Inductive Benchmarks for Automated Reasoning

Márton Hajdu[1], Petra Hozzová[1(✉)], Laura Kovács[1], Johannes Schoisswohl[1,2], and Andrei Voronkov[2,3]

[1] TU Wien, Vienna, Austria
{marton.hajdu,petra.hozzova,laura.kovacs}@tuwien.ac.at,
johannes.schoisswohl@manchester.ac.uk
[2] University of Manchester, Manchester, UK
[3] EasyChair, London, UK
andrei@voronkov.com

Abstract. We present a large set of benchmarks for automated theorem provers that require inductive reasoning. Motivated by the need to compare first-order theorem provers, SMT solvers and inductive theorem provers, the setting of our examples follows the SMT-LIB standard. Our benchmark set contains problems with inductive data types as well as integers. In addition to SMT-LIB encodings, we provide translations to some other less common input formats.

1 Introduction

Recently, automated reasoning approaches have been extended with inductive reasoning capabilities, for example in the context of superposition theorem proving [5,6,12] and SMT solving [13]. Evaluation of these developments prompts comparison not only among first-order theorem provers and/or SMT solvers but also with inductive provers (e.g., ACL2 [3], Zeno [15] or Imandra [11]). As a part of our work on automating induction in the first-order theorem prover Vampire [6], we created a benchmark set of 3516 benchmarks based on variations of properties of inductive data types as well as integers. To facilitate comparison of different solvers and provers, we translated our benchmarks into the input formats of other state-of-the-art inductive reasoners, supporting for example the SMT-LIB input format [2] and functional program encodings.

Our dataset is comparable to the TIP repository of inductive benchmarks [4], with which it shares 9 benchmarks. We note however that TIP focuses on classic problems with inductive data types, while our dataset contains variants of problems of increasing sizes for both inductive data types and integers. Furhter, TIP uses a non-standard variant of SMT-LIB, and offers tools for translating the benchmarks into standard SMT-LIB. Our dataset employs the current standard SMT-LIB 2.6 syntax, allowing us to potentially integrate our examples in any repository using the SMT-LIB standard. Our benchmark set is available at:

https://github.com/vprover/inductive_benchmarks

F. Kamareddine and C. Sacerdoti Coen (Eds.): CICM 2021, LNAI 12833, pp. 124–129, 2021.
https://doi.org/10.1007/978-3-030-81097-9_9

2 Benchmark Format

We provide all benchmarks in the standard SMT-LIB 2.6 syntax. We chose SMT-LIB as the main format for our benchmarks, since it is the most common format used by automated reasoners (SMT solvers and first-order provers, e.g., CVC4 [1] or Vampire [8]) and verification tools (e.g., CBMC [9], Dafny [10], or eThor [14]). In our examples, we use the SMT-LIB construct `declare-fun` to declare functions and `assert` to axiomatize functions (see the example benchmarks in Sect. 3). In addition to the SMT-LIB syntax, we also translated our examples to other formats depending on the data types used in these examples: three subsets of our benchmark set use inductively defined data types, and one subset uses integers (see Sect. 3). For the benchmarks with inductively defined data types, we also provide SMT-LIB encoding using the `define-fun-rec` construct for recursive function definitions.

Besides the SMT-LIB format, we also provide our benchmarks translated into other, less common input formats supported by state-of-the-art solvers for automating induction. Namely, for our benchmarks with inductively defined data types, we provide two encodings for Zipperposition [5] (using Zipperposition's native input format `.zf` with/without function definitions encoded as rewrite rules), and when possible[1] functional program encodings for ACL2 [3] (in Lisp), Imandra [11] (in OCaml) and Zeno [15] (in Haskell). For our inductive benchmarks over integers, we only provide translation into Lisp for ACL2. To the best of our knowledge, in addition to Vampire [7] and CVC4 [13], ACL2 is the only prover supporting inductive reasoning with integers.

3 Benchmark Categories

Our benchmark set consists of two categories, requiring different kinds of inductive reasoning, as follows. The benchmark category `dty` uses structural induction over inductively defined data types, whereas our `int` benchmark suite exploits integer induction. Further, our benchmark set `dty` is organized within three categories `nat`, `list` and `tree`, respectively collecting inductive properties over naturals, lists and trees.

3.1 `dty` - Benchmarks with Inductively Defined Data Types

The 3396 problems within the category `dty` involve three different inductively defined data types: natural numbers, lists of natural numbers, and binary trees of natural numbers. These data types are defined as follows:

```
(declare-datatypes ((nat 0) (list 0) (tree 0))
  (((zero) (s (s0 nat)))
   ((nil) (cons (head nat) (tail list)))
   ((Nil) (node (lc tree) (val nat) (rc tree)))))
```

[1] Some concepts, like conjectures that contain existential quantification, or some uninterpreted functions used to model out of bounds access for list indexing, are not straightforwardly translatable into these formats.

The benchmark category `dty` collects results of [6]. It is split into three subcategories `nat`, `list`, and `tree`, depending on the algebraic data types used in the examples. The category `nat` uses natural numbers only, `list` uses lists and natural numbers, and `tree` uses all three of the data types. Each of these categories within `dty` contains examples defining functions and predicates on the respective data type and a conjecture/goal to prove about these functions and predicates, as described next. To avoid repetition in the displayed examples, we use short descriptions of repeated content beginning with the comment sign `;-`.

`nat` *Examples.* The category `nat` contains a set of hand-crafted benchmarks encoding basic properties like commutativity of addition and multiplication. Additionally, `nat` contains three groups of generated benchmarks. In group `add_<m>var_<n>occ`, the conjecture of each benchmark consists of an equality of two sums of variables, with arbitrary bracketing, and `n` variables on each sides of the equality, where `m` distinct variables occur in the conjecture. In group `add_<n>sym`, the conjectures are equalities with an arbitrary combination of the successor function, zero, addition, and variables, on both hand sides. Each side of the equality in these benchmarks contains `n` symbols in total. The group `leq_<m>var_<n>_<o>occ` has a less-or-equal inequality as conjecture. It contains `m` distinct variables, with a total of `n` variables on the left-hand side arbitrarily added up, and a total of `o` variables occurring on the right-hand side, where each variable on the left-hand side is contained on the right-hand side at least as often as on the left one in order to ensure that the conjecture is indeed valid.

Inductive `nat` example from the set `add_2var_4occ`

```
(set-logic UFDT)

(declare-datatypes ((nat 0)) (((zero) (s (s0 nat)))))

(declare-fun add (nat nat) nat)
(assert (forall ((y nat)        ) (= (add  zero y) y            )))
(assert (forall ((x nat) (y nat)) (= (add (s x) y) (s (add x y)))))

(assert (not (forall ((v0 nat) (v1 nat))
    (= (add (add v0 (add v1 v1)) v1) (add (add (add v1 v1) v1) v0)))))

(check-sat)
```

The conjecture is a combination of associativity and commutativity of addition of natural numbers for two variables with four occurrences in total.

`list` *Examples.* These examples describe basic properties about lists, such as relating concatenation of lists to the resulting list length. Similarly to `nat`, the category `list` also contains two generated example sets: `concat_<m>var_<n>occ` contains examples as in `add_<m>var_<n>` occurrences, but using list concatenation instead of list addition, while `pref_<m>var_<n>_<o>occ` is defined in the same way as `leq_<m>var_<n>_<o>occ`, but replacing the less-or-equal order with the prefix relation and using list concatenation instead of natural addition.

Inductive `list` example from the set `crafted`

```
(set-logic UFDT)

;- nat and list declaration, as shown at the beginning of this Section
;- add function declaration and axiomatization, as in the example above

(declare-fun app (list list) list)
(assert (forall ((r list) ) (= (app nil r) r)))
(assert (forall ((a nat) (l list) (r list))
    (= (app (cons a l) r) (cons a (app l r)))))
(declare-fun len (list) nat)
(assert                              (= (len nil       ) zero        ))
(assert (forall ((e nat) (l list)) (= (len (cons e l)) (s (len l)))))

(assert (not (forall ((x list) (y list))
    (= (add (len x) (len y)) (len (app x y))))))

(check-sat)
```

The conjecture asserts that addition of lengths of two lists is equal to the length of the two lists concatenated.

`tree` *Examples.* This category has two main subcategories: one problem set relates binary trees indirectly by flattening them to lists, the other relates them directly to each other. The defined functions are two in-order flattening variants, two functions that recursively rotate a tree completely to the left and to the right at its root, one counting the number of non-leaf nodes in a tree and one checking if two trees are mirror images of each other. Occurrences of the flattenning and rotating functions are varied to get variants for each problem.

Inductive `tree` example from the set `flatten0_rotate_5var`

```
(set-logic UFDT)

;- data types declaration, as shown at the beginning of this Section
;- app function declaration and axiomatization, as in the example above

(declare-fun flat0 (tree) list)
(assert (= (flat0 Nil) nil))
(assert (forall ((p tree) (x nat) (q tree))
    (= (flat0 (node p x q)) (app (flat0 p) (cons x (flat0 q))))))

(assert (not (forall ((p tree) (q tree) (r tree) (x nat) (y nat))
    (= (flat0 (node (node p x q) y r)) (flat0 (node p x (node q y r))))
)))

(check-sat)
```

The conjecture asserts that the result of a tree flattening does not depend on the rotation in the root.

3.2 `int` - Benchmarks with Integers

The `int` category of our benchmark set contains 120 problems for inductive reasoning with integers. It is inspired by software verification problems [7] for three programs: `power`, computing powers of integers, `sum`, computing sums of integer intervals, and `val`, using integers as array indices to encode array properties. These benchmarks were used for evaluating the work from [7]. A sample problem from `power` expressing that the recursively defined power function on integers for positive exponents is distributive over multiplication, is:

Inductive `int` example from the set `power`

```
(set-logic UFNIA)

(declare-fun pow (Int Int) Int)
(assert (forall ((x Int)) (= (pow x 1) x)))
(assert (forall ((x Int) (e Int))
    (=> (<= 2 e) (= (pow x e) (* x (pow x (- e 1)))))))

(assert (not (forall ((x Int) (y Int) (e Int))
    (=> (<= 1 e) (= (pow (* x y) e) (* (pow x e) (pow y e)))))))

(check-sat)
```

The conjecture asserts that for positive exponents, the power function distributes over multiplication of integers.

All variations of the `int` benchmarks were created by varying the constraints and constants in the definitions and goals. For example, variations of the sample problem above use the function `pow` defined starting from 0 instead of 1, or introduce additional constraints on variables `x`, `y` and `e` in the goal.

4 Conclusions

We describe our benchmark set for evaluating inductive capabilities of automated reasoners. Although we primarily provide our problems in the standard SMT-LIB syntax, we also translated them to other input formats of state-of-the-art reasoners.

Future work includes extending our benchmark set with further examples coming from application domains of security and safety verification, as well as formalization of mathematics. Another task for future work is a possible integration of our dataset with the TIP benchmark set or with the SMT-LIB repository. One possibility for incorporating our benchmark set into SMT-LIB would be to add a new subset or an annotation for inductive problems in SMT-LIB, since SMT-LIB does not currently distinguish benchmarks focused on induction from those which can be easily solved without induction. Another possibility is to introduce subsets of the DT (data types) set from SMT-LIB for each notable algebraic data type (natural numbers, lists, trees).

Acknowledgements. This work has been partially funded by the ERC CoG ARTIST 101002685, the ERC StG 2014 SYMCAR 639270, the EPSRC grant EP/P03408X/1 and the Austrian FWF research project LogiCS W1255-N23.

References

1. Barrett, C., et al.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
2. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org (2016)
3. Boyer, R.S., Moore, J.S.: A Computational Logic Handbook, Perspectives in computing, vol. 23. Academic Press (1979)
4. 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.) CICM 2015. LNCS (LNAI), vol. 9150, pp. 333–337. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-20615-8_23
5. Cruanes, S.: Superposition with structural induction. In: Proceedings of FRoCoS, pp. 172–188 (2017)
6. Hajdú, M., Hozzová, P., Kovács, L., Schoisswohl, J., Voronkov, A.: Induction with generalization in superposition reasoning. In: Benzmüller, C., Miller, B. (eds.) CICM 2020. LNCS (LNAI), vol. 12236, pp. 123–137. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53518-6_8
7. Hozzová, P., Kovács, L., Voronkov, A.: Integer induction in saturation. EasyChair Preprint no. 5176 (EasyChair, 2021)
8. Kovács, L., Voronkov, A.: First-order theorem proving and vampire. In: Proceedings of CAV, pp. 1–35 (2013)
9. Kroening, D., Tautschnig, M.: CBMC – C bounded model checker. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 389–391. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_26
10. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
11. Passmore, G., et al.: The imandra automated reasoning system (system description). In: Peltier, N., Sofronie-Stokkermans, V. (eds.) IJCAR 2020. LNCS (LNAI), vol. 12167, pp. 464–471. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51054-1_30
12. Reger, G., Voronkov, A.: Induction in saturation-based proof search. In: Proceedings of CADE, pp. 477–494 (2019)
13. Reynolds, A., Kuncak, V.: Induction for SMT solvers. In: Proceedings of VMCAI, pp. 80–98 (2015)
14. Schneidewind, C., Grishchenko, I., Scherer, M., Maffei, M.: ethor: Practical and provably sound static analysis of ethereum smart contracts, pp. 621–640 (2020). https://doi.org/10.1145/3372297.3417250, https://doi.org/10.1145/3372297.3417250
15. Sonnex, W., Drossopoulou, S., Eisenbach, S.: Zeno: an automated prover for properties of recursive data structures. In: Proceedings of TACAS, pp. 407–421 (2012)