multiple results. \texttt{succeed} and \texttt{fail} are special tactics: \texttt{succeed} takes a value of \texttt{thm}, wraps it in a lazy sequence, and returns it without modifying the value. \texttt{fail} always returns an empty sequence. When using tactics, proof authors often have to adjust tactics using \textit{modifiers} for each proof obligation.

The search tree grows deeper when tactics are combined sequentially. The tactic combinators in Isabelle include \texttt{THEN} for sequential composition, \texttt{APPEND} for non-deterministic choice, \texttt{ORELSE} for deterministic choice, and \texttt{REPEAT} for iteration. The following is the syntax of \texttt{PSL}, which is similar to that of Isabelle’s tactic language.

\begin{verbatim}
strategy = default | dynamic | special | subtool | compound
default = Simp | Clarsimp | Fastforce | Auto | Induct |
          | Rule | Erule
dynamic = Dynamic (default)
special = IsSolved | Defer | Case
subtool = Sledgehammer | Nitpick | Quickcheck
compound = Thens [strategy] | Ors [strategy] | Alts [strategy]
           | Repeat (strategy) | RepeatN (strategy)
           | POrs [strategy] | PAlts [strategy]
\end{verbatim}

The \texttt{default} strategies correspond to Isabelle’s default tactics without arguments, while \texttt{dynamic} strategies correspond to Isabelle’s default tactics that are specialised for each conjecture. Given a \texttt{dynamic} strategy and conjecture, the runtime system generates variants of the corresponding Isabelle tactic. Each of these variants is specialised for the conjecture with a different combination of
promising arguments found in the conjecture and its proof context. It is the purpose of the PSL runtime system to select the right combination. 

subtool represents Isabelle tools such as sledgehammer [4] and counterexample finders. The compound strategies capture the notion of tactic combinators: Thens corresponds to THEN, Ors to ORELSE, Alts to APPEND, and Repeat to REPEAT. POrs and PAAlts are similar to Ors and Alts, respectively, but they admit parallel execution of sub-strategies. In the following, we explain how to write strategies and how PSL’s runtime system interprets strategies with examples.

3 PSL by Example

Example 1. For our first example, we take the following lemma, exec_acomp, from an assignment of a course [3]. This lemma is about the execution model of a simple language. From our experience, we presume that the proof involves some sort of mathematical induction followed by some simple procedures. However, we do not know exactly how we should conduct mathematical induction here; therefore, we describe this rough idea as a proof strategy, some_induct, with the keyword strategy, and apply it to exec_acomp with the keyword find_proof.

strategy some_induct = Thens [Dynamic (Induct), Auto, IsSolved]
lemma exec_acomp: "exec (acomp a) s stk = aval a s # stk"
find_proof some_induct

Invoked by find_proof, PSL’s runtime system interprets some_induct in terms of exec_acomp and the context that contains exec_acomp. For example, it interprets Auto as Isabelle’s default tactic, auto.

The interpretation of Dynamic (Induct) is more involved. First, PSL collects the free variables (noted in italics above) in exec_acomp and all applicable induction rules stored in the context. PSL uses the set of free variables to specify two things: on which variables instantiated tactics conduct mathematical induction, and which variables should be generalised in the induction scheme. The set of applicable rules are used to specify which rules to use. Second, PSL creates the powerset out of the set of all possible modifiers. Then, it attempts to instantiate a variant of the induct tactic for each subset of modifiers. Finally, it combines all the variants of induct with unique results using APPEND. In this example, PSL generates 1024 induct tactics for exec_acomp. These tactics created from Dynamic (Induct) are combined with auto with THEN.

As the above number indicates, PSL’s runtime system often creates large, inefficient, but powerful tactics, drawn from a large search space. To cope with this large search space we use assertion tactics: tactics that provide mechanisms for controlling proof search based on a condition: such a tactic takes a proof state, tests an assertion on it, then behaves as succeed or fail accordingly.

For exec_acomp, PSL’s runtime system interprets IsSolved as the assertion tactic is_solved. The is_solved tactic takes a proof state and checks if any proof obligations are left or not. It behaves as fail if obligations are left, it

3
behaves as $\textbf{succeed}$ if the proof is complete ensuring that no subgoals are left when sequentially combined with other tactics. For \texttt{exec:acomp}, PSL internally creates the following tactic from \texttt{some:induct} at runtime.

$$(\texttt{induct1 \ APPEND \ induct2 \ APPEND...}) \ THEN \ \texttt{auto} \ THEN \ \texttt{is:solved}$$

where \texttt{induct1} and \texttt{induct2} are variants of the \texttt{induct} tactic specialised with modifiers.

Within the runtime system, Isabelle first applies \texttt{induct} to \texttt{exec:acomp}, then \texttt{auto} to the resultant proof obligations. Note that each \texttt{induct} tactic and \texttt{auto} is deterministic: it either fails or returns a lazy sequence with a single element. However, combined together with \texttt{APPEND}, the 1024 \texttt{induct} tactics \textit{en masse} are non-deterministic: if \texttt{is:solved} finds remaining proof obligations, Isabelle backtracks to the next \texttt{induct} tactic, \texttt{induct2} and repeats this process until either it discharges all proof obligations or runs out of the variations of \texttt{induct} tactics. The numerous variants of \texttt{induct} tactics from \texttt{some:induct} allow Isabelle to explore a larger search space than its naive alternative, \texttt{induct THEN auto}, does. Consequently, Isabelle tends to spend a longer search time with \texttt{some:induct} than it does with \texttt{induct THEN auto}.

PSL addresses this performance problem by tracing Isabelle’s proof search: PSL keeps a log of successful proof attempts while scraping off backtracked proof attempts. PSL removes any failed proof attempts as soon as it backtracks, thus minimising its memory usage. Furthermore, since PSL follows Isabelle’s execution model based on lazy sequences, it stops proof search as soon as it finds a specialisation and combination of tactics, with which Isabelle can pass the non-proof-obligation test imposed by \texttt{is:solved}.

We still need a longer search time with PSL, but only once: upon success, PSL converts the log of successful attempts into an efficient proof script, which bypasses a large part of proof search. For \texttt{exec:acomp}, PSL generates the following proof script from \texttt{some:induct}.

\begin{verbatim}
apply((induct arbitrary:l stk rule:HW03.acomp.induct), HOL.auto)
\end{verbatim}

We implemented PSL as an Isabelle theory; to use PSL users only have to import the relevant theory files to use PSL to their files. Moreover, we have integrated PSL into Isabelle/Isar, Isabelle’s proof language, and Isabelle/jEdit, its standard editor. This allows users to define and invoke their own proof strategies inside their ongoing proof attempts, as shown above; and if the proof search succeeds PSL presents a proof script in jEdit’s output panel, which users can copy to the right location with one click. All generated proof scripts are independent of PSL, so users can maintain them without PSL.

Example 2. While verifying real world systems such as seL4 \cite{seL4}, Isabelle users often face numerous proof obligations that are beyond the scope of standard Isabelle tactics. However, many of these obligations are not very hard to discharge either: users know that they can discharge many of them by using external
theorem provers through \texttt{sledgehammer}. In such a case, users manually invoke \texttt{sledgehammer} several times to find out which proof obligations \texttt{sledgehammer} can discharge. We developed a strategy, \texttt{hammers}, to automate this notoriously time-consuming process. The following shows its definition and a use case simplified for illustrative purposes.

\begin{verbatim}
strategy hammers = RepeatN (Ors [Hammer, Defer])

lemma safe_trans: shows
1:"ps\_clear p s" and 2:"valid\_tran p s s' c" and 3:"ps\_clear p s'"
find_proof hammers
\end{verbatim}

Note that we made this example simple, so that two subgoals, 1:"\texttt{ps\_clear p s}" and 3:"\texttt{ps\_clear p s'}", are not hard to prove; however, proving them still requires the definition of \texttt{ps\_clear}, which is stored in the context. Finding out the definition of \texttt{ps\_clear} automatically is beyond the scope of Isabelle's standard tactics; we need stronger automatic tools, such as \texttt{sledgehammer}.

Generally, given a conjecture and a strategy of the form of \texttt{RepeatN strategy}, \texttt{PSL} applies \texttt{strategy} to the conjecture as many times as the number of proof obligations in the conjecture. In this case, \texttt{PSL} applies \texttt{Ors [Hammer, Defer]} to \texttt{safe\_trans} three times. \texttt{Ors} is translated into an \texttt{ORELSE}. In this case, \texttt{PSL} first applies \texttt{Hammer}; it applies \texttt{Defer} only if \texttt{Hammer} fails. \texttt{Hammer} calls external theorem provers via \texttt{sledgehammer}, while \texttt{Defer} postpones the current subgoal to the end of the list of proof obligations.

In this example, the second proof obligation 2:"\texttt{valid\_tran p s s' c}" is not provable by \texttt{sledgehammer}. After \texttt{sledgehammer} fails, \texttt{PSL} applies 2 to \texttt{Defer}, which sends 2 to the end of the list; then, \texttt{PSL} continues working on the subgoal 3 with \texttt{sledgehammer}. \texttt{PSL} stops its execution after applying \texttt{Ors [Hammers, Defer]} three times, generating the following proof script. This fast proof script leaves 2 as the meaningful task for human engineers.

apply((smt ps\_clear\_def),tactic{* defer 1 *},(smt ps\_clear\_def))

The default strategy: \texttt{try\_hard}. \texttt{PSL} comes with a default strategy, \texttt{try\_hard}. Users can use \texttt{try\_hard} as a completely automatic tool: engineers need not provide their intuitions by writing strategies. The lack of input from human engineers makes \texttt{try\_hard} less specific to each conjecture; however, we made \texttt{try\_hard} more powerful than existing proof automation tools for Isabelle by specifying larger search spaces with a large definition presented in Appendix A. Larger search spaces naturally lead to longer search times, which we addressed with lazy sequence and fast proof script generation as explained above.

We conducted a judgement-day style evaluation \cite{5} of \texttt{try\_hard} against course work assignments \cite{2}. \texttt{try\_hard} indeed proves more conjectures than \texttt{sledgehammer} does with three exceptions, reducing the labour-cost of formal verification. The evaluation log files suggest that \texttt{try\_hard} discharges less proof obligations in three files because it currently does not take user supplied hints, called \textit{chained facts}, into account while \texttt{sledgehammer} does. This lead to the
significant differences in the results for week 8, 9, and 11; these scripts already provide many chained facts which \texttt{sledgehammer} can rely on. We are currently solving this minor technical issue without changing the framework.

| Table 1. The number and percentage of automatically proved proof goals. |
|---------------------------------------------------------------|
| \textbf{proof goals}  | \textbf{week 1} | \textbf{week 2} | \textbf{week 3} | \textbf{week 4} | \textbf{week 5} | \textbf{week 6} | \textbf{week 8} | \textbf{week 9} | \textbf{week 11} |
| \textbf{try\_hard}   | 19   | 22   | 52   | 82   | 64   | 26   | 52   | 61   | 26   |
| \textbf{percentage}  | 100% | 95%  | 65%  | 85%  | 86%  | 50%  | 75%  | 33%  | 58%  |
| \textbf{sledgehammer}| 14   | 5    | 28   | 59   | 45   | 12   | 48   | 31   | 17   |
| \textbf{percentage}  | 74%  | 23%  | 54%  | 72%  | 70%  | 46%  | 92%  | 51%  | 65%  |

Other Concepts. We integrated two counterexample finders, \texttt{quickcheck} and \texttt{nitpick}, as \texttt{subtool} strategies. At runtime, \texttt{PSL} interprets them as assertion tactics: if they find a counterexample, they behave as \texttt{fail}, if not, they work as \texttt{succeed}. These counterexample finders help \texttt{PSL} prune search spaces with no prospects of completing a proof.

The flexible runtime interpretation might lead \texttt{PSL} into a non-terminating loop, such as \texttt{REPEAT succeed}. To handle such loops, \texttt{PSL} traverses a search space using iterative deepening depth first search (IDDFS). We implemented IDDFS, introducing the idea of \textit{history-sensitive tactic}: a tactic that takes the log of proof attempts into account.

4 Related Work and Conclusions

Dixon et al. developed \textit{IsaPlanner} \cite{6}, providing a framework to encode and apply common patterns of reasoning in Isabelle. Inspired by their work, we addressed the performance issue by generating efficient proof scripts. Furthermore, we made use of existing tools rather than implementing new proof machinery from scratch. This way, we supplement one tool’s weakness (e.g. mathematical induction for \texttt{sledgehammer}) with other tools’ strengths (e.g. the \texttt{induct} tactic), while enhancing their capabilities with runtime tactic generation. Freitas et al. introduced a system to identify and replay proof strategies \cite{7}. Following their approach, we plan to improve the default strategy of \texttt{try\_hard} mechanically.

We have introduced a proof strategy language, \texttt{PSL}, and its default strategy, \texttt{try\_hard}. Our preliminary evaluations indicate that our approach reduces the labour cost of interactive theorem proving, making complete formal verification more cost-effective. Our implementation is for Isabelle2015; however, the idea behind \texttt{PSL} should be transferable to other tactic-based ITPs. In fact, we are planning to export \texttt{PSL} to HOL4 \cite{11}, reusing the central part of \texttt{PSL}’s implementation.
### A Appendix: the Default Strategy, \texttt{try\_hard}

The following is the definition of \texttt{try\_hard}, the default proof search strategy of \texttt{PSL}. Unlike other user-defined strategies, one can invoke this strategy by simply typing \texttt{try\_hard} without \texttt{find\_proof} inside a proof attempt.

```plaintext
strategy \texttt{try\_hard} =
  O\texttt{rs}
  Thens \texttt{[Auto, IsSolved]},
  Thens \texttt{[Repeat (O\texttt{rs} \texttt{[Hammer, Fastforce, IsSolved]},
                         Thens \texttt{[Clarsimp, Hammer]]),
                         IsSolved]},
  Thens \texttt{[Induct, Auto, IsSolved]},
  Thens \texttt{[Case, Auto, IsSolved]}),
  Thens \texttt{[Dynamic (Induct), Auto, IsSolved]}),
  Thens \texttt{[Dynamic (Simp), IsSolved]}),
  Thens \texttt{[Case,}
    Repeat \texttt{[O\texttt{rs} \texttt{[Fastforce,}
                 Hammer,}
                 Thens \texttt{[Clarsimp, Hammer]]},
                 IsSolved]},
  Thens \texttt{[Dynamic (Induct),}
    Repeat \texttt{[O\texttt{rs} \texttt{[Fastforce,}
                 Hammer,}
                 Thens \texttt{[Clarsimp, Hammer]]},
                 IsSolved]};
```

### B Appendix: Details of the Preliminary Evaluation

We took the test cases from a course [2]. The experiment was conducted on a MacBook Pro (Retina, 15-inch, Late 2013) with 2.3 GHz Intel Core i7 and 16 GB 1600 MHz DDR3. We leave comprehensive evaluation against randomly chosen proof scripts as future work, including an in-depth comparison to \texttt{sledgehammer} and validation of the generated proof scripts.
References

1. Isabelle2015, http://isabelle.in.tum.de/website-Isabelle2015/
2. Blanchette, J., Fleury, M., Wand, D.: Concrete Semantics with Isabelle/HOL (2015), http://people.mpi-inf.mpg.de/~jblanche/cswi/
3. Blanchette, J., Fleury, M., Wand, D.: Concrete Semantics with Isabelle/HOL: hwso103.thy (2015), http://people.mpi-inf.mpg.de/~jblanche/cswi/hwso103.thy
4. Blanchette, J.C., Kaliszyk, C., Paulson, L.C., Urban, J.: Hammering towards QED. J. Formalized Reasoning 9(1), 101–148 (2016), http://dx.doi.org/10.6092/issn.1972-5787/4593
5. Böhme, S., Nipkow, T.: Sledgehammer: Judgement day. In: IJCAR, 2010
6. Dixon, L., Fleuriot, J.D.: IsaPlanner: A prototype proof planner in Isabelle. In: CADE, 2003
7. Freitas, L., Jones, C.B., Velykis, A., Whiteside, L.: A model for capturing and replaying proof strategies. In: Verified Software: Theories, Tools and Experiments - 6th International Conference, VSTTE 2014, Vienna, Austria, July 17-18, 2014, Revised Selected Papers. pp. 183–199 (2014), http://dx.doi.org/10.1007/978-3-319-12154-3_12
8. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: 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. pp. 207–220 (2009), http://doi.acm.org/10.1145/1629675.1629696
9. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic, Lecture Notes in Computer Science, vol. 2283. Springer (2002), http://dx.doi.org/10.1007/3-540-45949-9
10. Paulson, L.C.: The foundation of a generic theorem prover. CoRR cs.LO/9301105 (1993), http://arxiv.org/abs/cs.LO/9301105
11. Slind, K., Norrish, M.: A brief overview of HOL4. In: TPHOL (2008)