arXiv:2304.02986v1 [cs.LO] 6 Apr 2023

归纳定理证明器数学基准

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

布拉格捷克理工大学 email@thibaultgauthier.fr、mikolas.janota@gmail.com、josef.urban@gmail.com

摘要

我们介绍一个包含 29687 个问题的基准测试，这些问题源自在线整数序列百科全书（OEIS）。每个问题都表达了两个语法上不同、但能生成同一 OEIS 序列的程序之间的等价性。这类程序由一个采用循环运算符的学习引导型合成系统猜想得出。这些运算符实现了递归，因此许多证明需要对自然数进行归纳。该基准测试包含来自广泛数学领域、难度各异的问题。我们认为，这些特点将使其在未来数年成为评估该领域归纳定理证明器进展的有效标准。

1 引言：归纳、OEIS 与相关工作

在数学中，归纳原理是证明各种猜想的重要工具。如果问题可以表述为算术问题，这一点尤其如此。本项目的目标是提供一个基准测试，用于测试定理证明器证明数学公式的进展。我们希望从长远来看，数学家能够使用这些工具自动证明非平凡的猜想。

我们基准测试中的29,687 个问题，是通过在OEIS [18]上运行程序综合算法自动生成的。OEIS存储库收录了组合学、群论、几何学等领域中观察到的常见（以及不太常见的）整数序列。每当两个程序 P 和 Q 生成相同的 OEIS 序列时，我们就可以提出猜想 $\forall x \in \mathbb{N}.\ f_P(x) = f_Q(x)$。根据所测试的项数，这一猜想为真的可能性或高或低。在这项工作中，为了最大限度地减少错误猜想的数量，只考虑覆盖 OEIS 存储库中某个序列全部项的程序。等式还会在 OEIS 数据中未出现的其他输入上进行测试。

我们相信，这一基准测试对归纳定理证明器而言是一个很好的挑战，因为它自然地建立在数学理论之上。正如我们的首次评估所示，我们的基准测试包含一些简单的问题，但其中大多数超出了当前最佳归纳定理证明器的能力范围，而对大学生来说却相当容易。我们希望这些基准测试能够帮助开发者弥合这一差距。作为这项研究工作的起点，我们提供了该基准测试到 SMT-LIB 的转换结果 [2] 以及用于未来比较的评估基线。

目前已经存在两个面向归纳定理证明器的基准测试。与我们的基准测试相比，它们大多集中于与软件验证相关的问题。第一个是“众多归纳问题”基准测试[4] ，其中包括340个关于列表、自然数、二叉树和整数的问题，这些问题源自Isabelle、Agda和CLAM，并被翻译为SMT-LIB或WhyML。第二组问题是“用于自动推理的归纳基准测试” [10]。它包含3,516个关于列表、自然数、树和整数的问题。这些问题要么是手工编写的，要么受到软件验证问题的启发。为了测试归纳问题的极限，其中纳入了同一问题的多个版本，并逐步增加参数。所有这些问题都被翻译为SMT-LIB，其中一些还被翻译为Zipperposition和ACL2支持的格式。

2 编程语言

现在，我们介绍基准测试中使用的编程语言。该语言包含与我们的系统相同的运算符，用于从整数序列合成程序 [8]。在前文中，我们对这些运算符的语义进行了简化介绍。在本文中，我们给出一个与 SMT 问题中定义相匹配的形式化版本。

语法 我们语言中的程序集合 $\mathbb{P}$ 通过归纳定义为满足以下条件的最小集合：0、1、2、X、$Y \in \mathbb{P}$，以及如果 A、B、C、F、$G \in \mathbb{P}$，则 $A + B$、$A - B$、$A \times B$、A div B、A mod B、cond(A、B、C)、loop(F、A、B)、loop2 (F、G、A、B、C)、compr (F、A) $\in \mathbb{P}$。

在本文其余部分中，我们将 loop、loop2 和 compr 称为循环算子，并将其他算子（0、1、2、X、Y、$+$、$-$、$\times$、div、mod 和 cond）称为一阶算子。loop 和 compr 的第一个参数以及 loop2 的前两个参数称为高阶参数（在前面的定义中分别由 F 和 G 表示）。如果一个变量（X 或 Y）出现在高阶参数中，则称其为有界变量；否则称其为自由变量。如果某个变量出现在 P中且为自由变量，我们称程序 P 在语法上依赖于该变量。符号 0、1、2、$+$、$-$、$\times$、div、mod 具有重载含义，具体取决于上下文，它们可能指程序算子、Standard ML 函数或 SMT 函数。

语义每个程序 P都由一个函数 f_P进行解释：$(x, y) \in \mathbb{Z}^2 \mapsto f_P(x, y) \in \mathbb{Z}$。解释 f_P 针对每个程序 P 递归定义如下：

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

此递归定义中使用的常量和函数（程序索引之外）遵循 Standard ML [11]的语义。请注意，由 mod、div、compr 创建的函数可能不是全函数。

现在，我们来直观说明循环算子的语义。在这段非正式描述中，我们不展示以下辅助序列在负索引处的平凡行为。循环算子会构造一个递归序列 u_n，并返回值 $u_{f_A(x,\ y)}$。该序列定义如下：

$$u_0 = f_B(x,y), u_n = f_F(u_{n-1}, n)$$

loop2 算子会构造两个相互递归序列 u_n 和 v_n。它返回值 $u_{f_A(x,\ y)}$。这些序列定义如下：

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

compr 算子构造序列 u_n，并返回值 $u_{f_A(x,y)}$。序列u_n返回满足 $f_F(x, 0) \leq 0$ 的 $(n+1)^{th}$ 最小非负整数 x。辅助函数 $t(x)$ 搜索满足 $f_F(y, 0) \leq 0$ 的下一个数 $y \geq x$。

执行 执行程序时，在自学习实验（第 3.1 节）和循环性检查（第 3.4 节）期间，我们将每次调用的步数限制为 100,000 个抽象时间单位。等式验证（第 3.2节）期间，每次调用的时间限制会增加到 1,000,000个抽象时间单位。程序执行所消耗的抽象时间单位数量，是根据每个算子所需的 CPU 指令数量按比例得出的估计值。对于 mod 和 div，该值为5。对于所有其他一阶算子，该值为 1。如果算子返回的整数的绝对值大于 2^{64},，则使用该整数的位数作为估计值。生成序列时，会将前几次调用中未使用的时间加到当前调用的超时限制中。此外，当生成绝对值大于 10^{285} 的数时，执行会停止并失败。

属性程序的**大小**通过重复计数构成程序的运算符数量来衡量。程序的**速度**通过生成一个序列时所用的抽象时间单位总数来衡量。如果两个程序的大小（或速度）相同，则使用固定的全序来确定哪个程序最小（或最快）。

定义 1 (覆盖)。 当且仅当 $\forall x \in \mathbb{Z}.\ 0 \leq x \leq n \Rightarrow f_P(x, 0) = s_x$ 时，我们称一个程序**覆盖** （或**是……的一个解**）一个 *OEIS* 序列$(s_x)_{0 \leq x \leq n}$。

3 基准测试

我们的基准测试由以下形式的问题组成 $\forall x \in \mathbb{N}.\ f_{Small}(x) = f_{Fast}(x)$ 其中 f_{Small} 和f_{Fast} 是由小型程序 小型 和快速程序快速创建的函数。在自学习循环的检查阶段，我们仅当程序不依赖于 Y（其高阶参数可能依赖于 Y）时才测试并选择该程序。这样，我们能够将 f_{Small}和 f_{Fast} 表示为一元函数。第 4 节介绍了这些函数在我们的 SMT 问题中是如何定义的。

3.1 自学习系统简述

基准测试中包含的程序是通过自学习发现的。系统逐步自主发现 OEIS 序列的程序。自学习循环运行了 209 代，而非 [8]中的 25 代。在每一代，我们都记录了迄今为止发现的针对每个 OEIS 序列的最小且最快的程序（而 [8]中只记录最小的程序）。每一代由综合阶段、检查阶段和学习阶段组成。在综合阶段，根据学习阶段返回的目标序列，按照运算符上的概率分布创建程序。在第 0 代，使用随机概率分布。在检查阶段，我们检查创建的程序是否覆盖目标序列或任何其他 OEIS 序列。在学习阶段，我们训练一个树形神经网络

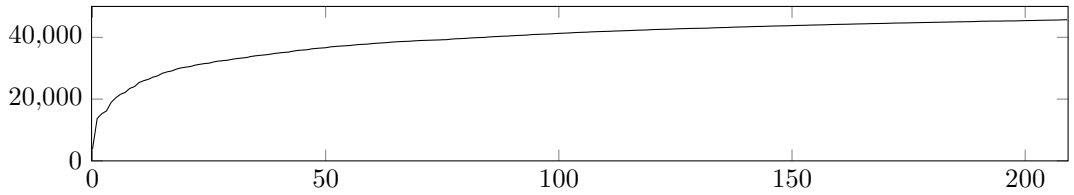


图 1：第 x 代后覆盖的 OEIS 序列

网络 [6] 根据给定的目标序列，预测迄今发现的生成该序列的最小和最快程序。随着针对越来越多 OEIS 序列找到解，这一过程不断重复，如图1所示。

3.2 基准测试中的问题

自学习循环结束时，对于 45691 个 OEIS 序列，我们找到了覆盖这些序列的程序。如果要求最小程序$Small$ 和最快程序 $Fast$ 在语法上不同，则这一数字降至 34171。对于每一对程序，我们构造问题 $\forall x \in \mathbb{N}.\ f_{Small}(x) = f_{Fast}(x)$。对生成完全相同方程的序列重新分组后（这是因为某些 OEIS 序列是其他 OEIS 序列的前缀），我们得到 32124 个独特问题。典型的 OEIS 序列可能包含几个到约 200 个项，其中最常见的项数为 20 到 50。对于少于 100 项的 OEIS 序列，我们进一步检查两个程序在给定时间限制内对前 100 个非负整数的计算结果是否确实相等。如果其中一个程序在计算时超出执行限制，只要在报错前通过了所有相等性检查，我们仍将该等式视为可能正确。事实上，一些有趣的等式出现在运行时间较长的程序或增长速度较快的程序之间。经过最后这项检查后，我们的数据集缩减为 29687 个问题。这些问题经转换为 SMT-LIB 后，构成了已发布的基准测试，可在grid01.ciirc.cvut.cz/~thibault/oeis-smt.tar.gz获取。运行自学习循环并将问题转换为 SMT-LIB 的代码可在 https://github.com/barakeel/oeis-synthesis。

在交互式 HOL4 会话中运行以下命令，将根据发现的解生成 SMT 基准测试，并将其存储在子目录 oeis-smt 的文件 model/itsol209 中：

load "smt"; smt.export_smt2 true "oeis-smt" "model/itsol209";

由于执行限制，基准测试中包含的 5435 个问题无法得到完整验证。未验证的问题列在文件 *all_nonverified100* 中，以供进一步分析。

3.3 示例

以下列举了一些基准问题示例，这些问题源自 OEIS 中著名（或不那么著名）的数列。对于每个问题，我们首先给出 OEIS 数列编号及其描述，并解释由两个程序之间的等式所表达的含义。在此列表中，等式 Small $=$ Fast 用作猜想 $\forall x \in \mathbb{N}.\ f_{Small}(x) = f_{Fast}(x)$ 的简写

- A217，三角数：

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

在此示例中，使用循环计算前 n 个非负整数的和，因此该方程可以改写为数学形式 $\sum_{i=0}^{n} i = \frac{n \times n + n}{2}$。

- A537，前 n 个立方数之和：

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

右侧的循环上限为 1。因此，它只对循环的初始值应用一次平方函数$X \times X$。用数学符号表示，该猜想可以写成 $\sum_{i=0}^{n} i^3 = (\frac{n \times n + n}{2})^2$

- A45，斐波那契数：

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

左侧是斐波那契数的预期定义。右侧的快速程序似乎通过将循环的起始位置推迟两步，并采用更大的初始值，节省了一些计算。由于两个程序中的循环相似，这个问题或许可以通过将循环展开两次来证明，而无需使用归纳法。不过，使用归纳法可能更容易找到证明。

- A79，2 的幂：

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

两个有界循环用于组合该方程右侧的函数。在这里，loop($X + X$, X div 2, 1)) 的结果先平方，然后再乘以 2 的 x mod 2 次方。因此，该猜想可以改写为 $2^x = 2^{(x\ mod\ 2)} \times (2^{(x\ div\ 2)})^2$。证明该引理 $2^x \times 2^y = 2^{x+y}$ 可能需要归纳推理。快速程序使用快速幂算法的第一步来加快计算。

- A165，偶数的双阶乘，$(2n)!! = 2^n \times n!$：

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

$2n$ 的双阶乘按定义为$(2n)!! = \prod_{k=1}^{n} 2k$ 因此巧合的是，这个等式在等式两边分别给出了OEIS中所给出的两个公式的实现。预计证明这一陈述需要使用归纳推理。

3.4 需要归纳的问题

一个动机[1] 是使用此基准测试检验证明器解决需要一次或多次归纳的数学问题的能力。由于我们的程序由循环结构组成，我们认为基准测试中的大多数问题都属于这种情况。然而，其中一些问题无需归纳即可解决。一个容易识别的情况是，当程序 $Small$ 和 $Fast$不包含任何循环，或其所有循环都由常量限定时。此类问题可能出现在我们的数据集中。因此，下面我们设计 句法和语义测试来检测一个等式是否至少包含一个“适当的”顶层循环。

给定等式 $\forall x \in \mathbb{N}.\ f_{Small}(x) = f_{Fast}(x)$，如果循环子程序（其根运算符为循环运算符的子程序）出现在某个位置，我们会在 Small 和 Fast 中选择该循环子程序

[1]这个基准测试有多种动机，其中一些非常务实。我们的 OEIS 程序综合系统会生成成千上万个越来越复杂的程序，这些程序看起来可能相当 陌生 [7]。判断此类程序是否正确可能需要相当多的时间，而人类数学家无法扩展到应对我们目前正在生成的这类问题的数量。

即不隶属于另一个循环子程序，并且其精确表述在方程问题中出现的次数不超过一次。我们将称一个问题 通过一项测试，如果存在至少一个满足该测试的此类顶层循环。根据语法测试和语义测试，我们从发布的基准测试中划分出两个子集。通过所有语法测试的问题列在文件 aind_syn 中，而通过所有语法测试和语义测试的问题列在文件 aind_sem 中。一般而言，事先确定一个问题是否需要归纳是很困难的，以下测试旨在权衡排除不需要归纳的问题与保留需要归纳的问题。

语法测试 对以下形式的循环子程序 loop(F_1, A_1, B_1)、loop2 (F_2, G_2, A_2, B_2, C_2)、compr(F_3, A_3)执行以下语法测试：
- 界限 A_1、A_2、A_3 以及子程序 F_1 必须依赖于 X。
- F_2 或 G_2 必须依赖于 X 和 Y。

语义测试 语义测试将尝试检测这样的情况：问题虽然通过了语法测试，但证明器并不需要使用归纳。例如，顶层循环子程序可能使用 $X \bmod 3$ 或 $2-X$作为界限，或者 F_1、F_2或 G_2 中的某些行为可能导致证明只需有限次展开循环。因此，我们不会测试对变量的语法依赖性，而是运行这些子程序，并测试其输出是否具有无环性。

定义 2 (整数序列的无环性——针对我们的设定). 当且仅当序列 $a_9, \ldots, a_{39}$ 不包含周期为1到15的周期时，我们称有限整数序列 $a_0, a_1, \ldots a_{39}$ 是无环的 。

定义3 （程序的无环性）。程序 P 在 x 上具有无环性，当且仅当： $\forall y \in \mathbb{Z}.\ 0 \leq y \leq 9 \Rightarrow$ （f_P（x， y））$_{0 \leq x \leq 39}$ 具有无环性。
程序 P在 y上是无环的 当且仅当： $\forall x \in \mathbb{Z}.\ 0 \leq x \leq 9 \Rightarrow (f_P(x, y))_{0 \leq y \leq 39}$ 是无环的。实际上，如果由于执行限制而无法生成其中一个序列，我们会让测试失败。在这种情况下，程序 P 将不被视为无环。

对以下形式的顶层循环子程序 loop(F_1, A_1, B_1), loop2 (F_2, G_2, A_2, B_2, C_2), compr(F_3, A_3) 执行以下语义测试：
- 边界 A_1, A_2, A_3 和子程序 F_1 在 x 上必须是无循环的。检查边界的循环性时，会先将所有负的程序输出映射为 0，然后再检查是否存在循环。
- F_2 或 G_2 必须在 x 上无循环，且在 y 上无循环。
- 循环子程序本身在 x 上必须是无循环的。

4 翻译为 SMT-LIB

现在，我们将基准测试中的 29687 个问题（见第 3 节）翻译为 SMT-LIB。这些 SMT 问题包含 f_{Small} 和 f_{Fast} 的定义，以及否定猜想：

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

实例化第 2 节中给出的语义定义后，我们可以递归地为两个顶层程序中的每个子程序创建定义。在这些定义中，Standard ML函数被替换为其 SMT 对应项。此过程会为每个子程序创建一个新定义。为了简化 SMT 问题，我们展开一阶

运算符的定义。我们还会尽量减少每个函数的参数数量。实际上，如果程序P不依赖于 Y（分别是 X、X 和 Y），我们就可以定义一个只有一个参数的函数 f_P，使得 $f_P(x) := f_P(x,y)$（分别是 $f_P(y) := f_P(x,y)$、$f_P() := f_P(x,y)$）。我们以程序 $Small = \text{loop}(X+Y, X, 0)$ 为例，说明创建 SMT 定义的过程：

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

第三行是通过展开一阶算子的定义，直到遇到循环子程序而推导出的。在我们的示例中，我们得到 $f_{X+Y}(x,y) = f_X(x,y) + f_Y(x,y) = x + y$。在一个更一般的示例中，如果一个循环子程序 Q 出现在一阶部分下方，例如我们会得到 $f_{2\times X+Q}(x,y) = 2 \times x + f_Q(x,y)$。循环定义中已展开定义的内联由证明器负责。在发布的 SMT 问题中，定义中的程序索引被替换为整数索引。

函数的全定义性 我们的某些 Standard ML 函数最初可能并非全定义，这是因为使用了 div、mod、compr 运算符。它们会被转换为全定义的 SMT 函数。因此，证明两个函数 f_{Small}、f_{Fast} 相等，且它们的初始定义域分别为 D_{Small}、D_{Fast}，仅能证明它们在 $D_{Fast} \cap D_{Small}$上相等。我们的问题中的函数预期都是全定义的。如果我们只考虑那些在前 100 个非负整数上完全通过验证测试的函数，尤其如此。

面向更一般的猜想 在我们的基准测试中，所有猜想都具有 $\forall x.x \geq 0 \Rightarrow f_P(x) = f_Q(x)$的形式。特别是，我们无法表达对 P或 Q 进行量化的猜想。为了解决这一问题，可以使用求值函数 $\text{eval}(P,x,y) := f_P(x,y)$，使 P 成为一个适当的参数，而不是索引。这种替代性翻译要求将语法构造声明为程序SMTsort上的SMT 函数，并在定义每个运算符的方程（语义）中，将 $f_P(x,y)$ 的每个实例替换为 $eval(P,x,y)$，以创建SMT 公理。经过这一转换后，例如，我们可以将寻找递增函数的猜想表达为 $\exists P.\, \text{eval}(P,x,0) \leq \text{eval}(P,x+1,0)$。我们没有在基准测试中使用这种$eval$ 编码，因为当前的猜想不需要它，而且我们还观察到，它会使证明器处理这些问题更加困难。

5 实验

在以下实验中，我们测试了三种最先进的证明器 Vam-pire [14], CVC5 [15, 1] 和 Z3 [5] 在我们的基准测试上的表现。 Vampire 使用其开发者建议的归纳调度方案运行 [12]； CVC5 启用了归纳选项运行 [17]。这两种证明器最近才加入了对算术归纳的支持。证明器Z3 尚未获得此类支持，因此只能解决不需要归纳的问题。所有证明器均在全部问题上运行，每个问题的超时限制为 60 秒。

表1 展示了运行 Z3、Vampire 和 CVC5 对该基准测试进行测试的结果。对于 CVC5，我们还展示了增强猜想、加入额外项相等条件后的结果。例如，C1 加入后继项相等条件，C2 还加入后继的后继项相等条件，以此类推。C2x 将猜想改为 $2x$ 与 $2x+1$相等。这些额外方法对CVC5有显著影响，但对 Z3 和 Vampire没有影响。我们还尝试将猜想改为强归纳（所有前序数均相等）。然而，实际上它

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

表 1：各方法解决的问题。NoFilt 表示不进行筛选（在整个基准测试上的结果），SynFilt 表示通过语法筛选的 23163 个问题上的结果，SemFilt 表示通过语义筛选的 16197 个问题上的结果，NonVer 表示扩展验证（测试）失败的 5435 个问题上的结果。

对 cvc5没有影响。对于每种方法，我们还展示了通过语法过滤（第 3.4 节）的 23163 个问题的结果、通过语义过滤（第 3.4 节）的 16197 个问题的结果，以及扩展验证（对 100 个项进行测试——第 3.2 节）失败的 5435 个问题的结果。

Z3 在最强语义过滤后仅解决了 7 个问题，这一事实表明，在经过语义过滤的 16197 个问题中，很可能需要使用归纳。在这些问题中，我们总共可以证明 2686 个，占 16.58%。一个有趣的结果是，有 212 个问题（占 5435 个问题的 3.90%）中，我们可以证明函数相等，但扩展等式验证（在 100 个项上进行测试）过程却在这些问题上失败了。这通常意味着这些是增长速度很快的函数，在较大输入上计算数值时，常规计算会发生溢出。虽然 3.90% 并不算高，但这表明，与仅运行扩展测试相比，自动推理确实带来了额外价值。所有系统在完整基准测试上的联合性能为 26.67%（29687 个问题中解决了 8215 个），而在经过语法过滤的问题上的性能为 16.16%（23163 个问题中解决了 3743 个）。Z3 是完整基准测试中表现最佳的系统，而 CVC5 在经过过滤且通常可能需要归纳的问题上表现最佳。CVC5 与 Z3 和 Vampire 也具有相当好的正交性，为两者都增加了许多解答。

在第 3.3 节介绍的五个示例中，由三角数得出的问题可由 cvc5 求解。加强猜想（使用后继方法 C1）也可以求解由斐波那契数诱导的问题。 Vampire 能够求解关于双阶乘的问题，而 Z3 无法求解其中任何一个问题。 Vampire 在使用其默认的归纳调度而非建议的调度时，也能够求解三角数问题。

6 结论

在这项工作中，我们依靠自学习系统为每个OEIS 序列创建了小型且快速的程序。随后，我们创建了一个包含29,687 个 SMT 问题的基准测试，断言这两个程序生成相同的序列。然后，我们要求自动定理证明器和 SMT 求解器解决这些问题，并分析了结果。最终，我们发现了一种简单的方法，可以提升 cvc5 在归纳问题上的性能。

未来，我们计划利用这一基准测试揭示证明助手中现有归纳定理证明器的局限，并探索改进方法。这样，我们便能评估多年来开发的各种技术所产生的影响，例如项综合 [3],波纹 [13], 和基于模板的猜想生成 [16]。为此，我们需要将这些问题转换为各个归纳定理证明器所要求的特定格式。最后但同样重要的是，我们计划在程序等价的背景下探索所取得的结果 [9]。

参考文献

[1] Haniel Barbosa、Clark W. Barrett、Martin Brain、Gereon Kremer、Hanna Lachnitt、Makai Mann、Abdalrhman Mohamed、Mudathir Mohamed、Aina Niemetz、Andres Nötzli、Alex Ozdemir、Math- ias Preiner、Andrew Reynolds、Ying Sheng、Cesare Tinelli 和 Yoni Zohar。cvc5：一种功能多样、达到工业级强度的 SMT 求解器。载于 Dana Fisman 和 Grigore Rosu 编，*工具与算法：系统构造与分析——第 28 届国际会议 TACAS 2022，作为欧洲软件理论与实践联合会议 ETAPS 2022 的一部分举行，德国慕尼黑，2022 年 4 月 2 日至 7 日，论文集，第一部分*，计算机科学讲义第 13243 卷，第 415–442 页。Springer，2022 年。[2] Clark Barrett、Aaron Stump、Cesare Tinelli 等。《SMT-LIB 标准：2.0 版》。载于*第 8 届理论模可满足性国际研讨会论文集（英国爱丁堡）*，第 13 卷，第 14 页，2010 年。[3] Koen Claessen、Moa Johansson、Dan Rosén 和 Nicholas Smallbone。利用理论探索实现归纳证明自动化。载于 Maria Paola Bonacina 编，*自动演绎会议（CADE）*，LNCS 第 7898 卷，第 392–406 页。Springer，2013 年。[4] Koen Claessen、Moa Johansson、Dan Rosén 和 Nicholas Smallbone。TIP：海量归纳问题。载于 Manfred Kerber、Jacques Carette、Cezary Kaliszyk、Florian Rabe 和 Volker Sorge 编，*智能计算数学——CICM 2015 国际会议，2015 年 7 月 13 日至 17 日，美国华盛顿特区，论文集*，计算机科学讲义第 9150 卷，第 333–337 页。Springer，2015 年。[5] Leonardo De Moura 和 Nikolaj Bjørner。Z3：一种高效的 SMT 求解器。载于 C. R. Ramakrishnan 和 Jakob Rehof 编，*系统构造与分析工具和算法会议（TACAS）*，LNCS 第 4963 卷，第 337–340 页。Springer，2008 年。[6] Thibault Gauthier。HOL4 中的树神经网络。载于 Christoph Benzmüller 和 Bruce R. Miller 编，*智能计算数学——第 13 届国际会议 CICM 2020，2020 年 7 月 26 日至 31 日，意大利贝尔蒂诺罗，论文集*，计算机科学讲义第 12236 卷，第 278–283 页。Springer，2020 年。[7] Thibault Gauthier 、 Miroslav Olsák 和 Josef Urban。Alien coding。*CoRR*，abs/2301.11479，2023 年。[8] Thibault Gauthier 和 Josef Urban。从零开始学习整数序列的程序综合，2022 年。[9] Benny Godlin 和 Ofer Strichman。回归验证：证明相似程序的等价性。*软件测试、验证与可靠性*，23(3)：241–258，2013 年。[10] Márton Hajdú、Petra Hozzová、Laura Kovács、Johannes Schoisswohl 和 Andrei Voronkov。用于自动推理的归纳基准测试。载于 Fairouz Kamareddine 和 Claudio Sacerdoti Coen 编，*智能计算数学——第 14 届国际会议 CICM 2021，2021 年 7 月 26 日至 31 日，罗马尼亚蒂米什瓦拉，论文集*，计算机科学讲义第 12833 卷，第 124–129 页。Springer，2021 年。[11] Robert Harper、David MacQueen 和 Robin Milner。*Standard ML*。爱丁堡大学计算机科学系，1986 年。[12] Petra Hozzová、Laura Kovács 和 Andrei Voronkov。饱和中的整数归纳。载于 André Platzer 和 Geoff Sutcliffe 编，*自动演绎——CADE 28——第 28 届自动演绎国际会议，线上举行，2021 年 7 月 12 日至 15 日，论文集*，计算机科学讲义第 12699 卷，第 361–377 页。Springer，2021 年。[13] Moa Johansson、Lucas Dixon 和 Alan Bundy。动态波纹、中间向外推理与引理发现。载于 Simon Siegler 和 Nathan Wasser 编，*验证、归纳与终止性分析——为庆祝 Christoph Walther 60 岁生日而作的纪念文集*，计算机科学讲义第 6463 卷，第 102–116 页。Springer，2010 年。[14] Laura Kovács 和 Andrei Voronkov。一阶定理证明与 Vampire。载于 Natasha Sharygina 和 Helmut Veith 编，*计算机辅助验证——第 25 届国际会议*，

CAV 2013，俄罗斯圣彼得堡，*2013* 年 *7* 月 *13–19* 日。论文集，计算机科学讲义第 8044 卷，第 1–35 页。Springer，2013 年。[15] Gereon Kremer、Andrew Reynolds、Clark W. Barrett 和 Cesare Tinelli。在 cvc5 SMT 求解器中求解非线性实数算术的协同技术（系统描述）。载于 Jasmin Blanchette、Laura Kovács 和 Dirk Pattinson 编，自动推理——第 *11* 届国际联合会议 *IJCAR 2022*，海法，以色列，*2022* 年 *8* 月 *8–10* 日，论文集，计算机科学讲义第 13385 卷，第 95–105 页。Springer，2022 年。[16] Yutaka Nagashima、Zijin Xu、Ningli Wang、Daniel Sebastian Goc 和 James Bang。用于 Isabelle/HOL 自动归纳的基于性质的猜想。*CoRR*，abs/2212.11151，2022 年。[17] Andrew Reynolds 和 Viktor Kuncak。SMT 求解器的归纳。载于 Deepak D'Souza、Akash Lal 和 Kim Guldstrand Larsen 编，验证、模型检查与抽象解释——第 *16* 届国际会议 *VMCAI 2015*，孟买，印度，*2015* 年 *1* 月 *12–14* 日。论文集，计算机科学讲义第 8931 卷，第 80–98 页。Springer，2015 年。[18] Neil J. A. Sloane。《在线整数序列百科全书》。载于 Manuel Kauers、Manfred Kerber、Robert Miner 和 Wolfgang Windsteiger 编，迈向机械化数学助理，第 *14* 届研讨会 *Calculemus 2007*、第 *6* 届国际会议 *MKM 2007*，哈根贝格，奥地利，*2007* 年 *6* 月 *27–30* 日，论文集，计算机科学讲义第 4573 卷，第 130 页。Springer，2007 年。

程序及其编码的更多示例

这里我们展示由 A000165– 偶数的双阶乘创建的问题的 SMT 编码[2]。这些数字增长得太快，导致其基于求值的等式检查失败。Vampire 在其证明中证明了该问题并使用了归纳。

;; sequence(s): A165 ;; terms: 1 2 8 48 384 3840 46080 645120 10321920 185794560 3715891200 ;; 81749606400 1961990553600 51011754393600 1428329123020800 42849873690624000 ;; 1371195958099968000 46620662575398912000 1678343852714360832000 63777066403145711616000 ;; 小型程序：loop(2 * (x * y), x, 1) ;; 快速程序：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))))) (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))))))

[2] oeis.org/A000165

接下来，我们展示由 A45（斐波那契数列）构造的问题的 SMT 编码[3]。CVC5 通过将猜想加强为后继形式，证明了该问题，如下所示：

;; 数列：A45-A77373 ;; 项：0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 ;; 小型程序：循环2(x + y, x, x, 0, 1) ;; 快速程序：如果 x <= 0 则为 0，否则循环2(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))))))) 最后，我们展示了 A180713[4] 的编码，该编码只有在将 $2x$ 和 $2x+1$ 上的猜想改为相等后，才能由 CVC5 证明：

;; 数列：A180713 ;; 项：0 4 6 11 12 16 18 23 24 28 30 35 36 40 42 47 48 52 54 59 ;; 小型程序：((((((x div 2) * x) mod 2) + (xmod 2)) + x) + x) + x ;; 快速程序：(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)))))

[3] oeis.org/A45[4] oeis.org/A180713

```
(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))))) ))))
```