arXiv:2304.10768v1 [cs.PL] 21 Apr 2023

通过迭代式前向-后向抽象解释进行归纳式程序合成

Yongho Yoon，首尔国立大学，韩国Woosuk Lee*，
汉阳大学，韩国Kwangkeun Yi，首尔国立大学，韩国

基于示例的程序合成面临的一项关键挑战是程序的搜索空间极其庞大。为了解决这一挑战，已有多项研究提出利用抽象解释来剪枝搜索空间。然而，现有的大多数方法仅关注前向抽象解释，因此无法充分发挥抽象解释的潜力。本文提出了一种通过迭代式前向-后向抽象解释实现归纳式程序合成的新方法。前向抽象解释根据输入计算程序可能产生的输出，而后向抽象解释则根据输出计算程序可能接受的输入。通过交替迭代执行这两种抽象解释，我们能够有效判断每个部分程序的候选补全是否能够满足输入-输出示例。我们将该方法应用于一种标准形式化框架——语法引导合成（SyGuS），从而支持广泛的归纳式综合任务。我们实现了该方法，并在一组先前研究中的基准测试上对其进行了评估。实验结果表明，得益于精细的抽象解释技术，我们的方法显著优于最先进的方法。

CCS 概念：•**软件及其工程** →**示例编程**；**自动编程**；•**计算理论** →**抽象**；**程序分析**。

附加　　关键词与短语：程序综合、示例编程、抽象解释　　n

ACM 引用格式：
Yongho Yoon、Woosuk Lee 和 Kwangkeun Yi。2023年。通过迭代式前向—后向抽象解释进行归纳式程序合成。*ACM*编程语言会议录 7，PLDI，第174篇文章（2023年6月），25 页。https://doi.org/10.1145/3591288

1 问题与我们的方法

归纳式程序综合旨在综合出一个满足给定输入输出示例集的程序。流行的自顶向下搜索策略会枚举缺少部分的<|style id='1'>部分程序，然后将其补全为完整程序。

尽管这种策略对于合成小型程序很有效，但由于搜索空间呈指数级增长，如果无法快速排除伪候选项，它几乎无法扩展到大型程序。

因此，已有多种技术被提出用于剪枝搜索空间 [Feng 等人 2017；Gulwani 2011年；Lee，2021；Polikarpova 等人 2016年；Wang 等人 2017年a]。特别是，抽象解释 [Cousot（2021）；Rival 和 Yi 2020] 已被广泛用于剪枝搜索空间

*通讯作者

作者地址：Yongho Yoon，yhyoon@ropas.snu.ac.kr，首尔国立大学，计算机科学与工程系，韩国；Woosuk Lee，woosuk@hanyang.ac.kr，汉阳大学，计算机科学与工程系，韩国；Kwangkeun Yi，kwang@ropas.snu.ac.kr，首尔国立大学，计算机科学与工程系，韩国。

允许出于个人或课堂用途免费制作本作品的部分或全部内容的数字副本或纸质副本，但不得为营利或商业利益而制作或传播副本，且副本首页必须载有本声明及完整引文。本作品中第三方组件的版权必须得到尊重。其他用途请联系所有者/作者。© 2023年 版权归所有者/作者所有。
2475-1421/2023/6-ART174https://doi.org/10.1145/3591288

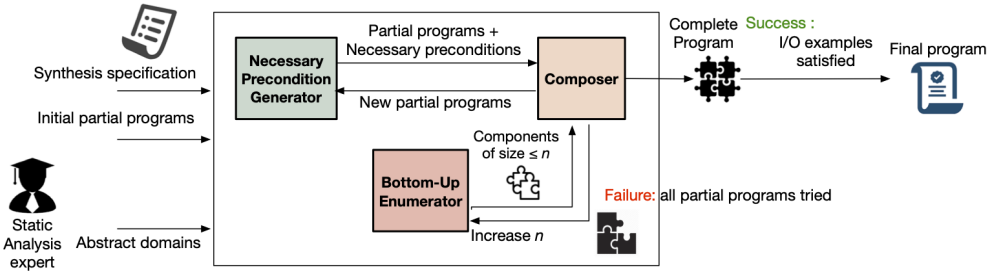


图 1. 我们综合算法的高级架构。

归纳式程序合成 [Feng 等人 2017；Singh 和 Solar-Lezama 2011；So 和 Oh 2017；Wang 等人 2017年a，b]。抽象解释有两种：模拟程序执行的前向抽象解释，以及模拟反向执行的后向抽象解释。

之前的大多数方法都仅基于前向抽象解释。它们使用抽象语义对每个部分程序进行符号执行，并计算所有可从该部分程序推导出的程序的可能输出的可靠上近似。如果上近似的输出不包含期望的输出，就可以安全地丢弃该程序。

然而，仅依靠前向抽象解释是不够的，因为它只能告诉我们部分程序的可行性，却无法说明如何补全它。另一方面，后向抽象解释可用于推导部分程序中缺失部分的必要前置条件。

本文提出了一种新的基于抽象解释的剪枝方法，用于归纳式程序合成，并以迭代方式结合前向推理和后向推理。对于每个含有缺失表达式的部分程序，前向分析根据给定的输入示例，计算程序最终输出及中间操作结果上的（过近似）不变式。基于前向分析的结果和期望输出示例，后向分析计算出必要前置条件，缺失表达式必须满足这些条件，程序才能产生期望输出。这两种分析以协同方式结合，使一种分析的结果能够细化另一种分析的结果，并不断迭代直至收敛。如果任何必要前置条件都无法满足，则丢弃该部分程序，因为它无法产生期望输出。

图 1 展示了我们综合算法的总体架构。该算法受到近期提出的一种综合策略的启发 [Lee 2021]。该算法以综合规范（由输入输出示例和包含缺失部分的初始部分程序组成）作为输入。此外，它还需要一个由领域专家设计、用于刻画目标语言抽象语义的抽象域。我们的算法由三个关键模块组成，即 自底向上枚举器、必要前置条件生成器和组合器：

- 自底向上枚举器：给定输入输出示例和一个数值 n（初始值为 1），自底向上枚举器会生成组件。组件是表达式（大小为≤ n），用于补全部分程序中的缺失部分。自底向上枚举器会在观察等价的意义下，穷举生成大小不超过该界限的组件。

- 必要前置条件生成器：给定一个包含缺失部分的部分程序，必要前置条件生成器会计算缺失表达式必须满足的必要前置条件，以确保程序满足规范。为此，它

迭代地执行前向和后向抽象解释。由此得到的必要前置条件将每个缺失表达式映射为抽象值，这些抽象值表示允许缺失表达式生成的可能值的一个过近似，以确保程序满足规范。

- 组合器：给定一个标注了必要前置条件和组件的部分程序，组合器会选择要用哪个组件填充哪个空缺，并生成新的部分程序。在缺失部分放入组件时，组合器会检查该组件是否满足缺失部分的必要前置条件。如果没有组件满足任何必要前置条件，则丢弃该部分程序。如果在尝试所有组件与缺失部分的组合之前仍无法找到解决方案，则确定当前组件集合不足。在这种情况下，将调用自底向上枚举器，通过增加 n来添加更大的组件，然后重复该过程。

我们的算法保证在存在解决方案时找到该解决方案，因为自底向上枚举器最终会生成能够完成综合并满足输入-输出示例的组件。

我们已将该方法应用于 SyGuS [Alur 等人提出的 2013] 规范语言。SyGuS 是一种标准形式，通过年度竞赛建立了各种综合基准测试 [历届 SyGuS 竞赛 2020]。SyGuS 使用形式文法描述可能程序的空间。这种文法可以用某种 SMT 理论表达。我们针对位向量和 SAT 理论中的运算符设计了高度精确的专用抽象域。通过采用标准形式，我们的综合算法适用于这些理论中具有任意文法的广泛 SyGuS 问题。

我们将算法实现为一个名为 Simba 的工具。我们在此前研究中针对多种应用提出的一组 5 个基准测试上评估了 Simba：来自程序去混淆的 500 个基准测试[David 等人 2020], 369 来自程序优化的基准测试 [Lee 等人 2020], 以及来自 SyGuS 竞赛的 1006 个基准测试（用于合成抗侧信道攻击电路和位运算技巧） [往届 SyGuS 竞赛 2020]。评估结果表明，对于归纳式 SyGuS 问题，Simba 比最先进的工具更具可扩展性，包括 Duet [Lee 2021] 和Probe [Barke 等人 2020]。例如，在 544 个非条件位向量操作问题中，Simba 能够在每个问题平均用时不到 34.8 秒的情况下解决 519 个问题；相比之下，Duet 和 Probe 平均分别耗时 165.3 秒和 63.7 秒，仅解决了 456 个和 409 个问题。 Simba 相比最先进的工具，速度有显著提升。

我们总结了本工作的主要贡献：

- 一种新颖且通用的综合算法，通过同时使用前向和后向抽象解释，有效剪枝搜索空间：与现有综合算法不同，我们的算法同时采用前向和后向推理，从而充分发挥抽象解释的能力来剪枝搜索空间。
- 适用于位向量和可满足性的高精度抽象域：我们为位向量运算符和布尔运算符设计了精确的前向和后向抽象传递函数。由此得到的抽象域具有很高的精度，可用于位向量和可满足性理论中具有任意文法的归纳式 SyGuS 问题。
- 算法的实现与评估：我们将该算法实现为名为 Simba 的工具，并在来自多种应用的一组 5 个基准测试上对其进行评估。结果表明，与现有最先进的综合技术相比，该算法在性能上有显著提升。

局限性。 我们的方法要求目标应用具有高度精确的抽象域。本文展示了，我们的算法借助位向量和布尔表达式的合成能力，

得益于高度精确的抽象域，能够有效合成表达式。然而，我们的算法能否有效应用于其他场景（例如合成包含循环的程序，或合成字符串处理程序）仍是一个开放问题。我们在第 7 节中讨论了未来工作的可能方向。

2 概述

我们通过合成一个位操作程序的问题来说明我们的方法。所需的程序是一个函数 f，它以固定宽度为 4 的位向量 x 作为输入，并关闭其中最右侧 0 位左边的所有位 x。我们将位向量表示为二进制数。例如，给定一个位向量 1011_2,，该函数应返回 0011_2。

这个问题用 SyGuS 语言表示，该语言将程序合成问题表述为语法规范和语义规范的组合。f的语法规范如下文法：

$$
\begin{array}{rcll}
S & \rightarrow & x \mid 0001_2 & \text{input bit-vector and bit-vector literals} \\
 & \mid & S \wedge S \mid S \vee S \mid S \oplus S & \text{bitwise logical binary operators} \\
 & \mid & S + S \mid S \times S \mid S/S \mid S >> S & \text{bitwise arithmetic binary operators}
\end{array}
$$

其中，S是起始非终结符，运算符均为位向量理论中支持的运算符（$\oplus$ 表示按位异或运算符，$>>$ 表示算术右移运算符，左侧空出的位用左操作数的最高有效位填充）。f的语义规范遵循示例驱动编程（PBE）范式，由输入输出示例构成。为便于说明，我们假设只给出一个输入输出示例：$f(1011_2) = 0011_2$。该程序合成问题的一个解为 $f(x) = ((x + 0001_2) \oplus x) >> 0001_2$。

抽象域。在我们的抽象解释中，我们使用以下位向量抽象域来表示程序的抽象语义。按位域 由一组元素构成，每个元素都是一个长度为 4 的抽象位 序列。每个抽象位的取值来自集合 $\{0,1,\bot,\top\}$，其中 $\top$ 表示未知值，$\bot$ 表示无值。该域包含用于按位逻辑运算符和算术运算符的抽象算子。这些抽象算子在抽象域中模拟具体运算符的行为。从现在起，我们用相应的具体运算符加上上标 # 来表示该域中的每个抽象算子。例如，$1\top10 \wedge^{\#} 00\top\top = 00\top0$。

<escape>**初始部分程序的生成**。</escape>我们首先生成一个固定集合 Q，其中包含一个或多个作为占位符的非终结符。从起始符号 S出发，通过将文法的产生式规则应用于非终结符，直到达到一定深度，穷举生成所有可能的部分程序。在本例中，为便于说明，我们假设 Q 集合包含三个部分程序：

$$Q = \{(S \oplus x) >> 0001_2, (S/x) >> 0001_2, S_1 \times S_2\}$$

组件生成。组件生成器随后通过自底向上枚举搜索生成一组 C 组件，该搜索维护一组完整程序，并通过组合现有程序逐步生成新程序。组件集由大小为 $\leq n$ 的表达式组成，其中 n 是大小上界，初始值为 1。当当前组件集不足以合成解决方案时，该上界会增加 1。组件的数量可能呈 n指数级增长，但我们可以利用表达式的观察等价来减少组件数量。例如，如果 x 在组件集中，

$x \vee 0000_2$ 未添加到集合中，因为它们观察等价。由于最初$n = 1$，组件集是

$$C = \{x, 0001_2\}.$$

这些组件用于在接下来的组合阶段中补全 Q 中的部分程序所缺失的部分（即非终结符）。

必要前置条件的推导。 在开始组合阶段之前，我们使用抽象域，通过前向和后向分析，推导 Q 中部分程序的每个子表达式（包括非终结符）的必要前置条件。必要前置条件表示为按位抽象域中的一个元素。以下是 Q中部分程序的必要前置条件的推导。

- $S_1 \times S_2$（必要前置条件： $S_1 \mapsto \top\top\top\top$， $S_2 \mapsto \top\top\top\top$）：我们首先对部分程序执行前向分析，以获得关于程序最终输出和中间操作结果的不变式。非终结符可以替换为任意表达式，因此两个非终结符的抽象输出均为 $\top\top\top\top$。由此，整个程序的抽象输出为 $\top\top\top\top$。现在，我们通过检查部分程序的抽象输出是否与期望输出示例一致，来判断该部分程序的可行性。只需对部分程序的抽象输出和期望输出的抽象应用交运算符（$\sqcap$），即可简单地完成此项检查。由于结果不包含$\bot$ （$\top\top\top\top \sqcap 0011 = 0011$），因此该部分程序是可行的。接下来，可以执行后向分析，以获得关于非终结符的必要前置条件。根据期望输出0011_2，考虑到乘法运算中可能发生的溢出， S_1 和 S_2 可以取任意值。因此， S_1（以及 S_2）的必要前置条件为 $\top\top\top\top$。

- $(S \oplus x) \gg 0001_2$ (必要前置条件： $S \mapsto 110\top$)：通过前向分析，$(S \oplus x)$ 的抽象输出为 $\top\top\top\top$，而 $(S \oplus x) \gg 0001_2$ 的抽象输出为 $\top\top\top\top \gg^{\#} 0001 = \top\top\top\top$。现在我们检查部分程序的可行性。由于该结果与期望输出 ($\top\top\top\top \sqcap 0011 = 0011$) 不矛盾，因此该部分程序是可行的。现在执行后向分析。根据期望输出 0011_2，我们可以推导出 $(S \oplus x)$ 上的必要前置条件为 $011\top$，因为 $011\top \gg^{\#} 0001 = 0011$。 S 上的必要前置条件为 $110\top$，因为对于 x 的输入 1011_2，$110\top \oplus^{\#} 1011 = 011\top$。
- $(S/x) \gg 0001_2$ （该程序不可行）：通过前向分析，(S/x) 的抽象输出为 $000\top$。这是因为 S 的最大可能值为 1111_2（十进制为 15），而输入 x的值为 1011_2（十进制为 11），所以 (S/x) 的可能值为 0 和 1，在按位域中表示为 $000\top$。(S/x) 的抽象输出 $\gg 0001_2$ 为 0000，因为 $000\top \gg^{\#} 0001 = 0000$。现在我们检查部分程序的可行性。由于结果与期望输出 ($0000 \sqcap 0011 = 00\bot\bot$) 不一致，因此该部分程序不可行。

如上所示，部分程序 $(S/x) \gg 0001_2$ 被确定为不可行。在组合阶段，只会考虑另外两个部分程序。

组合过程。 给定 Q 中带有必要前置条件的部分程序以及 C 中的组件集合，组合器通过使用组件替换部分程序中的非终结符，生成新的（部分）程序。

组合器首先选择 $(S \oplus x) \gg 0001_2$。随后，它会在$C = \{x、0001_2\}$中搜索一个组件 $c$in，使得关于 S 的必要前置条件被 c 的抽象输出所蕴含。不存在这样的组件，因为必要前置条件 $110\top$ 既不被 x (1011) 的抽象输出所蕴含，也不被 0001_2 (0001) 的抽象输出所蕴含。因此，组合器丢弃该部分程序。

下一个部分程序是 $S_1 \times S_2$。假设组合器将 S_1 替换为 x，得到一个新的部分程序 $x \times S_2$。每当生成一个新的部分程序时，都会调用必要前置条件生成器，以推导关于非终结符的必要前置条件。由于乘法运算是模 2^4, 的，计算关于 S_2 的必要前置条件相当于求解以下方程：

$$1011_2 \times y = 0011_2 \mod 16$$

其中， y 表示 S_2的未知值。利用扩展欧几里得算法，我们可以求得该方程的解为 $y = 1001_2$。因此，关于 S_2 的必要前置条件为1001。不幸的是， C 中没有任何组件的抽象输出能够涵盖该必要前置条件。因此，组合器会丢弃该部分程序。

由于组合器穷尽了 Q 中的所有部分程序，仍未找到解决方案，因此它调用组件生成器生成更多组件。

接下来，假设组件生成器生成大小为 ≤ 3 的组件，结果为 $C = \{x, 0001_2, x + 0001_2, \cdots\}$。

组合器重新检查部分程序 $(S_1 \oplus x) \gg 0001_2$。回想一下，关于 S_1 的必要前置条件是 $110\top$。现在，输出为 1100_2 的组件 $x + 0001_2$ 满足必要前置条件。因此，组合器将 S_1 替换为 $x + 0001_2$，以获得新的程序 $((x + 0001_2) \oplus x) \gg 0001_2$。评估该程序后，组合器发现程序正确，并将其作为解决方案返回。

本文其余部分组织如下。第 3 节介绍预备概念，并描述我们的总体综合算法。第 4 节介绍针对位向量和布尔逻辑理论专门设计的抽象域。第 5 节展示实验结果。第 6节讨论相关工作。第 7 节讨论未来工作。第 8节总结全文。

3 总体综合算法

本节将阐述我们的方法。我们首先介绍预备概念，包括项、正则树文法，以及基于有限示例集的语法引导综合。随后，我们将介绍通用综合算法，该算法基于迭代式前向-后向抽象解释。

3.1 预备知识

项 一个签名 Σ 是由函数符号组成的集合，其中每个 $f \in \Sigma$ 都关联一个非负整数 n，即 f的元数（记为 $arity(f)$）。对于 $n \geq 0$，我们用 $\Sigma^{(n)}$表示所有 n 元元素 Σ 组成的集合。元数为 0 的函数符号称为常量。设 V 为一个变量集合。所有基于 V的 Σ-项构成的集合 $T_{\Sigma,V}$通过归纳方式定义如下：$V \subseteq T_{\Sigma,V}$和 $\forall n \geq 0, f \in \Sigma^{(n)}. t_1, \cdots t_n \in T_{\Sigma,V}. f(t_1, \cdots t_n) \in T_{\Sigma,V}$。一个项可以看作一棵树。

位置 项 s 的位置集合是由自然数构成的字母表上的字符串集合$Pos(s)$，其归纳定义如下：

- If $s = x \in V$, $Pos(s) = \{\epsilon\}$.
- If $s = f(s_1, \cdots, s_n)$, then $Pos(s) = \{\epsilon\} \cup \bigcup_{i=1}^{n} \{ip \mid p \in Pos(s_i)\}$.

位置 ϵ称为项 s的根位置。对于 $p \in Pos(s)$，位置 p处的子项（记作 $s|_p$）定义为：(i) $s|_\epsilon = s$以及 (ii) $f(s_1, \cdots s_n)|_{iq} = s_i|_q$。对于 $p \in Pos(s)$，我们将通过把位置 p 处的子项替换为 t而从 s 获得的项记作 $s[p \leftarrow t]$。形式化地，$s[\epsilon \leftarrow t] = t$以及 $f(s_1, \cdots s_n)[iq \leftarrow t] = f(s_1, \cdots s_i[q \leftarrow s], \cdots s_n)$。

算法1 Simba算法

要求： 一个 SyGuS 实例 G，$\Phi = \bigcup_{j=1}^{m}\{i_j \mapsto o_j\}$，其中每个 i_j，$o_j \in D$

要求： 抽象域 $\hat{D}$ 使得 $\mathrm{P}(D)$ $\underset{\alpha}{\overset{\gamma}{\leftrightarrows}}$ $\hat{D}$

要求： 一个整数 d，表示部分程序的最大高度

确保： 满足 Φ 的解程序 $P \in L(G)$

$Q :=$ 生成草图(G, d)

$n := 1$

C:= $\emptyset$

重复

  $Q' := Q$

  C := 生成组件$(G, \mathrm{C}\ n)$ ▷C： $N \to \mathrm{P}(L(G))$

  当 Q' 非空时 **执行**

    从 Q' 中移除 P

    如果 完整性检查(P) 且 $P \models \Phi$ **则返回** P

    else

      A := 分析(P, Φ) ▷A： $Pos(P) \to \hat{D}^m$

      如果 $\exists p \in$ 正位置$(P).\ \perp_{\hat{D}} \in \mathcal{A}(p)$ **则继续**

      pos :=选择(空缺(P)) 正位置(P)

      对于 $c \in$C$(P|_{pos})$ 使得 $\alpha([\![c]\!](i_1)), \cdots\ \alpha([\![c]\!](i_m)) \sqsubseteq$A$(pos)$ **执行**

        $Q' := Q' \cup \{P[pos \leftarrow c]\}$

      结束循环

    结束条件判断

  结束循环

  $n := n + 1$

直到假

正则树文法。正则树文法是一个元组 $G = (N, \Sigma, S, \delta)$，其中 N是非终结符号的有限集合（元数为 0），Σ 是签名，$S \in N$是初始非终结符，而 δ是形如 $A_0 \to \sigma(i)(A_1, \cdots\ A_i)$ 的产生式有限集合，其中每个 $A_j \in N$都是非终结符。给定一棵树（或一个项）$t \in T_\Sigma$，v，将产生式 $r = A \to \beta$应用于 t，会把 t中出现的 A替换为右部 β。树$t \in T_\Sigma$，v由文法 G生成，当且仅当它可以通过依次应用产生式 r_1、$\cdots$ r_n，从根节点表示初始非终结符 S的树获得。所有能够从 S推导出的树都称为 G的语言，并记作 $L(G)$。

归纳式语法引导综合。语法引导综合问题 [Alur 等人 2013]将其定义为一个元组 G, Φ。目标是在可判定理论中，找到一个满足给定规范 Φ 的程序 P。程序必须使用由正则树文法 G 描述的语言 $L(G)$ 编写。具体来说，如果规范是输入-输出对集合 $\Phi = \bigcup_{j=1}^{m}\{i_j \mapsto o_j\}$，其中 i_j和 o_j是值 [1]，我们就称该 SyGuS 实例为归纳式。假设为 $L(G)$ 中的每个程序 P赋予确定性语义 $[\![P]\!]$，目标是找到一个程序 P，使得对于所有$i \mapsto o \in \Phi$，$[\![P]\!](i) = o$（记作 $P \models \Phi$）。

3.2 总体算法

现在我们介绍一种基于双向搜索的归纳式综合算法，该算法通过使用前向/后向抽象解释来加速。算法1 展示了我们综合算法的高层结构，该算法以.....为输入

- 具有正则树文法 G和 m 输入输出示例 Φ的归纳式 SyGuS 实例
- 抽象域 $\hat{}D$，用于抽象 D 中所有项在 $L(G)$中的取值集合，以及一个伽罗瓦连接 $\alpha : \mathrm{P}(D) \to \hat{}D$和 $\gamma : \hat{}D \to \mathrm{P}(D)$
- 整数 d，用于指定要探索的草图 （含非终结符的部分程序）的最大高度

高度为 $\leq d$ 的草图按照文法 G自顶向下枚举，并插入优先队列 Q 中（第 1 行）。随后，将组件的大小上界 n 初始化为 1（第 2 行）。整数 n 将在每次迭代中递增 1（第 19 行），直到找到一个解。组件池 C（初始为空集）包含所有通过自底向上方式生成的大小为 $\leq n$ 的组件。主循环（第 4–20 行）将重复执行，直到找到一个解。当前迭代中使用的优先队列 Q' 通过将 Q 中的草图插入 Q' 来初始化（第 5 行）。GenerateComponent 过程通过组合先前生成的表达式，逐步构建大小为$\leq n$ 的表达式（第 6 行）。通过利用观察等价性，C 不会包含多个相对于规范在语义上彼此等价的组件。在 while 循环（第 7–18 行）中，算法探索由当前组件池 C 和草图集合所确定的搜索空间。如果从 Q' 中出队的候选项 P是一个完整程序，且相对于规范是正确的，则返回 P作为解（第 9 行）。否则，继续探索 Q' 中的另一个候选项。如果候选项 P 是一个部分程序，我们将对 P 进行分析，以推导出 P 中每个子表达式的必要前置条件。Analyze $\hat{}$ 过程计算一个映射 A，该映射将 P 中的每个子表达式映射到由 D 中的 m 个抽象值组成的元组（第 11 行）。第 j 个抽象值表示一个必要前置条件：为了满足第 j 个输入输出示例 $i_j \mapsto o_j$，替换该非终结符的任何表达式都必须满足此条件。如果P存在一个位置，其对应的 $\bot_{\hat{D}}$ 表示该位置无法放入任何表达式，则判定 P不可行，并将其丢弃（第 12 行）。否则，选择 P中的一个非终结符（第 13 行）。对于每个可以替换该非终结符且满足必要前置条件的组件 c（第 14 行），我们将该非终结符替换为 c，并将生成的程序加入 Q'（第 15 行）。如果使用当前组件池 C 未能找到解，则将大小 n 递增 1（第 19 行），并重复执行主循环。

我们的算法具有以下性质。首先，从以下意义上来说，我们的算法是可靠且完备的。

定理 3.1。 *算法 1 在如下意义上是可靠且完备的：如果给定的归纳式 SyGuS 实例存在解，算法 1 最终会找到该解。*

其次，如果队列中的每个草图 [Hu 等人 2020] 在某个位置都具有 Q'，则可以判定给定的归纳式 SyGuS 实例不可实现；这些草图会在主循环的第 $\bot_{\hat{D}}$行被丢弃12。也就是说，没有任何草图能够补全为可行程序。最后，该方法也可以单独用于自顶向下综合，而不是双向综合，因此适用于现有的自顶向下综合算法。为此，我们无需使用固定的草图集合，而是在主循环的每次迭代中（第4–20行），通过增大待探索草图的最大高度，将更大的草图加入队列 Q'，同时保持大小上界 d为 n，直至找到解1。

[1]具有逻辑公式的 SyGuS 实例中，对于给定输入，若存在唯一一个满足该公式的输出，则可以通过反例引导的归纳综合（CEGIS），将其转换为具有输入输出示例的实例 [Solar-Lezama 等人 2006]。

3.3 迭代式前向-后向分析

我们将详细介绍算法Analyze 中的过程，见算法 1。已知，通过交替进行前向和后向分析，可以计算出一个近似，该近似覆盖了既能从程序入口到达、又能到达程序出口处期望状态的状态集合 [Cousot 和 Cousot 1992]。在我们的设定中，程序是一棵树，程序的执行从程序树的叶节点（常量或输入变量）开始，并向根节点推进。因此，对于程序树中的每个节点，前向分析以自底向上的方式计算以该节点为根的子树可能计算出的值集合的过近似。后向分析则以自顶向下的方式，计算其父节点为生成期望输出而可能使用的值集合的过近似。

给定候选项 P 和 n 输入输出示例 Φ，分析过程会针对每个输入输出示例执行迭代式前向-后向分析，并合并结果以获得映射 A。分析结果 A 将 P 中的每个位置映射到一个抽象值元组 $\hat{}\,\hat{}d_1, \cdots \ d_m$。

相对于输入输出示例 $i \mapsto o$ 的前向抽象语义由以下函数 $\mathrm{F}_{\langle i,o\rangle}$ 的最小不动点刻画：$(Pos(P) \to \hat{}D) \to (Pos(P) \to \hat{}D)$：

$$\mathcal{F}_{\langle i,o\rangle} = \lambda X.\ \mathcal{I}^i_{\mathcal{F}} \sqcup F^{\#}(X) \quad \text{where} \quad \mathcal{I}^i_{\mathcal{F}} = \{p \mapsto \begin{cases} \alpha(i) & (P|_p \in V) \\ \top & (P|_p \in N) \\ \alpha(P|_p) & (P|_p \text{ is a constant}) \\ \bot & (\text{otherwise}) \end{cases} \mid p \in Pos(P)\}.$$

初始状态 $\mathrm{I}^i{}_{\mathcal{F}}$ 将 N 中的每个非终结符映射到 $\top$，因为非终结符表示一个空缺，可以由任意表达式填充。每个变量都映射到输入示例的抽象（即$\alpha(i)$），每个常量都映射到该常量本身的抽象。前向抽象函数 $F^{\#}$ 定义如下：

$$F^{\#}(X) = \{p \mapsto \begin{cases} \overrightarrow{f}^{\#}(X(p1), \cdots, X(pk)) & (P|_p = f(\cdots), arity(f) = k) \\ \bot & (\text{otherwise}) \end{cases} \mid p \in Pos(P)\}$$

其中 $\overrightarrow{f}_{\#}$ 表示算子 f 的前向抽象算子。它接收参数的抽象值，并返回结果抽象值。

相对于输入输出示例 $i \mapsto o$ 的后向抽象语义，由以下函数 $\mathrm{B}_{\langle i,o\rangle} : (Pos(P) \to \hat{}D) \to (Pos(P) \to \hat{}D)$ 的最大不动点刻画：

$$\mathcal{B}_{\langle i,o\rangle} = \lambda X.\ \mathcal{I}^o_{\mathcal{B}} \sqcap B^{\#}(X) \quad \text{where} \quad \mathcal{I}^o_{\mathcal{B}} = \{p \mapsto \begin{cases} \alpha(o) & (p = \epsilon) \\ \top & (\text{otherwise}) \end{cases} \mid p \in Pos(P)\}.$$

最终状态 $\mathrm{I}^o_{\mathcal{B}}$ 将根位置映射到输出示例的抽象（即 $\alpha(o)$），并将其他每个位置映射到 $\top$。后向抽象函数 $B^{\#}$ 定义如下：

$$B^{\#}(X) = \{p \mapsto \begin{cases} \overleftarrow{f}^{\#}_i(X(p'), X(p'1), \cdots, X(p'k)) & (p = p'i, P|_{p'} = f(\cdots), arity(f) = k) \\ \top & (\text{otherwise}) \end{cases} \mid p \in Pos(P)\}$$

其中 $\overleftarrow{f}_{\#i}$ 表示算子 f 的后向抽象算子。它接收结果的抽象值和参数的抽象值，并返回第 i 个参数的抽象值；该值对应于使第 i个参数满足输入输出示例的必要前置条件 。

通过求得以下递减链的极限，计算前向与后向抽象语义的交集。该递减链对所有 $n \in \mathbb{N}$ 均有定义，直到链收敛 [Cousot 和 Cousot 1992]：$\dot{X}^0{}_{i,o} = \mathrm{lfp}\ \mathrm{F}_{\langle i,o\rangle}$, $\dot{X}^{2n+1}{}_{i,o} = \mathrm{gfp}\ \lambda X.\ \dot{X}^{2n}{}_{i,o} \sqcap \mathrm{B}_{\langle i,o\rangle}(X)$，以及 $\dot{X}^{2n+2}{}_{i,o} = \mathrm{lfp}\ \lambda X.\ \dot{X}^{2n+1}{}_{i,o}\ \ulcorner\mathrm{F}_{\langle i,o\rangle}(X)$。

分析过程定义如下：

$$\textsc{Analyze}(P, \bigcup_{j=1}^{m} i_j \mapsto o_j) = \{p \mapsto \langle \mathcal{A}_1(p), \cdots, \mathcal{A}_m(p)\rangle \mid p \in Pos(P), \forall 1 \le j \le m.\, \mathcal{A}_j = \lim_{n\to\infty} \dot{X}^n_{\langle i_j, o_j\rangle}\}$$

示例 *3.2.* 考虑第 $P = (S \oplus x) \gg 0001_2$ 节概览示例中的部分程序 2，其中输入示例为 $i = 1011_2$，输出示例为 $o = 0011_2$。P 的子项为 $(S \oplus x)$、0001_2、S 和 x，其位置分别为 1、2、11 和 12。假设抽象域 $\hat{}D$ 为按位域，则前向分析的初始状态 $\mathrm{I}^i_{\mathcal{F}}$ 为 $\{\epsilon \mapsto \bot\bot\bot\bot, 1 \mapsto \bot\bot\bot\bot, 2 \mapsto 0001, 11 \mapsto \top\top\top\top, 12 \mapsto 1011\}$。后向分析的初始状态 $\mathrm{I}^o_{\mathcal{B}}$ 将根位置 ϵ 映射为 0011，将其他所有位置映射为 $\top\top\top\top$。下表展示了递减链 $\dot{X}^n{}_{\langle i,\ o\rangle}$中的前三个元素：

位置 p	$\dot{X}^0_{i,o}(p)$	$\dot{X}^1_{i,o}(p)$	$\dot{X}^2_{i,o}(p)$
ϵ	$\top\top\top\top$	0011	0011
1	$\top\top\top\top$	$011\top$	$011\top$
2	0001	0001	0001
11	$\top\top\top\top$	$110\top$	$110\top$
12	1011	1011	1011

我们假设使用为按位域中的按位异或运算符和算术右移运算符定义的前向和后向抽象算子（见第4节）。由于该链在 $\dot{X}^2{}_{\langle i,\ o\rangle}$处收敛，分析过程将其作为最终结果返回。

3.4 优化

在实现中，我们采用以下优化来提高 Algo 的效率。 1.

通过具体化避免不必要的扫描。除了组件池 $\mathsf{C}: N \to \mathsf{P}(L(G))$（用于记忆可由每个非终结符推导出的组件表达式）外，我们还维护一个额外的映射 $\mathsf{V}: N \to (D \times D) \to \mathsf{P}(L(G))$，用于返回具有特定输入输出行为的组件。例如，在第 2 节的概览示例中，$\mathsf{V}(S)(1011_2, 1100_2) = \{x + 0001_2\}$，因为当以 1011_2 作为输入时，$x + 0001_2$ 输出 1100_2。只有当分析结果 $\mathsf{A}(pos)$ 足够精确（即其具体化结果是一个较小的集合）时，才使用此映射来节省第 14 行所需的时间。在这种情况下，我们不再计算集合 $\{c \in \mathsf{C}(P|_{pos}) \mid \alpha(\llbracket c \rrbracket(i_1)), \cdots \alpha(\llbracket c \rrbracket(i_m)) \sqsubseteq \mathsf{A}(pos)\}$，而是计算 $\bigcap\{\mathsf{V}(P|_{pos})(i_j, v_j) \mid \hat{}\mathsf{A}(pos) = d_1, \cdots d_m\ \forall 1 \le j \le m.\, v_j \in \gamma(d_j)\}$，以获得与分析结果兼容的组件集合。通过避免扫描整个组件池 C，这可以节省第 14 行所需的时间。

条件程序的分治法。对于条件程序的综合，我们将分治枚举方法 [Alur 等人 2017] 按如下方式纳入我们的算法。首先，对于每个输入输出示例，我们使用算法 1综合出一个满足该示例的无条件程序。其次，我们使用 [Alur 等人 2017]先前提出的决策树学习算法，将这些无条件程序组合成一个适用于所有示例的条件程序。条件谓词通过自底向上枚举生成。

4 抽象域

本节中，我们提出精确且高效的抽象域，可用于处理基于以下背景理论的广泛归纳式 SyGuS 问题：(1) 固定宽度位向量和 (2) 命题逻辑。

由于篇幅所限，我们仅介绍抽象域的一些值得注意之处。抽象域的完整细节见补充材料。

4.1 符号表示

对于固定宽度 w的位向量理论，我们遵循 SMT-LIB v2.0 标准 [Barrett 等人 2010]中所描述的位向量运算符的标准语法和语义。无符号和有符号位向量分别表示为长度为 w 的位串（即 $\{0,1\}^w$）。无符号位向量的取值范围为 $\{0,\cdots\ 2^w-1\}$，有符号位向量的取值范围分别为 $\{-2^{w-1},\cdots\ 2^{w-1}-1\}$。我们分别将 -2^{w-1}、$2^{w-1}-1$、0 和 2^w-1 称为 smin、smax、umin 和 umax。我们将 $[n]_i$表示为位串表示中 n 的第 i 位，并将 $[n]_{i:j}$表示为 n 的子串，该子串由 n 的第 i 位、第 $i{+}1$ 位、…、第 j 位组成。两个字符串 n_1 和 n_2 的连接表示为 $n_1 \cdot n_2$，字符串 n 的长度表示为 $|n|$。我们将 $n[a/b]$ 表示为通过将 n 中每个 b 替换为 a 所得到的位串。对于区间 $[l,h]$，我们分别将 $lb([l,h])$ 和 $ub([l,h])$ 表示为 $[l,h]$ 的下界和上界。最后，对于位串 n，我们将 Trail0s(n) 表示为 n 中尾随零的数量。

4.2 固定位宽位向量的抽象域

抽象域。 我们针对定宽位向量的抽象域是一个约化积域。抽象域的约化积可以通过协同结合多个抽象域的表达能力，实现精确的抽象 [Cousot 2021]。我们的抽象域由以下几个域组成。

$\hat{}$ 按位域 $B, \sqsubseteq_{\hat{B}}, \sqcup_{\hat{B}}, \sqcap_{\hat{B}}$ 是一种独立跟踪位向量中每一位取值的域，此前的工作中也使用过该域 [Miné（2012）；Regehr 和 Duongsaa 2006]。按位域 B 的每个元素都是一个长度为 w 的抽象位字符串，如第 2 节所述。该域形式化定义如下：$\hat{}_B \text{def}= \{b_1 \cdot b_2 \cdots b_w \mid \forall 1 \le i \le w.\ b_i \in \{0,1,\bot,\top\}\}$。为定义 B 与 $\mathrm{P}(\mathbb{Z})$ 之间的伽罗瓦连接，我们定义如下函数，该函数使用二进制补码表示法 [Miné（2012）计算整数 x 的位向量表示 $p(x)$：

$$p(x) \stackrel{\text{def}}{=} \begin{cases} b_1 \cdots b_w & \text{where } \forall 1 \le i \le w.\ b_i = \lfloor x/2^{w-i} \rfloor \mod 2 \quad (x \ge 0) \\ b'_1 \cdots b'_w & \text{where } \forall 1 \le i \le w.\ b'_i = \neg b_i, b_i = [p(-x-1)]_i \quad (x < 0) \end{cases}$$

$\gamma_{\hat{B}}$ $\hat{}$ 请注意，最左侧的位是最高有效位。伽罗瓦连接 $\mathrm{P}(\mathbb{Z}) \underset{\alpha_{\hat{B}}}{\overset{\gamma_{\hat{B}}}{\leftrightarrows}} B$ 定义如下：

$$\alpha_{\hat{B}}(Z) \stackrel{\text{def}}{=} \bigsqcup_{\hat{B}} \{p(x) \mid x \in Z\} \qquad \gamma_{\hat{B}}(b) \stackrel{\text{def}}{=} \begin{cases} \emptyset & (\exists i.\ [b]_i = \bot) \\ \gamma^{\text{unsigned}}_{\hat{B}}(b) \cup \gamma^{\text{signed}}_{\hat{B}}(b) & (\text{otherwise}) \end{cases}$$

其中，无符号和有符号位向量的具体化函数定义如下：

$$\gamma^{\text{unsigned}}_{\hat{B}}(b) \stackrel{\text{def}}{=} \{\textstyle\sum_{i=0}^{w-1} 2^i m_{w-i} \mid \forall 1 \le i \le w.\ m_i \in \gamma_B([b]_i)\}$$
$$\gamma^{\text{signed}}_{\hat{B}}(b) \stackrel{\text{def}}{=} \{-2^{w-1} m_1 + \textstyle\sum_{i=0}^{w-2} 2^i m_{w-i} \mid \forall 1 \le i \le w.\ m_i \in \gamma_B([b]_i)\}.$$

$\hat{}$ 有符号区间域 S，$\sqsubseteq_{\hat{S}}$，$\sqcup_{\hat{S}}$，$\sqcap_{\hat{S}}$ 是一种将位向量表示为有符号位向量区间的域。形式上，$\hat{}_S$定义为$= \{[l,h] \mid l,\ h \in \{0,\ 1\}^w,\ [\![\text{bvsle}]\!](l,\ h) = \text{真}\}$，其中 bvsle

$\gamma_{\hat{S}}$ $\hat{}$ 是表示有符号小于等于的二元谓词。伽罗瓦连接 $\mathbf{P}(\mathbb{Z}) \underset{\alpha_{\hat{S}}}{\overset{\gamma_{\hat{S}}}{\leftrightarrows}} S$ 是标准的。

$\hat{}$ 无符号区间域 U， $\sqsubseteq_{\hat{U}}$， $\sqcup_{\hat{U}}$， $\sqcap_{\hat{U}}$ 是一种将位向量表示为无符号位向量区间的域。形式上，$\hat{}_U \text{def}= \{[l,h] \mid l, \ h \in \{0,1\}^w, \ [\![\text{bvule}]\!](l, \ h) = \text{true}\}$，其中 bvule 是表示无符号小于等于的二元谓词。伽罗瓦连接 $\mathbf{P}(\mathbb{Z}) \underset{\alpha}{\overset{\gamma_{\hat{U}}}{\leftrightarrows}} \hat{}_U$ 也是标准的。

$\hat{}$ $\hat{}$ $\hat{}$ 上述三个域组合起来构成乘积抽象域 $D = B \times S \times U$。其伽罗瓦连接 $\mathbf{P}(\mathbb{Z}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \hat{}D$ 可以通过组合这三个域的伽罗瓦连接来简单定义。

为了让三个域之间的信息流能够相互细化，我们需要使用一个约简算子，利用其中一个域所跟踪的信息来细化其他域所跟踪的信息。约简需要计算一个不动点 [Granger 1992]，$\hat{}$ $\hat{}$，我们的迭代约简算子 ρ： $D \to D$ 定义如下：

$$\rho \overset{\text{def}}{=} \mathit{fix}\, \lambda\langle b,s,u\rangle.\, \langle b \sqcap \pi_{\hat{S}\to\hat{B}}(s) \sqcap \pi_{\hat{U}\to\hat{B}}(u), s \sqcap \pi_{\hat{B}\to\hat{S}}(b) \sqcap \pi_{\hat{U}\to\hat{S}}(u), u \sqcap \pi_{\hat{B}\to\hat{U}}(b) \sqcap \pi_{\hat{S}\to\hat{U}}(s)\rangle$$

$\hat{}$ 其中 $\pi_{\hat{}D_1\to\hat{}D_2}$ 是我们的投影算子，它将信息从抽象域 $D_1\hat{}$ 传播到另一个域 D_2。例如，$\pi_{\hat{B}\to\hat{U}}$ 算子接受一个按位元素，并返回一个无符号区间，其下界（分别为上界）是通过将每个 $\top$ 抽象位替换为 0（分别为 1）得到的位向量。更多详情请参阅补充材料。

$\hat{}$ 抽象域 D 中的约简算子 ρ 应当是可靠的，也就是说，它必须满足以下两个性质：对于所有 $d \in D$， (1) $\rho(d) \sqsubseteq d$（应用该算子的结果是一个更精确的抽象元素）； (2) $\gamma(\rho(d)) = \gamma(d)$（一个抽象元素及其约简具有相同的含义）。

定理4.1。 我们的约简算子 ρ 是可靠的。

抽象算子。 现在我们定义前向和后向抽象算子。由于我们处理的是位数固定的二进制数，所有域都考虑可能发生的溢出/下溢。对于每个元数为 k的位向量运算 f，前向抽象算子 $\overrightarrow{f}^{\#}$： $\hat{}D^k \to \hat{}D$接收 k 个参数的抽象元素，并返回结果的一个抽象元素，其定义如下：

$$\overrightarrow{f}^{\#}(\langle b_1,s_1,u_1\rangle,\cdots,\langle b_k,s_k,u_k\rangle) = \rho(\langle \overrightarrow{f}^{\#}_{\hat{B}}(b_1,\cdots,b_k), \overrightarrow{f}^{\#}_{\hat{S}}(s_1,\cdots,s_k), \overrightarrow{f}^{\#}_{\hat{U}}(u_1,\cdots,u_k)\rangle)$$

其中， $\overrightarrow{f}^{\#}_{\hat{B}}$、 $\overrightarrow{f}^{\#}_{\hat{S}}$和 $\overrightarrow{f}^{\#}_{\hat{U}}$分别是按位域、有符号域和无符号域中的前向算子。对于每个 $1 \le i \le k$，后向抽象算子 $\overleftarrow{f}^{\#}_i$： $\hat{}D^{k+1} \to \hat{}D^k$接收结果当前的抽象元素以及 k 个参数的抽象元素，并返回 i个参数中一个更精确的抽象元素。

$$\overleftarrow{f}^{\#}_i(\langle b,s,u\rangle,\langle b_1,s_1,u_1\rangle\cdots\langle b_k,s_k,u_k\rangle) = \rho(\langle \overleftarrow{f}^{\#}_{\hat{B},i}(b,b_1\cdots b_k), \overleftarrow{f}^{\#}_{\hat{S},i}(s,s_1\cdots s_k), \overleftarrow{f}^{\#}_{\hat{U},i}(u,u_1\cdots u_k)\rangle)$$

其中， $\overleftarrow{f}^{\#}_{\hat{B},\ i}$、 $\overleftarrow{f}^{\#}_{\hat{S},\ i}$和 $\overleftarrow{f}^{\#}_{\hat{U},\ i}$分别是按位域、有符号域和无符号域中的后向算子。每当应用前向或后向算子时，我们总会应用约简算子，以使分析结果保持在最精确的形式。

下面我们将介绍各个域的前向和后向算子。我们采用与 SMT-LIB v2.0 标准中定义相同的位向量算子名称。

有符号/无符号区间域的前向抽象算子下面我们将详细说明前向抽象算子的若干显著特征。

- $\overrightarrow{\text{bvadd}}^{\#}_{\hat{S}}$的定义与标准整数区间算术类似，略有不同的是，它会考虑可能发生的溢出或下溢。形式化地说，

$$\overrightarrow{\text{bvadd}}^{\#}_{\hat{S}}([l_1,h_1],[l_2,h_2]) \quad = \quad \text{Wrap}_{\hat{S}}([[\![\text{bvadd}]\!](l_1,l_2),[\![\text{bvadd}]\!](h_1,h_2)]) \quad (\overrightarrow{\text{bvadd}}^{\#}_{\hat{U}} \text{ is similar.})$$

其中，如果 $([l,h] \sqsubseteq_{\hat{S}} [\text{smin}, \text{smax}])$，则为 $\text{Wrap}_{\hat{S}}([l,h]) = [l,h]$；否则为 $[\text{smin}, \text{smax}]$。其他抽象算子（包括减法、乘法和除法）的定义也类似。

- $\overrightarrow{\text{bvneg}}^{\#}_{\hat{U}}$会交换区间的下界和上界，但下界表示零、上界表示非零值的情况除外。这是因为 1 的取负值为 -1，在二进制补码表示中写作 $11\cdots1_2$。$11\cdots1_2$是无符号区间域中的最大值。因此，这种情况下会返回顶元素。形式化地说，

$$\overrightarrow{\text{bvneg}}^{\#}_{\hat{U}}([l,h]) \quad = \quad \begin{cases} \top_{\hat{U}} & (l = p(0), h \neq p(0)) \\ [[\![\text{bvneg}]\!](h),[\![\text{bvneg}]\!](l)] & (\text{otherwise}) \end{cases}$$

- $\overrightarrow{\text{bvsdiv}}^{\#}_{\hat{S}}$模拟有符号除法的以下具体语义。

$$[\![\text{bvsdiv}]\!](s,t) = \begin{cases} [\![\\ [\![\text{bvudiv}]\!](s,t) & (\text{当 } s\text{和 } t\text{均为非负数时}) \\ \text{bvudiv}]\!]([\![\text{bvneg}]\!](s) & [\![\text{bvneg}]\!](t)) \quad (\text{当两者均为负数时}) \\ \text{bvneg}]\!]([\![\\ \text{bvudiv}]\!]([\![\text{bvneg}]\!](s) & t)) \quad (\text{当 } s\text{为负数且 } t\text{为非负数时}) \\ \text{bvneg}]\!]([\![\text{bvudiv}]\!](s,[\![\text{bvneg}]\!](t))) & (\text{当 } s\text{为非负数且 } t\text{为负数时}) \end{cases}$$

根据上述语义，$\overrightarrow{\text{bvsdiv}}^{\#}_{\hat{S}}$将参数的区间划分为四种情况，并使用标准定义的$\overrightarrow{\text{bvudiv}}^{\#}$运算符分别计算每种情况的商。

前向抽象按位域运算符以下几点值得注意：

- $\overrightarrow{\text{bvadd}}^{\#}_{\hat{B}}$由 RippleCarryAdd 运算符定义，该运算符由 [Regehr 和 Duongsaa 2006],定义，用于模拟按位域中两个位向量的行波进位加法。
- $\overrightarrow{\text{bvneg}}^{\#}_{\hat{B}}$由 $[\![\text{bvneg}]\!](b) = [\![\text{bvadd}]\!]([\![\text{bvnot}]\!](b)\ p(1))$ [Warren 2012]定义。
- $\overrightarrow{\text{bvmul}}^{\#}_{\hat{B}}$的定义依据是：$[\![\text{bvmul}]\!](b_1,b_2)$ 结果中的尾随零数量等于 b_1 和 b_2中的尾随零数量之和。形式化地，

$$\overrightarrow{\text{bvmul}}^{\#}_{\hat{B}}(b_1,b_2) \quad = \quad \top^{w-(n+m)} \cdot 0^{(n+m)} \quad \text{where } n = \text{Trail0s}(b_1), m = \text{Trail0s}(b_2)$$

- 算术右移 $\overrightarrow{\text{bvashr}}^{\#}_{\hat{B}}$的抽象算子使用按位域中定义的右移算子，其定义如下：$b >>^{\#}_{x} i$将 b的所有抽象位向右移动 i位，最左侧的位用 x填充。形式化地，

$$\overrightarrow{\text{bvashr}}^{\#}_{\hat{B}}(b_1,b_2) \quad = \quad \bigsqcup\{b_1 >>^{\#}_{[b_1]_1} (i \mod w) \mid i \in \gamma^{\text{unsigned}}_{\hat{B}}(b_2)\}.$$

其他抽象移位算子的定义也类似。

有符号/无符号区间域的后向抽象算子 有以下几点值得注意：

- $\overleftarrow{\texttt{bvurem}}^{\#\hat{U},1}(i,\ i_1,\ i_2)$ 根据第二个操作数（i_2）和结果（i）的抽象值，细化第一个操作数的抽象值（i_1），定义如下：

$$\overleftarrow{\texttt{bvurem}}^{\#}_{\hat{U},1}(i,i_1,i_2) \quad = \quad \begin{cases} i \sqcap_{\hat{U}} i_1 & (\llbracket \texttt{bvule} \rrbracket\,(p(2^{w-1}), lb(i)) = \texttt{true}) \\ \top & (\text{otherwise}) \end{cases}$$

此行为基于这样一个事实：对于某些位向量 b_1、b_2 和 b，如果 $\llbracket \texttt{bvurem} \rrbracket\,(b_1,\ b_2) = b$ 且 b 的最高有效位为 1，则 $b = b_1$。证明见补充材料。

按位域的后向抽象算子 一些值得注意的算子如下：

- $\overleftarrow{\texttt{bvand}}^{\#}_{\hat{B},\ 1}$ 根据另一个操作数和结果的抽象值，推导 bvand 左操作数的抽象值。例如，对于结果的每个抽象位，如果该位为 1，我们也可以推断第一个操作数的对应位为 1。形式上，

$$\overleftarrow{\texttt{bvand}}^{\#}_{\hat{B},1}(b,b_1,b_2) \quad = \quad d_1 d_2 \cdots d_w \quad \text{where } \forall 1 \le i \le w.\ d_i = \begin{cases} 1 & ([b]_i = 1) \\ 0 & ([b]_i = 0, [b_2]_i = 1) \\ \top & (\text{otherwise}) \end{cases}$$

其他按位逻辑运算符的抽象算子也以类似方式定义。

- 对于 $\overleftarrow{\texttt{bvshl}}^{\#\hat{B},\ 1}(b,\ b_1,\ b_2)$，如果移位量是常数，我们可以通过将 b 向右移位（即反向移位）该移位量，推断出第一个操作数的抽象值。然而，一般情况下，移位量并非确切已知，因此我们应对其进行过近似。我们将 b 向右移位所有可能移位量所得的结果进行并运算。最小移位量为零，最大移位量为假设每个未知位（$\top$）均为 0 时 b 可得到的尾随零数量。通过考虑第二个操作数的抽象值，可以进一步细化此范围。形式化地，

$$\overleftarrow{\texttt{bvshl}}^{\#}_{\hat{B},1}(b,b_1,b_2) \quad = \quad \bigsqcup\{b >>^{\#}_{\top} (n \bmod w) \mid n \in \gamma_{\hat{U}}([0, \texttt{Trail0s}(b[0/\top])] \sqcap \pi_{\hat{B}\to\hat{U}}(b_2))\}$$

- $\overleftarrow{\texttt{bvmul}}^{\#\hat{B},\ 1}$ 是后向抽象算子中最复杂的，其定义如下：

$$\overleftarrow{\texttt{bvmul}}^{\#}_{\hat{B},1}(b,b_1,b_2) = \begin{cases} \texttt{InferMulOp}(b_2,b) & (|\gamma^{\text{unsigned}}_{\hat{B}}(b_2)| = |\gamma^{\text{unsigned}}_{\hat{B}}(b)| = 1) \\ \top^{w-l} \cdot 0^l & (\text{otherwise}) \end{cases}$$
$$\text{where } l = \max\{0, \texttt{Trail0s}(b) - \texttt{Trail0s}(b_2[0/\top])\}$$

其中InferMulOp 将在稍后定义。这个后向抽象算子在第二个操作数和结果确切已知时，可以精确推断第一个操作数的抽象值。假设第二个操作数和结果分别表示非负数 n_2 和n [2]。由于乘法是模 2^w运算，我们的目标是找到 x 使得

$$x \times n_2 \equiv n \pmod{2^w} \tag{1}$$

当 n_2 和 n 都不为零时，我们可以按如下方式求得 x（n_2 或 n 为零的情况显然成立）：设 b_2 和 b 的尾随零个数分别为 t_2 和 t。因为

[2]按位乘法不考虑操作数的有符号性。因此，可以安全地将操作数视为非负数。

$n_2 \neq 0 \wedge n \neq 0$、 $n_2 = 2^{t_2} \times m_2$和 $n = 2^t \times m$，其中 m_2和 m是奇数。我们分两种情况讨论。

– 情况 1） $t \geq t_2$：我们可以将方程（1）变换为 $x \times (n_2/2^{t_2}) \equiv (n/2^{t_2})$ （模 2^{w-t_2}），因为 $n_2/2^{t_2}$是奇数， $n_2/2^{t_2}$且 2^{w-t_2}互质。根据扩展欧几里得算法，我们可以求出 $n_2/2^{t_2}$模 2^{w-t_2}的乘法逆元。令 y为该模乘法逆元。那么，

$$
\begin{aligned}
& x \times (n_2/2^{t_2}) \times y \equiv (n/2^{t_2}) \times y \pmod{2^{w-t_2}} && \text{(multiplying } y \text{ into both sides)} \\
& x \equiv (n/2^{t_2}) \times y \pmod{2^{w-t_2}} \\
& x \times 2^{t_2} \equiv (n/2^{t_2}) \times y \times 2^{t_2} \pmod{2^w} && \text{(multiplying } 2^{t_2} \text{ into both sides)}
\end{aligned}
$$

总之，在 x的二进制表示中，最后 $w - t_2$ 位必须与 $n/2^{t_2} \times y$ 的二进制表示相同。 x 的前 t_2 位可以取任意值。

– 情况 2） $t < t_2$：在这种情况下，方程（1）无解。这是因为线性同余式（1）有解，当且仅当 n_2与 2^w的最大公约数（gcd）整除 n。 n_2 与 2^w的最大公约数为 2^{t_2}，由于 n，它无法整除 $t_2 > t$。

ab ove 情况分析通过如下定义的函数 InferMultOperand 实现：

$$
\mathsf{InferMulOp}(b_2, b) = \begin{cases} \top & (n_2 = n = 0) \\ 0 & (n_2 \neq 0, n = 0) \\ \top^{t_2} \cdot [p(n/2^{t_2} \times y)]_{w-t_2:w} & (n_2 \neq 0, n \neq 0, t \geq t_2) \\ \bot & (n_2 = 0 \wedge n \neq 0 \vee n_2 \neq 0 \wedge n \neq 0 \wedge t < t_2) \end{cases}
$$

其中 $n_2 = \gamma^{\text{unsigned}}{}_{\hat{B}}(b_2)$、 $n = \gamma^{\text{unsigned}}{}_{\hat{B}}(b)$、 $t = \mathsf{Trail0s}(b)$、 $t_2 = \mathsf{Trail0s}(b_2)$，且 y 是 $n_2/2^{t_2}$ 模 2^{w-t_2}的模乘法逆元。

如果第二个操作数和结果并非完全已知，为了对第一个操作数的抽象值进行过近似，我们只需利用以下事实：结果的尾随零数量等于第一个操作数和第二个操作数的尾随零数量之和。

4.3 布尔代数的抽象域

我们用于布尔代数的抽象域基于位向量的抽象域。该抽象域是集合 $B = \{0, 1, \bot, \top\}$，可以理解为按位域的一种变体，其中长度 $w = 1$。对于 and、or、xor、not 等布尔运算符，其抽象算子的定义方式与按位域中的定义方式相同。

5 评估

我们已将该方法实现为一个名为 Simba[3] 的工具，该工具包含 1 万行 OCaml 代码，并使用 Z3 [De Moura 和 Bjorner 2008] 作为约束求解引擎。我们的工具可公开下载[4]。

本节评估 Simba，以回答以下问题：

Q1： Simba 在来自各种不同应用领域的综合任务上表现如何？**Q2：** Simba 与现有综合技术相比表现如何？**Q3：** 与其他替代方法（例如使用 SMT 求解器或不进行剪枝）相比，Simba 中基于抽象解释的剪枝技术在缩减搜索空间方面有多有效？

我们的所有实验均在一台搭载 Intel Xeon 2.6GHz CPU 和 256GB 内存的 Linux 机器上运行。

[3]**基**于**归**纳式规范的**综**合，由**双**向**抽**象解释增强 [4]https://github.com/yhyoon/simba

5.1 实验设置

基准测试。 我们从三个不同的应用领域中选取了 **1,875** 个综合任务：i）不含条件语句的位向量操作（BitVec）；ii）含条件语句的位向量操作（BitVec-Cond）；以及 iii）电路变换（Circuit），

这些基准测试用于评估Duet 工具、先前关于程序去混淆的研究 [David 等人 2020]，以及年度 SyGuS 竞赛 [往届 SyGuS 竞赛2020]。

BitVec 域包含 544 个关于位向量算术背景理论的任务。所有这些问题的解都是无条件程序。

- HD：来自 SyGuS 竞赛套件（通用赛道）的 44 个基准测试。这些问题源自《黑客的喜悦》 [Warren 2012]，，这本书通常被誉为位运算技巧的圣经。语义规范是一个与目标程序在功能上等价的全称量化一阶公式。 [5]
- 反混淆：来自程序反混淆器评估基准测试的 500 个基准测试QSynth （其中的数据集“VR-EA”见 [David 等人，2020 2020]）。这些问题旨在根据输入输出示例，寻找与随机生成的位操作程序等价的程序；这些问题已用于评估最先进的反混淆器 [Blazytko 等人，2017年；David 等人，2020；Menguy 等人 2021]。由于混淆后的程序在语法上十分复杂，迄今为止的最佳实践是采用黑盒方法，从混淆后的程序中采样输入输出行为，再根据这些样本合成目标程序。按照这种方法，我们为每个混淆程序随机生成了 20 个输入输出示例。

BitVec-Cond 域包含 Duet 评估基准测试中的 750 个任务 [6]。这些问题要求根据数量从 10 到 1,000 不等的输入输出示例，寻找与随机生成的位操作程序等价的程序。与反混淆基准测试不同，这些问题的解通常包含条件语句。

来自 Duet 评估基准测试的电路领域包含 581 个基于可满足性理论的任务。

- Lobster：来自 [Lee 等人 2020] 的 369 个问题。这些问题的设计旨在优化同态求值电路。每个问题给定一个电路 C，要求合成一个电路C'，使其计算的函数与 C 相同，但其乘法深度更小，并且在功能上等价于 C。
- 密码学：SyGuS 竞赛中使用的 212 个问题，灵感来自针对嵌入式系统中密码模块的侧信道攻击。每个问题都要求给定一个电路 C，合成一个恒定时间电路 C'（即能够抵御时序攻击），其计算的函数与 C相同。

基准求解器。 我们将 Simba 与现有的通用综合工具进行比较，这些工具在某种程度上采用了领域专门化。我们的算法通常适用于任何领域，但需要针对目标问题采用合适的抽象域，以进一步提高效率。因此，我们将 Simba 与以下采用某种领域专门化方法的通用工具进行比较：

[5]我们对原始基准测试进行了略微修改：原始基准测试针对 32 位整数，现已改为针对 64位整数。这是为了与 Probe进行公平比较，因为 Probe只能处理64 位整数。[6]这些基准测试是对 SyGuS 竞赛套件（PBE-BitVector 赛道）的略微修改。每个问题中的语法限制都被替换为更通用的文法。

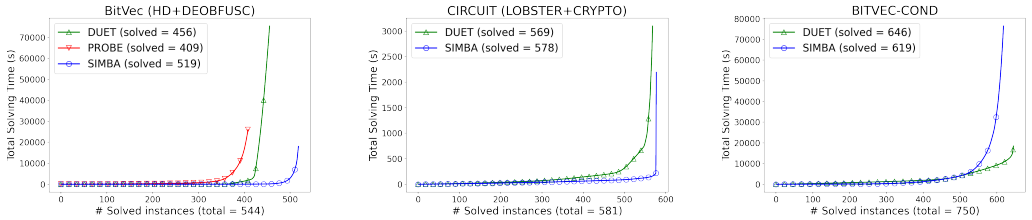


图 2. Simba 与其他基准求解器在不同领域上的比较。

- Duet 是用于归纳式 SyGuS 问题的最先进工具，它采用一种结合了领域专门化技术的双向搜索策略，该技术称为 自顶向下传播。自顶向下传播是一种分治策略，通过递归方式将给定的程序合成问题分解为多个子问题。它需要使用逆语义运算符，这些运算符应针对目标语言中每个可用的运算符进行设计。
- Probe 采用自底向上的搜索，并由概率模型提供指导。这种概率模型可以在搜索过程中即时学习，通过从搜索过程中遇到的部分解中学习得到。因此，这种模型可以视为针对每个问题实例进行领域专门化的结果。

我们将Simba 与Duet 在所有基准测试上进行比较，并将Probe 仅在BitVec 领域上进行比较，因为电路和BitVec-Cond 领域超出了Probe[7]。

5.2 Simba 的有效性

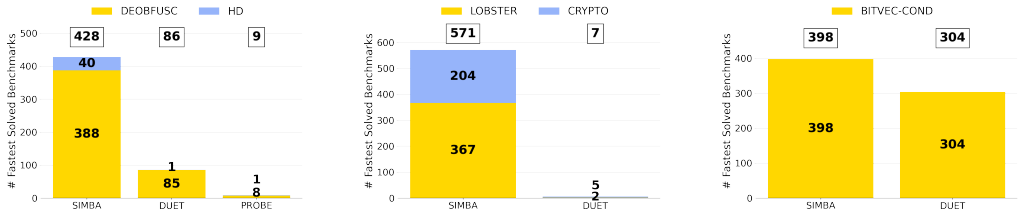
我们在所有基准测试上评估 Simba，并将其与 Duet 和 Probe 进行比较。对于每个实例，我们测量综合的运行时间和综合程序的大小，超时时间设为一小时。

结果汇总见图3。图3d 展示了求解时间和解大小的统计数据，图 3a 和 3b 展示了各求解器在每个领域中以最快时间求解的实例数量。由于 Probe 不适用于 电路 和 BitVec-Cond 领域，因此未展示Probe 在这些领域中的结果。

总体而言，Simba 在求解实例数量和平均求解时间方面均优于其他基准工具。如图 3d 所示，Simba 求解了 1716 个实例，而 Duet和Probe 分别求解了 1671 个和 409 个实例。Simba 在 1397 个实例（总数的 75%）中是最快的求解器，而 Duet 在 397 个实例中最快。图 2展示了 Simba、Duet和Probe 的求解时间仙人掌图。横轴表示已求解实例的数量，纵轴表示累计求解时间。该图表明，Simba在更短的时间内求解的实例多于Duet 和Probe 。

与 Duet 相比，Simba 在 BitVec 域和电路域中效率更高，而 Duet 在 BitVec-Cond 域中效率更高。由于 Simba 会针对每个输入-输出示例执行前向-后向分析，示例数量会影响其效率。在 BitVec-Cond 域中，示例数量异常庞大（最多达 1000 个），这使得 Simba 效率低下。另一方面，Duet 能够高效处理大量示例。然而，在其他域（尤其是 BitVec 域）中，Simba 的表现优于 Duet，因为 Duet 的逆向语义常常会产生难以求解的子问题，而 Simba 不会生成此类子问题。

[7]BitVec-Cond 实例的解是规模庞大的条件程序，需要大量分类讨论，而 Probe 不支持这种操作。



(a) BitVec 域中求解速度最快的数量 (b) Circuit 域中求解速度最快的数量 (c) BitVec- 域中求解速度最快的数量 Cond 域

基准测试类别	已解决数量			时间（平均值）			时间（中位数）			规模（平均值）			规模（中位数）		
	Simba	DUET	Probe	S	D	P	S	D	P	S	D	P	S	D	P
HD	42	36	31	5.7	42.7	11.8	0.1	0.2	1.0	7.7	8.5	6.5	6	8	5
反混淆	477	420	378	37.3	175.8	68.0	0.0	0.1	2.9	9.6	10.0	7.9	9	9	8
BitVec-Cond	619	646	-	123.6	28.6	-	5.3	5.7	-	269.6	350.6	-	137	46	-
Lobster	369	369	-	0.4	7.4	-	0.2	0.8	-	12.9	10.9	-	13	11	-
密码学	209	200	-	9.7	1.8	-	0.1	0.2	-	11.9	12.8	-	11	12	-
总体	**1716**	**1671**	**409**	**56.4**	**58.0**	**63.7**	**0.2**	**1.7**	**2.7**	**107.1**	**145.4**	**7.8**	**15**	**13**	**8**

（d）求解时间和解大小的统计数据。所有时间均以秒为单位。Probe 无法在电路域和 BitVec-Cond 域中运行。HD、Deobfusc、Lobster、Crypto 和 BitVec-Cond 类别中的问题数量分别为 **44**、**500**、**369**、**212** 和 **750** 个（总计 **1875** 个）。

图 3. Simba、Duet 和 Probe 性能对比的主要结果（按类别细分）。超时时间设为一小时。

我们通过 AST 节点数量来衡量解的质量。根据奥卡姆剃刀，较小的解更好，因为它们不太可能对输入输出示例过拟合。总体而言，Probe 生成的解最小（两个领域的平均大小分别为 6.5 和 7.9），不过 Simba 也能生成大小相近的解（平均大小分别为 7.7 和 9.6）。这两种工具之间的微小差距可归因于 Simba 能够解决 Probe 因可扩展性有限而无法解决的问题。例如，对于 Probe 未能解决的两个领域，Simba 生成的解的平均大小分别为 10.5 和 13.1；而对于两种工具都能解决的问题，Simba 生成的解的平均大小分别为 6.8 和 8.6。

详细结果。我们逐一详细研究各个领域的结果。表1 展示了随机选取的 25 个问题（每个类别 5 个）的详细结果。结果表明，前向和后向分析产生了显著影响。例如，对于 hd-20-d1-prog，Simba 在综合过程中以仅 0.22 秒的开销，丢弃了生成的 11,365 个部分程序（总计 11,710 个，占 97.1%）。对于 target-410，Simba 以可接受的 3.06 秒开销，提前剪枝了 188,501 个部分程序中的 187,858 个（占 99.7%）。此外，我们观察到，即使部分程序未被丢弃，其空缺也会受到必要前置条件的严格约束，从而大幅减少补全部分程序时需要探索的组件数量。另一方面，Duet 和 Probe 不执行此类剪枝，因此会生成许多不太可能的候选项，耗时是 Simba 的 10 到 1000 倍。

示例数量的影响分析。我们研究了示例数量对 Simba 性能的影响。由于前向-后向分析会按示例数量执行相同次数，因此示例数量会影响效率；但在实践中，只要示例数量不太大，其对效率的影响并不显著。

表1. 25 个随机选取的基准问题的结果（每个类别 5 个），其中 **综合时间** 表示综合所用时间， T_A 表示正向和反向分析所花费的时间， $|P|$ 表示综合程序的大小（以 AST 节点数量衡量）。

基准测试类别	基准测试	Probe		Duet		Simba		
		Time	$\|P\|$	**Time**	$\|P\|$	**Time**	T_A	$\|P\|$
HD	hd-03-d5-prog	0.85	4	1.19	4	0.06	0.00	4
	hd-07-d0-prog	0.92	6	0.09	6	0.07	0.00	6
	hd-14-d5-prog	4.91	9	>1h	-	0.60	0.26	9
	hd-19-d1-prog	>1h	-	>1h	-	10.79	7.07	19
	hd-20-d1-prog	>1h	-	228.98	15	98.22	0.22	15
反混淆	目标_9	>1h	-	2310.23	16	31.45	2.85	16
	目标_119	0.87	5	0.02	8	0.05	0.03	9
	目标_385	15.35	9	37.46	10	0.33	0.25	9
	目标_410	>1h	-	2635.78	16	69.38	3.06	16
	目标_449	2.61	7	0.01	7	0.03	0.00	7
BitVec-Cond	133_1000	-	-	48.18	35	166.55	6.89	139
	23_10	-	-	340.75	62	0.89	0.40	15
	60_100	-	-	6.90	14	0.15	0.10	14
	icfp_gen_10.7	-	-	4.68	67	1.19	0.20	192
	icfp_gen_14.1	-	-	6.29	154	49.61	37.75	415
Lobster	hd09.eqn_45_0	-	-	21.95	15	11.09	0.32	13
	longest_1位最优.eqn_63_1	-	-	0.59	13	0.17	0.01	11
	longest_1位最优.eqn_75_1	-	-	1.31	14	0.24	0.01	15
	p03.eqn_38_2	-	-	0.22	9	0.11	0.01	9
	p09.eqn_49_1	-	-	0.24	7	0.19	0.01	9
密码学	CrCy_2-P6_2-P6	-	-	193.68	20	0.65	0.07	20
	CrCy_5-P9-D5-sIn	-	-	0.13	11	0.11	0.00	11
	CrCy_8-P12-D5-sIn4	-	-	0.21	11	0.14	0.01	11
	CrCy_8-P12-D7-sIn5	-	-	1.99	19	0.63	0.04	19
	CrCy_10-sbox2-D5-sIn11	-	-	0.28	9	0.13	0.00	9

我们使用了 500 个反混淆基准测试进行实验，示例数量从 5 到 20 不等。当提供给 Simba 的示例数量分别为 5、10 和 20 时，平均综合时间分别为 29.8 秒、31.0 秒和 37.3 秒。在所有情况下，平均解的大小均为 9.5。换言之，示例数量增加四倍（从 5 增至 20）时，平均综合时间仅增加了 25%。

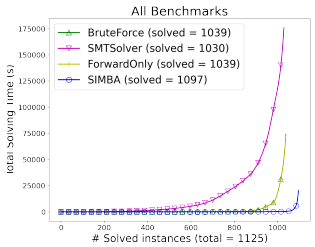
结果总结。与不同领域中最先进的基准工具相比，Simba 能够更快速地解决难度更高的程序合成问题。

5.3 我们基于抽象解释的剪枝效果

现在我们评估基于抽象解释的剪枝效果。为此，我们比较 Simba 的三个变体的性能，每个变体采用不同的组合：

- 使用前向和后向分析的 Simba（即原始 Simba）
- *ForwardOnly* 仅使用前向分析
- *BruteForce* 不使用任何剪枝技术（即仅使用双向搜索策略） [8]
- *SMT* 求解器 结合基于可满足性模理论的剪枝。它通过检查可满足性模理论公式的可满足性来判断每个部分程序的可行性。该公式对正在构建的部分程序以及程序期望的输入输出行为进行编码。类似的

[8]尽管禁用了剪枝，但其他优化（如对称性破除和观察等价约简）仍然启用。



(a) Simba 不同变体的性能

基准测试类别	解出数量				时间（平均值）			
	S	**FW**	**SMT**	**BF**	**S**	**FW**	**SMT**	**BF**
HD	42	41	40	41	5.7	12.9	163.6	13.4
反混淆	477	421	424	421	37.3	128.8	178.7	131.2
Lobster	369	368	358	368	0.4	43.7	234.9	43.5
密码学	209	209	208	209	9.7	11.8	45.7	11.9
总体	**1097**	**1039**	**1030**	**1039**	**18.4**	**70.5**	**170.8**	**71.5**

（b）求解时间的统计数据。**S**、**FW**、**可满足性模理论**和**暴力搜索**分别表示 Simba、*ForwardOnly*、*SMT* 求解器和 *BruteForce* 变体。

图 4. 消融研究结果。

该方法被用于Morpheus [Feng 等人 2017] 中，以剪枝搜索空间。由于 SMT 求解不涉及任何近似，因此预计其准确性高于基于抽象解释的剪枝，但代价是更高的计算成本。

通过比较这些变体的性能，我们旨在了解

- 基于抽象解释的剪枝的总体影响（Simba 与 *BruteForce* 的比较）
- 后向分析的必要性（Simba 与 *ForwardOnly* 及 *BruteForce* 的比较）
- 使用抽象解释器的成本效益（*ForwardOnly* 与 *SMT* 求解器 的比较）

图 4 展示了消融研究的结果（图 4b展示求解时间的统计数据，图 4a 展示仙人掌图）。为专注于评估我们方法的核心思想，我们排除了 BitVec-Cond 基准测试；这些测试需要大量分类拆分，而分治策略可以处理这类拆分 [Alur 等人 2017] 。

第一个观察结果是，我们基于抽象解释的剪枝能够有效缩小搜索空间（Simba 求解了 1097 个，而 *BruteForce* 求解了 1039 个）。第二个观察结果是，后向分析对于有效剪枝搜索空间是必不可少的，因为 *BruteForce*和 *ForwardOnly* 的表现几乎相同，而 Simba 的表现远优于 *ForwardOnly*。第三个观察结果是，使用 SMT 求解器比使用抽象解释器的成本更高（*ForwardOnly* 求解了 1039 个，而 *SMTSolver*求解了 1030 个）。尽管 SMT 求解器比抽象解释器具有更高的精度，但 SMT 求解器的剪枝成功率不足以抵消计算成本的增加。因此，使用抽象解释器能够在精度与计算成本之间取得良好的平衡。以基准测试 hd-14-d1 为例，采用*SMTSolver* 变体时，SMT 求解器被调用了 2,247 次，用于剪枝 1,091 个部分程序。这一过程耗费了 49.08 秒，而求解该基准测试的总耗时为 52.33 秒。另一方面，使用 Simba 时，前向-后向分析被调用了 1,230 次，用于剪枝 1,127 个部分程序。这一过程仅耗费了 0.06 秒，而求解该基准测试的总耗时为 0.15 秒。这表明使用抽象解释器具有较高的成本效益。

结果总结。前向和后向分析对于有效剪枝搜索空间都是必要的。此外，使用SMT求解器比使用抽象解释器成本更高。

6 相关工作

基于抽象解释的综合。 以往大多数基于抽象解释的程序综合剪枝方法只采用前向抽象解释 [Feng 等人 2017； Singh 和 Solar-Lezama 2011；So 和 Oh 2017； Vechev 等人 2010； Wang 等人 2017年a，b]。与这些方法不同，我们的技术结合前向推理和后向推理，为缺失表达式推导必要前置条件，并利用这些条件有效地剪枝搜索空间。

Pailoor 等人 [2021] 通过后向推理来加速综合，检查每个候选程序是否满足必要前置条件。在他们的设定中，部分程序 P 的必要前置条件是对 P 的输入的约束；任何对 P 的补全都必须满足该约束，才能使 P 满足针对所需新数据结构给定的约束。如果任何输入使该条件不成立，就会丢弃该部分程序。可以通过标准的最弱前置条件方法计算这样的必要前置条件。然而，前向推理和后向推理并未协同结合，因此剪枝能力可能受到限制。

据我们所知，Mukherjee 等人 [2020] 是唯一一项同时利用前向和后向抽象解释来剪枝程序合成搜索空间的先前研究。然而，该研究缺少前向和后向分析的协同结合。在这项工作中，前向和后向分析是分别进行的，二者之间没有任何交互。此外，他们的工作仅限于 LLVM 超优化，而我们的工作通过针对 SyGuS 规范语言，适用于范围广泛的归纳式程序合成问题。

与我们在整个程序合成过程中使用固定抽象域的工作不同，另一类研究基于抽象细化 [Guo 等人 2019年；Vechev 等人 2010年；Wang 等人 2018，2017b]。在这些方法中，不满足规范的程序会被用于迭代细化抽象域，直至找到解。Blaze [Wang 等人 2017b] 及其扩展 [Wang 等人 2018] 能够根据给定的谓词模板和训练用程序合成问题集，自动学习谓词抽象域。当没有专家充分了解目标应用领域时，这种方法很有用。我们的工作表明，无需进行抽象细化，高精度的抽象解释抽象就能显著提升程序合成性能。我们预计，采用前向和后向分析这一核心思想，也可以应用于基于抽象细化的方法，进一步提升程序合成性能。

迭代前向/后向分析。前向和后向静态分析的结合，最早由 [Cousot 1978], 引入，后来在程序验证的背景下得到研究 [Cousot 和 Cousot 1992年；Dimovski 和 Legay 2020年；Kafle 和 Gallagher 2015年；Kana- gasabapathi 和 Thushara 2020年；Massé 2001], 模型检测 [Cousot 和 Cousot 1999],针对不满足规范的情况生成反例 [Yin 2019年；Yin 等人 2019], 以及过滤静态分析中的伪警报 [Rival 2005]。通常，迭代前向和后向分析可以提高分析精度，但代价是增加分析时间。

据我们所知，我们的工作首次将迭代 前向和后向分析用于归纳式综合。我们的关键发现是，同时采用前向和后向推理的高精度分析能够提高剪枝成功率，其提升足以抵消分析时间的增加。

位向量算术的抽象域。关于位向量算术抽象域的研究成果非常丰富。下面我们简要介绍其中几种。Miné [2012]和Regehr 和 Duongsaa [2006] 提出结合区间域和按位域的方法。我们的一些前向抽象传递函数借鉴了这些工作。包装区间域 [Gange 等人 2015] 通过将位向量值环绕到最小和最大的位向量值之间，可以精确跟踪溢出和下溢的影响。Sharma 和 Reps [2017] 提出了一个框架，用于将整数上的数值抽象域转换为位向量域。Simon 和 King [2007] 提出了一个适用于多面体的环绕运算符，用于跟踪环绕效应。

据我们所知，现有的位向量抽象域都没有提供反向抽象传递函数。正如我们的实验已经表明的，反向抽象解释对于搜索空间的剪枝至关重要。因此，要在我们的方法中使用现有抽象域，就需要为它们开发反向抽象传递函数。

归纳综合的领域专门化。 近期研究表明，通过利用各种形式的领域知识（例如领域专用语言），可以显著提升性能 [Gulwani 2011年； Kini 和 Gulwani 2015年； Rolim 等人 2017], 概率模型 [Barke 等人，2020； Lee 等人 2018], 逆语义 [Lee 2021], 和模板 [Inala 等人 2016]。特别是，Duet [Lee 2021]将双向搜索与专门的 逆语义（也称为见证函数）相结合，这些逆语义返回能够生成给定输出的所有可能输入集合。我们的工作将双向搜索与抽象解释相结合。实验表明，高度精确的抽象语义能够带来显著的性能提升，并且这种提升与其他领域专门化方法（如逆语义和概率模型）所实现的性能提升具有互补性；将我们的方法融入先前的方法中也很有前景。

7 未来工作

我们方法的一种可能扩展是支持 SyGuS 中的其他理论，例如整数算术和字符串理论。由于单一抽象域的精度不足，因此需要使用多个抽象域，包括前向/后向抽象传递函数，以及用于定义这些抽象域约简积的约简算子。对于整数算术，可以使用现有非关系抽象域的约简积，例如区间域 [Cousot 和 Cousot 1977]以及同余域 [Granger 1989] 提出的域。除此之外，关系域 [Miné2006; Singh 等人 2017] 有助于跟踪不同程序空缺与输入变量之间的关系。对于字符串，可以使用基于前缀、后缀和简单正则表达式的抽象域 [Costantini 等人 2011], 提出的下推自动机 [Kim 和 Choe 2011],以及解析栈 [Doh 等人 2009] 提出的方法。另一种可能的扩展是合成带循环的程序。与当前无循环的设定相比，前向和后向分析可能需要高级拓宽/收窄算子（例如，使用推断阈值进行拓宽 [Lakhdar-Chaouch 等人2011]），以加快不动点迭代的收敛，同时保持精度。

8 结论

我们提出了一种新颖的程序综合算法，通过结合前向与后向抽象解释，有效地剪枝搜索空间。我们的实现 Simba 及其评估结果表明，其性能显著优于现有最先进的综合系统 Duet 和 Probe。要实现归纳式程序合成的高性能与可扩展性，关键在于将前向抽象解释与后向抽象解释相结合，从而快速缩小搜索空间。我们的实验还表明，使用可满足性模理论求解器来代替这类复杂的静态分析，无法实现良好的可扩展性。

致谢

我们感谢审稿人提出的宝贵意见。本研究获得了信息通信技术企划评价院（2022-0-00995）、韩国国家研究基金会（2020R1C1C1014518, 2021R1A5A1021944）、韩国大检察厅（科学技术信息通信部资助，0536-20220043）、BK21 FOUR 智能计算（首尔大学计算机科学与工程系）（4199990214639，韩国政府（科学技术信息通信部）资助）、Sparrow Co., Ltd.、三星电子（IO220411-09496-01）、Greenlabs（0536-20220078）和 Cryptolab（0536-20220081）的支持

D数据可用性声明。该研究成果可在 Zenodo 获取[Yoon 等人 2023]。

参考文献

Rajeev Alur、Rastislav Bodik、Garvit Juniwal、Milo M. K. Martin、Mukund Raghothaman、Sanjit A. Seshia、Rishabh Singh、Armando Solar-Lezama、Emina Torlak 和 Abhishek Udupa。2013年。语法引导合成。载于计算机辅助设计中的形式化方法（FMCAD '13）。Rajeev Alur、Arjun Radhakrishna 和 Abhishek Udupa。2017年。通过分治法扩展枚举式程序综合。载于《系统构造与分析工具和算法》，Axel Legay 和 Tiziana Margaria（编）。Springer Berlin Heidelberg，柏林，海德堡，319–336。Shraddha Barke、Hila Peleg 和 Nadia Polikarpova。2020年。用于自底向上枚举综合的即时学习。ACM编程语言会议录 4，OOPSLA，论文编号227（2020年11月），29页。https://doi.org/10.1145/3428295 Clark W. Barrett、Aaron Stump 和 Cesare Tinelli。2010年。SMT-LIB标准2.0版。Tim Blazytko、Moritz Contag、Cornelius Aschermann 和 Thorsten Holz。2017年。Syntia：混淆代码语义综合。在第26届USENIX安全研讨会（USENIX Security 17）上。USENIX协会，温哥华，不列颠哥伦比亚省，643–659。https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/blazytko Giulia Costantini、Pietro Ferrara 和 Agostino Cortesi。2011年。字符串值的静态分析。载于《形式化方法与软件工程》，Shengchao Qin 和 Zongyan Qiu（编）。Springer Berlin Heidelberg，柏林，海德堡，505–521。Patrick Cousot。1978年。单调算子在格上不动点的迭代构造与近似方法：程序语义分析。博士生导师资格认证。格勒诺布尔国立理工学院 - INPG；约瑟夫·傅里叶大学 - 格勒诺布尔第一大学。https://tel.archives-ouvertes.fr/tel-00288657 大学：格勒诺布尔科学与医学大学和格勒诺布尔国立理工学院。Patrick Cousot。2021年。抽象解释原理。麻省理工学院出版社。Patrick Cousot 和 Radhia Cousot。1977年。抽象解释：通过构造或近似不动点，为程序静态分析建立统一的格模型。载于ACM SIGPLAN-SIGACT编程语言原理研讨会论文集。238–252。Patrick Cousot 和 Rahida Cousot。1992年 13，2 3 1992年 103 179 10 1016 0743 1066 92 90030 7。抽象解释及其在逻辑程序中的应用。《逻辑程序杂志》–（7月），–。https://doi.org/ . / - ( ) - Patrick Cousot 和 Radhia Cousot。1999 年。通过抽象解释精化模型检测。《自动化软件工程》 6，1 （1999年），69–95。https://doi.org/10.1023/A:1008649901864 Robin David、Luigi Coniglio 和 Mariano Ceccato。2020年。QSynth——一种基于程序综合的二进制代码去混淆方法。2020年二进制分析研究研讨会论文集（2020年）。Leonardo De Moura 和 Nikolaj Bjorner。2008年。Z3：一种高效的SMT求解器。载于《软件理论与实践》，第14届系统构造与分析工具和算法国际会议（匈牙利布达佩斯）（TACAS'08）论文集。施普林格出版社，柏林，海德堡，337–340。Aleksandar S. Dimovski 和 Axel Legay。2020年。利用前向—后向前置条件分析和模型计数计算程序可靠性。载于《软件工程基础方法》，Heike Wehrheim 和 Jordi Cabot（编）。Springer International Publishing，Cham，182–202。Kyung-Goo Doh、Hyunha Kim 和 David A. Schmidt。2009年。抽象解析：使用LR解析技术对动态生成的字符串输出进行静态分析。载于第16届静态分析国际研讨会论文集（洛杉矶，加利福尼亚州）（SAS '09）。施普林格出版社，柏林，海德堡，256–272。https://doi.org/10.1007/978-3-642-03237-0_18 Yu Feng、Ruben Martins、Jacob Van Geffen、Isil Dillig 和 Swarat Chaudhuri。2017年。基于组件的综合：从示例中综合表格整合与转换任务。SIGPLAN通知 52，6（2017年6月），422–436。https: //doi.org/10.1145/3140587.3062351 Graeme Gange、Jorge A. Navas、Peter Schachte、Harald Sondergaard 和 Peter J. Stuckey。2015年。区间分析与机器算术：为何忽略有符号性有益。《ACM编程语言系统汇刊》37，1，论文编号1（2015年1月），35页。https://doi.org/10.1145/2651360 Philippe Granger。1989年。算术同余的静态分析。《国际计算数学杂志》30，3-4（1989年），165–190。https://doi.org/10.1080/00207168908803778 arXiv:https://doi.org/10.1080/00207168908803778 Philippe Granger。1992年。通过局部递减迭代改进程序静态分析的结果。载于《软件技术与理论计算机科学基础》，Rudrapatna Shyamasundar（编）。Springer Berlin Heidelberg，柏林，海德堡，68–79。Sumit Gulwani。2011年。利用输入输出示例自动化电子表格中的字符串处理。载于第38届ACM SIGPLAN-SIGACT编程语言原理年度研讨会论文集（美国得克萨斯州奥斯汀）（POPL '11）。Zheng Guo、Michael James、David Justo、Jiaxiao Zhou、Ziteng Wang、Ranjit Jhala 和 Nadia Polikarpova。2019年。通过类型引导的抽象精化进行程序综合。ACM编程语言会议录 4，POPL，论文编号12（2019年12月），28页。https://doi.org/10.1145/3371080

Qinheping Hu、John Cyphert、Loris D'Antoni 和 Thomas Reps。2020。《证明语法引导综合问题不可实现性的精确与近似方法》。载于第 41 届 ACM SIGPLAN 编程语言设计与实现会议论文集（英国伦敦）（PLDI 2020）。计算机协会，美国纽约州纽约，1128–1142。https://doi.org/10.1145/3385412.3385979 Jeevana Priya Inala、Rohit Singh 和 Armando Solar-Lezama。2016。《位向量求解器的领域专用 CNF 编码器综合》。载于《可满足性测试理论与应用——SAT 2016》，Nadia Creignou 和 Daniel Le Berre（编）。Springer International Publishing，Cham，302–320。Bishoksan Kafle 和 John P. Gallagher。2015。《霍恩子句验证中的约束特化》。载于 2015 年部分求值与程序操纵研讨会论文集（印度孟买）（PEPM '15）。计算机协会，美国纽约州纽约，85–90。https://doi.org/10.1145/2678015.2682544 Somasundaram Kanagasabapathi 和 M. G. Thushara。2020。《浮点程序关键数值精度的正向与反向静态分析》。Comput. Sci. 21（2020）。Se-Won Kim 和 Kwang-Moo Choe。2011。《作为抽象解释的字符串分析》。载于《验证、模型检测与抽象解释》，Ranjit Jhala 和 David Schmidt（编）。Springer Berlin Heidelberg，柏林，海德堡，294–308。Dileep Kini 和 Sumit Gulwani。2015。《FlashNormalize：用于文本规范化的示例驱动编程》。载于第 24 届国际人工智能会议论文集（阿根廷布宜诺斯艾利斯）（IJCAI'15）。AAAI Press，776–783。Lies Lakhdar-Chaouch、Bertrand Jeannet 和 Alain Girault。2011。《面向复杂控制流图程序的带阈值扩宽》。载于《验证与分析自动化技术》，Tevfik Bultan 和 Pao-Ann Hsiung（编）。Springer Berlin Heidelberg，柏林，海德堡，492–502。DongKwon Lee、Woosuk Lee、Hakjoo Oh 和 Kwangkeun Yi。2020。《通过程序综合与项重写优化同态求值电路》。载于第 41 届 ACM SIGPLAN 编程语言设计与实现会议论文集（英国伦敦）（PLDI 2020）。计算机协会，美国纽约州纽约，503–518。https://doi.org/10.1145/3385412.3385996 Woosuk Lee。2021。《结合自顶向下传播与自底向上枚举的归纳式程序综合》。ACM 编程语言学报 5，POPL（2021），1–28。Woosuk Lee、Kihong Heo、Rajeev Alur 和 Mayur Naik。2018。《利用学习得到的概率模型加速基于搜索的程序综合》。载于第 39 届 ACM SIGPLAN 编程语言设计与实现会议论文集（美国宾夕法尼亚州费城）（PLDI 2018）。计算机协会，美国纽约州纽约，436–449。https://doi.org/10.1145/3192366.3192410 Damien Massé。2001。《结合时序属性的前向与后向分析》。载于《程序作为数据对象》，Olivier Danvy 和 Andrzej Filinski（编）。Springer Berlin Heidelberg，柏林，海德堡，103–116。Grégoire Menguy、Sébastien Bardin、Richard Bonichon 和 Cauim de Souza Lima。2021。《基于搜索的局部黑盒去混淆：理解、改进与缓解》。载于 2021 年 ACM SIGSAC 计算机与通信安全会议论文集（线上活动，韩国）（CCS '21）。计算机协会，美国纽约州纽约，2513–2525。https://doi.org/10.1145/3460120.3485250 Antoine Miné。2006。《八边形抽象域》。Higher-Order and Symbolic Computation 19，1（2006），31–100。https: //doi.org/10.1007/s10990-006-8609-1 Antoine Miné。2012。《位级机器整数与浮点运算的抽象域》。载于 WING'12——第 4 届不变式生成国际研讨会。英国曼彻斯特，16。https://hal.archives-ouvertes.fr/hal- 00748094 Manasij Mukherjee、Pranav Kant、Zhengyang Liu 和 John Regehr。2020。《基于数据流的剪枝：加速超优化》。ACM编程语言会议录 4，OOPSLA，文章 177（2020年11月），24 页。https://doi.org/10.1145/3428245 Shankara Pailoor、Yuepeng Wang、Xinyu Wang 和 Isil Dillig。2021。《从完整性约束综合数据结构细化》。载于第 42 届 ACM SIGPLAN 编程语言设计与实现国际会议论文集（线上，加拿大）（PLDI 2021）。计算机协会，美国纽约州纽约，574–587。https://doi.org/10.1145/3453483.3454063 往届 SyGuS 竞赛。2020。https://sygus.org/comp/。Nadia Polikarpova、Ivan Kuraj 和 Armando Solar-Lezama。2016。《从多态细化类型进行程序综合》。载于第 37 届 ACM SIGPLAN 编程语言设计与实现会议论文集（美国加利福尼亚州圣巴巴拉）（PLDI '16）。计算机协会，美国纽约州纽约，522–538。https: //doi.org/10.1145/2908080.2908093 John Regehr 和 Usit Duongsaa。2006。《推导用于分析嵌入式软件的抽象传递函数》。载于 2006 年 ACM SIGPLAN/SIGBED 嵌入式系统语言、编译器与工具支持会议论文集（加拿大安大略省渥太华）（LCTES '06）。计算机协会，美国纽约州纽约，34–43。https://doi.org/10.1145/1134650.1134657 Xavier Rival。2005。《理解 Astrée 中警报的来源》。载于《静态分析》，Chris Hankin 和 Igor Siveroni（编）。Springer Berlin Heidelberg，柏林，海德堡，303–319。

Xavier Rival 和 Kwangkeun Yi。2020年。《静态分析导论：抽象解释视角》。麻省理工学院出版社。Reudismam Rolim、Gustavo Soares、Loris D'Antoni、Oleksandr Polozov、Sumit Gulwani、Rohit Gheyi、Ryo Suzuki 和 Björn Hartmann。2017年。从示例中学习语法程序转换。载于第 39 届国际软件工程会议论文集（阿根廷布宜诺斯艾利斯）（ICSE'17）。IEEE出版社，404–415。https://doi.org/10.1109/ ICSE.2017.44 Tushar Sharma 和 Thomas Reps。2017年。可靠的位精确数值域。载于《验证、模型检测与抽象解释》，Ahmed Bouajjani 和 David Monniaux（编）。Springer International Publishing，Cham，500–520。Axel Simon 和 Andy King。2007年。驯服整数算术的回绕。载于《静态分析》，Hanne Riis Nielson 和 Gilberto Filé（编）。Springer Berlin Heidelberg，柏林，海德堡，121–136。Gagandeep Singh、Markus Püschel 和 Martin Vechev。2017年。快速多面体抽象域。载于第 44 届 ACM SIGPLAN 编程语言原理研讨会论文集（法国巴黎）（POPL '17）。计算机协会，纽约，美国纽约州，46–59。https://doi.org/10.1145/3009837.3009885 Rishabh Singh 和 Armando Solar-Lezama。2011年。从故事板合成数据结构操作。载于第 19 届 ACM SIGSOFT 研讨会暨第 13 届欧洲软件工程基础会议论文集（匈牙利塞格德）（ESEC/FSE '11）。计算机协会，纽约，美国纽约州，289–299。https://doi.org/10.1145/2025113.2025153 Sunbeom So 和 Hakjoo Oh。2017年。在静态分析引导下从示例合成命令式程序。载于《静态分析》，Francesco Ranzato（编）。Springer International Publishing，Cham，364–381。Armando Solar-Lezama、Liviu Tancau、Rastislav Bodik、Sanjit Seshia 和 Vijay Saraswat。2006年。有限程序的组合式草图。载于第 12 届国际编程语言与操作系统体系结构支持会议论文集（美国加利福尼亚州圣何塞）（ASPLOS XII）。Ma[rtin Vechev、Eran Yahav 和 Greta Yorsh。2010年。抽象引导的同步合成。载于第 37 届 ACM SIGPLAN-SIGACT 编程语言原理年度研讨会论文集（西班牙马德里）（POPL '10）。计算机协会，纽约，美国纽约州，327–338。https://doi.org/10.1145/1706299.1706338 Chenglong Wang、Alvin Cheung 和 Rastislav Bodik。2017年a。从输入输出示例合成高表达力的 SQL 查询。载于第 38 届 ACM SIGPLAN 编程语言设计与实现会议论文集（西班牙巴塞罗那）（PLDI 2017）。计算机协会，纽约，美国纽约州，452–466。https://doi.org/10. 1145/3062341.3062365 Xinyu Wang、Greg Anderson、Isil Dillig 和 K. L. McMillan。2018年。学习用于程序合成的抽象。载于《计算机辅助验证》，Hana Chockler 和 Georg Weissenbacher（编）。Springer International Publishing，Cham，407–426。Xinyu Wang、Isil Dillig 和 Rishabh Singh。2017年b。使用抽象精化进行程序合成。ACM编程语言会议录 2，POPL，文章 63（2017年12月），30 页。https://doi.org/10.1145/3158151 Henry S. Warren。2012。《黑客的喜悦》（第 2 版）。培生 Addison-Wesley 专业出版社。Banghu Yin。2019年。通过迭代抽象解释进行面向性质的验证。载于第 41 届国际软件工程会议：伴随论文集（加拿大魁北克省蒙特利尔）（ICSE '19）。IEEE出版社，162–164。https://doi.org/10.1109/ICSE-Companion.2019.00067 Banghu Yin、Liqian Chen、Jiangchao Liu、Ji Wang 和 Patrick Cousot。2019年。通过迭代抽象测试验证数值程序。载于《静态分析：第 26 届研讨会，SAS 2019，葡萄牙波尔图，2019年10月8日至11日，论文集》（葡萄牙波尔图）。施普林格出版社，柏林，海德堡，247–267。https://doi.org/10.1007/978-3-030-32304-2_13 Yongho Yoon、Woosuk Lee 和 Kwangkeun Yi。2023年。通过迭代式前向-后向抽象解释进行归纳式程序合成的制品。https://doi.org/10.5281/zenodo.7816533

收稿日期：2022年11月10日；录用日期：2023年3月31日