TIP: Tools for Inductive Provers

Dan Rosén and Nicholas Smallbone

Department of Computer Science and Engineering, Chalmers University of Technology
{danr,nicsma}@chalmers.se

Abstract. TIP is a toolbox for users and developers of inductive provers. It consists of a large number of tools which can, for example, simplify an inductive problem, monomorphise it or find counterexamples to it. We are using TIP to help maintain a set of benchmarks for inductive theorem provers, where its main job is to encode aspects of the problem that are not natively supported by the respective provers. TIP makes it easier to write inductive provers, by supplying necessary tools such as lemma discovery which prover authors can simply import into their own prover.

1 Introduction

More and more people are making inductive theorem provers. Besides traditional systems such as ACL2 [14], new provers such as Zeno [18], HipSpec [10], Hipster [13], Pirate [19] and Graphsc [12] have appeared, and some formerly non-inductive provers such as CVC4 [16] and Dafny [15] can now do induction.

To make it easier to scientifically compare these provers, we recently compiled a benchmark suite of 343 inductive problems [11]. We ran into a problem: all of the provers are very different. Some expect the problem to be monomorphic, some expect it to be first-order, some expect it to be expressed as a functional program rather than a logic formula. If we stuck only to features supported by all the provers, we would have very little to work with.

Instead, we designed a rich language which can express a wide variety of problems. The TIP format (short for *Tons of Inductive Problems*) is an extension of SMT-LIB [2] and includes inductive datatypes, built-in integers, higher-order functions, polymorphism, recursive function definitions and first-order logic.

The TIP tools In this paper, we demonstrate a set of tools for transforming and processing inductive problems. The tools are based around the TIP format that we used for our benchmark suite, and provide a wide variety of operations that are useful to users and developers of inductive provers. The tools can currently:

– Convert SMT-LIB and Haskell to TIP.
– Convert TIP to SMT-LIB, TPTP TFF, Haskell, WhyML or Isabelle/HOL.
– Remove features from a problem that a prover does not support, such as higher-order functions or polymorphism.
– Instantiate an induction schema: given a conjecture and a set of variables to do induction over, generate verification conditions for proving the conjecture by induction.

– Model check a problem, to falsify conjectures in it.
– Use theory exploration to invent new conjectures about a theory.

We describe the TIP format itself in Section 2, and many of the available transformations in Section 3. TIP improves the ecosystem of inductive provers in two ways:

– *Interoperability between provers.* Almost all existing inductive theorem provers are incompatible. They all use different input syntax but, more importantly, support entirely different sets of features. This makes it difficult to scientifically compare provers.
  TIP provides conversion tools which allow us to write one problem and try it on several provers. The conversion is not just syntactic but uses tools such as defunctionalisation [17] and monomorphisation to mask the differences between provers. We are using TIP to convert our inductive benchmarks to various provers' input formats.
– *Easier to make new provers.* There are many ingredients to a good inductive prover: it must instantiate induction schemas, perform first-order reasoning to discharge the resulting proof obligations, and discover the necessary lemmas to complete the proof. This makes it hard to experiment with new ideas.
  TIP provides many parts of an inductive prover as ready-made components, so that an author who has—say—an idea for a new induction principle can implement just that, leaving the first-order reasoning and lemma discovery to TIP. This is analogous to first-order logic where a tool author might use, for example, an off-the-shelf clausifier instead of writing their own. In Section 4 we demonstrate the versatility of the TIP tools by stitching them together to make a simple inductive prover as a shell script!

We are continually adding more tools and input and output formats to TIP. We are working to make TIP a universal format for induction problems, backed by a powerful toolchain which can be used by prover authors and users alike. We describe our plans for improving TIP further in Section 6. TIP is publicly available and can be downloaded from `https://github.com/tip-org/tools`.

2 The TIP format

The TIP format is a variant of SMT-LIB. The following problem about lists illustrates all of its features. We first declare the polymorphic list datatype `(list a)`, using the widely supported `declare-datatypes` syntax.

```
(declare-datatypes (a) ((list (nil) (cons (head a) (tail (list a))))))
```

We then define the list `map` function by pattern matching. The `par` construct is used to introduce polymorphism. The `match` expression provides pattern matching and is proposed for inclusion in SMT-LIB 2.6. To support partial functions like `head`, a `match` expression may have missing branches, in which case its value is unspecified. The syntax for higher-order functions is a TIP extension and we discuss it below.

```
(define-fun-rec (par (a b)
  (map ((f (=> a b)) (xs (list a))) (list b)
    (match xs
      (case nil (as nil (list b)))
      (case (cons x xs) (cons (@ f x) (map f xs)))))))
```

Finally, we conjecture that mapping the identity function over a list gives the same list back. As in SMT-LIB we assert the negation of the conjecture and ask the prover to derive `false`. Many inductive provers treat the goal specially, so TIP uses the syntax `(assert-not p)`, which is semantically equivalent to `(assert (not p))` but hints that `p` is a conjecture rather than a negated axiom.

```
(assert-not (par (a) (forall ((xs (list a)))
     (= xs (map (lambda ((x a)) x) xs)))))
(check-sat)
```

To summarise, the TIP format consists of:

– SMT-LIB plus `declare-datatypes` (inductive datatypes), `define-funs-rec` (recursive function definitions), `match` (pattern matching) and `par` (polymorphism), which are all standard or proposed extensions to SMT-LIB.
– Our own TIP-specific extensions: higher-order functions, and `assert-not` for marking the conjecture.

Our tools also understand the SMT-LIB theory of integer arithmetic. We intend TIP to be compatible with the standard theories of SMT-LIB.

First-class functions TIP supports higher-order functions, as these often crop up in inductive problems. We chose to make all use of these syntactically explicit: they must be written explicitly as a lambda function and are applied with the operator `@`. There is no implicit partial application. If `succ` is a function from `Int` to `Int`, we cannot write `(map succ xs)`, but instead write `(map (lambda ((x Int)) (succ x)) xs)`. And in the definition of `map`, we use `(@ f x)` to apply `f` to the list element. There is a type `(=> a b)` of first-class functions from `a` to `b`; `lambda` introduces values of this type and `@` eliminates them. This design allows us to keep the bulk of TIP first-order.

3 Transforming and translating TIP

TIP is structured as a large number of independent transformations. This is true even for our file format conversions. When TIP and the target prover have different feature sets, our approach is to keep the problem in TIP as long as possible, running many small transformations to reduce the problem to some fragment of TIP which we can translate directly. For example, many formats do not support pattern matching, so we must translate it to if-then-else, or if the format does not support that either, we can transform each function definition into a series of axioms. This happens as a TIP-to-TIP transformation.

This approach makes TIP quite modular. It is quite easy to add a new converter as most of the hard work is taken care of by existing transformations. Furthermore, many of those transformations are useful in their own right. In this section we illustrate many of the available transformations; we will use as a running example the conversion of the `map` example to SMT-LIB.

Although TIP is a variant of SMT-LIB, the two are quite different. SMT solvers often do not support polymorphism, higher-order functions or pattern-matching so our converter must remove those features. Here is what our tool produces when asked to translate the `map` example to vanilla SMT-LIB. It has monomorphised the problem, used defunctionalisation to eliminate the `lambda` and is using `is-cons/head/tail` instead of pattern matching.

```
(declare-sort sk_a 0)
(declare-sort fun 0)
(declare-datatypes () ((list (nil) (cons (head sk_a) (tail list)))))
(declare-const lam fun)
(declare-fun apply (fun sk_a) sk_a)
(declare-fun map (fun list) list)
(assert (forall ((x sk_a)) (= (apply lam x) x)))
(assert (forall ((f fun) (xs list))
  (= (map f xs)
    (ite (is-cons xs) (cons (apply f (head xs)) (map f (tail xs))) nil))))
(assert (not (forall ((xs list)) (= xs (map lam xs)))))
(check-sat)
```

3.1 Defunctionalisation

To support theorem provers that have no support for first-class functions and lambdas, a TIP problem can be defunctionalised [17]. This replaces all λ-functions in the problem with axiomatised function symbols. Defunctionalisation is sound but incomplete: if the goal existentially quantifies over a function, it may be rendered unprovable. We expect this to be rare for typical inductive problems.

In the example above, defunctionalisation has introduced the new abstract sort `fun` which stands for functions taking one argument. The identity function is replaced by a constant `lam` of sort `fun`. The `@` operator has been replaced by the function `apply`, together with an axiom which states that `(apply lam x)` is `x`.

3.2 Monomorphisation

Many functional programs are naturally expressed using polymorphism. However, most provers do not support polymorphism. Though there has been work on supporting polymorphism natively in FO provers and SMT solvers, in particular Alt-Ergo [5], and also initial work for CVC4, it is not yet standard practice. Thus, we provide a monomorphisation transformation that removes polymorphic definitions by cloning them at different ground types.

As calculating the required instances is undecidable [4], our monomorphiser is heuristic. It generates a set of rules, in the form of first-order Horn clauses, which say when we should generate various instances. The minimal model of these Horn clauses then tells us which instances are required. The reason we use rules is that it makes it easy to adjust the behaviour of the monomorphiser: different settings may include or omit instantiation rules.

For a function definition, the principle is that when we instantiate the function, we should also instantiate everything required by the function. For `map`, some of the rules will be:

```
map(a,b) -> cons(a)
map(a,b) -> cons(b)
map(a,b) -> map(a,b)
```

In the snippet above, `a` and `b` are the type arguments to `map`. The first two lines make sure that when we instantiate `map` at types `a` and `b` in the program, we will also instantiate the `cons` constructor at `a` and at `b`. For data types, we have other rules that make sure that if `cons` is needed at some type, we also instantiate `list` at that type. We also generate rules for lemmas.

The last line is present because `map` calls itself. In general, when `f` calls `g`, we add a rule that when we instantiate `f` we must instantiate `g`. The rule makes no difference for this example, but is problematic for *polymorphically recursive functions*, which call themselves at a larger type. This is an obstacle for monomorphisation as the set of instances is infinite. A similar problem can occur when instantiating lemmas. To curb this, our procedure gives up after a predefined number of steps.

To start the procedure, we first Skolemise any type variables in the conjecture, and then add facts to the rule set for the functions called in the conjecture. These seed the procedure, which will either return with a set of ground instances that cover the problem, or give up. The transformation succeeds on all but one of our benchmarks; the failing one has a polymorphically recursive data type.

3.3 Eliminating pattern matching

TIP provides two passes for eliminating pattern matching. The first one is used in the translated `map` function above, and replaces `match` with if-then-else (`ite`), discriminators (`is-nil` and `is-cons`) and projection functions (`head` and `tail`). For converting SMT-LIB to TIP format, we also provide a reverse transformation which replaces discriminators and projection functions with `match` expressions.

For some theorem provers, using if-then-else is not an option. We can also translate a function definition using `match` into several axioms, one for each case. Using this transformation, the `map` function turns into the following two axioms, which specify its behaviour on `nil` and `cons`:

```
(assert (forall ((f fun)) (= (map f nil) nil)))
(assert (forall ((f fun) (x sk_a) (xs list))
  (= (map f (cons x xs)) (cons (apply f x) (map f xs)))))
```

The transformation works by first lifting all `match` expressions to the outermost level of the function definition. A function with an outermost `match` can easily be split into several axioms.

3.4 Applying structural induction

We also supply a transformation that applies structural induction to the conjecture. It requires the conjecture to start with a ∀-quantifier, and does induction on the variables quantified there. It splits the problem into several new problems, one for each proof obligation. When using the command line tool, the problems are put in separate files in a directory specified as a command line argument.

The transformation can do induction on several variables and induction of arbitrary depth, depending on what the user chooses. There is some choice about how strong the induction hypothesis should be: we copy HipSpec in assuming the induction hypothesis for all strict syntactic subterms of the conclusion. For example, if `p` is a binary predicate on natural numbers, proving `(p x y)` by induction on `x` and `y` gives the following proof obligation (among others):

```
(assert-not (forall ((x nat) (y nat))
  (=> (p x y) (p x (succ y)) (p (succ x) y)
      (p (succ x) (succ y)))))
```

This works well in practice: it can for instance prove commutativity of the normal natural number plus without lemmas by doing induction on both variables.

3.5 Other transformations and external tools

Minor transformations TIP also includes simplification passes including inlining, dead code elimination and merging equivalent functions. Another transformation partially axiomatises inductive data types for provers and formats that lack built-in support for them, such as TPTP TFF. This is useful for sending proof obligations to a first-order prover after applying an induction schema.

Theory exploration TIP is integrated with the theory exploration system QuickSpec [9]. QuickSpec only accepts Haskell input, so TIP is used to translate the problem to Haskell, and QuickSpec's conjectures are translated back into TIP formulas. This allows theorem provers to easily use theory exploration.

Counterexamples to non-theorems TIP properties can also be randomly tested with QuickCheck [8], via the Haskell converter. Furthermore, the Haskell Bounded Model Checker, HBMC, can read TIP files. These tools can be useful to identify non-theorems among conjectures.

4 Rudimentocrates, a simple inductive prover

Rudimentocrates[1] is a rudimentary inductive theorem prover, using the E theorem prover for first-order reasoning and QuickSpec for lemma discovery. It is a rough caricature of HipSpec, but while HipSpec is 6000 lines of Haskell code, Rudimentocrates is a 100-line shell script built on top of TIP.

The source code of Rudimentocrates is found in appendix A, and an example run in appendix B. It works as follows:

[1] Named after the lesser-known Ancient Greek philosopher.

– Run QuickSpec to find conjectures about the input problem.
– Pick a conjecture, and a variable in that conjecture.
  - Generate proof obligations for proving the conjecture by induction.
  - Translate each obligation to TPTP and send it to E (with a timeout).
  - If all obligations are proved, add the conjecture as an axiom to the problem for use in proving further conjectures.
– Repeat this process until no more conjectures can be proved.

The result is the input problem, but with each proved conjecture (taken either from the input problem or QuickSpec) added as an extra axiom.

Each of the steps—discovering conjectures, generating proof obligations, and translating them to TPTP—is performed by calling TIP. Rudimentocrates is not intended as a serious inductive theorem prover, but it demonstrates how easy it is to experiment with new inductive tools with the help of TIP.

5 Related work

The system most obviously connected to ours is Why3 [6]. Like us, they have a language for expressing problems and a set of transformations and prover backends. The main difference is that Why3 emphasises imperative program verification with pre- and postconditions. There is a functional language like TIP inside Why3 but it it mostly used to write the specifications themselves. By contrast, TIP is specialised to induction and recursive function definitions. This smaller domain allows us to provide more powerful tools, such as theory exploration, random testing and model checking, which would be difficult in a larger language. Another difference is that Why3 manages the entire proof pipeline, taking in a problem and sending it to provers. We intend TIP as a modular collection of tools which can be combined however the user wishes. Nonetheless, on the inside the systems have some similarities and we expect there to be fruitful exchange of ideas between them.

6 Future work and discussion

We are experimenting with heuristics for monomorphisation. A particular problem is what to do when the set of instances is infinite. One possibility is to limit the depth of instantiations by using fuel arguments [1], guaranteeing termination and predictability. Function definitions that could not be instantiated because of insufficient fuel would be turned into uninterpreted functions.

Monomorphisation is inherently incomplete. A complete alternative is to encode polymorphic types [3]. These encodings introduce overhead that slows down the provers, but we would like to add them as an alternative. We would also like to extend our monomorphiser so that it can specialise specialise higher-order functions, generating all their first-order instances that are used in the problem [7]. This would be a low-overhead alternative to defunctionalisation.

We want to add more, stronger, kinds of induction, including recursion-induction and well-founded induction. We would also like to extend the format by adding inductive predicates, as well as coinduction.

Inductive theorem proving has seen a new lease of life recently and we believe it has more potential for growth. With TIP we hope to encourage that growth by fostering competition between provers and providing tools.

References

[1] Nada Amin, K. Rustan M. Leino, and Tiark Rompf. "Computing with an SMT Solver". In: *Tests and Proofs*. 2014.

[2] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The SMT-LIB Standard – Version 2.5*. `http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.5-r2015-06-28.pdf`.

[3] Jasmin Christian Blanchette et al. "Encoding monomorphic and polymorphic types". In: *TACAS*. Springer, 2013.

[4] François Bobot and Andrei Paskevich. "Expressing Polymorphic Types in a Many-Sorted Language". In: *FROCOS*. 2011.

[5] François Bobot et al. "Implementing Polymorphism in SMT Solvers". In: *SMT Workshop*. 2008.

[6] François Bobot et al. "Why3: Shepherd Your Herd of Provers". In: *Boogie 2011*. Aug. 2011.

[7] Wei-Ngan Chin and John Darlington. "A higher-order removal method". In: *LISP and Symbolic Computation* 9.4 (1996).

[8] Koen Claessen and John Hughes. "QuickCheck: a lightweight tool for random testing of Haskell programs". In: *ICFP*. 2000.

[9] Koen Claessen, Nicholas Smallbone, and John Hughes. "QuickSpec: Guessing Formal Specifications Using Testing". In: *Tests and Proofs*. 2010.

[10] Koen Claessen et al. "Automating Inductive Proofs using Theory Exploration". In: *CADE*. Springer, 2013.

[11] Koen Claessen et al. "TIP: Tons of Inductive Problems". In: *Conference on Intelligent Computer Mathematics*. 2015.

[12] S.A. Grechanik. "Proving properties of functional programs by equality saturation". In: *Programming and Computer Software* 41.3 (2015).

[13] Moa Johansson et al. "Hipster: Integrating Theory Exploration in a Proof Assistant". In: *Conference on Intelligent Computer Mathematics*. 2014.

[14] Matt Kaufmann, Manolios Panagiotis, and J Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[15] K. Rustan Leino. "Automating Induction with an SMT Solver". In: *VMCAI*. 2012.

[16] Andrew Reynolds and Viktor Kuncak. "Induction for SMT Solvers". In: *VMCAI*. 2015.

[17] John C. Reynolds. "Definitional Interpreters for Higher-order Programming Languages". In: *the ACM Annual Conference - Volume 2*. 1972.

[18] Willam Sonnex, Sophia Drossopoulou, and Susan Eisenbach. "Zeno: An automated prover for properties of recursive datatypes". In: *TACAS*. 2012.

[19] Daniel Wand and Christoph Weidenbach. "Automatic Induction inside Superposition". `https://people.mpi-inf.mpg.de/~dwand/datasup/draft.pdf`.

A Rudimentocrates source code

```bash
#!/bin/bash

# Run the input file through QuickSpec.
# Discovered lemmas get added as new goals.
echo Dreaming up conjectures...
file=$(tip-spec $1)

# Sends a TIP problem to E, without doing induction.
# Reads the problem from stdin, succeeds if E proves the goal.
# Expects one argument, which is the timeout (in seconds).
e() {
  # Convert the problem to TFF.
  tip --skolemise-conjecture --tff |
  # Use Monotonox to convert the TFF to FOF.
  jukebox fof /dev/stdin 2>/dev/null |
  # Send the FOF problem to E.
  eprover --tstp-in --soft-cpu-limit=$1 --auto --silent >& /dev/null
}

# Sends a whole directory of TIP problems to E, without doing induction.
# Expects two parameters, the directory and the timeout.
dir_e() {
  for i in $1/*; do
    if ! e $2 < $i; then
      return 1
    fi
  done
}

# Attempts to prove a TIP problem by induction with the help of E.
# Uses the TIP tool's induction pass to generate proof obligations,
# and the dir_e command from above to prove them.
# Reads the problem from stdin and expects the timeout as an argument.
inductive_e() {
  local file="$(cat)"
  local n=0  # The position of the variable to do induction on.

  while true; do
    dir=$(mktemp -d)
    # Use TIP to generate proof obligations for doing induction
    # on the nth variable. Fails if n is out of bounds.
    if echo "$file"|tip --induction "[$n]" - $dir >& /dev/null; then
      # Use E to discharge the proof obligations.
      if dir_e $dir $1; then
```

```
        # Success!
        rm -r $dir
        return
      else
        # Failed - try the next variable.
        rm -r $dir
        ((n=$n+1))
      fi
    else
      # Tried all variables unsuccessfully - fail.
      rm -r $dir
      return 1
    fi
  done
}

# Read a problem from stdin and try to prove as many goals as possible.
# Takes a single parameter, which is the timeout to give to E.
prove() {
  file=$(cat)

  progress= # Set to yes if we manage to prove some goal.
  n=0       # The index of the goal we're trying to prove now.

  while true; do
    # Check that n isn't out of bounds.
    if echo "$file"|tip --select-conjecture $n >& /dev/null; then
      # Make a theory where goal n is the only goal.
      goal=$(echo "$file"|tip --select-conjecture $n)
      # Can we prove it without induction?
      if echo "$goal"|e $1; then
        # Proved without induction - delete the goal.
        echo -n ':) ' >&2
        file=$(echo "$file"|tip --delete-conjecture $n)
        progress=yes
      # Can we prove the goal with induction?
      elif echo "$goal"|inductive_e $1; then
        # Proved with induction - change the goal into a lemma.
        echo -n ':D ' >&2
        file=$(echo "$file"|tip --proved-conjecture $n)
        progress=yes
      else
        # Failed to prove the goal - try the next one.
        echo -n ':( ' >&2
        ((n=$n+1))
      fi
    else
      # We've tried all goals - if we failed to prove any,
      # then stop, otherwise go round again.
      echo >&2
      if [ -z $progress ]; then break; fi
      progress=
      n=0
    fi
  done

  # Print out the final theory.
  echo "$file"
}

# Run the proof loop, gradually increasing the timeout.
echo Trying to prove conjectures...
for i in 0 1 5 30; do
  file=$(echo "$file" | prove $i)
done

# Print the final theory out.
echo
echo ";; Final theory"
echo "$file"
```

B Example run of Rudimentocrates

Here is an example showing the output of Rudimentocrates on a simple theory of `append` and `reverse`. The input file has a single conjecture that `reverse (reverse xs) = xs`:

```
(declare-datatypes (a)
  ((list (nil) (cons (head a) (tail (list a))))))
(define-fun-rec (par (a)
  (append ((xs (list a)) (ys (list a))) (list a)
    (match xs
      (case nil ys)
      (case (cons z zs) (cons z (append zs ys)))))))
(define-fun-rec (par (a)
  (reverse ((xs (list a))) (list a)
   (match xs
     (case nil (as nil (list a)))
     (case (cons y ys)
       (append (reverse ys) (cons y (as nil (list a)))))))))
(assert-not (par (a)
  (forall ((xs (list a))) (= (reverse (reverse xs)) xs))))
```

```
(check-sat)
```

Rudimentocrates first runs QuickSpec to discover likely lemmas about `append` and `reverse`:

```
Dreaming up conjectures...
append x Nil = x
append Nil x = x
append (Cons x y) z = Cons x (append y z)
append (append x y) z = append x (append y z)
reverse Nil = Nil
reverse (reverse x) = x
reverse (Cons x Nil) = Cons x Nil
append (reverse y) (reverse x) = reverse (append x y)
```

It then goes into a proof loop, taking one conjecture at a time and trying to prove it. It prints `:(` when it failed to prove a conjecture, `:)` when it proved a conjecture without induction, and `:D` when it proves a conjecture with the help of induction:

```
Trying to prove conjectures...
:( :D :) :) :D :) :( :) :D
:D :)
```

Rudimentocrates prints a newline when it has tried all conjectures, then goes back and retries the failed ones (in case it can now prove them with the help of lemmas). In this case it manages to prove all the discovered conjectures, and prints out the following final theory. Notice that: a) the property `(= xs (reverse (reverse xs)))` is now an axiom (`assert`) rather than a conjecture (`assert-not`), indicating that it has been proved, and b) several other proved lemmas have been added to the theory file.

```
(declare-datatypes (a)
  ((list (nil) (cons (head a) (tail (list a))))))
(define-fun-rec
  (par (a)
    (append
       ((xs (list a)) (ys (list a))) (list a)
       (match xs
         (case nil ys)
         (case (cons z zs) (cons z (append zs ys)))))))
(define-fun-rec
  (par (a)
    (reverse
       ((xs (list a))) (list a)
       (match xs
         (case nil (as nil (list a)))
         (case (cons y ys)
           (append (reverse ys) (cons y (as nil (list a)))))))))
```

```
(assert
  (par (x)
    (forall ((y (list x))) (= (append y (as nil (list x))) y))))
(assert
  (par (x)
    (forall ((y (list x)) (z (list x)) (x2 (list x)))
      (= (append (append y z) x2) (append y (append z x2))))))
(assert
  (par (x)
    (forall ((y (list x)) (z (list x)))
      (= (append (reverse z) (reverse y)) (reverse (append y z))))))
(assert
  (par (a) (forall ((xs (list a))) (= (reverse (reverse xs)) xs))))
(check-sat)
```