

# A Mathematical Benchmark for Inductive Theorem Provers

Thibault Gauthier, Chad E. Brown, Mikoláš Janota, Josef Urban

Czech Technical University in Prague

`email@thibaultgauthier.fr`, `mikolas.janota@gmail.com`, `josef.urban@gmail.com`

## Abstract

We present a benchmark of 29687 problems derived from the On-Line Encyclopedia of Integer Sequences (OEIS). Each problem expresses the equivalence of two syntactically different programs generating the same OEIS sequence. Such programs were conjectured by a learning-guided synthesis system using a language with looping operators. The operators implement recursion, and thus many of the proofs require induction on natural numbers. The benchmark contains problems of varying difficulty from a wide area of mathematical domains. We believe that these characteristics will make it an effective judge for the progress of inductive theorem provers in this domain for years to come.

## 1 Introduction: Induction, OEIS and Related Work

In mathematics, the principle of induction is an essential tool for proving various conjectures. This is especially true if the problem can be expressed as an arithmetical problem. Our goal in this project is to provide a benchmark to test the progress of theorem provers at proving mathematical formulas. Our hope is that in the long run, mathematicians will be able to use these tools to automatically prove non-trivial conjectures.

The 29,687 problems in our benchmark were derived automatically by running a program synthesis algorithm on the OEIS [21]. The OEIS repository archives common (and less common) integer sequences observed in combinatorics, group theory, geometry, etc. Every time two programs  $P$  and  $Q$  generate the same OEIS sequence, we can make the conjecture that  $\forall x \in \mathbb{N}. f_P(x) = f_Q(x)$ . Depending on the number of terms tested, this conjecture is more or less likely to be true. In this work, to minimize the number of false conjectures, only programs that cover all terms of a sequence available in the OEIS repository are considered. Equalities are also tested on additional inputs not present in the OEIS data.

We believe that this benchmark is a good challenge for inductive theorem provers as it is naturally grounded in mathematical theories. Our benchmark contains some easy problems, as shown in our first evaluation, but most of them are out of reach of the current best inductive theorem provers while being quite easy for a university student. We hope that our benchmarks will help developers bridge that gap. Although the problems in our benchmark were derived from data compiled by mathematicians, a large portion of them is also relevant for software verification as mathematical optimizations are common place in computer code. Section 3.3 provides examples of such optimizations.

As a starting point for this research endeavor, we provide a translation of our benchmark to SMT-LIB [2] and evaluation baselines for future comparisons.

There already exist two benchmarks for inductive theorem provers. Compared to our benchmark, both of them are mostly focused on problems related to software verification. The first one is the “Tons of Inductive Problems” benchmark [6] including 340 problems about lists, natural numbers, binary trees, and integers originating from Isabelle, Agda and CLAM translated to SMT-LIB or to WhyML. The second set of problems is “Inductive Benchmarks

for Automated Reasoning” [13]. It consists of 3,516 problems about lists, natural numbers, trees, and integers. These problems were either handcrafted or inspired by software verification problems. To test the limit of inductive problems, multiple versions of the same problem with increased parameters were included. All those problems were translated to SMT-LIB and some of them to formats supported by Zipperposition and ACL2.

## 2 Programming Language

We now present the programming language used in our benchmark. This language contains the same operators as our system for synthesizing programs from integers sequences [11]. There, a simplified presentation of the semantic of the operators was given. In this paper, we present a formal version that matches the definitions given in the SMT problems.

**Syntax** The set  $\mathbb{P}$  of programs in our language is inductively defined to be the smallest set such that  $0, 1, 2, X, Y \in \mathbb{P}$ , and if  $A, B, C, F, G \in \mathbb{P}$  then  $A + B, A - B, A \times B, A \text{ div } B, A \text{ mod } B, \text{cond}(A, B, C), \text{loop}(F, A, B), \text{loop2}(F, G, A, B, C), \text{compr}(F, A) \in \mathbb{P}$ .

In the rest of this paper, we refer to  $\text{loop}$ ,  $\text{loop2}$ , and  $\text{compr}$  as *looping operators*, and we refer to the other operators ( $0, 1, 2, X, Y, +, -, \times, \text{div}, \text{mod}$ , and  $\text{cond}$ ) as *first-order operators*. The first argument of  $\text{loop}$ ,  $\text{compr}$  and the first two arguments of  $\text{loop2}$  are called higher-order arguments (designated by  $F$  and  $G$  in the previous definition). If a variable ( $X$  or  $Y$ ) appears in a higher-order argument it is said to be bounded otherwise it is said to be free. We say that a program  $P$  depends on a variable if this variable appears free in  $P$ . The symbols  $0, 1, 2, +, -, \times, \text{div}, \text{mod}$  are overloaded and may refer to program operators, Standard ML functions or SMT functions depending on the context.

**Semantics** Each program  $P$  is interpreted by a function  $f_P : (x, y) \in \mathbb{Z}^2 \mapsto f_P(x, y) \in \mathbb{Z}$ . The interpretation  $f_P$  is recursively defined for every program  $P$  by:

$$\begin{aligned}
f_0(x, y) &:= 0, & f_1(x, y) &:= 1, & f_X(x, y) &= x, & f_Y(x, y) &= y \\
f_{A+B}(x, y) &:= f_A(x, y) + f_B(x, y), & f_{A-B}(x, y) &:= f_A(x, y) - f_B(x, y) \\
f_{A \times B}(x, y) &:= f_A(x, y) \times f_B(x, y), & f_{A \text{ div } B}(x, y) &:= f_A(x, y) \text{ div } f_B(x, y) \\
f_{A \text{ mod } B}(x, y) &:= f_A(x, y) \text{ mod } f_B(x, y) \\
f_{\text{cond}(A, B, C)}(x, y) &:= \text{if } f_A(x, y) \leq 0 \text{ then } f_B(x, y) \text{ else } f_C(x, y) \\
f_{\text{loop}(F, A, B)}(x, y) &= u(f_A(x, y), f_B(x, y)) \\
&\quad \text{where } u(x, y) = \text{if } x \leq 0 \text{ then } y \text{ else } f_F(u(x-1, y), x) \\
f_{\text{loop2}(F, G, A, B, C)}(x, y) &:= u(f_A(x, y), f_B(x, y), f_C(x, y)) \\
&\quad \text{where } u(x, y, z) = \text{if } x \leq 0 \text{ then } y \text{ else } f_F(u(x-1, y, z), v(x-1, y, z)) \\
&\quad \text{and } v(x, y, z) = \text{if } x \leq 0 \text{ then } z \text{ else } f_G(u(x-1, y, z), v(x-1, y, z)) \\
f_{\text{compr}(F, A)}(x, y) &:= u(f_A(x, y)) \\
&\quad \text{where } t(x) = \text{if } f_F(x, 0) \leq 0 \text{ then } x \text{ else } t(x+1) \\
&\quad \text{and } u(x) = \text{if } x \leq 0 \text{ then } t(0) \text{ else } t(u(x-1) + 1)
\end{aligned}$$

The constants and functions used (outside program indices) in this recursive definition follow the semantics of Standard ML [14]. Note that the functions created from  $\text{mod}$ ,  $\text{div}$ ,  $\text{compr}$  may not be total.

We now give an intuition for the semantics of the looping operators. In this informal description, we do not show the trivial behavior of the following auxiliary sequences on negative indices. The operator *loop* is constructing a recursive sequence  $u_n$  and returns the value  $u_{f_A(x,y)}$ . This sequence is defined by:

$$\begin{aligned} u_0 &= f_B(x, y) \\ u_n &= f_F(u_{n-1}, n) \end{aligned}$$

The operator *loop2* is constructing two mutually recursive sequences  $u_n$  and  $v_n$ . It returns the value  $u_{f_A(x,y)}$ . These sequences are defined by:

$$\begin{aligned} u_0 &= f_B(x, y) \\ v_0 &= f_C(x, y) \\ u_n &= f_F(u_{n-1}, v_{n-1}) \\ v_n &= f_G(u_{n-1}, v_{n-1}) \end{aligned}$$

The operator *compr* constructs a sequence  $u_n$  and returns the value  $u_{f_A(x,y)}$ . The sequence  $u_n$  returns the  $(n + 1)^{th}$  smallest non-negative integer  $x$  satisfying  $f_F(x, 0) \leq 0$ . The auxiliary function  $t(x)$  searches for the next number  $y \geq x$  satisfying  $f_F(y, 0) \leq 0$ .

The presence of the *compr* looping operator is enough to make the programming language Turing complete. The other two looping operators are included because they can express common constructs more efficiently and more concisely.

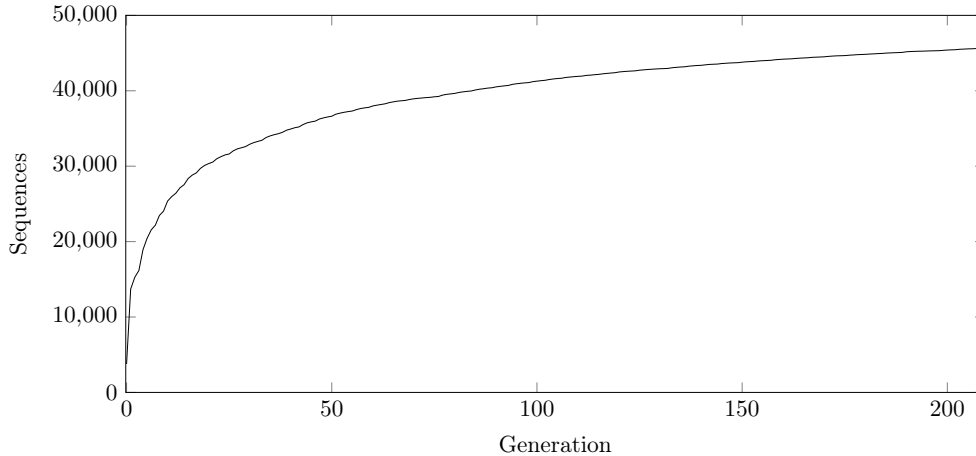
**Execution** The number of abstract time units consumed by the execution of a program is an estimate proportional to the number of CPU instructions needed for each operator. It is 5 for *mod* and *div*. It is 1 for all other first-order operators. In case the absolute value of the integer returned by the operator is bigger than  $2^{64}$ , the number of digits in this integer is used as an estimate. When generating a sequence, the timeout for the current call is increased by adding the unused time from the previous calls on smaller indices of that sequence. This gives extra time to generate longer sequences. On top of this, the execution stops and fails when a number with absolute value greater than  $10^{285}$  is produced. Two different abstract time limits per call are used in this paper: 100 000 during the self-learning experiment (Section 3.1) and during the cyclicity checks (Section 3.4) and 1 000 000 during equality verification (Section 3.2).

**Properties** The **size** of a program is measured by counting with repetition the number of operators composing it. The **speed** of a program is measured by the total number of abstract time units used when generating a sequence. If two programs have the same size (respectively speed) a fixed total order is used to determine which one is the smallest (respectively fastest).

**Definition 1** (Cover). *We say that a program covers (or is a solution for) an OEIS sequence  $(s_x)_{0 \leq x \leq n}$  if and only if  $\forall x \in \mathbb{Z}. 0 \leq x \leq n \Rightarrow f_P(x, 0) = s_x$ .*

### 3 Benchmark

Our benchmark consists of problems of the form  $\forall x \in \mathbb{N}. f_{Small}(x) = f_{Fast}(x)$  where  $f_{Small}$  and  $f_{Fast}$  are functions created from a small program *Small* and a fast program *Fast*. Given an OEIS sequence  $s$ , we choose the smallest (respectively fastest) program discovered by our self-learning system for *Small* (respectively *Fast*). During the checking phase of the self-learning

Figure 1: OEIS sequences covered after  $x$ -th generation

loop, we only test and select a program if it does not depend on the variable  $Y$  (its higher-order arguments may depend on the variable  $Y$ ). This way, we are able to express  $f_{Small}$  and  $f_{Fast}$  as unary functions. An explanation of how these functions are defined in our SMT problems is given in Section 4.

### 3.1 Short Overview of the Self-Learning System

The programs present in the benchmark were discovered through self-learning. The system gradually discovers on its own programs for OEIS sequences. The self-learning loop was run for 209 generations instead of 25 generations in [11]. At each generation, we recorded the smallest and fastest programs discovered so far for each OEIS sequence (instead of only the smallest as in [11]). Each generation consists of a synthesis phase, a checking phase and a learning phase. During the synthesis phase, programs are created using a probability distribution on operators given a target sequence returned by the learning phase. At generation 0, a random probability distribution is used. During the checking phase, we check that the programs created cover the target sequence or any other OEIS sequence. During the learning phase, we train a tree neural network [9] to predict, given a target sequence  $s$ , the smallest and the fastest programs discovered so far generating  $s$ . This process repeats finding solutions for more and more OEIS sequences as depicted in Figure 1.

### 3.2 Problems in the Benchmark

At the end of the self-learning loop, a program covering an OEIS sequence is found for 45691 sequences. This number is reduced to 34171 when looking for sequences where the smallest program  $Small$  and the fastest program  $Fast$  are syntactically different. From each pair of programs, we construct the problem  $\forall x \in \mathbb{N}. f_{Small}(x) = f_{Fast}(x)$ . After regrouping the sequences that generate the same exact equations (this may happen because some OEIS sequences are prefixes of other OEIS sequences), we get 32124 unique problems. A typical OEIS sequence may contain anywhere from a few terms to about 200 terms with the most common number of terms being between 20 and 50. For OEIS sequences with less than 100 terms, we further check that both programs are indeed equal on the first 100 non-negative integers with a

given time limit. If, when computing, one of the programs exceeds the execution limits, the equality is still considered potentially correct if all equality checks pass before an error is raised. Indeed, some interesting equalities occur between long-running programs or fast-increasing programs. After this last check, our dataset is reduced to 29687 problems. These problems, after translation to SMT-LIB, constitute the released benchmark available at <https://github.com/ai4reason/oeis-atp-benchmark> and [grid01.ciirc.cvut.cz/~thibault/oeis-smt.tar.gz](https://github.com/grid01.ciirc.cvut.cz/~thibault/oeis-smt.tar.gz). The code for running the self-learning loop and translating the problems to SMT-LIB is available at <https://github.com/barakeel/oeis-synthesis>. After running the following commands in an interactive HOL4 session will produce the SMT benchmark from the discovered solutions stored in the file `model/itsol1209` in the subdirectory `oeis-smt`:

```
load "smt"; smt.export_smt2 true "oeis-smt" "model/itsol1209";
```

5435 problems (included in the benchmark) could not be computationally checked on the first 100 terms because of the execution limits. These 5435 problems are listed in the file `all_nonverified100` for further analysis.

### 3.3 Examples

Here are a few examples of the benchmark problems derived from famous (and less famous) sequences in the OEIS. For each of those problems, we first give the OEIS sequence number and its description and interpret the meaning of the derived equality between the two programs. In this list, the equality  $Small = Fast$  is used as a shorthand for the conjecture  $\forall x \in \mathbb{N}. f_{Small}(x) = f_{Fast}(x)$

- A217, triangular numbers:

$$loop(X + Y, X, 0) = ((X \times X) + X) \text{ div } 2$$

In this example, a loop is used to compute the sum of the first  $n$  non-negative integers, thus the equation can be rewritten in mathematical form as  $\sum_{i=0}^n i = \frac{n \times n + n}{2}$ .

- A537, sum of first  $n$  cubes:

$$loop((Y \times Y) \times Y + X, X, 0) = loop(X \times X, 1, ((X \times X) + X) \text{ div } 2)$$

The loop on the right-hand side has a bound of 1. So it simply applies the squaring function  $X \times X$  once to the initial value of the loop. In mathematical notation, this conjecture can be expressed as  $\sum_{i=0}^n i^3 = (\frac{n \times n + n}{2})^2$

- A45, Fibonacci numbers:

$$loop2(X + Y, X, X, 0, 1) = cond(X, 0, loop2(X + Y, X, X - 2, 1, 1))$$

On the left-hand side is the expected definition for Fibonacci numbers. On the right-hand side the fast program seems to be saving some computation by starting the loop two steps later with higher initial values. Due to the similarity between the loops in the two programs, this problem may be proven without induction by unrolling the loop twice. However, a proof using induction might be easier to find.

- A79, powers of 2:

$$loop(X + X, X, 1) = loop(X + X, X \text{ mod } 2, loop(X \times X, 1, loop(X + X, X \text{ div } 2, 1)))$$

Two bounded loops are used to compose functions in the right-hand side of this equation. There, the result of  $loop(X + X, X \text{ div } 2, 1)$  is squared and then multiplied  $x \text{ mod } 2$  times

by 2. This conjecture can thus be rewritten as  $2^x = 2^{(x \bmod 2)} \times (2^{(x \operatorname{div} 2)})^2$ . The proof will likely require inductive reasoning to prove the lemma  $2^x \times 2^y = 2^{x+y}$ . The fast program uses the first step of the fast exponentiation algorithm to speed up the computation.

- A165, double factorial of even numbers,  $(2n)!! = 2^n \times n!$ :

$$\text{loop}(2 \times (X \times Y), X, 1) = \text{loop}(X + X, X, 1) \times \text{loop}(X \times Y, X, 1)$$

The double factorial of  $2n$  is by definition  $(2n)!! = \prod_{k=1}^n 2k$ . Thus coincidentally, this equation gives an implementation on each side of the equation of the two formulas given in the OEIS. A proof of this statement is expected to require inductive reasoning.

### 3.4 Problems Requiring Induction

One motivation<sup>1</sup> for this benchmark is to test provers on mathematical problems requiring one or multiple inductions. Since our programs consist of looping constructs, we believe that is the case for the majority of the problems in our benchmark. However, some of the problems can be solved without induction. An easily recognizable case is when the programs *Small* and *Fast* do not contain any loop or when all their loops are bounded by a constant. Such problems may occur in our dataset. Therefore, in the following, we design *syntactic and semantic tests* to detect if an equality contains at least one “proper” top-level loop.

Given an equality  $\forall x \in \mathbb{N}. f_{\text{Small}}(x) = f_{\text{Fast}}(x)$ , we select a looping subprogram (a subprogram whose root operator is a looping operator) in *Small* and *Fast* if it appears at a position that is not under another looping subprogram and if its exact formulation does not appear more than once in the equational problem. We will say that a problem *passes a test* if there exists at least one such top-level loop which satisfies this test. From the syntactic and semantic tests, we carve out two subsets of the released benchmark. Problems that pass all syntactic tests are listed in the file `aind_syn` and problems that pass all syntactic and semantic tests are listed in the file `aind_sem`. In general, it is a hard problem to determine if a problem will require induction a priori, and the following tests are trying to achieve a trade-off between ruling out problems that do not require induction and keeping problems that do.

**Syntactic Tests** The following syntactic tests are performed on the looping subprograms of the form  $\text{loop}(F_1, A_1, B_1)$ ,  $\text{loop}2(F_2, G_2, A_2, B_2, C_2)$ ,  $\text{compr}(F_3, A_3)$ :

- The bounds  $A_1, A_2, A_3$  and the subprogram  $F_1$  must depend on  $X$ .
- Either  $F_2$  or  $G_2$  must depend on  $X$  and  $Y$ .

**Semantic Tests** The semantic tests will try to detect cases where a prover does not require induction even though the problem passes the syntactic tests. For instance, top-level looping subprograms may use  $X \bmod 3$  or  $2 - X$  as a bound or some behavior in  $F_1, F_2$  or  $G_2$  may result in a proof that unrolls the loop a finite amount of time. Thus, instead of testing for syntactic dependency on a variable, we will run the subprograms and test for acyclicity in their output.

**Definition 2** (Acyclicity of an integer sequence – tailored to our setting). *We say that a finite sequence of integers  $a_0, a_1, \dots, a_{39}$  is acyclic if and only if the sequence  $a_9, \dots, a_{39}$  does not contain a cycle with a period ranging from 1 to 15.*

<sup>1</sup>There are multiple motivations for this benchmark, some of them being very pragmatic. Our OEIS program synthesis systems produce thousands of more and more complex programs that may look quite *alien* [10]. It may take a nontrivial amount of time to decide if such programs are correct and human mathematicians do not scale to the number of such problems we are currently generating.

**Definition 3** (Acyclicity of a program). *A program  $P$  is acyclic on  $x$  if and only if:  $\forall y \in \mathbb{Z}. 0 \leq y \leq 9 \Rightarrow (f_P(x, y))_{0 \leq x \leq 39}$  is acyclic.*  
*A program  $P$  is acyclic on  $y$  if and only if:  $\forall x \in \mathbb{Z}. 0 \leq x \leq 9 \Rightarrow (f_P(x, y))_{0 \leq y \leq 39}$  is acyclic.*  
*In practice, we chose to make the test fail if one of the sequence cannot be produced because of the execution limits. In such situations, the program  $P$  will not be considered acyclic.*

The following semantic tests are performed on the top-level looping subprograms of the form  $loop(F_1, A_1, B_1), loop2(F_2, G_2, A_2, B_2, C_2), compr(F_3, A_3)$ :

- The bounds  $A_1, A_2, A_3$  and the subprogram  $F_1$  must be acyclic on  $x$ .

When checking for cyclicity in the bounds, the negative program outputs of  $A_1, A_2, A_3$  are mapped to 0 before checking for cycles.

- Either  $F_2$  or  $G_2$  must be acyclic on  $x$  and acyclic on  $y$ .
- The looping subprogram itself must be acyclic on  $x$ .

## 4 Translation to SMT-LIB

We now translate the 29687 problems in our benchmark (see Section 3) to SMT-LIB. These SMT problems consist of definitions for  $f_{Small}$  and  $f_{Fast}$ , and the negated conjecture:

$$\exists c. c \geq 0 \wedge \neg(f_{Small}(c) = f_{Fast}(c))$$

The semantic definitions given in Section 2 recursively define  $f_{Small}$  and  $f_{Fast}$ . In these definitions, the Standard ML functions are replaced by their SMT counterparts. This process creates a new definition for each subprogram. In order to simplify the SMT problem we expand definitions for first-order operators. We also minimize the number of arguments of each function. Indeed, if a program  $P$  does not depend on  $Y$  (respectively  $X$ ,  $X$  and  $Y$ ), we can define a function  $f_P$  with one argument such that  $f_P(x) := f_P(x, y)$  (respectively  $f_P(y) := f_P(x, y)$ ,  $f_P() := f_P(x, y)$ ). We illustrate the process of creating SMT definitions on the program  $Small = loop(X + Y, X, 0)$ :

$$\begin{aligned} f_1() &= 1 \\ f_X(x) &= x \\ f_{X+Y}(x, y) &= x + y \\ u(x, y) &= \text{if } x \leq 0 \text{ then } y \text{ else } f_{X+Y}(u(x - 1, y), x) \\ f_{loop(X+Y, X, 1)}(x) &= u(f_X(x), f_1()) \end{aligned}$$

The third equation is derived by expanding definitions of first-order operators until a looping subprogram is reached. In our example, we have  $f_{X+Y}(x, y) = f_X(x, y) + f_Y(x, y) = x + y$ . In a more general example where a looping subprogram  $Q$  appears under the first-order part, we would for instance get  $f_{2 \times X + Q}(x, y) = 2 \times x + f_Q(x, y)$ . The in-lining of expanded definitions inside definitions for loops is left to the provers. In the released SMT problems, program indices in the definitions are replaced by integer indices.

**Totality of the Functions** Some of our Standard ML functions may initially be partial because of the operators  $div, mod, compr$ . These are translated to total SMT functions. Therefore, a proof of the equality between two functions  $f_{Small}, f_{Fast}$  with respective initial domain  $D_{Small}, D_{Fast}$  is only a proof that they are equal on  $D_{Fast} \cap D_{Small}$ . Functions from our problems are expected to be total. This is especially the case if we consider only those that fully passed the verification test on the first 100 non-negative integers. A few of the programs in our benchmark may not terminate on large inputs, in which case the derived conjectures as expressed by our translation will not be provable.



**Towards More General Conjectures** In our benchmark, all of the conjectures are of the form  $\forall x. x \geq 0 \Rightarrow f_P(x) = f_Q(x)$ . In particular, we cannot express conjectures that quantify over  $P$  or  $Q$ . To solve this issue, one can use an evaluation function  $eval(P, x, y) := f_P(x, y)$  that makes  $P$  a proper argument instead of an index. This alternative translation requires to declare the syntactic constructs as SMT functions on a SMT sort for programs and replace in the defining equations (semantics) for each operator every instance of  $f_P(x, y)$  by  $eval(P, x, y)$  to create the SMT axioms. After this transformation, we can, for example, express the conjecture of finding an increasing function as  $\exists P. eval(P, x, 0) \leq eval(P, x + 1, 0)$ . We did not use this *eval* encoding in our benchmark since we do not need it for our current conjectures, and also because we observed that it makes the problems more difficult for the provers.

## 5 Experiments

In the following experiments, we test the performance of three state-of-the-art provers **Vampire** [17], **CVC5** [18, 1] and **Z3** [7] on our benchmark. **Vampire** is run with an induction schedule [15, 13] suggested by its developers; **CVC5** is run with its induction flag on [20]. The addition of support for arithmetical induction is recent in these two provers. The prover **Z3** has not yet been given such support and therefore can only solve problems that do not require induction. All provers are run on all the problems with a timeout of 60 seconds for each problem.

Table 1 shows the results of running **Z3**, **Vampire** and **CVC5** on the benchmark. For **CVC5** we also show the results after strengthening the conjecture to include equality of additional terms. Noting  $f = f_{Small}$  and  $g = f_{Fast}$ , the conjecture  $\forall x. x \geq 0 \Rightarrow (f(x) = g(x))$  becomes:

- $\forall x. x \geq 0 \Rightarrow (f(x) = g(x) \wedge f(x + 1) = g(x + 1))$  in the column *C1*,
- $\forall x. x \geq 0 \Rightarrow (f(x) = g(x) \wedge f(x + 1) = g(x + 1) \wedge f(x + 2) = g(x + 2))$  in the column *C2*,
- ...
- and  $\forall x. x \geq 0 \Rightarrow (f(2x) = g(2x) \wedge f(2x + 1) = g(2x + 1))$  in the column *C2x*.

Although these conjectures are equivalent to the original conjecture, they have a significant influence on **CVC5**. By changing the shape of the conjectures, **CVC5** will use different instances of the induction schema due to how induction is implemented in **CVC5**. These changes do not benefit **Z3** because the problems are strictly harder without induction. The reasons why **Vampire** does not benefit from the modified conjectures remain to be investigated. We have also tried to change the conjecture to strong induction (equality on all previous numbers). However, it has practically no effect on **CVC5**. For each method, we also show the results on the 23163 problems that pass the syntactic filtering (Section 3.4), results on the 16197 problems that pass the semantic filtering (Section 3.4), and results on the 5435 problems where the extended verification (testing on 100 terms - Section 3.2) fails.

The fact that **Z3** solves only 7 problems after the strongest semantic filtering suggests that induction is very likely to be needed on the 16197 semantically filtered problems. In total, we can prove 2686 of those problems, which is 16.58%. An interesting result is the 212 problems (3.90% of the 5435) where we can prove equality of the functions, but our extended equality verification (testing on 100 terms) procedure fails on them. Most often this means that these are fast-growing functions where the normal computation of the numerical values overflows on larger inputs. While 3.90% is not much, it demonstrates a real value added by automated reasoning compared to just running extended testing. The joint performance of all systems on the full benchmark is 26.67% (8215 out of 29687) and the performance on the syntactically filtered problems is 16.16% (3743 out of 23163). **Z3** is the best system on the full benchmark,



System	Z	V	C	C1	C2	C3	C4	C5	C6	C8	C2x	All
NoFilt	4757	2195	2428	3793	4030	4100	4084	3962	3796	3451	3557	8215
SynFilt	487	278	893	2258	2547	2701	2699	2590	2447	2156	2015	3743
SemFilt	7	83	504	1799	2059	2235	2240	2146	2021	1786	1501	2686
NonCh	2	97	21	22	48	33	76	39	29	8	5	212

Table 1: Problems solved by each of the methods. NoFilt means no filtering (results on the whole benchmark), SynFilt are results on the 23163 problems that pass the syntactic filtering, SemFilt are results on the 16197 problems that pass the semantic filtering, and NonCh are results on the 5435 problems where the extended checking fails.

while CVC5 performs best on the filtered problems where induction is likely often needed. CVC5 is also quite orthogonal to Z3 and Vampire, adding many solutions to both.

From the five examples presented in Section 3.3, the problem derived from the triangular numbers is solved by CVC5. Strengthening the conjecture (to the successor - method C1) can also solve the problem induced by the Fibonacci numbers. Vampire is able to solve the problem about double factorials and Z3 cannot solve any of them. Vampire can also solve the triangular numbers problem when using its default induction schedule instead of the suggested one.

## 6 Conclusion

In this work, we relied on a self-learning system to create small and fast programs for each OEIS sequence. Subsequently, we created a benchmark of 29,687 SMT problems, asserting that the two programs produce identical sequences. We then asked automated theorem provers and SMT solvers to solve these problems and analyzed the results. Ultimately, we discovered a simple method to enhance the performance of CVC5 on inductive problems. The benchmark is publicly available, and we also provide bounties on 1000 unsolved problems that can be claimed by anybody in a decentralized way (Appendix B).

In the future, we aim to use our benchmark to reveal the limitations of inductive theorem provers available in proof assistants and explore ways to improve them. This will enable us to assess the impact of various techniques developed over the years, such as term synthesis [5], rippling [16], template-based conjecturing [19], and also more recent methods such as neural synthesis [8] and conjecturing [22]. To achieve this, we will need to translate the problems into the specific format required by each of these inductive theorem provers. Last but not least, we aim to explore the achieved results in the context of program equivalence [12].

## 7 Acknowledgments

This work was partially supported by the European Regional Development Fund under the Czech project AI&Reasoning no. CZ.02.1.01/0.0/0.0/15\_003/0000466 (CB, JU), CTU Global Postdoc funding scheme (TG), Czech Science Foundation project 20-06390Y (TG), Amazon Research Awards (TG, JU), the Czech MEYS under the ERC CZ project *POSTMAN* no. LL1902 (CB, TG, MJ), and the EU ICT-48 2020 project TAILOR no. 952215 (JU).

## References

- [1] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. CVC5: A versatile and industrial-strength SMT solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022.
- [2] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. The SMT-LIB standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, Scotland)*, volume 13, page 14, 2010.
- [3] Chad E. Brown, Cezary Kaliszyk, Thibault Gauthier, and Josef Urban. Proofgold: Blockchain for formal methods. In *FMBC@CAV*, volume 105 of *OASiCs*, pages 4:1–4:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.
- [4] Chad E. Brown and Karol Pak. A tale of two set theories. In *CICM*, volume 11617 of *Lecture Notes in Computer Science*, pages 44–60. Springer, 2019.
- [5] Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. Automating inductive proofs using theory exploration. In Maria Paola Bonacina, editor, *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, volume 7898 of *Lecture Notes in Computer Science*, pages 392–406. Springer, 2013.
- [6] Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. TIP: tons of inductive problems. In Manfred Kerber, Jacques Carette, Cezary Kaliszyk, Florian Rabe, and Volker Sorge, editors, *Intelligent Computer Mathematics - International Conference, CICM 2015, Washington, DC, USA, July 13-17, 2015, Proceedings*, volume 9150 of *Lecture Notes in Computer Science*, pages 333–337. Springer, 2015.
- [7] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.
- [8] Thibault Gauthier. Deep reinforcement learning for synthesizing functions in higher-order logic. In *LPAR*, volume 73 of *EPiC Series in Computing*, pages 230–248. EasyChair, 2020.
- [9] Thibault Gauthier. Tree neural networks in HOL4. In Christoph Benzmüller and Bruce R. Miller, editors, *Intelligent Computer Mathematics - 13th International Conference, CICM 2020, Bertinoro, Italy, July 26-31, 2020, Proceedings*, volume 12236 of *Lecture Notes in Computer Science*, pages 278–283. Springer, 2020.
- [10] Thibault Gauthier, Miroslav Olsák, and Josef Urban. Alien coding. *CoRR*, abs/2301.11479, 2023.
- [11] Thibault Gauthier and Josef Urban. Learning program synthesis for integer sequences from scratch. *CoRR*, abs/2202.11908, 2022.
- [12] Benny Godlin and Ofer Strichman. Regression verification: proving the equivalence of similar programs. *Softw. Test. Verification Reliab.*, 23(3):241–258, 2013.
- [13] Márton Hajdú, Petra Hozzová, Laura Kovács, Johannes Schoisswohl, and Andrei Voronkov. Inductive benchmarks for automated reasoning. In Fairouz Kamareddine and Claudio Sacerdoti Coen, editors, *Intelligent Computer Mathematics - 14th International Conference, CICM 2021, Timisoara, Romania, July 26-31, 2021, Proceedings*, volume 12833 of *Lecture Notes in Computer Science*, pages 124–129. Springer, 2021.
- [14] Robert Harper, David MacQueen, and Robin Milner. *Standard ML*. Department of Computer

- Science, University of Edinburgh, 1986.
- [15] Petra Hozzová, Laura Kovács, and Andrei Voronkov. Integer induction in saturation. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*, volume 12699 of *Lecture Notes in Computer Science*, pages 361–377. Springer, 2021.
  - [16] Moa Johansson, Lucas Dixon, and Alan Bundy. Dynamic rippling, middle-out reasoning and lemma discovery. In Simon Sieglér and Nathan Wasser, editors, *Verification, Induction, Termination Analysis - Festschrift for Christoph Walther on the Occasion of His 60th Birthday*, volume 6463 of *Lecture Notes in Computer Science*, pages 102–116. Springer, 2010.
  - [17] Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 1–35. Springer, 2013.
  - [18] Gereon Kremer, Andrew Reynolds, Clark W. Barrett, and Cesare Tinelli. Cooperating techniques for solving nonlinear real arithmetic in the CVC5 SMT solver (system description). In Jasmin Blanchette, Laura Kovács, and Dirk Pattinson, editors, *Automated Reasoning - 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8-10, 2022, Proceedings*, volume 13385 of *Lecture Notes in Computer Science*, pages 95–105. Springer, 2022.
  - [19] Yutaka Nagashima, Zijin Xu, Ningli Wang, Daniel Sebastian Goc, and James Bang. Property-based conjecturing for automated induction in Isabelle/HOL. *CoRR*, abs/2212.11151, 2022.
  - [20] Andrew Reynolds and Viktor Kuncak. Induction for SMT solvers. In Deepak D’Souza, Akash Lal, and Kim Guldstrand Larsen, editors, *Verification, Model Checking, and Abstract Interpretation - 16th International Conference, VMCAI 2015, Mumbai, India, January 12-14, 2015. Proceedings*, volume 8931 of *Lecture Notes in Computer Science*, pages 80–98. Springer, 2015.
  - [21] Neil J. A. Sloane. The On-Line Encyclopedia of Integer Sequences. In Manuel Kauers, Manfred Kerber, Robert Miner, and Wolfgang Windsteiger, editors, *Towards Mechanized Mathematical Assistants, 14th Symposium, Calculemus 2007, 6th International Conference, MKM 2007, Hagenberg, Austria, June 27-30, 2007, Proceedings*, volume 4573 of *Lecture Notes in Computer Science*, page 130. Springer, 2007.
  - [22] Josef Urban and Jan Jakubuv. First neural conjecturing datasets and experiments. In *CICM*, volume 12236 of *Lecture Notes in Computer Science*, pages 315–323. Springer, 2020.

## A Further Examples of Programs and their Encoding

Here we show the SMT encoding of the problem created from A000165 – double factorial of even numbers<sup>2</sup>. The numbers grow too fast, making their evaluation-based equality checking fail. Vampire proves the problem and uses induction in its proof.

```
;; sequence(s): A165
;; terms: 1 2 8 48 384 3840 46080 645120 10321920 185794560 3715891200
;; 81749606400 1961990553600 51011754393600 1428329123020800 42849873690624000
;; 1371195958099968000 46620662575398912000 1678343852714360832000
;; 63777066403145711616000
;; small program: loop(2 * (x * y), x, 1)
;; fast program: loop(x + x, x, 1) * loop(x * y, x, 1)
(assert (forall ((x Int) (y Int)) (= (f0 x y) (* 2 (* x y))))
(assert (forall ((x Int)) (= (g0 x) x)))
(assert (= h0 1))
(assert (forall ((x Int) (y Int)) (= (u0 x y) (ite (<= x 0) y (f0 (u0 (- x 1) y) x)))))
```

<sup>2</sup>[oeis.org/A000165](https://oeis.org/A000165)

```
(assert (forall ((x Int)) (= (v0 x) (u0 (g0 x) h0))))
(assert (forall ((x Int)) (= (small x) (v0 x))))
(assert (forall ((x Int)) (= (f1 x) (+ x x))))
(assert (forall ((x Int)) (= (g1 x) x)))
(assert (= h1 1))
(assert (forall ((x Int) (y Int)) (= (u1 x y) (ite (<= x 0) y (f1 (u1 (- x 1) y))))))
(assert (forall ((x Int)) (= (v1 x) (u1 (g1 x) h1))))
(assert (forall ((x Int) (y Int)) (= (f2 x y) (* x y))))
(assert (forall ((x Int)) (= (g2 x) x)))
(assert (= h2 1))
(assert (forall ((x Int) (y Int)) (= (u2 x y) (ite (<= x 0) y (f2 (u2 (- x 1) y) x))))
(assert (forall ((x Int)) (= (v2 x) (u2 (g2 x) h2))))
(assert (forall ((x Int)) (= (fast x) (* (v1 x) (v2 x))))
(assert (exists ((c Int)) (and (>= c 0) (not (= (small c) (fast c))))))
```

Next, we show the SMT encoding of the problem created from A45 – the Fibonacci sequence<sup>3</sup>. CVC5 proves the problem after strengthening the conjecture to the successor as follows:

```
;; sequence(s): A45-A77373
;; terms: 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
;; small program: loop2(x + y, x, x, 0, 1)
;; fast program: if x <= 0 then 0 else loop2(x + y, x, x - 2, 1, 1)
(assert (forall ((x Int) (y Int)) (= (f0 x y) (+ x y))))
(assert (forall ((x Int)) (= (g0 x) x)))
(assert (forall ((x Int)) (= (h0 x) x)))
(assert (= i0 0))
(assert (= j0 1))
(assert (forall ((x Int) (y Int) (z Int)) (= (u0 x y z)
      (ite (<= x 0) y (f0 (u0 (- x 1) y z) (v0 (- x 1) y z))))))
(assert (forall ((x Int) (y Int) (z Int)) (= (v0 x y z)
      (ite (<= x 0) z (g0 (u0 (- x 1) y z))))))
(assert (forall ((x Int)) (= (w0 x) (u0 (h0 x) i0 j0))))
(assert (forall ((x Int)) (= (small x) (w0 x))))
(assert (forall ((x Int) (y Int)) (= (f1 x y) (+ x y))))
(assert (forall ((x Int)) (= (g1 x) x)))
(assert (forall ((x Int)) (= (h1 x) (- x 2))))
(assert (= i1 1))
(assert (= j1 1))
(assert (forall ((x Int) (y Int) (z Int)) (= (u1 x y z)
      (ite (<= x 0) y (f1 (u1 (- x 1) y z) (v1 (- x 1) y z))))))
(assert (forall ((x Int) (y Int) (z Int)) (= (v1 x y z)
      (ite (<= x 0) z (g1 (u1 (- x 1) y z))))))
(assert (forall ((x Int)) (= (w1 x) (u1 (h1 x) i1 j1))))
(assert (forall ((x Int)) (= (fast x) (ite (<= x 0) 0 (w1 x))))
(assert (exists ((c Int)) (and (>= c 0) (or (not (= (small (+ c 1)) (fast (+ c 1))))
      (not (= (small c) (fast c))))))
```

Finally, we show the encoding of A180713<sup>4</sup> proved by CVC5 only after changing the conjecture to equality on  $2x$  and  $2x + 1$ :

```
;; sequence(s): A180713
```

---

<sup>3</sup>[oeis.org/A45](https://oeis.org/A45)

<sup>4</sup>[oeis.org/A180713](https://oeis.org/A180713)

```

;; terms: 0 4 6 11 12 16 18 23 24 28 30 35 36 40 42 47 48 52 54 59
;; small program: ((((((x div 2) * x) mod 2) + (x mod 2)) + x) + x) + x
;; fast program: (loop(loop(1, 2 - (x mod (2 + 2)), 2) + x, x mod 2, x) + x) + x
(assert (forall ((x Int)) (= (small x)
    (+ (+ (+ (+ (mod (* (div x 2) x) 2) (mod x 2)) x) x) x))))
(assert (= f1 1))
(assert (forall ((x Int)) (= (g1 x) (- 2 (mod x (+ 2 2)))))
(assert (= h1 2))
(assert (forall ((x Int) (y Int)) (= (u1 x y) (ite (<= x 0) y f1))))
(assert (forall ((x Int)) (= (v1 x) (u1 (g1 x) h1))))
(assert (forall ((x Int)) (= (f0 x) (+ (v1 x) x))))
(assert (forall ((x Int)) (= (g0 x) (mod x 2))))
(assert (forall ((x Int)) (= (h0 x) x)))
(assert (forall ((x Int) (y Int)) (= (u0 x y) (ite (<= x 0) y (f0 (u0 (- x 1) y)))))
(assert (forall ((x Int)) (= (v0 x) (u0 (g0 x) (h0 x)))))
(assert (forall ((x Int)) (= (fast x) (+ (+ (v0 x) x) x))))
(assert (exists ((c Int)) (and (>= c 0) (or (not (= (small (* c 2)) (fast (* c 2))))
(not (= (small (* 2 (+ c 1)) (fast (* 2 (+ c 1)))))))))

```

## B Bounties on 1000 Examples

We selected 1000 of the unsolved problems from the benchmark and translated the SMT2 representation into a set theoretic representation used by the Megalodon<sup>5</sup> system [4]. We then placed a bounty of 1 Proofgold<sup>6</sup> [3] bar on each of the corresponding formal propositions. As a consequence anyone can claim 1 Proofgold bar by giving an appropriate proof term for the corresponding problem.<sup>7</sup>

<sup>5</sup><https://github.com/ai4reason/Megalodon>

<sup>6</sup><http://proofgold.net/>

<sup>7</sup>The details are available at <http://grid01.ciiirc.cvut.cz/~chad/oeis-mgpg1000.tgz>.