A Proof Strategy Language and Proof Script Generation for Isabelle/HOL

Yutaka Nagashima[1] and Ramana Kumar[1,2(✉)]

[1] Data61, CSIRO, Sydney, Australia
[2] Data61, CSIRO/UNSW, Sydney, Australia
ramana.kumar@data61.csiro.au

Abstract. We introduce a language, PSL, designed to capture high level proof strategies in Isabelle/HOL. Given a strategy and a proof obligation, PSL's runtime system generates and combines various tactics to explore a large search space with low memory usage. Upon success, PSL generates an efficient proof script, which bypasses a large part of the proof search. We also present PSL's monadic interpreter to show that the underlying idea of PSL is transferable to other ITPs.

1 Introduction

Currently, users of interactive theorem provers (ITPs) spend a lot of time iteratively interacting with their ITP to manually specialise and combine tactics. This time consuming process requires expertise in the ITP, making ITPs more esoteric than they should be. The integration of powerful automatic theorem provers (ATPs) into ITPs ameliorates this problem significantly; however, the exclusive reliance on general purpose ATPs makes it hard to exploit users' domain specific knowledge, leading to combinatorial explosion even for conceptually straightforward conjectures.

To address this problem, we introduce `PSL`, a programmable, extensible, meta-tool based framework, to Isabelle/HOL [22]. We provide `PSL` (available on GitHub [18]) as a language, so that its users can encode *proof strategies*, abstract descriptions of how to attack proof obligations, based on their intuitions about a conjecture. When applied to a proof obligation, `PSL`'s runtime system creates and combines several tactics based on the given proof strategy. This makes it possible to explore a larger search space than has previously been possible with conventional tactic languages, while utilising users' intuitions on the conjecture.

We developed `PSL` to use engineers' downtime: with `PSL`, we can run an automatic proof search for hours while we are attending meetings, sleeping, or reviewing papers. `PSL` makes such expensive proof search possible on machines with limited memory: `PSL`'s runtime system truncates failed proof attempts as soon as it backtracks to minimise its memory usage.

Furthermore, `PSL`'s runtime system attempts to generate efficient proof scripts from a given strategy by searching for the appropriate specialisation and combination of tactics for a particular conjecture without direct user interaction.

L. de Moura (Ed.): CADE 2017, LNAI 10395, pp. 528–545, 2017.
DOI: 10.1007/978-3-319-63046-5_32

Thus, PSL not only reduces the initial labour cost of theorem proving, but also keeps proof scripts interactive and maintainable by reducing the execution time of subsequent proof checking.

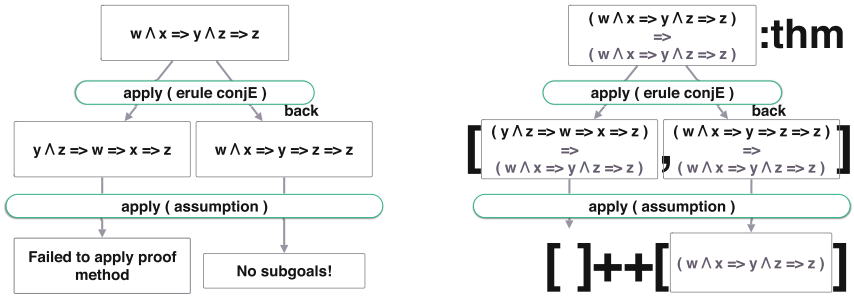
In Isabelle, `sledgehammer` adopts a similar approach [2]. It exports a proof goal to various external ATPs and waits for them to find a proof. If the external provers find a proof, `sledgehammer` tries to reconstruct an efficient proof script in Isabelle using hints from the ATPs. `sledgehammer` is often more capable than most tactics but suffers from discrepancies between the polymorphic higher-order logic of Isabelle and the monomorphic first-order logic of most backend provers. While we integrated `sledgehammer` as a sub-tool in PSL, PSL conducts a search using Isabelle tactics, thus avoiding the problems arising from the discrepancies and proof reconstruction.

The underlying implementation idea in PSL is the monadic interpretation of proof strategies, which we introduce in Sect. 6. We expect this prover-agnostic formalization brings the following strengths of PSL to other ITPs such as Lean [14] and Coq [28]:

- runtime tactic generation based on user-defined procedures,
- memory-efficient large-scale proof search, and
- generation of efficient proof scripts for proof maintenance.

2 Background

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. Tactics are context sensitive: they behave differently depending on information stored in background proof contexts. These proof contexts contain such information as the constants defined and auxiliary lemmas proved prior to the current step. Since tactic behaviour depends on the proof context, it is hard to predict the shape of the search tree in advance.



(a) External view (b) Internal view

Fig. 1. External and internal view of proof search tree

The goal is to find a node representing a solved state: one in which the proof is complete. The search 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 PSL strategy is to direct an automated search of this tree to find a solved state; PSL will reconstruct an efficient path to this state as a human-readable proof script.

Figure 1 shows an example of proof search. At the top, the tactic `erule conjE` is applied to the proof obligation $\mathtt{w} \wedge \mathtt{x} \Rightarrow \mathtt{y} \wedge \mathtt{z} \Rightarrow \mathtt{z}$. This tactic invocation produces two results, as there are two places to apply conjunction elimination. Applying conjunction elimination to $\mathtt{w} \wedge \mathtt{x}$ returns the first result, while doing so to $\mathtt{y} \wedge \mathtt{z}$ produces the second result. Subsequent application of proof by `assumption` can discharge the second result; however, `assumption` does not discharge the first one since the `z` in the assumptions is still hidden by the conjunction. Isabelle's proof language, *Isar*, returns the first result by default, but users can access the subsequent results using the keyword `back`.

Isabelle represents this non-deterministic behaviour of tactics using lazy sequences: tactics are functions of type `thm -> [thm]`, where `[·]` denotes a (possibly infinite) lazy sequence [25]. Figure 1b illustrates how Isabelle internally handles the above example where `++` stands for the concatenation of lazy sequences. Each proof state is expressed as a (possibly nested) implication which assumes proof obligations to conclude the conjecture. One may complete a proof by removing these assumptions using tactics. Tactic failure is represented as an empty sequence, which enables backtracking search by combining multiple tactics in a row [30]. For example, one can write `apply(erule conjE,assumption)` using the sequential combinator `,` (comma) in *Isar*; this tactic traverses the tree using backtracking and discharges the proof obligation without relying on the keyword `back`.

The search tree grows wider when choosing between multiple tactics, and it grows deeper when tactics are combined sequentially. In the implementation language level, the tactic combinators in Isabelle include `THEN` for sequential composition (corresponding to `,` in *Isar*), `APPEND` for non-deterministic choice, `ORELSE` for deterministic choice, and `REPEAT` for iteration.

Isabelle/HOL comes with several default tactics such as `auto`,`simp`,`induct`, `rule`, and `erule`. When using tactics, proof authors often have to adjust tactics using *modifier*s for each proof obligation. `succeed` and `fail` are special tactics: `succeed` takes a value of type `thm`, wraps it in a lazy sequence, and returns it without modifying the value. `fail` always returns an empty sequence.

3 Syntax of PSL

The following is the syntax of PSL. We made PSL's syntax similar to that of Isabelle's tactic language aiming at the better usability for users who are familiar with Isabelle's tactic language.

```
strategy := default | dynamic | special | subtool | compound
default  := Simp | Clarsimp | Fastforce | Auto | Induct
          | Rule | Erule | Cases | Coinduction | Blast
```

```
dynamic  := Dynamic (default)
special  := IsSolved | Defer | IntroClasses | Transfer
          | Normalization | Skip | Fail | User <string>
subtool  := Hammer | Nitpick | Quickcheck
compound := Thens [strategy] | Ors [strategy] | Alts [strategy]
          | Repeat (strategy) | RepeatN (strategy)
          | POrs [strategy] | PAlts [strategy]
          | PThenOne [strategy] | PThenAll [strategy]
          | Cut int (strategy)
```

The *default* strategies correspond to Isabelle's default tactics without arguments, while *dynamic* strategies correspond to Isabelle's default tactics that are specialised for each conjecture. Given a *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 automatically.

subtool represents Isabelle tools such as `sledgehammer` [2] 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 `PAlts` are similar to `Ors` and `Alts`, respectively, but they admit parallel execution of sub-strategies. `PThenOne` and `PThenAll` take exactly two sub-strategies, combine them sequentially and apply the second sub-strategy to the results of the first sub-strategy in parallel in case the first sub-strategy returns multiple results. Contrary to `PThenAll`, `PThenOne` stops its execution as soon as it produces one result from the second sub-strategy. Users can integrate user-defined tactics, including those written in Eisbach [13], into PSL strategies using `User`. `Cut` limits the degree of non-determinism within a strategy.

In the following, we explain how to write strategies and how PSL's runtime system interprets strategies with examples.

4 PSL by Example

Example 1. For our first example, we take the following lemma from an entry [23] in the Archive of Formal Proofs (AFP):

```
lemma dfs_app: "dfs g (xs @ ys) zs = dfs g ys (dfs g xs zs)"
```

where `dfs` is a recursively defined function for depth-first search. As `dfs` is defined recursively, it is natural to expect that its proof involves some sort of mathematical induction. However, we do not know exactly how we should conduct mathematical induction here; therefore, we describe this rough idea as a proof strategy, `DInductAuto`, with the keyword `strategy`, and apply it to `dfs_app` with the keyword `find_proof` as depicted in Fig. 2. The `find_proof` command tells PSL's runtime system to interpret `DInductAuto`. For example, it interprets `Auto` as Isabelle's default tactic, `auto`.

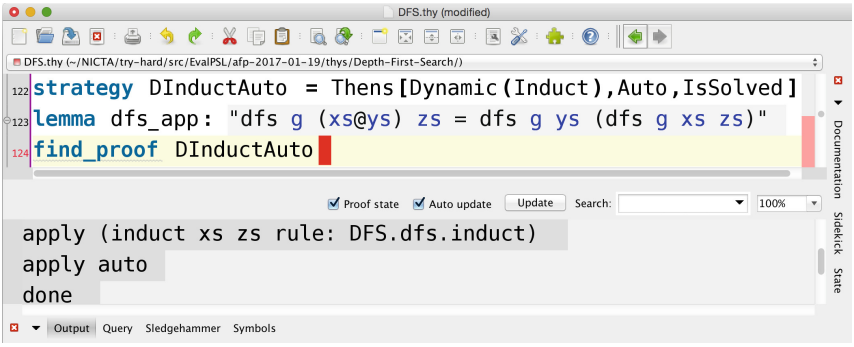


Fig. 2. Screenshot for Example 1.

The interpretation of `Dynamic(Induct)` is more involved: the runtime generates tactics using the information in `dfs_app` and its background context. First, `PSL` collects the free variables (noted in *italics* above) in `dfs_app` and 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 case, `PSL` tries to generate 4160 induct tactics for `dfs_app` by passing several combinations of modifiers to Isabelle; however, Isabelle cannot produce valid induction schemes for some combinations, and some combinations lead to the same induction scheme. The runtime removes these, focusing on the 223 unique results. `PSL`'s runtime combine these tactics with `auto` using `THEN`.

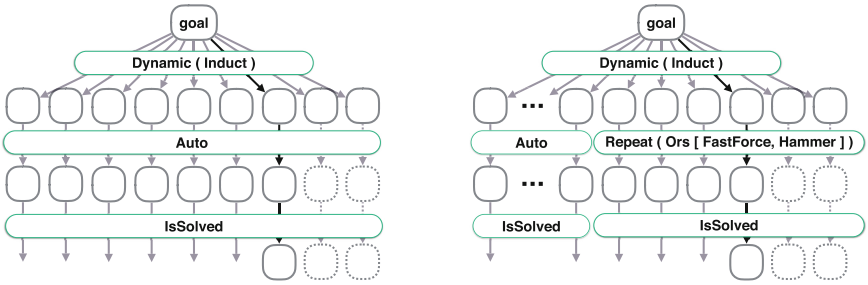
`PSL`'s runtime interprets `IsSolved` as the `is_solved` tactic, which checks whether any proof obligations are left or not. If obligations are left, `is_solved` behaves as `fail`, triggering backtracking. If not, `is_solved` behaves as `succeed`, allowing the runtime to stop the search. This is how `DInductAuto` uses `IsSolved` to ensure that no sub-goals are left before returning an efficient proof script. For `dfs_app`, `PSL` interprets `DInductAuto` as the following tactic:

```
(induct1 APPEND induct2 APPEND...) THEN auto THEN is_solved
```

where `induct_n`s are variants of the induct tactic specialised with modifiers.

Within the runtime system, Isabelle first applies `induct1` to `dfs_app`, then `auto` to the resultant proof obligations. Note that each `induct` tactic and `auto` is deterministic: it either fails or returns a lazy sequence with a single element. However, combined together with `APPEND`, the numerous variations of `induct` tactics *en masse* are non-deterministic: if `is_solved` finds remaining proof obligations, Isabelle backtracks to the next `induct` tactic, `induct2` and repeats this

process until either it discharges all proof obligations or runs out of the variations of `induct` tactics. The numerous variants of `induct` tactics from `DInductAuto` allow Isabelle to explore a larger search space than its naive alternative, `induct THEN auto`, does. Figure 3a illustrates this search procedure. Each edge and curved square represents a tactic application and a proof state, respectively, and edges leading to no object stand for tactic failures. The dashed objects represent possible future search space, which `PSL` avoids traversing by using lazy sequences.



(a) `DInductAuto` (b) `some_induct2`

Fig. 3. Proof search tree for `some_induct`

The larger search space specified by `DInductAuto` leads to a longer search time. `PSL` addresses this performance problem by tracing Isabelle's proof search: it keeps a log of successful proof attempts while removing backtracked proof attempts. The monadic interpretation discussed in Sect. 6 let `PSL` remove failed proof steps as soon as it backtracks. This minimises `PSL` memory usage, making it applicable to hours of expensive automatic proof search. 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 no-proof-obligation test imposed by `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 `dfs_app`, `PSL` generates the following proof script from `DInductAuto`.

```
apply (induct xs zs rule: DFS.dfs.induct) apply auto done
```

We implemented `PSL` as an Isabelle theory; to use it, `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 in Fig. 2; 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. DInductAuto is able to pick up the right induction scheme for relatively simple proof obligations using backtracking search. However, in some cases even if PSL picks the right induction scheme, auto fails to discharge the emerging sub-goals. In the following, we define InductHard, a more powerful strategy based on mathematical induction, by combining Dynamic (Induct) with more involved sub-strategies to use external theorem provers.

```
strategy SolveAllG = Thens[Repeat(Ors[Fastforce,Hammer]),IsSolved]
strategy PInductHard = PThenOne[Dynamic(Induct),SolveAllG]
strategy InductHard = Ors[DInductAuto, PInductHard]
```

PSL's runtime system interprets Fastforce and Hammer as the fastforce tactic and sledgehammer, respectively. Both fastforce and sledgehammer try to discharge the first sub-goal only and return an empty sequence if they cannot discharge the sub-goal.

The repetitive application of sledgehammer would be very time consuming. We mitigate this problem using Ors and PThenOne. Combined with Ors, PSL executes PInductHard only if DInductAuto fails. When PInductHard is called, it first applies Dynamic(Induct), producing various induction schemes and multiple results. Then, SolveAllG tries to discharge these results in parallel. The runtime stops its execution when SolveAllG returns at least one result representing a solved state. We apply this strategy to the following conjecture, which states the two versions of depth-first search programs (dfs2 and dfs) return the same results given the same inputs.

```
lemma "dfs2 g xs ys = dfs g xs ys"
```

Then, our machine with 28 cores returns the following script within 3 minutes:

```
apply (induct xs ys rule: DFS.dfs2.induct)
apply fastforce
apply (simp add: dfs_app)
done
```

Figure 3b roughly shows how the runtime system found this proof script. The runtime first tried to find a complete proof as in Example 1, but without much success. Then, it interpreted PInductHard. While doing so, it found that induction on *xs* and *ys* using DFS.dfs2.induct leads to two sub-goals both of which can be discharged either by fastforce or sledgehammer. For the second sub-goal, sledgehammer found out that the result of *Example* 1 can be used as an auxiliary lemma to prove this conjecture. Then, it returns an efficient proof script (simp add: dfs_app) to PSL, before PSL combines this with other parts and prints the complete proof script.

Example 3. In the previous examples, we used `IsSolved` to get a complete proof script from `PSL`. In Example 3, we show how to generate incomplete but useful proof scripts, using `Defer`. Incomplete proofs are specially useful when ITP users face numerous proof obligations, many of which are within the reach of high-level proof automation tools, such as `sledgehammer`, but a few of which are not.

Without `PSL`, Isabelle users had to manually invoke `sledgehammer` several times to find out which proof obligations `sledgehammer` can discharge. We developed a strategy, `HamCheck`, to automate this time-consuming process. The following shows its definition and a use case simplified for illustrative purposes.

```
strategy HamCheck = RepeatN(Ors[Hammer,Thens[Quickcheck,Defer]])
lemma safe_trans: shows
1:"ps_safe p s" and 2:"valid_tran p s s' c" and 3:"ps_safe p s'"
find_proof HamCheck
```

We made this example simple, so that two sub-goals, `1:"ps_safe p s`" and `3:"ps_safe p s'`", are not hard to prove; however, they are still beyond the scope of commonly used tactics, such as `fastforce`.

Generally, for a conjecture and a strategy of the form of `RepeatN (strategy)`, `PSL` applies `strategy` to the conjecture as many times as the number of proof obligations in the conjecture. In this case, `PSL` applies `Ors [Hammer, Thens [Quickcheck, Defer]]` to `safe_trans` three times.

Note that we integrated `quickcheck` and `nitpick` into `PSL` as *assertion tactics*. Assertion tactics 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. We have already seen one of them in the previous examples: `is_solved`.

`Ors [Hammer, Thens [Quickcheck, Defer]]` first applies `sledgehammer`. If `sledgehammer` does not find a proof, it tries to find counter-examples for the sub-goal using `quickcheck`. If `quickcheck` finds no counter-examples, `PSL` interprets `Defer` as `defer_tac 1`, which postpones the current sub-goal to the end of the list of proof obligations.

In this example, `sledgehammer` fails to discharge `2:"valid_tran p s s' c`". When `sledgehammer` fails, `PSL` passes `2` to `Thens [Quickcheck, Defer]`, which finds no counter-example to `2` and sends `2` to the end of the list; then, `PSL` continues working on the sub-goal `3` with `sledgehammer`. The runtime stops its execution after applying `Ors [Hammers,Thens [Quickcheck, Defer]]` three times, generating the following proof script. This script discharges `1` and `3`, but it leaves `2` as the meaningful task for human engineers, while assuring there is no obvious counter-examples for `2`.

```
apply (simp add: state_safety ps_safe_def)
defer apply (simp add: state_safety ps_safe_def)
```

5 The Default Strategy: try_hard.

PSL comes with a default strategy, try_hard. Users can apply try_hard as a completely automatic tool: engineers need not provide their intuitions by writing strategies. Unlike other user-defined strategies, one can invoke this strategy by simply typing try_hard without find_proof inside a proof attempt. The lack of input from human engineers makes try_hard less specific to each conjecture; however, we made try_hard more powerful than existing proof automation tools for Isabelle by specifying larger search spaces presented.

We conducted a Judgement Day style evaluation [3] of try_hard against selected theory files from the AFP, coursework assignments and exercises [1], and Isabelle's standard library. Tables 1, 2 and 3 show that given 300 s for each proof goal try_hard solves 1115 proof goals out of 1526,while sledgehammer found proofs for 901 of themusing the same computational resources and reconstructed proofs in Isabelle for 866 of them. This is a 14% point improvement of proof search and a 16% point increase for proof reconstruction. Moreover, 299 goals (20% of all goals) were solved only by try_hard within 300 s. They also show that a longer time-out improves the success ratio of try_hard, which is desirable for utilising engineers' downtime.

Table 1. The number of automatically proved proof obligations from assignments. TH and SH stand for the number of obligations discharged by try_hard and sledgehammer, respectively. TH\SH represents the number of goals to which try_hard found proofs but sledgehammer did not. POs stands for the number of proof obligations in the theory file. $x(y)$ for SH means sledgehammer found proofs for x proof obligations, out of which it managed to reconstruct proof scripts in Isabelle for y goals. We omit these parentheses when these numbers coincide. Note that all proofs of PSL are checked by Isabelle/HOL. Besides, sledgehammer inside PSL avoids the smt proof method, as this method is not allowed in the Archive of Formal Proofs.

assignments [1]	POs	TH	SH	TH\SH	TH	SH	TH\SH
time out	-	30s	30s	30s	300s	300s	300s
assignment_1	19	17	14(13)	4	18	14(13)	5
assignment_2	22	21	5	16	22	5	17
assignment_3	52	30	27	8	35	27	10
assignment_4	82	66	61	10	71	61	10
assignment_5	64	36	41(39)	6	55	44(42)	17
assignment_6	26	11	12(11)	2	14	13(12)	3
assignment_8	52	36	45(39)	1	40	46(39)	0
assignment_9	61	31	32(30)	6	35	32(30)	6
assignment_11	26	14	15	1	20	17	3
sum	404	262	252(241)	54	310	259(246)	71

Table 2. The number of automatically proved proof obligations from exercises.

exercises [1]	POs	TH	SH	TH\SH	TH	SH	TH\SH
time out	-	30s	30s	30s	300s	300s	300s
exercise_1	15	12	8	4	12	8	4
exercise_2	7	4	3	2	5	3	2
exercise_3	42	27	26(25)	5	29	27(26)	5
exercise_4	23	11	15	0	17	15	2
exercise_5a	13	9	11	0	11	11	0
exercise_5b	83	65	74	1	74	74	1
exercise_6	4	1	2	0	1	3	0
exercise_7a	3	0	0	0	0	0	0
exercise_7b	9	5	6	1	8	6	2
exercise_8a	10	7	7	1	7	7	1
exercise_8b	26	11	9	4	12	12	2
exercise_9	31	14	17	3	19	17	3
exercise_10	15	5	5(4)	1	6	6(5)	1
exercise_11	10	4	6	0	9	6	3
exercise_12	30	8	10	1	12	10	3
sum	321	183	199(197)	23	222	205(203)	29

Table 3. The number of automatically proved proof goals from AFP entries and Isabelle's standard libraries.

theory name	POs	TH	SH	TH\SH	TH	SH	TH\SH
time out	-	30s	30s	30s	300s	300s	300s
DFS.thy [23]	51	24	28	6	34	29	7
Efficient_Sort.thy [27]	75	27	28(26)	8	33	31(28)	9
List_Index.thy [20]	105	48	72(70)	12	67	75(72)	14
Skew_Heap.thy [21]	16	8	6(5)	4	12	8(7)	5
Hash_Code.thy [10]	16	7	4	4	11	4	7
CoCallGraph.thy [4]	141	88	78(71)	29	104	79(73)	33
Coinductive_Language.thy [29]	139	57	69(68)	11	106	70(69)	43
Context_Free_Grammar.thy [29]	29	26	2	26	29	2	27
LTL.thy [26]	97	56	61	15	78	65(62)	15
HOL/Library/Tree.thy	124	93	70(68)	32	101	73(70)	32
HOL/Library/Tree_Multiset.thy	8	8	1	7	8	1	7
sum	801	442	419(404)	154	583	437(417)	199

`try_hard` is particularly more powerful than `sledgehammer` at discharging proof obligations that can be nicely handled by the following:

- mathematical induction or co-induction,
- type class mechanism,
- specific procedures implemented as specialised tactics (such as `transfer` and `normalization`), or
- general simplification rules (such as `field_simps` and `algebra_simps`).

Furthermore, careful observation of `PSL` indicates that `PSL` can handle the so-called "hidden fact" problem in relevance filtering. Hidden facts are auxiliary lemmas that are useful to discharge a given proof obligation but are not obviously relevant. For example, a hidden fact may share no constants with the proof obligation, because it is related only via an intermediate fact. With `PSL`, a user can write a strategy that applies rewriting before relevance filtering to reveal more information. This information allows the relevance filter to find useful facts that were previously hidden. For example, the following strategy "massages" the given proof obligation before invoking the relevance filter of `sledgehammer`: `Thens [Auto, Repeat(Hammer),IsSolved]`.

For 3 theories out of 35, `try_hard` discharged fewer proof obligations, even given 300 seconds of time-out. This is due to the fact that `PSL` uses a slightly restricted version of `sledgehammer` internally for the sake of the integration with other tools and to avoid the `smt` method, which is not allowed in the AFP. In these files, `sledgehammer` can discharge many obligations and other obligations are not particularly suitable for other sub-tools in `try_hard`. Of course, given high-performance machines, users can run both `try_hard` and `sledgehammer` in parallel to maximise the chance of proving conjectures.

6 Monadic Interpretation of Strategy

The implementation of the tracing mechanism described in Sect. 4 is non-trivial: `PSL`'s tracing mechanism has to support arbitrary strategies conforming to its syntax. What is worse, the runtime behaviour of backtracking search is not completely predictable statically since `PSL` generates tactics at runtime, using information that is not available statically. Moreover, the behaviour of each tactic varies depending on the proof context and proof obligation at hand.

It is likely to cause code clutter to specify where to backtrack explicitly with references or pointers, whereas explicit construction of search tree [17] consumes too much memory space when traversing a large search space. Furthermore, both of these approaches deviate from the standard execution model of Isabelle explained in Sect. 2. This deviation makes the proof search and the efficient proof script generation less reliable. In this section, we introduce our monadic interpreter for `PSL`, which yields a modular design and concise implementation of `PSL`'s runtime system.

Program 1. Monad with zero and plus, and lazy sequence as its instance.

```
signature MONADOPLUS =
sig
  type 'a m0p;
  val return :  'a -> 'a m0p;
  val bind   :  'a m0p -> ('a -> 'b m0p) -> 'b m0p;
  val mzero  :  'a m0p;
  val ++     : ('a m0p * 'a  m0p) -> 'a m0p;
end;
structure Nondet : MONADOPLUS =
struct
  type 'a m0p     = 'a Seq.seq;
  val return      = Seq.single;
  fun bind xs f   = Seq.flat (Seq.map f xs);
  val mzero       = Seq.empty;
  fun (xs ++ ys)  = Seq.append xs ys;
end;
```

Monads in Standard ML. A monad with zero and plus is a constructor class[1] with four basic methods (`return`, `bind`, `mzero`, and `++`). As Isabelle's implementation language, Standard ML, does not natively support constructor classes, we emulated them using its module system [19]. Program 1 shows how we represent the type constructor, `seq`, as an instance of monad with zero and plus.

The body of `bind` for lazy sequences says that it applies `f` to all the elements of `xs` and concatenates all the results into one sequence. Attentive readers might notice that this is equivalent to the behaviour of `THEN` depicted in Fig. 1b and that of `Thens` shown in Fig. 3. In fact, we can define all of `THEN`, `succeed`, `fail`, and `APPEND`, using `bind, return, mzero`, and `++`, respectively.

Monadic Interpretation of Strategies. Based on this observation, we formalised `PSL`'s search procedure as a monadic interpretation of strategies, as shown in Program 2, where the type `core_strategy` stands for the internal representation of strategies. Note that `Alt` and `Or` are binary versions of `Alts` and `Ors`, respectively; `PSL` desugars `Alts` and `Ors` into nested `Alts` and `Ors`. We could have defined `Or` as a syntactic sugar using `Alt`, `mzero`, `Fail`, and `Skip`, as explained by Martin *et al.* [11]; however, we prefer the less monadic formalisation in Program 2 for better time complexity.

`eval` deals with all the atomic strategies, which correspond to ***`default`***, ***`dynamic`***, and ***`special`*** in the surface language. For the ***`dynamic`*** strategies, `eval` expands them into dynamically generated tactics making use of contextual information from the current proof state. `PSL` combines these generated tactics

[1] Constructor classes are a class mechanism on type constructors such as `list` and `option`, whereas type classes are a class mechanism on types such as `int` and `double`. Commonly used constructor classes include functor, applicative, monoid, and arrow.

Program 2. The monadic interpretation of strategies.

```
interp :: core_strategy -> 'a -> 'a m0p
interp (Atom atom_str) n     = eval atom_str n
interp  Skip  n              = return n
interp  Fail  n              = mzero
interp (str1 Then str2) n    = bind (interp str1 n) (interp str2)
interp (str1 Alt  str2) n    = interp str1 n ++ interp str2 n
interp (str1 Or   str2) n    = let val result1 = interp str1 n
  in if (result1 != mzero) then result1 else interp str2 n end
interp (Rep str) n           = interp ((str THEN (Rep str)) Or Skip) n
interp (Comb (comb, strs)) n = eval_comb (comb, map interp strs) n
```

Program 3. The writer monad transformer as a ML functor.

```
functor writer_trans (structure Log:MONOID; structure Base:MONADOPLUS) =
struct
  type 'a m0p  = (Log.monoid * 'a) Base.m0p;
  fun return (m:'a) = Base.return (Log.mempty, m) : 'a m0p;
  fun bind (m:'a m0p) (func: 'a -> 'b m0p) : 'b m0p =
    Base.bind    m          (fn (log1, res1) =>
    Base.bind   (func res1) (fn (log2, res2) =>
    Base.return (Log.mappend log1 log2, res2)));
  val mzero = Base.mzero;
  val (xs ++ ys) = Base.++ (xs, ys);
end : MONADOPLUS;
```

either with `APPEND` or `ORELSE`, depending on the nature of each tactic. `eval_comb` handles non-monadic strategy combinators, such as `Cut`. We defined the body of `eval` and `eval_comb` for each atomic strategy and strategy combinator separately using pattern matching. As is obvious in Program 2, `interp` separates the complexity of compound strategies from that of runtime tactic generation.

Adding Tracing Modularly for Proof Script Generation. We defined `interp` at the constructor class level, abstracting it from the concrete type of proof state and even from the concrete type constructor. When instantiated with lazy sequence, `interp` tries to return the first element of the sequence, working as depth-first search. This abstraction provides a clear view of how compound strategies guide proof search while producing tactics at runtime; however, without tracing proof attempts, `PSL` has to traverse large search spaces every time it checks proofs.

We added the tracing mechanism to `interp`, combining the non-deterministic monad, `Nondet`, with the writer monad. To combine multiple monads, we emulate monad transformers using ML functors: Program 3 shows our ML functor, `writer_trans`, which takes a module of `MONADOPLUS`, adds the logging mechanism to it, and returns a module equipped with both the capability of the base

monad and the logging mechanism of the writer monad. We pass `Nondet` to `writer_trans` as the base monad to combine the logging mechanism and the backtracking search based on non-deterministic choice. Observe Programs 1, 2 and 3 to see how `Alt` and `Or` truncate failed proof attempts while searching for a proof. The returned module is based on a new type constructor, but it is still a member of `MONADOPLUS`; therefore, we can re-use `interp` without changing it.

History-Sensitive Tactics using the State Monad Transformer. The flexible runtime interpretation might lead `PSL` into a non-terminating loop, such as `REPEAT succeed`. To handle such loops, `PSL` traverses a search space using iterative deepening depth-first search (IDDFS). However, passing around information about depth as an argument of `interp` as following quickly impairs its simplicity:

```
interp (t1 CSeq t2) level n = if level < 1 then return n else ...
interp (t1 COr  t2) level n = ...
```

where `level` stands for the remaining depth `interp` can proceed for the current iteration.

We implemented IDDFS without code clutter, introducing the idea of a *history-sensitive tactic*: a tactic that takes the log of proof attempts into account. Since the writer monad does not allow us to access the log during the search time, we replaced the writer monad transformer with the state monad transformer, with which the runtime keeps the log of proof attempt as the "state" of proof search and access it during search. By measuring the length of "state", `interp` computes the current depth of proof search at runtime.

The modular design and abstraction discussed above made this replacement possible with little change to the definition of `interp`: we only need to change the clause for `Atom`, providing a wrapper function, `iddfc`, for `eval`, while other clauses remain intact.

```
inter (CAtom atom_str) n = iddfc limit eval atom_str n
```

`iddfc limit` first reads the length of "state", which represents the number of edges to the node from the top of the implicit proof search tree. Then, it behaves as `fail` if the length exceeds `limit`; if not, it executes `eval atom_str n`.[2]

7 Related Work

ACL2 [9] is a functional programming language and mostly automated first-order theorem prover. ACL2 is known for the so-called waterfall model, which is essentially repeated application of various heuristics. Its users can guide proof search by supplying arguments called "hints", but the underlining operational procedure of the waterfall model itself is fixed. ACL2 does not produce efficient proof scripts after running the waterfall algorithm.

[2] In this sense, we implemented IDDFS as a tactic combinator.

PVS [24] provides a collection of commands called "strategies". Despite the similarity of the name to `PSL`, strategies in PVS correspond to tactics in Isabelle. The highest-level strategy in PVS, grind, can produce re-runnable proof scripts containing successful proof steps only. However, scripts returned by grind describe steps of much lower level than human engineers would write manually, while `PSL`'s returned scripts are based on tactics engineers use. Furthermore, grind is known to be useful to complete a proof that does *not* require induction, while `try_hard` is good at finding proofs involving mathematical induction.

SEPIA [8] is an automated proof generation tool in Coq. Taking existing Coq theories as input, SEPIA first produces proof traces, from which it infers an extended finite state machine. Given a conjecture, SEPIA uses this model to search for its proof. The authors of SEPIA chose to use breadth-first search (BFS) to find shorter proofs. For `PSL` we could emulate the BFS strategy within the IDDFS framework. However, our experience tells us that the search tree tends to be very wide and some tactics, such as `induct`, need to be followed by other tactics to complete proofs. Therefore, we chose IDDFS for `PSL`. Both SEPIA and `PSL` off-load the construction of proof scripts to search and try to reconstruct efficient proof scripts. Compared to SEPIA, `PSL` allows users to specify their own search strategies to utilize the engineer's intuition, which enables `PSL` to return incomplete proof scripts, as discussed in Sect. 4.

Martin *et al.* first discussed a monadic interpretation of tactics for their language, *Angel*, in an unpublished draft [12]. We independently developed `interp` with the features discussed above, lifting the framework from the tactic level to the strategy level to traverse larger search spaces. The two interpreters for different ITPs turned out to be similar to each other, suggesting our approach is not specific to Isabelle but can be used for other ITPs.

Similar to Ltac [6] in Coq, Eisbach [13] is a framework to write *proof method*s in Isabelle. Proof methods are the *Isar* syntactic layer of tactics. Eisbach does not generate methods dynamically, trace proof attempts, nor support parallelism natively. Eisbach is good when engineers already know how to prove their conjecture, while `try_hard` is good when they want to find out how to prove it.

IsaPlanner [7] offers a framework for encoding and applying common patterns of reasoning in Isabelle, following the style of proof planning [5]. IsaPlanner addresses the performance issue by a memoization technique, on the other hand `try_hard` strips off backtracked steps while searching for a proof, which Isabelle can check later without `try_hard`. While IsaPlanner works on its own data structure *reasoning state*, `try_hard` managed to minimize the deviation from Isabelle's standard execution model using constructor classes.

8 Conclusions

`PSL` improves proof automation in higher-order logic, allowing us to exploit both the engineer's intuition and various automatic tools. The simplicity of the design is our intentional choice: we reduced the process of interactive proof development

to the well-known dynamic tree search problem and added new features (efficient-proof script generation and IDDFS) by safely abstracting the original execution model and employing commonly used techniques (monad transformers).

We claim that our approach enjoys significant advantages. Despite the simplicity of the design, our evaluations indicate that `PSL` reduces the labour cost of ITP significantly. The conservative extension to the original model lowers the learning barrier of `PSL` and makes our proof script generation reliable by minimising the deviation. The meta-tool approach makes the generated proof script independent of `PSL`, separating the maintenance of proof scripts from that of `PSL`; furthermore, by providing a common framework for various tools we supplement one tool's weakness (e.g. induction for `sledgehammer`) with other tools' strength (e.g. the `induct` tactic), while enhancing their capabilities with runtime tactic generation. The parallel combinators reduce the labour-intensive process of interactive theorem proving to embarrassingly parallel problems. The abstraction to the constructor class and reduction to the tree search problem make our ideas transferable: other ITPs, such as Lean and Coq, handle inter-tactic backtracking, which is best represented in terms of `MONADOPLUS`.

Acknowledgements. We thank Jasmin C. Blanchette for his extensive comments that improved the evaluation of `try_hard`. Pang Luo helped us for the evaluation. Leonardo de Moura, Daniel Matichuk, Kai Engelhardt, and Gerwin Klein provided valuable comments on an early draft of this paper. We thank the anonymous reviewers for useful feedback, both at CADE-26 and for previous versions of this paper at other conferences. This work was partially funded by the ERC Consolidator grant 649043 - AI4REASON.

A Appendix: Details of the Evaluation

All evaluations were conducted on a Linux machine with Intel (R) Core (TM) i7-600 @ 3.40 GHz and 32 GB memory. For both tools, we set the time-out of proof search to 30 and 300 s for each proof obligation.

Prior to the evaluation, the relevance filter of `sledgehammer` was trained on 27,041 facts and 18,695 non-trivial Isar proofs from the background libraries imported by theories under evaluation for both tools. Furthermore, we forbid `sledgehammer` inside `PSL` from using the `smt` method for proof reconstruction, since the AFP does not permit this method.

Note that `try_hard` does not use parallel strategy combinators which exploit parallelism. The evaluation tool does not allow `try_hard` to use multiple threads either. Therefore, given the same time-out, `try_hard` and `sledgehammer` enjoy the same amount of computational resources, assuring the fairness of the evaluation results.

The evaluation tool [16] and results [15] are available at our websites. We provide the evaluation tool and results in the following websites:

- http://ts.data61.csiro.au/Downloads/cade26_evaluation/
- http://ts.data61.csiro.au/Downloads/cade26_results/

References

1. Blanchette, J., Fleury, M., Wand, D.: Concrete Semantics with Isabelle/HOL (2015). http://people.mpi-inf.mpg.de/~jblanche/cswi/
2. 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
3. Böhme, S., Nipkow, T.: Sledgehammer: Judgement day. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS (LNAI), vol. 6173, pp. 107–121. Springer, Heidelberg (2010). doi:10.1007/978-3-642-14203-1_9
4. Breitner, J.: The safety of call arity. Archive of Formal Proofs, February 2015. http://isa-afp.org/entries/Call_Arity.shtml. Formal proof development
5. Bundy, A.: The use of explicit plans to guide inductive proofs. In: Lusk, E., Overbeek, R. (eds.) CADE 1988. LNCS, vol. 310, pp. 111–120. Springer, Heidelberg (1988). doi:10.1007/BFb0012826
6. Delahaye, D.: A tactic language for the system Coq. In: Parigot, M., Voronkov, A. (eds.) LPAR 2000. LNAI, vol. 1955, pp. 85–95. Springer, Heidelberg (2000). doi:10.1007/3-540-44404-1_7
7. Dixon, L., Fleuriot, J.: IsaPlanner: A prototype proof planner in isabelle. In: Baader, F. (ed.) CADE 2003. LNCS (LNAI), vol. 2741, pp. 279–283. Springer, Heidelberg (2003). doi:10.1007/978-3-540-45085-6_22
8. Gransden, T., Walkinshaw, N., Raman, R.: SEPIA: search for proofs using inferred automata. In: Felty, A.P., Middeldorp, A. (eds.) CADE 2003. LNCS, vol. 9195, pp. 246–255. Springer, Heidelberg (2015). http://dx.doi.org/10.1007/978-3-319-21401-6_16
9. Kaufmann, M., Moore, J.S., Manolios, P.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers, Norwell (2000)
10. Lammich, P.: Collections framework. Archive of Formal Proofs, November 2009. http://isa-afp.org/entries/Collections.shtml. Formal proof development
11. Martin, A.P., Gardiner, P.H.B., Woodcock, J.: A tactic calculus-abridged version. Formal. Asp. Comput. **8**(4), 479–489 (1996). http://dx.doi.org/10.1007/BF01213535
12. Martin, A., Gibbons, J.: A monadic interpretation of tactics (2002)
13. Matichuk, D., Wenzel, M., Murray, T.: An isabelle proof method language. In: Klein, G., Gamboa, R. (eds.) ITP 2014. LNCS, vol. 8558, pp. 390–405. Springer, Cham (2014). doi:10.1007/978-3-319-08970-6_25
14. de Moura, L., Kong, S., Avigad, J., van Doorn, F., von Raumer, J.: The Lean theorem prover (system description). In: Felty, A.P., Middeldorp, A. (eds.) CADE 2003. LNCS, vol. 9195, pp. 378–388. Springer, Heidelberg (2015). http://dx.doi.org/10.1007/978-3-319-21401-6_26
15. Nagashima, Y.: Evaluation Results (2016). http://ts.data61.csiro.au/Downloads/cade26_results/
16. Nagashima, Y.: Evaluation Tool (2016). http://ts.data61.csiro.au/Downloads/cade26_evaluation/
17. Nagashima, Y.: Keep failed proof attempts in memory. In: Isabelle Workshop, Nancy, France, August 2016
18. Nagashima, Y.: PSL (2016). https://github.com/data61/PSL
19. Nagashima, Y., O'Connor, L.: Close encounters of the higher kind - emulating constructor classes in standard ML, September 2016

20. Nipkow, T.: List index. Archive of Formal Proofs, February 2010. http://isa-afp.org/entries/List-Index.shtml. Formal proof development
21. Nipkow, T.: Skew heap. Archive of Formal Proofs, August 2014. http://isa-afp.org/entries/Skew_Heap.shtml. Formal proof development
22. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002). http://dx.doi.org/10.1007/3-540-45949-9
23. Nishihara, T., Minamide, Y.: Depth first search. Archive of Formal Proofs, June 2004. http://isa-afp.org/entries/Depth-First-Search.shtml. Formal proof development
24. Owre, S., Rushby, J.M., Shankar, N.: PVS: A prototype verification system. In: Kapur, D. (ed.) CADE 1992. LNCS, vol. 607, pp. 748–752. Springer, Heidelberg (1992). doi:10.1007/3-540-55602-8_217
25. Paulson, L.C.: The foundation of a generic theorem prover. CoRR cs.LO/9301105 (1993). http://arxiv.org/abs/cs.LO/9301105
26. Sickert, S.: Linear temporal logic. Archive of Formal Proofs, March 2016. http://isa-afp.org/entries/LTL.shtml. Formal proof development
27. Sternagel, C.: Efficient mergesort. Archive of Formal Proofs, November 2011. http://isa-afp.org/entries/Efficient-Mergesort.shtml. Formal proof development
28. The Coq development team: The Coq proof assistant reference manual (2009)
29. Traytel, D.: A codatatype of formal languages. Archive of Formal Proofs, November 2013. http://isa-afp.org/entries/Coinductive_Languages.shtml. Formal proof development
30. Wadler, P.: How to replace failure by a list of successes a method for exception handling, backtracking, and pattern matching in lazy functional languages. In: Jouannaud, J.-P. (ed.) FPCA 1985. LNCS, vol. 201, pp. 113–128. Springer, Heidelberg (1985). doi:10.1007/3-540-15975-4_33