Goal-Oriented Conjecturing for Isabelle/HOL

Yutaka Nagashima[1,2] and Julian Parsert[2(✉)]

[1] CIIRC, Czech Technical University in Prague, Prague, Czech Republic
[2] Department of Computer Science, University of Innsbruck, Innsbruck, Austria
{yutaka.nagashima,julian.parsert}@uibk.ac.at

Abstract. We present PGT, a Proof Goal Transformer for Isabelle/HOL. Given a proof goal and its background context, PGT attempts to generate conjectures from the original goal by transforming the original proof goal. These conjectures should be weak enough to be provable by automation but sufficiently strong to identify and prove the original goal. By incorporating PGT into the pre-existing PSL framework, we exploit Isabelle's strong automation to identify and prove such conjectures.

1 Introduction

Consider the following two reverse functions defined in literature [9]:

```
primrec itrev:: "'a list  'a list  'a list" where
 "itrev [] ys = ys" | "itrev (x#xs) ys = itrev xs (x#ys)"
primrec rev :: "'a list  'a list" where
 "rev [] = []" | "rev (x # xs) = rev xs @ [x]"
```

How would you prove their equivalence "itrev xs [] = rev xs"? Induction comes to mind. However, it turns out that Isabelle's default proof methods, induct and induct_tac, are unable to handle this proof goal effectively.

Previously, we developed PSL [8], a programmable, meta-tool framework for Isabelle/HOL. With PSL one can write the following strategy for induction:

```
strategy DInd = Thens [Dynamic (Induct), Auto, IsSolved]
```

PSL's Dynamic keyword creates variations of the induct method by specifying different combinations of promising arguments found in the proof goal and its background proof context. Then, DInd combines these induction methods with the general purpose proof method, auto, and is_solved, which checks if there is

Y. Nagashima—Supported by the European Regional Development Fund under the project AI & Reasoning (reg. no.CZ.02.1.01/0.0/0.0/15_003/0000466)
J. Parsert—Supported by the European Research Council (ERC) grant no 714034 SMART.

any proof goal left after applying `auto`. As shown in Fig. 1a, `PSL` keeps applying the combination of a specialization of `induct` method and `auto`, until either `auto` discharges all remaining sub-goals or `DInd` runs out of the variations of `induct` methods as shown in Fig. 1a.

This approach works well only if the resulting sub-goals after applying some `induct` are easy enough for Isabelle's automated tools (such as `auto` in `DInd`) to prove. When proof goals are presented in an automation-unfriendly way, however, it is not enough to set a certain combination of arguments to the `induct` method. In such cases engineers have to investigate the original goal and come up with auxiliary lemmas, from which they can derive the original goal.

In this paper, we present `PGT`, a novel design and prototype implementation[1] of a conjecturing tool for Isabelle/HOL. We provide `PGT` as an extension to `PSL` to facilitate the seamless integration with other Isabelle sub-tools. Given a proof goal, `PGT` produces a series of conjectures that might be useful in discharging the original goal, and `PSL` attempts to identify the right one while searching for a proof of the original goal using those conjectures.

2 System Description

2.1 Identifying Valuable Conjectures via Proof Search

To automate conjecturing, we added the new language primitive, `Conjecture` to `PSL`. Given a proof goal, `Conjecture` first produces a series of conjectures that might be useful in proving the original theorem, following the process described in Sect. 2.2. For each conjecture, `PGT` creates a `subgoal_tac` method and inserts the conjecture as the premise of the original goal. When applied to `"itrev xs [] = rev xs"`, for example, `Conjecture` generates the following proof method along with 130 other variations of the `subgoal_tac` method:

```
apply (subgoal_tac "!!Nil. itrev xs Nil = rev xs @ Nil")
```

where `!!` stands for the universal quantifier in Isabelle's meta-logic. Namely, `Conjecture` introduced a variable of name `Nil` for the constant `[]`. Applying this method to the goal results in the following two new sub-goals:

```
1. (!!Nil. itrev xs Nil = rev xs @ Nil) ==> itrev xs [] = rev xs
2. !!Nil. itrev xs Nil = rev xs @ Nil
```

`Conjecture` alone cannot determine which conjecture is useful for the original goal. In fact, some of the generated statements are not even true or provable. To discard these non-theorems and to reduce the size of `PSL`'s search space, we combine `Conjecture` with `Fastforce` (corresponding to the `fastforce` method) and `Quickcheck` (corresponding to Isabelle's sub-tool *quickcheck* [3]) sequentially as well as `DInd` as follows:

[1] Available at Github https://github.com/data61/PSL/releases/tag/v0.1.1. The example of this paper appears in `PSL/PGT/Example.thy`.

```
strategy CDInd = Thens [Conjecture, Fastforce, Quickcheck, DInd]
```

Importantly, `fastforce` does not return an intermediate proof goal: it either discharges the first sub-goal completely or fails by returning an empty sequence. Therefore, whenever `fastforce` returns a new proof goal to a sub-goal resulting from `subgoal_tac`, it guarantees that the conjecture inserted as a premise is strong enough for Isabelle to prove the original goal. In our example, the application of `fastforce` to the aforementioned first sub-goal succeeds, changing the remaining sub-goals to the following:

```
1. !!Nil. itrev xs Nil = rev xs @ Nil
```

However, `PSL` still has to deal with many non-theorems: non-theorems are often strong enough to imply the original goal due to the principle of explosion. Therefore, `CDInd` applies `Quickcheck` to discard easily refutable non-theorems. The atomic strategy `Quickcheck` returns the same sub-goal only if Isabelle's sub-tool quickcheck does not find a counter example, but returns an empty sequence otherwise.

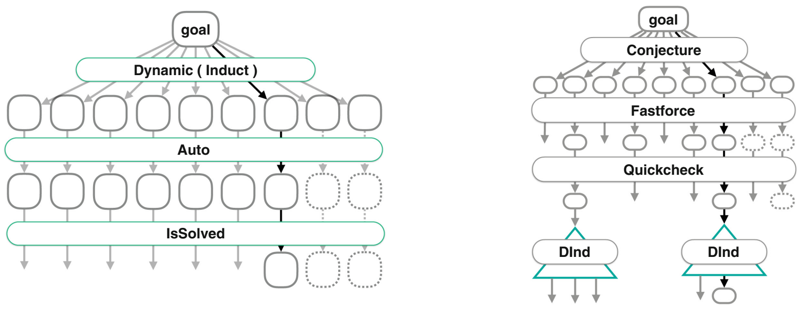
Now we know that the remaining conjectured goals are strong enough to imply the original goal and that they are not easily refutable. Therefore, `CDInd` applies its sub-strategy `DInd` to the remaining sub-goals and it stops its proof search as soon as it finds the following proof script, which will be printed in Isabelle/jEdit's output panel.

```
apply (subgoal_tac "!!Nil. itrev xs Nil = rev xs @ Nil")
apply fastforce apply (induct xs) apply auto done
```

Figure 1b shows how `CDInd` narrows its search space in a top-down manner. Note that `PSL` lets you use other Isabelle sub-tools to prune conjectures. For example, you can use both *nitpick* [1] and quickcheck: `Thens [Quickcheck, Nitpick]` in `CDInd`. It also let you combine `DInd` and `CDInd` into one: `Ors [DInd, CDInd]`.

2.2 Conjecturing

Section 2.1 has described how we identify useful conjectures. Now, we will focus on how `PGT` creates conjectures in the first place. `PGT` introduced both automatic conjecturing (`Conjecture`) and automatic generalization (`Generalize`). Since the conjecturing functionality uses generalization, we will only describe the former. We now walk through the main steps that lead from a user defined goal to a set of potentially *useful* conjectures, as illustrated in Fig. 2. We start with the extraction of constants and sub-terms, continue with generalization, goal oriented conjecturing, and finally describe how the resulting terms are sanitized.



(a) Search tree of `DInd` (b) Search tree of `CDInd`

Fig. 1. `PSL`'s proof search with/without `PGT`.

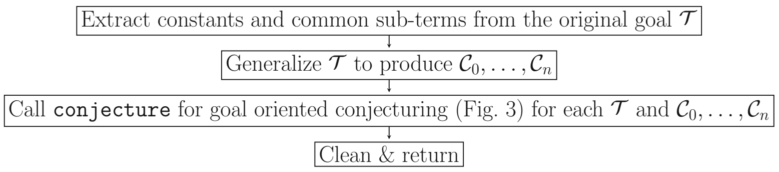

Extract constants and common sub-terms from the original goal $\mathcal{T}$
Generalize $\mathcal{T}$ to produce $\mathcal{C}_0, \ldots, \mathcal{C}_n$
Call conjecture for goal oriented conjecturing (Fig. 3) for each $\mathcal{T}$ and $\mathcal{C}_0, \ldots, \mathcal{C}_n$
Clean & return

Fig. 2. The overall workflow of `Conjecture`.

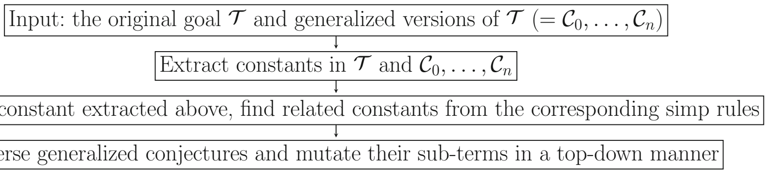
Extraction of Constants and Common Sub-terms. Given a term representation $\mathcal{T}$ of the original goal, `PGT` extracts the constants and sub-terms that appear multiple times in $\mathcal{T}$. In the example from Sect. 1, `PGT` collects the constants `rev`, `itrev`, and `[]`.

Generalization. Now, `PGT` tries to generalize the goal $\mathcal{T}$. Here, `PGT` alone cannot determine over which constant or sub-terms it should generalize $\mathcal{T}$. Hence, it creates a generalized version of $\mathcal{T}$ for each constant and sub-term collected in the previous step. For `[]` in the running example, `PGT` creates the following generalized version of $\mathcal{T}$: `!!Nil. itrev xs Nil = rev xs`.

Goal Oriented Conjecturing. This step calls the function `conjecture`, illustrated in Fig. 3, with the original goal $\mathcal{T}$ and each of the generalized versions of $\mathcal{T}$ from the previous step ($\mathcal{C}_0, \ldots, \mathcal{C}_n$). The following code snippet shows part of `conjecture`:

```
fun cnjcts t = flat (map (get_cnjct generalisedT t) consts)
fun conj (trm as Abs (_,_,subtrm)) = cnjcts trm @ conj subtrm
 |  conj (trm as App (t1,t2)) = cnjcts trm @ conj t1 @ conj t2
 |  conj trm = cnjcts trm
```

For each $\mathcal{T}$ and $\mathcal{C}_i$ for $0 \leq i \leq n$, `conjecture` first calls `conj`, which traverses the term structure of each $\mathcal{T}$ or $\mathcal{C}_i$ in a top-down manner. In the running


Input: the original goal $\mathcal{T}$ and generalized versions of $\mathcal{T}$ ($= \mathcal{C}_0, \ldots, \mathcal{C}_n$)

Extract constants in $\mathcal{T}$ and $\mathcal{C}_0, \ldots, \mathcal{C}_n$

For each constant extracted above, find related constants from the corresponding simp rules

Traverse generalized conjectures and mutate their sub-terms in a top-down manner

Fig. 3. The workflow of the `conjecture` function.

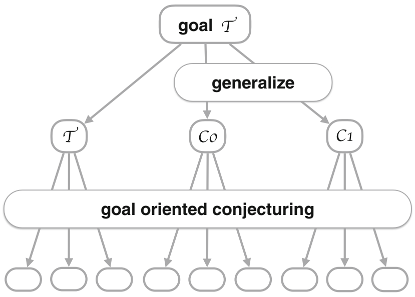


Fig. 4. `PSL`'s sequential generalization and goal oriented conjecturing.

example, `PGT` takes some $\mathcal{C}_k$, say `!!Nil. itrev xs Nil = rev xs`, as an input and applies `conj` to it.

For each sub-term the function `get_cnjct` in `cnjcts` creates new conjectures by replacing the sub-term (`t` in `cnjcts`) in $\mathcal{T}$ or $\mathcal{C}_i$ (`generalisedT`) with a new term. This term is generated from the sub-term (`t`) and the constants (`consts`). These are obtained from simplification rules that are automatically derived from the definition of a constant that appears in the corresponding $\mathcal{T}$ or $\mathcal{C}_i$.

In the example, `PGT` first finds the constant `rev` within $\mathcal{C}_k$. Then, `PGT` finds the simp-rule (`rev.simps(2)`) relevant to `rev` which states, `rev (?x # ?xs) = rev ?xs @ [?x]`, in the background context. Since `rev.simps(2)` uses the constant `@`, `PGT` attempts to create new sub-terms using `@` while traversing in the syntax tree of `!!Nil. itrev xs Nil = rev xs` in a top-down manner.

When `conj` reaches the sub-term `rev xs`, `get_cnjct` creates new sub-terms using this sub-term, `@` (an element in `consts`), and the universally quantified variable `Nil`. One of these new sub-terms would be `rev xs @ Nil`[2]. Finally, `get_cnjct` replaces the original sub-term `rev xs` with this new sub-term in $\mathcal{C}_k$, producing the conjecture: `!!Nil. itrev xs Nil = rev xs @ Nil`.

Note that this conjecture is not the only conjecture produced in this step: `PGT`, for example, also produces `!!Nil. itrev xs Nil = Nil @ rev xs`, by replacing `rev xs` with `Nil @ rev xs`, even though this conjecture is a non-theorem. Figure 4 illustrates the sequential application of generalization in the previous paragraph and goal oriented conjecturing described in this paragraph.

[2] Note that `Nil` is a universally quantified variable here.

Clean and Return. Most produced conjectures do not even type check. This step removes them as well as duplicates before passing the results to the following sub-strategy (`Then [Fastforce, Quickcheck, DInd]` in the example).

3 Conclusion

We presented an automatic conjecturing tool `PGT` and its integration into `PSL`. Currently, `PGT` tries to generate conjectures using previously derived simplification rules as hints. We plan to include more heuristics to prioritize conjectures before passing them to subsequent strategies.

Most conjecturing tools for Isabelle, such as *IsaCoSy* [6] and *Hipster* [7], are based on the bottom-up approach called *theory exploration* [2]. The drawback is that they tend to produce uninteresting conjectures. In the case of IsaCoSy the user is tasked with pruning these by hand. Hipster uses the difficulty of a conjecture's proof to determine or measure its usefulness. Contrary to their approach, `PGT` produces conjectures by mutating original goals. Even though `PGT` also produces unusable conjectures internally, the integration with `PSL`'s search framework ensures that `PGT` only presents conjectures that are indeed useful in proving the original goal. Unlike Hipster, which is based on a Haskell code base, `PGT` and `PSL` are an Isabelle theory file, which can easily be imported to any Isabelle theory. Finally, unlike Hipster, `PGT` is not limited to equational conjectures.

Gauthier and Kaliszyk described conjecturing across proof corpora [4]. While `PGT` creates conjectures by mutating the original goal, Gauthier *et al.* produced conjectures by using statistical analogies extracted from large formal libraries [5].

References

1. Blanchette, J.C.: Nitpick: a counterexample generator for Isabelle/HOL based on the relational model finder Kodkod. In: LPAR-17, pp. 20–25 (2010)
2. Buchberger, B.: Theory exploration with theorema (2000)
3. Bulwahn, L.: The new quickcheck for Isabelle. In: Hawblitzel, C., Miller, D. (eds.) CPP 2012. LNCS, vol. 7679, pp. 92–108. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-35308-6_10
4. Gauthier, T., Kaliszyk, C.: Sharing HOL4 and HOL light proof knowledge. In: Davis, M., Fehnker, A., McIver, A., Voronkov, A. (eds.) LPAR 2015. LNCS, vol. 9450, pp. 372–386. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48899-7_26
5. Gauthier, T., Kaliszyk, C., Urban, J.: Initial experiments with statistical conjecturing over large formal corpora, pp. 219–228 (2016). http://ceur-ws.org/Vol-1785/W23.pdf
6. Johansson, M., Dixon, L., Bundy, A.: Conjecture synthesis for inductive theories. J. Autom. Reason. **47**(3), 251–289 (2011)

7. Johansson, M., Rosén, D., Smallbone, N., Claessen, K.: Hipster: integrating theory exploration in a proof assistant. In: Watt, S.M., Davenport, J.H., Sexton, A.P., Sojka, P., Urban, J. (eds.) CICM 2014. LNCS (LNAI), vol. 8543, pp. 108–122. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08434-3_9
8. Nagashima, Y., Kumar, R.: A proof strategy language and proof script generation for Isabelle/HOL. In: de Moura, L. (ed.) CADE 2017. LNCS (LNAI), vol. 10395, pp. 528–545. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63046-5_32
9. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9