北京大学

博士研究生学位论文

题目：**算法范式制导的程序合成方法研究**

姓　　名：	吉如一
学　　号：	2001111323
院　　系：	计算机学院
专　　业：	计算机软件与理论
研究方向：	程序合成
导　　师：	胡振江 教授
	熊英飞 副教授

二〇二五 年 四 月

版权声明

任何收存和保管本论文各种版本的单位和个人，未经本论文作者同意，不得将本论文转借他人，亦不得随意复制、抄录、拍照或以任何方式传播。否则，引起有碍作者著作权之问题，将可能承担法律责任。

摘要

算法设计是提升软件运行效率的核心方法。它通常在算法模式的指导下进行。每个算法模式都抽象了一种通用的算法思想，可被用于解决一类具有相似特征的问题。例如，分治模式会将问题分解为更小的子问题，在递归求解后合并结果；它常被用于需要并行加速的场景。在算法模式的指导下，算法问题可以被视为一个模板应用问题：其目标是将算法模式实例化为一个完整程序，并保证该程序符合给定的形式化规约。

然而，即使有了算法模式的帮助，实践中的算法设计依然充满挑战。一方面，算法设计的高度专业性导致多数开发者难以独立解决算法问题，使得算法优化只能由少数专家完成；另一方面，算法层面的优化往往以牺牲代码可维护性为代价，而当算法的复杂程度超出阈值时，极易引发软件缺陷。

为了解决这些问题，程序合成在算法问题上的应用受到了广泛关注。通过自动生成符合算法模式和规约的程序，不仅能够帮助开发者摆脱复杂的算法设计工作，同时也能够在理论上确保优化后程序的正确性。然而，现有的程序合成方法并不能直接用于求解算法问题，它们面临着如下两方面的挑战。

- 一方面是**通用性**挑战。由于算法问题涉及形式各异的算法模式，因此程序合成方法必须具备足够的通用性，以便支持多种不同算法模式下的程序合成。
- 另一方面是**效率**挑战。鉴于算法问题的目标程序往往规模庞大且结构复杂，程序合成方法还需要具有高效合成复杂程序的能力。

为了解决这些挑战，本文面向算法问题提出了程序合成系统 SuFu。它包含了四个主要的技术创新以实现通用性与程序合成时的高效性。首先，为了实现通用性，本文设计了 SuFu 语言用于描述不同算法模式上的算法问题、并将它们自动地规约到一个形式固定的程序合成问题，名为**提升问题**。通过将 SuFu 语言作为前端，任何针对提升问题的程序合成方法都可以被用于求解大量不同的算法问题，从而实现通用性。

接下来，针对效率挑战，本文提出了三点创新用于实现高效的程序合成。

- 为了减小问题规模，本文针对提升问题提出了一个问题分解系统 AutoLifter，它可以有效地将提升问题分解为规模更小的子问题。
- 为了加速子问题的求解，本文发现导致现有合成方法效率不足的主要原因在于可泛化性缺陷，于是提出了具有可泛化性保障的程序合成方法 PolyGen。
- 为了进一步加速，本文发现程序合成方法在接收信息量更高的输入输出样例时往往能展现出显著更好的合成效率，并依此提出了询问选择方法 LearnSy，用于为程序合成挑选高质量的样例。

本文的主要研究工作和创新如下：

1. **提出了 SuFu 语言用于将不同形式的算法问题规约到形式固定的程序合成问题。**本文观察到许多算法问题的核心都是消除中间数据结构：它们需要将中间数据结构替换为更加高效的形式，使得一些原先低效的计算可以被高效地执行。本文将该核心问题形式化为了**提升问题**，并设计了 SuFu 语言用于将算法问题自动地归约到提升问题。在语言层面，本文在 SuFu 中加入了用于声明与操作中间数据结构的语言组件；而为了完成归约，本文为 SuFu 设计了一个类型系统，用于分析与提取所有与中间数据结构相关的计算。
2. **针对提升问题提出了一个高效的分解系统** AutoLifter。该系统可以将提升问题拆分为一系列子问题，每个子问题都只涉及原提升问题中目标程序的一小部分，从而显著减小了合成规模。本文在理论上证明了 AuotLifter 的正确性与完备性，并通过实验评估验证了它的有效性。
3. **提出了一个高效且具有可泛化性保障的程序合成方法** PolyGen。本文发现现有程序合成方法效率不足的原因在于它们的可泛化性缺陷。为了改进这一点，本文借助奥卡姆学习理论推导出了一个足以确保高可泛化性的充分条件，并在该结果的指导下设计了具有泛化能力保障的合成方法 PolyGen。实验结果表明，相比于现有的程序合成方法，PolyGen 在数据集上多解出了 49.2% 的任务，并取得了 6.14 倍的加速。
4. **提出了一个高效的样例选择方法** LearnSy。本文发现输入输出样例的质量会对程序合成方法的效率产生显著影响：当样例包含的信息量更高时，程序合成方法的效率也会越高。于是，为了提供质量更高的样例，本文尝试将交互式程序合成里的询问选择方法应用于加速程序合成，并针对现有方法时间开销过大的问题，提出了一个效率显著更高的近似方法 LearnSy。实验结果表明，LearnSy 平均可以为 PolyGen 减少 28.0% 的样例使用量，并降低 50.4% 的时间开销。

本文将上述创新集成为了程序合成系统 SuFu，并系统地评估了它在求解算法问题时的性能。在构建数据集时，本文从现有工作中收集了 290 个算法问题，涵盖了分治、流算法、结构递归等 8 种算法模式，并涉及列表、树等 35 种数据结构。实验结果表明，SuFu 在 5 分钟的时间限制内可以求解数据集内的 264 个任务，平均时间开销仅为 24.4 秒，显著优于所有现有的程序合成方法。

关键词：归纳程序合成，算法设计，程序优化

Program Synthesis Guided by Algorithmic Paradigms

Ruyi Ji (Computer Software and Theory)
Directed by: Prof. Zhenjiang Hu and Prof. Yingfei Xiong

ABSTRACT

Efficiency is a core pursuit in software development, and algorithm design is the major way to improve the efficiency of programs. Algorithm design is usually guided by algorithmic paradigms. Each paradigm formalizes a generic idea and can be applied to solve a class of algorithmic problems with similar properties. For example, divide-and-conquer denotes the idea of decomposing a problem into smaller subproblems, solving them recursively, and combining the results; it is often applied to scenarios requiring parallelization. Given a paradigm, the goal of an algorithmic problem is to instantiate the paradigm into a runnable program that satisfies some given specification.

However, even under the guidance of algorithmic paradigms, algorithm design remains challenging in practice. On the one hand, algorithm design is difficult for most developers, making algorithm-level optimizations typically carried out by only experts; on the other hand, algorithm-level optimizations often make programs complex, break the modularity, and reduce maintainability, thus bringing a significant risk of code defects.

To address these issues, much research effort has been devoted to applying program synthesis to solve algorithmic problems. By automatically synthesizing programs that follow the paradigm and satisfy the specification, program synthesis can not only help developers design efficient algorithms but can also ensure the correctness of the optimization in theory. However, existing program synthesizers cannot be directly applied to solve algorithmic problems, because of two challenges.

- **Generality**. Since there are various algorithmic paradigms available, the program synthesizer must be general enough to synthesize programs under different paradigms.
- **Efficiency**. Given that the target programs for algorithmic problems are mostly large in scale and complex in structure, the program synthesizer must also be efficient enough to synthesize complex programs.

To address these challenges, this paper proposes a program synthesis system, SuFu, for algorithmic problems. It comprises four technical contributions to achieve both generality and

efficiency. First, to achieve generality, we propose the SuFu language to describe algorithmic problems across different paradigms and automatically reduce them into a fixed class of program synthesis problems, named **lifting problems**. By taking the SuFu language as the front end, any program synthesizer for lifting problems can be applied to solve a wide range of different algorithmic problems, thereby achieving generality.

Next, we propose three approaches to achieve efficient program synthesis:

- To reduce the synthesis scale, we propose a decomposition system AutoLifter. It can effectively decompose a lifting problem into smaller subproblems.
- To efficiently solve subproblems, we find that the main cause of the inefficiency of existing synthesizers is the lack of generalizability. Therefore, we propose a novel program synthesizer PolyGen with a theoretical guarantee of generalizability.
- For further acceleration, we observe that the efficiency of program synthesis can be improved significantly when given examples with higher quality. Therefore, we propose an question selector, LearnSy, to select high-quality examples.

In summary, this paper makes the following contributions.

- **Proposing the SuFu language to reduce different algorithmic problems into a fixed class of program synthesis problems.** We observe that the core of many algorithmic problems is to eliminate intermediate data structures: they require replacing an intermediate data structure with a more efficient form so that some originally inefficient computations can be performed efficiently. We formalize this core problem as the **lifting problems** and design the SuFu language to automatically reduce an algorithmic problem into a lifting problem. At the language level, SuFu augments a functional language with constructs for describing and operating intermediate data structures; to perform the reduction, SuFu comprises a type system that is designed for analyzing and extracting all program fragments related to intermediate data structures.
- **Proposing an effective decomposition system AutoLifter for the lifting problem**. This system comprises two decomposition rules, named as **component elimination** and **variable elimination**. By recursively applying these rules, AutoLifter breaks a lifting problem into a series of simper subproblems. Each subproblem relates only to a small fragment of the target program, and thus its scale is much smaller than the original. We proves the correctness and completeness of AutoLifter in theory and validates its effectiveness via evaluation.
- **Proposing an efficient and generalizable program synthesizer, PolyGen.** we ob-

serve that the lack of generalizability is the main cause of inefficiency in existing program synthesizers. To address this issue, we apply the **theory of Occam learning** to derive a sufficient condition for ensuring high generalizability and designs PolyGen with the guidance of this condition. Our evaluation results show that, compared to existing program synthesizers, PolyGen can solve 49.2% more tasks on the dataset and achieves a speedup of 6.14×.

- **Proposing an efficient question selector,** LearnSy**.** We observe that the quality of input-output examples can significantly affect the efficiency of program synthesizers: the more informatic these examples are, the more efficient a synthesizer will be. To generate higher-quality examples, we follow the existing study on question selectors and propose an efficient approximation, LearnSy, to address the efficiency issue of existing selectors. Our evaluation results show that LearnSy can, on average, reduce the number of examples used by PolyGen by 28.0% and reduce its time cost by 50.4%.

We integrate the above contributions into a program synthesis system SuFu and comprehensively evaluate its effectiveness in solving algorithmic problems. To construct the dataset, we collect 290 algorithmic problems from existing studies, covering 8 algorithmic paradigms (such as divide-and-conquer, streaming algorithms, and structural recursion) and involving 35 data structures (such as lists and trees). Our evaluation results show that SuFu can successfully solve 264 problems, with an average time cost of only 24.4 seconds, significantly outperforming all existing program synthesizers.

KEY WORDS: Inductive Program Synthesis, Algorithm Design, Program Optimization

目录

表格索引

插图索引

第一章　绪论

1.1　算法范式制导的程序合成

软件运行时的高效性是软件开发的核心目标之一，而算法设计是提升软件运行效率的关键途径。算法层面的优化可以有效降低程序的渐进时间复杂度，从而在大规模数据的场景下为运行效率带来数量级的提升 [1–3]。然而，算法设计是一项极具挑战性的任务。它依赖于对问题性质的深入分析与创造性的方案设计，被广泛认为是软件开发中最困难的任务之一 [4,5]。在实践中，存在着大量必须由专家解决的算法问题[6,7]。但专家的精力终究是有限的，导致实际的软件性能往往难以达到其理想状态。

另一方面，算法设计也是一把双刃剑：它在提升运行效率的同时也显著增加了缺陷风险。算法层面的优化往往会破坏模块边界，使代码变得冗长复杂、难以维护。例如，在对操作系统内核 seL4 的形式化验证过程中，实现层面的缺陷仅导致了约 50 处修改，而由算法层面缺陷引发的修改则超过了 500 处 [8]。

为了降低算法设计的门槛与风险，程序合成（Program Synthesis）方法被广泛应用于相关研究中 [9–17]。通过低效程序合成等价的高效实现，程序合成能够帮助开发者摆脱困难的算法问题，使其能专注于软件功能的实现与验证；同时，程序合成的正确性保证也能提升优化的可靠性，从源头上降低缺陷风险。

然而，现有的程序合成方法尚不足以直接求解算法问题。一方面，相比于一般的程序，高效算法的代码实现通常具有显著更大的程序规模，这导致同等规模下的候选程序数量指数级上升，远远超出了程序合成方法的搜索能力。另一方面，算法设计往往需要引入全新的递归结构，但对递归程序的自动合成却被广泛认为是困难的[18,19] —— 这进一步放大了程序合成在求解能力上不足。

为了提升对算法问题的求解能力，本文在现有工作[10,13] 的启发下将**算法模式**（Algorithmic Paradigm[1]）作为输入引入程序合成，并对**算法模式制导的程序合成**问题（定义 1）展开研究。算法模式是对算法思想的抽象，每种算法模式都为设计特定算法提供了模板，并可被用于求解一类具有相似性质的算法问题。现有研究已经总结出了大量算法模式，例如分治、动态规划、贪心等，涵盖了不同领域上的大量高效算法。

定义 1 (算法模式制导的程序合成)**.** 给定算法模式 $\mathcal{P}$ 和参考程序 p，算法模式制导下的程序合成旨在合成遵循模式 $\mathcal{P}$ 的程序 p^*，并保证 p^* 与 p 具有相同的输入输出行为。

算法模式通常揭示了目标算法中的递归结构。因此，其制导的程序合成问题可以被视为一个模板填写问题：即根据参考程序将合适的底层计算填充到算法模板中，同

```
def dac(xs, l, r):
  if r - l == 1:
    return ??
  mid = (l + r) // 2
  lres = dac(xs, l, mid)
  rres = dac(xs, mid, r)
  return ??

res = dac(xs, 0, len(xs))
return ??
```

图 1.1　列表上的分治模板

```
def dac(xs, l, r):
  if r - l == 1:
    return (INF, xs[l])
  mid = (l + r) // 2
  (l2, l1) = dac(xs, l, mid)
  (r2, r1) = dac(xs, mid, r)
  smin = min(l2, r2, max(l1, r1))
  return smin, min(l1, r1)

res = dac(xs, 0, len(xs))
return res[0]
```

图 1.2　用分治计算列表次小值

时确保整体程序在输入输出行为上保持一致。这一问题不再需要合成完整的递归程序，而只需要一些递归无关的程序片段，从而显著降低了程序合成的难度。

例 1 (列表分治制导的程序合成)**.** 考虑一个关于列表次小值的编程任务，如下所示：

- 给定一个非空整数列表，计算并返回列表中第二小的元素；特殊地，若列表只包含了一个元素，则返回缺省值 INF。

下图展示了该任务上的一个正确程序。它先使用 Python 中的库函数 sort 将输入列表按照从小到大的顺序排序，再使用列表访问操作符 $[\cdot]$ 取出列表中的第二个元素。

```
INF if len(xs) == 1 else sorted(xs)[1]
```

该程序在处理长度为 n 的列表时具有 $\Theta(n \log n)$ 的时间复杂度，仍具有优化空间。

如果需要优化上述程序在并发场景下的运行效率，那么开发者可以在列表分治（例 **??**）这一模式的指导下设计算法。此时，开发者需要解决的算法问题在于如何适当地填写分治模板（图 1.2），使得结果程序与图 **??** 中的参考程序语义等价。具体而言，他们需要用合适的程序片段填写模板中的三处空缺。按照从上到下的顺序，第一处空缺需要计算分治算法在终止情况下（即列表长度为 *1* 时）的返回值；第二处空缺需要指明如何从子列表上的结果合并得到完整列表上的结果；而最后一个空缺则需要从分治的返回值中提取出最终输出。

图 1.2 展示了当前算法问题的一个合法程序。它在分治时不仅计算了列表次小值，还引入了列表最小值作为辅助值帮助计算。而在合并子结果时，该程序利用了最小值与次小值之间的一个特殊关系：完整列表的次小值只可能是某个子列表的次小值或两个子列表的最小值中的较大值。

1.2　自动求解算法问题的关键挑战

算法问题在形式上属于程序合成问题的一个子集。程序合成的目标是自动生成符合给定规约的程序。而在算法问题中，这些规约由两部分组成：一方面来自算法模式，即结果程序必须实例化一个特定的算法模式；另一方面则来自于参考程序，即结果程序的行为需要与参考程序保持一致。然而，现有的程序合成方法尚不足以有效求解算法问题，因为这些问题在**通用性**和**合成效率**方面对程序合成提出了更高要求。

通用性挑战　在算法设计领域中存在着大量不同的算法模式，包括但不限于分治[20]、动态规划[21]、增量算法[9]、结构递归[22]、滑动窗口[23]、线段树[24] 等。因此，面向算法问题的程序合成方法必须具备足够的通用性，以支持多种不同模式下的程序合成。否则，若每次仅针对单一模式设计合成方法，则需要付出巨大的研究代价才能覆盖足够多的算法模式，难以实现广泛应用。

然而，设计通用的程序合成方法绝非易事，因为不同的算法模式之间存在着巨大的形式差异。例如，在应用分治算法（例 **??**）时，关键在于找到合适的辅助值与合并操作，以确保完整列表的结果能够从左右子列表的结果中合并得出；而在应用动态规划时，核心在于设计合适的状态和状态转移方程，以确保计算过程中不存在后效性，并且包含大量重叠的子问题。尽管目前已经有研究分别针对分治[10,25,26] 和动态规划[11,27] 设计了相对高效的程序合成方法，它们均不能被推广到对方的算法模式上。

合成效率挑战　相比于一般的程序，算法问题的目标程序通常具有更大的规模与更复杂的程序结构。例如，在计算列表次大值时，一个简单的正确程序（**??**）可以仅由库函数 `sort` 和少量操作符组成；而对应的分治程序（图 1.2）则需要引入复杂的整数运算和显式的递归结构，其规模显著更大。相反地，尽管现有研究已经提出了大量不同的程序合成方法 [18,28–43]，它们仍局限于非递归表达式或小规模递归程序，与算法问题所需的合成效率存在显著差距。在本文收集到的算法问题中，目标程序平均会涉及 60.9 个操作符，最多可达 600，远远超出了现有程序合成方法的能力范围 [18,19]。

1.3　本文的研究框架

本文的目标是设计通用且高效的程序合成系统，用于解决算法问题。如图 1.3 所示，本文提出的系统能够支持多种算法模式下的算法问题，并能高效地生成目标程序。

实现通用性　为了解决通用性难题，本文的思路是发掘不同算法模式之间的共性，并将它们的算法问题归约到一个形式相同的核心问题。具体而言，本文发现许多算法模

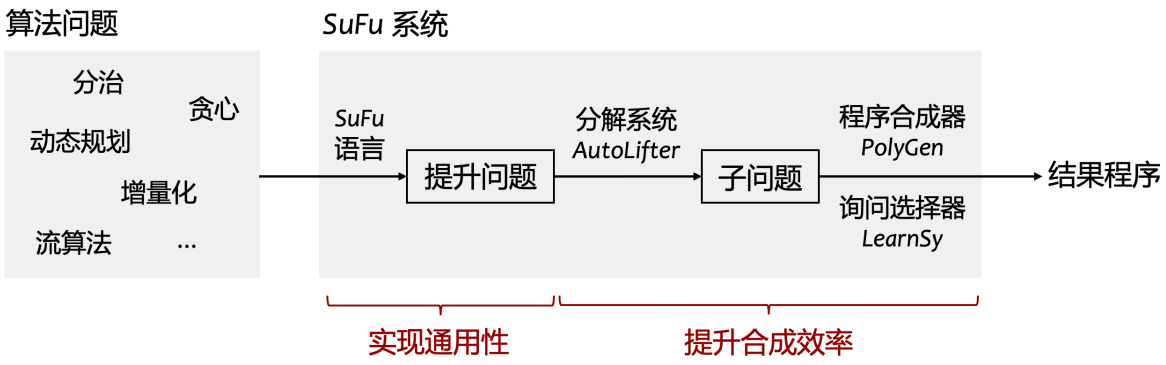

图 1.3　本文的研究框架

式的关键在于**消除中间数据结构**。于是，本文提出了**提升问题**的概念，用于形式化一类涉及消除中间数据结构的程序合成问题；并在此基础上设计了 SuFu 语言，用于自动地将算法问题归约到提升问题。通过将 SuFu 语言作为前端，任何支持求解提升问题的程序合成方法都可以被应用于大量不同的算法问题，从而实现广泛的通用性。

提升合成效率　为了加速求解提升问题，本文考虑了三个影响程序合成效率的因素。首先，**目标程序的规模**与合成效率直接相关：程序规模越大，程序合成方法就需要探索更大的程序空间，所需的时间也就越长。

其次，在目标程序固定的情况下，程序合成的效率主要受两个因素的影响：**程序合成方法的泛化能力**与**输入输出样例的质量**。具体而言，现代程序合成方法通常基于输入输出样例进行迭代式合成。这些方法会不断收集目标程序的输入输出样例，快速生成满足当前样例集合的候选程序，并验证其正确性，直至验证通过。由于每轮合成的时间开销通常较低，上述过程的效率主要取决于合成轮数（即输入输出样例的使用量）。而合成轮数的多少进一步与两个因素有关：

- 程序合成方法的泛化能力：即合成方法从少量样例中找到目标程序的能力；泛化能力越强，程序合成方法就越有可能在更早的轮次中找到目标程序。
- 输入输出样例的质量：质量越高的样例能够为程序合成提供更多关于目标程序的信息，降低合成目标程序的难度，进而减少所需的合成轮数。

本文针对这三个因素展开了研究，并分别提出了相应的改进方法：

- **减小目标程序的规模**：本文提出了分解系统 AutoLifter，用于将提升问题分解为一系列子问题，并确保每个子问题仅需合成目标程序的一小部分。
- **提升泛化能力**：本文提出了具有泛化能力保障的程序合成方法 PolyGen，它相比于现有方法可以从更少的样例中找到正确程序。
- **提升样例质量**：本文提出了询问选择方法 LearnSy，用于为程序合成挑选质量

更高的输入输出样例。

1.4　本文的主要贡献

如前文研究框架所述，本文拟解决以下科学问题。

- 如何将不同算法模式上的算法问题归约到形式固定的程序合成问题？具体而言，需要定义出不同算法模式间共享的核心问题，设计一种方法来描述算法问题，并实现自动化地将算法问题归约到其核心问题。
- 如何将算法问题分解为目标程序规模更小的程序合成问题？
- 如何设计一个泛化能力更强的程序合成方法？具体而言，需要首先提出理论框架以分析程序合成的泛化能力，并在此基础上设计一种具有泛化能力保障的程序合成方法。
- 如何为程序合成挑选质量更高的输入输出样例？一方面，该方法需要足够有效，能够显著提升样例的质量；另一方面，该方法需要足够高效，其时间开销应远小于程序合成本身的时间代价。

1.4.1　算法问题的规约方法

中间数据结构消除　为了将不同算法模式上的算法问题规约到形式固定的程序合成问题，本文调研了大量已有的算法模式，并发现应用许多算法模式的关键在于消除中间数据结构。具体而言，本文发现许多算法模式本质上都在描述输入数据的遍历方式。在应用这类算法模式时，总是存在着一个平凡的实现：它先按照算法模式遍历输入，将完整的输入原样返回，再直接使用参考程序计算结果。

例如，列表分治要求程序先独立地遍历左右子列表，然后合并来自两侧的信息。图 1.4 展示了将列表分治应用于任意参考程序 `ref` 的一种平凡实现：

- 函数 `dac` 按照列表分治的遍历方式对输入进行访问并将其原样返回。它先将输入列表分为两半，分别递归处理每一部分，然后直接连接递归结果。
- 该程序将参考程序 `ref` 直接用于 `dac` 的结果，从而保证了其输出与 `ref` 一致。

然而，这一平凡程序仍然在完整的输入上运行了参考程序，需被进一步优化。此时，优化的关键在于**中间数据结构消除**。具体而言，我们需要将被原样返回的输入替换为一种更高效的数据形式，使得最终输出可以在不调用参考程序的情况下快速计算。在上述列表分治的例子中，如果参考程序 `ref` 计算的是列表次小值（**??**），我们可以将函数 `dac` 优化为只返回输入列表的次大值与最大值，从而将原先对参考程序的调用替换为廉价的元组投影操作，如图 1.5 所示。该优化后的程序运行效率显著优于原始参考程序和平凡程序，可以作为算法问题的最终结果。

```
def dac(xs, l, r):
  if r - l == 1:
    return xs[l]
  mid = (l + r) // 2
  lres = dac(xs, l, mid)
  rres = dac(xs, mid, r)
  return lres + rres

res = dac(xs, 0, len(xs))
return ref(dac)
```

图 1.4　列表分治的平凡实现。

```
def dac(xs, l, r):
  if r - l == 1:
    return (INF, xs[l])
  mid = (l + r) // 2
  (l2, l1) = dac(xs, l, mid)
  (r2, r1) = dac(xs, mid, r)
  smin = min(l2, r2, max(l1, r1))
  return smin, min(l1, r1)

res = dac(xs, 0, len(xs))
return res[0]
```

图 1.5　对图 1.4 消除中间数据结构的结果。

提升问题　基于上述观察，本文提出了**提升问题**的概念，用于形式化算法问题中出现的中间数据结构消除问题。提升问题的目标是合成一个表征函数 *repr* 与一组合并算子 $comb_i$。其中，表征函数 *repr* 将原先的中间数据结构映射到新的高效表示形式，而每个合并算子 $comb_i$ 则需要在新的表示形式下高效地执行原先基于中间数据结构的计算。

以列表分治计算次小值为例，此时提升问题的目标是合成一个以列表为输入的表征函数以及三个合并算子，规约如下所示[①]，其中 $xs +\!\!+ ys$ 代表了列表拼接操作。

$$\begin{array}{lllll} \forall x : \texttt{Int} & comb_1\ x & = & repr\ [x] \\ \forall xs, ys : \texttt{List} & comb_2\ (repr\ xs, repr\ ys) & = & repr\ (xs +\!\!+ ys) \\ \forall xs : \texttt{List} & comb_3\ (repr\ xs) & = & sndmin\ xs \end{array} \tag{1.1}$$

该规约对应的约束如下：

- **终止情况**（第一行）：要求 $comb_1$ 计算单元列表的表征函数值。
- **合并操作**（第二行）：对应分治中的合并步骤。给定两侧列表 *xs* 与 *ys*，它要求 $comb_2$ 在接收递归结果 *repr xs* 和 *repr ys* 时，返回完整列表上的表征函数值。
- **最终输出**（第三行）：要求 $comb_3$ 根据递归结果计算列表的次小值。

图 1.6 展示了该提升问题的一组合法解。通过将这些程序填入图 1.7 中的分治模板，便可以得到图 1.5 中的分治程序。

SuFu 语言　在此基础上，本文设计了 SuFu 语言，用于自动将算法问题归约到提升问题。这一归约的关键在于自动提取算法问题中与中间数据结构相关的操作。为此，本文在函数式语言中引入了一个与中间数据结构相关的类型系统。

在 SuFu 语言中，算法问题通过其算法模式的平凡应用描述。图 1.8 展示了一个 SuFu 程序，它描述的算法问题是将列表分治应用于计算次小值。

① 在公式环境中，本文将默认使用函数式语言的语法书写函数求值操作，即用 $(f\ x)$ 表示函数 f 在输入 x 上的输出。而在代码环境中，本文将与使用的编程语言保持一致，例如在 Python 的环境下就会使用 $f(x)$ 来表示求值。

```
def repr(xs):
  return sndmin(xs), min(xs)

def comb1(x):
  return INF, x

def comb2(lres, rres):
  (l2, l1), (r2, r1) = lres, rres
  smin = min(l2, r2, max(l1, r1))
  return smin, min(l1, r1)

def comb3(res):
  return res[0]
```

图 1.6　将图 1.7用于计算列表次小值时，提升问题的目标程序。

```
def dac(xs, l, r):
  if r - l == 1:
    return comb1(xs[l])
  mid = (l + r) // 2
  lres = dac(xs, l, mid)
  rres = dac(xs, mid, r)
  return comb2(lres, rres)

res = dac(xs, 0, len(xs))
return comb3(res)
```

图 1.7　与提升问题对应的列表上的分治模板。

- 该程序的核心部分是分治模板 `dac`，它以参考程序 `ref` 和列表 `xs` 作为输入，首先按照列表分治的模式遍历列表 `xs`，再调用参考程序 `ref` 完成计算。由于该模板适用于所有基于列表分治的算法问题，因此 SuFu 将其纳入了一个算法模板库中，用户无需自行实现。
- 用户需要提供一个计算列表次小值的参考程序 `sndmin`（具体实现略），并将该程序与输入列表传递给分治模板 `dac`。不难看出，`sndmin'` 是列表分治在次小值上的平凡应用。虽然它的输出与参考程序一致，但并未进行任何优化。

为了提取与中间数据结构相关的操作，SuFu 提供了类型关键字 `Packed`，并要求在描述程序中标注所有需要被消除的中间数据结构。在上述例子中，需要被消除的是函数 `dac'` 的输出列表，于是该函数的输出类型应为 `Packed List`。值得注意的是，这类标注仅存在于算法模板中，因此在使用模板库的情况下无需由用户提供。

给定描述程序，SuFu 会利用类型信息与约束求解系统自动推导出所有对 `Packed` 数据结构的操作，如图 1.9 中的红色代码片段所示。这些操作分别对应分治算法中的边界情况、中间结果的合并以及最终结果的计算。通过对这些代码片段的类型和变量使用情况进行分析，SuFu 能够自动将该算法问题归约到公式 1.1 中的提升问题。

1.4.2　提升问题的分解方法

为了减小合成规模，本文针对提升问题提出了分解系统 AutoLifter。该系统包含两条分解规则，分别称为**组件消除**和**变量消除**。通过递归地应用这些规则，AutoLifter 能够将提升问题逐步分解为一系列子问题。每个子问题仅需合成目标程序中的一个片段，从而显著减小了合成规模。

算法模板库：

```
dac' :: List -> Packed
    List
dac' Single(x) = Single(x)
dac' xs =
  let ls, rs = split xs in
  concat (dac ls) (dac rs)

dac ref xs = ref (dac' xs)
```

用户代码：

```
sndmin xs = ...
sndmin' xs = dac sndmin xs
```

图 1.8　使用 SuFu 语言描述的算法问题，关于将列表分治应用于列表次小值。

```
dac' Single(x) = Single(x)
dac' xs =
  let ls, rs = split xs in
  let lres = dac ls in
  let rres = dac rs in
  concat lres rres

dac ref xs =
  let res = dac xs in
  ref res

sndmin xs = ...
sndmin' xs = dac (sndmin xs)
```

图 1.9　SuFu 推理出的数据结构操作。

组件消除　组件消除规则利用了提升问题中目标程序的元组结构。具体而言，复杂的算法问题通常涉及多个与中间数据结构相关的属性值。因此，在提升问题中，目标的表征函数和合并算子通常以多组件元组的形式出现。例如，在图 1.6 中，表征函数包含两个组件：*sndmin xs* 与 *min xs*，分别对应需要计算的目标值和辅助值；而合并算子 $comb_1$ 与 $comb_2$ 同样由两个组件组成，分别对应表征函数的两个输出。

进一步地，本文观察到这些组件之间存在着依赖关系。在上述提升问题（公式 1.1）中，表征函数的第一个组件 *sndmin xs* 是为了满足第三行规约而引入的。在此基础上，由于表征函数 *repr* 需要返回列表次小值，为满足前两行约束，我们需要引入表征函数的第二个组件 *min xs* 以及前两个合并算子的第一个组件，用于计算列表次小值。最后，由于更新后的表征函数 *repr* 还需要返回列表最小值，我们需要进一步引入前两个合并算子的第二个组件，用于完成列表最小值的计算。

根据这一观察，本文设计了组件消除规则。如图 1.10 所示，该规则可以将提升问题的求解拆分为若干步骤，每个步骤仅对上一步新引入的组件进行补充，从而减少了每个子问题中需要合成的组件数量。

变量消除　变量消除规则利用了提升问题中每个合并算子的独立性。以公式 1.1 中的规约为例，提升问题中的每一行规约都可以看作是表征函数 *repr* 与一个不同合并算子的复合。因此，在表征函数已知的情况下，每一行规约都可以被独立地视为仅针对一个合并算子的子问题，并被分别求解。

基于上述观察，本文设计了变量消除规则。对于由组件消除产生的每个中间问题，变量消除规则首先从其规约中提取与表征函数相关的内容，并优先合成表征函数。随后，它将合成得到的表征函数代入原始规约中，从而为每个合并算子生成独立的合成

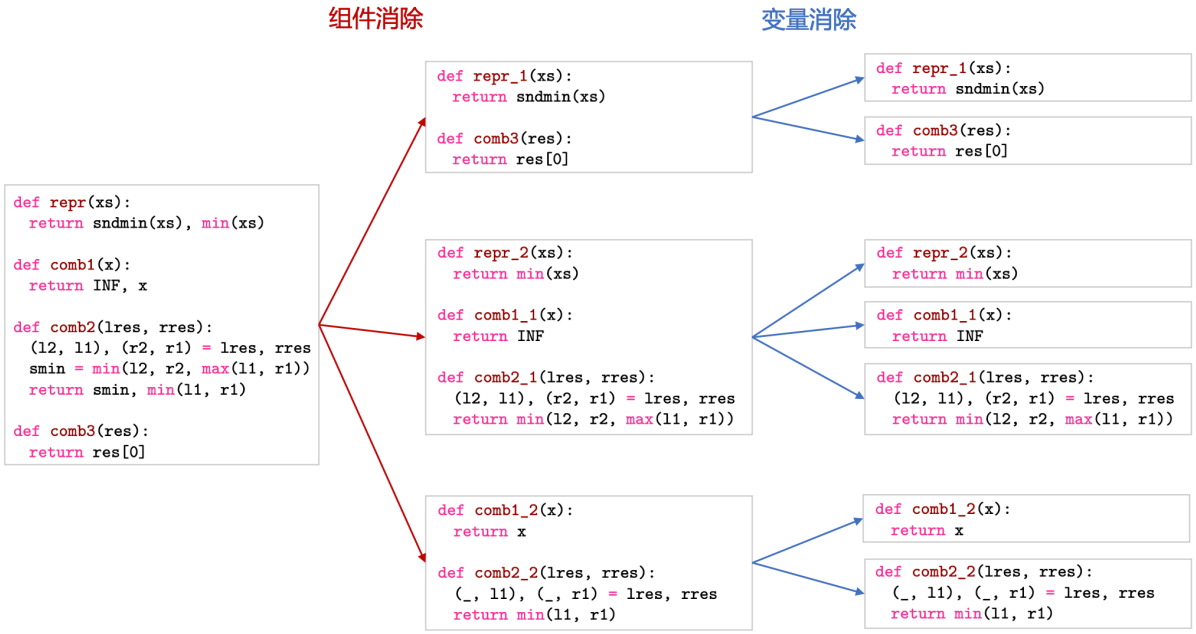

图 1.10　应用 AutoLifter 分解提升问题，其中箭头代表分解过程，每个方框对应一个合成问题，而其内部的程序表示该问题的目标程序。

问题。如图 1.10 所示，变量消除规则能够进一步分解由组件消除生成的中间问题，最终得到仅涉及单一变量的程序合成问题。

1.4.3　具有泛化能力保障的程序合成方法

奥卡姆合成器　为了讨论程序合成的泛化能力，本文首先基于奥卡姆学习理论 [44] 提出了**奥卡姆合成器**的概念，作为衡量程序合成方法泛化能力的理论基础。奥卡姆合成器需要遵循奥卡姆剃刀原则，保证返回一个规模较小的合法程序。具体而言，其合成的程序大小不能超过最小合法程序的多项式级别，并且在渐进意义上小于给定的输入输出样例数量。奥卡姆学习理论为这样的程序合成方法提供了泛化能力保障：当接收多项式级别的输入输出样例时，它们能够保证返回一个与目标程序几乎等价的结果。

合成方法设计　在理论结果的基础上，本文设计了一个高效的奥卡姆合成器 PolyGen。由于复杂程序通常包含分支结构，PolyGen 对分支程序进行了针对性优化，如图 1.11 所示。具体而言，它会将分支程序的合成问题分解为关于分支语句集合与分支条件的子问题，然后进一步将这些子问题分解为关于单一语句的合成问题以及单一子句的合成问题。在 PolyGen 的设计过程中，本文将奥卡姆合成器对结果大小的要求分配到了每个子问题，并保证子问题的求解过程严格遵循了这些条件。通过这种方式，本文证明了 PolyGen 满足奥卡姆合成器的要求，从而为其泛化能力提供了理论保障。

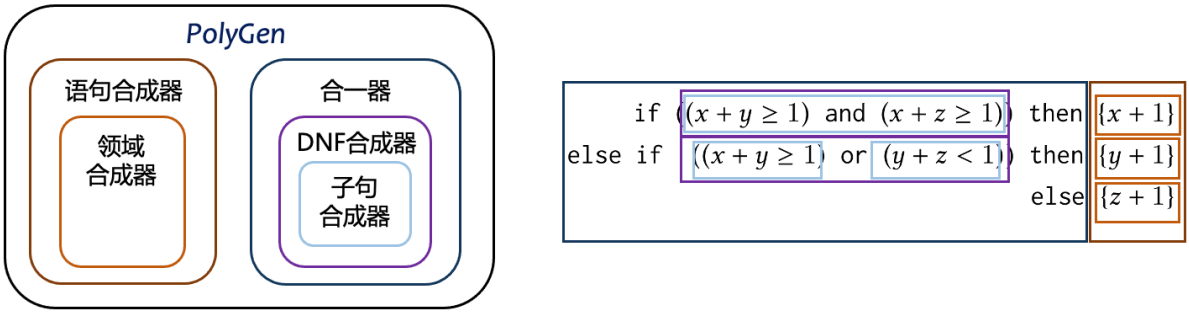

图 1.11　PolyGen 的结构，其中右图的每一个矩形表示了由左图对应颜色组件合成的部分。

1.4.4　高效的样例选择方法

交互式程序合成中的样例选择　本文跟进了交互式程序合成领域中对样例选择问题的研究 [45,46]。在交互式程序合成中，所有输入输出样例都需要通过向用户提问来获得：由合成系统选择输入，用户反馈对应的输出，二者组成新的样例用于下一步合成。在这一过程中，选择高质量的输入可以减少与用户交互的轮数，降低用户负担。

现有研究 [45] 在样例选择问题与最优决策树问题[47] 之间建立了理论联系，并提供了一个有效的贪心策略。具体而言，现有的样例选择方法均基于**最少等价对策略** [48] 及其变种。该策略定义输入的目标值为程序空间中有多少对程序在该输入上的输出相同，并建议在每一轮中选择目标值尽可能小的输入生成样例。

然而，在程序合成中执行最少等价对策略是非常困难的。要计算一个输入的目标值，我们需要逐一运行程序空间中的每个程序并统计重复的输出数量。但由于程序空间通常非常庞大，这种方法的时间开销难以接受。尽管现有的样例选择方法通过离线计算等策略在交互式场景下取得了一定的效果，但它们的时间开销甚至可能超过程序合成本身，难以实现本文加速程序合成的目标。

高效的样例选择　为了高效地选择样例，本文为最少等价对策略设计了一个可以被高效执行的近似策略。具体而言，本文分析了最少等价对策略中目标值的计算过程，并发现其计算瓶颈来源于运算符的复杂行为：由于每个运算符在不同的输入下的行为可能完全不同，我们在统计等价对数量时不得不完全计算所有程序的输出，难以进行优化。针对这一问题，本文设计了**统一等价模型**。该模型使用简单的随机过程来近似复杂的运算符行为，从而将原先难以计算的目标值近似为一个可以被高效计算的数学期望值。本文为统一等价模型设计了对应的学习算法，并在此基础上提出了一个高效的样例选择器 LearnSy。

1.5 本文产出的技术工具

综上所述，基于本文研究产出的技术工具如下：

- 面向算法问题的程序合成系统 SuFu（第三章与第七章），其学术论文发表于国际会议 PLDI2024。SuFu 是首个在算法问题上实现通用性与高效性的程序合成方法，也是目前在算法问题上最好的程序合成方法。在本文收集的 290 个算法问题中，SuFu 能够在 5 分钟的时间限制内成功求解 264 个问题，显著优于所有通用的程序合成方法，并在对应的子集上超越了针对特定算法模式的特化方法。
- 针对提升问题的分解系统 AutoLifter（第四章），其学术论文发表于国际期刊 TOPLAS2024。该系统在理论上保证了分解的正确性与完备性，并在实验中能够成功分解数据集中的所有提升问题，其子问题的平均规模仅为原问题的 20.5%.
- 具有泛化能力保障的程序合成方法 PolyGen（第五章），其学术论文发表于国际会议 OOPSLA2021，并且所有审稿人都给予了强烈接受（Strong Accept）的审稿意见。该方法是目前对分支程序最好的合成方法，它能够在 5 分钟的时间限制内合成最多包含 500 个操作符的大规模程序。
- 样例选择方法 LearnSy（第六章），其学术论文发表于国际会议 OOPSLA2023。它是首个成功加速程序合成的样例选择方法，并且能够在显著减少时间开销的情况下，提供与现有方法的结果质量相近的样例。

1.6 论文组织架构

本文共分为八个章节，组织结构如图 1.12 所示。第一章绪论介绍研究背景，给出算法问题的定义，介绍其中的核心挑战，并提出本文的研究思路；第二章回顾相关研究工作；第三章给出提升问题的定义，并介绍如何自动地将算法问题规约到提升问题；第四章介绍如何将提升问题分解为规模更小的子问题；第五章和第六章分别介绍如何提升程序合成的泛化能力和提升输入输出样例的质量；第七章将本文的创新集成为一个面向算法问题的程序合成系统，并对其进行实验评估；第八章总结。

对本文各章节的简介如下：

- **第一章 绪论**。主要介绍本文的研究背景，提出本文的研究思路，并介绍本文的主要贡献和创新。
- **第二章 相关研究工作**。主要回顾和总结相关研究工作，内容覆盖演绎程序变换、归纳程序合成和程序合成技术的近期进展。
- **第三章 算法问题的归约方法**。给出提升问题的形式化定义，并展示常见的算法模式（包括分治、动态规划、增量算法等）到提升问题的归约。接着，给出 SuFu 语言的设计（包括语法、语义与类型系统），介绍其中的自动规约算法，并在具

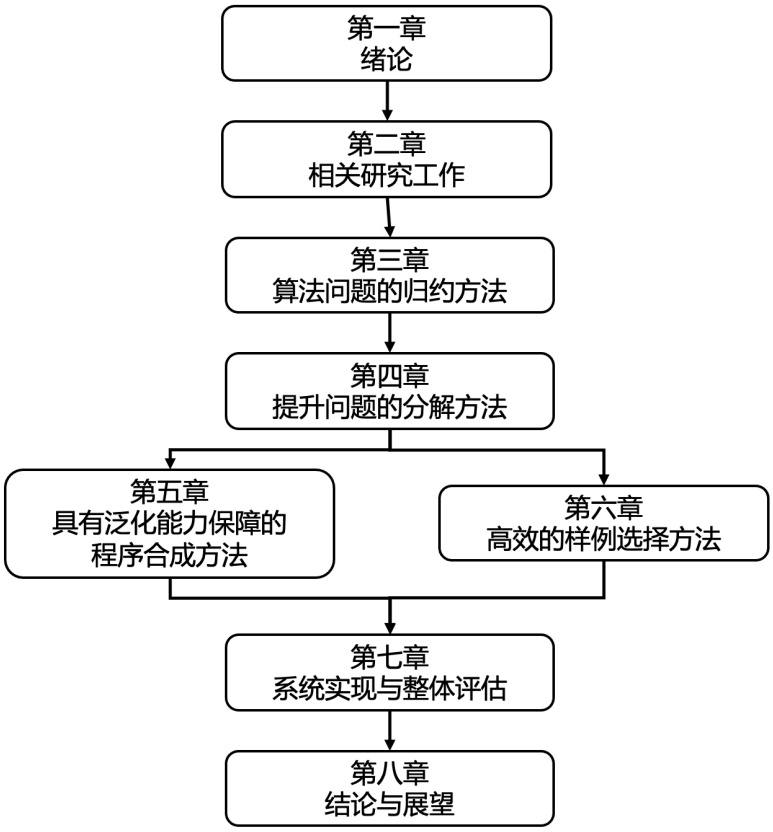

图 1.12　本文组织结构示意图。

有代表性的算法模式上对 SuFu 语言进行演示。

- **第四章 提升问题的分解方法**。根据第三章中提升问题的定义，介绍提升问题的分解系统 AutoLifter。
- **第五章 具有泛化能力保障的程序合成方法**。介绍奥卡姆学习理论，引入奥卡姆求解器的概念。接着，分析现有的程序合成方法的泛化能力，并提出一种高效且具有泛化能力保障的程序合成方法 PolyGen。
- **第六章 高效的样例选择方法**。介绍样例选择问题与最小相等对策略。引入统一等价模型的概念，并提出一种高效的样例选择器 LearnSy。
- **第七章 系统实现与整体评估**。介绍本文对 SuFu 系统的实现，并进行实验评估。
- **第八章 结论与展望**。总结本文的工作并展望未来研究方向。

第二章　相关研究现状

自从算法的概念被提出以来，自动求解算法问题就成了研究人员们长久以来的目标。对这一问题的研究可以追溯到上世纪八十年代[49]，并按照时间顺序涌现出了两种不同的技术路线，如图2.1所示。

- 首先，基于演绎推理的程序优化技术[49–61] 的思路是设计一系列的程序变换规则，并通过不断应用这些变换规则，将参考程序重写为足够高效的算法。
- 随后，随着计算机算力的进步，归纳程序合成技术[28–43] 在本世纪得到了广泛关注。其思路是设计一个足够大的程序空间和一个足够高效的搜索方法，并直接在这一程序空间中搜索满足规约的程序。

近几年，算法问题的求解方法被进一步丰富。一些工作尝试结合演绎推理与归纳程序合成的优点，针对特定的算法模式提出了更有效的程序合成方法[10,25,27,62]；而随着人工智能技术的发展，大语言模型也涌现出了值得关注的算法问题求解能力[63,64]。

在本章中，本文先总结演绎推理与归纳合成这两个方向上的代表性技术，再介绍算法问题求解在近期的进展。

2.1　基于演绎推理的程序优化技术

基于演绎推理的程序优化技术使用预先定义的程序变换规则，逐步将低效的参考程序变换为更加高效的程序。例如，规则 $?e \times 0 \mapsto 0$ 表明任何乘以 0 的表达式都可以被直接替换成 0。通过应用该规则，演绎推理系统可以将低效程序 $x + (slow\ y) \times 0$ 重写为 $x + 0$，从而避免函数调用 *slow y* 中的无效开销。

而在处理复杂的优化，尤其是算法层面的优化时，演绎推理技术遇到的核心难题在于对递归的处理。具体而言，复杂的程序往往会涉及数据结构与这些数据结构上的递归函数，对这些程序的优化需要展开递归函数的定义，对其函数体进行变换，并最终引入新的递归函数。如何系统性地完成这些递归函数上的变换成为了演绎推理研究中的重要问题。对这一问题的研究最终发展为了两个不同的演绎推理框架，分别为**展开/折叠框架**[52] 与**程序演算框架**[50]。

2.1.1　展开/折叠框架

展开/折叠框架对递归函数的变换可以被分为**展开**和**折叠**两个阶段。在展开阶段中，该框架将参考程序中的所有递归调用展开足够多的层数，以得到一个规模更大但更加

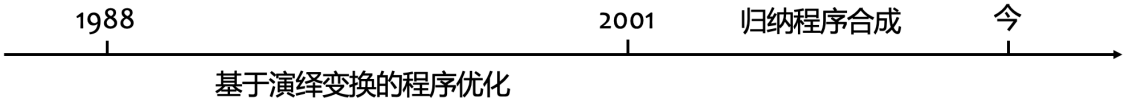

图 2.1　按照时间顺序，与求解算法问题有关的两条技术路线。

底层的程序。接着，在折叠阶段中，该框架使用表达式级别的规则对展开后的程序进行变换，并最终将变换结果折叠为新的递归函数。

例 2. 考虑按照如下方式定义的递归函数 $dot\ x\ y\ n$，它接受两个长度为 n 的向量 x, y，并计算它们的内积 $\sum_{i=1}^{n} x_i y_i$。

$$dot\ x\ y\ n \quad \coloneqq \quad if\ n = 0\ then\ 0\ else\ \left(dot\ x\ y\ (n-1) + x_n y_n\right)$$

假设现在需要推导一个递归函数 $f\ a\ b\ c\ d\ n$，它接受四个长度为 n 的向量，并计算前两个向量的内积与后两个向量的内积的和，即 $\sum_{i=1}^{n} a_i b_i + \sum_{i=1}^{n} c_i d_i$。下面展示了一个遵循展开折叠框架的推导过程。

$$\begin{aligned}
f\ a\ b\ c\ d\ n &= dot\ a\ b\ n + dot\ c\ d\ n \\
&= if\ n = 0\ then\ 0\ else\ \left(dot\ a\ b\ (n-1) + a_n b_n\right) + \\
&\quad\ if\ n = 0\ then\ 0\ else\ \left(dot\ c\ d\ (n-1) + c_n d_n\right) \\
&= if\ n = 0\ then\ 0\ else\ \left(dot\ a\ b\ (n-1) + dot\ c\ d\ (n-1) + a_n b_n + c_n d_n\right) \\
&= if\ n = 0\ then\ 0\ else\ \left(f\ a\ b\ c\ d\ (n-1) + a_i b_i + c_i d_i\right)
\end{aligned}$$

该推导过程先展开了函数 dot 的定义，再利用分支语句的性质将两个条件相同的分支合为一处，最后根据最初的定义将表达式 $dot\ a\ b\ (n-1) + dot\ c\ d\ (n-1)$ 折叠回 $f\ a\ b\ c\ d\ (n-1)$，从而得到了 f 的递归定义。

尽管展开/折叠框架可以支持许多复杂变换，但在实践中，其缺点在于难以确定合适的展开层数。如果展开的层数过少，展开后的表达式可能不足以得到目标程序；而如果展开的层数过多，展开后的程序可能会变得过于复杂，导致搜索空间过大。

2.1.2　程序演算框架

为了从根本上消除与展开层数相关的问题，程序演算转而聚焦于一些常见的组件（通常为高阶函数）。该框架通过组件层面的变换规则进行程序变换，从而避免了对递归函数的展开。在使用程序演算框架进行变换时，用户需要使用给定组件定义初始程序。接着，变换系统会在组件的层面进行程序变换，直到得到满足要求的程序。

例 3. 考虑一个与例 2 中函数 f 类似的程序 g，它的输入是一个四元组列表 xs，输出为四元组前两个元素乘积的和加上后两个元素乘积的和，即 $\sum_{i=1}^{n} xs_{i,1}xs_{i,2} + \sum_{i=1}^{n} xs_{i,3}xs_{i,4}$。在程序演算框架中，该程序可以使用如下的组件 *fold* 定义。

$$fold\,(\oplus)\;e\;[xs_1,\ldots,xs_n] \quad \coloneqq \quad \big(\big((e \oplus xs_1) \oplus xs_2\big) \cdots \oplus xs_n\big)$$

该组件以二元操作符 $\oplus$、初值 e 和一个列表为输入。它会依次访问列表中的每一个元素，并不断应用操作符 $\oplus$ 计算得到最终结果。

使用这一组件，我们可以按照如下方式在程序演算框架中定义函数 g。

$$\begin{aligned} g \;\coloneqq\; \big(fold\,(\oplus_1)\;0\;x\big) + \big(fold\,(\oplus_2)\;0\;x\big) \quad \textbf{\textit{where}} \quad & v \oplus_1 (a,b,c,d) \coloneqq v + ab \\ & v \oplus_2 (a,b,c,d) \coloneqq v + cd \end{aligned}$$

程序演算中的元组化规则[65,66] 可以将该程序中的两个 *fold* 合并为一个，从而直接得到函数 g 的递归定义：

$$\begin{aligned} g \;\coloneqq\; add\,\big(fold\,(\oplus_3)\;(0,0)\;x\big) \quad \textbf{\textit{where}} \quad & add\,(v_1,v_2) && \coloneqq\; v_1 + v_2 \\ & (v_1,v_2) \oplus_3 (a,b,c,d) && \coloneqq\; (v_1 + ab,\, v_2 + cd) \end{aligned}$$

因为程序演算只在组件层面进行变换，所以在将它用于求解算法问题时，需要针对具体算法模式提出特定的组件与变换规则。过去的研究探索了大量常见的算法模式，例如分治[20,67]、贪心[68,69]、动态规划[70–73] 等。这些研究中的一个重大挑战在于对数据结构的处理。因为许多算法模式并不局限于特定的数据结构（例如动态规划并不限制状态的类型），所以算法相关的变换规则需要具有应用于任何数据结构的能力。为了做到这一点，Bird 与 de Moor 以范畴论为基础构造了一个对数据结构抽象的程序模型，并对该模型上的程序演算进行了研究[74]。

在本节的剩余部分中，本文将先介绍基于范畴论的程序模型并展示其能力，然后对一些已有的算法变换规则进行介绍。

2.1.2.1　范畴论概念下的程序模型

一个范畴 **C** 包括以下三个要素。

- 一个由对象组成的类 $ob(\mathbf{C})$。
- 由对象间的态射构成的类 $hom(\mathbf{C})$。每一个态射 f 都有唯一的一个源对象 a 和目标对象 b，记作 $f :: b \leftarrow a$。
- 态射类上的二元操作复合。给定任意两个态射 $f :: a \leftarrow b$ 和 $g :: b \leftarrow c$，它们的复合（记作 $f \cdot g$）是一个对象 c 到对象 a 的态射。

范畴中的态射复合操作满足结合律，且保证对于任何对象 a，都存在一个复合操作下的单位态射 $id_a :: a \leftarrow a$。

在一般情况下，程序演算只考虑所有函数形成的范畴 **Fun**。在该范畴中，每一个对象都是一个集合，而从集合 A 到集合 B 的态射包含所有从 A 映射到 B 的函数。但在一些特殊情况下，例如在描述组合优化问题时，程序演算还会使用另一个范畴，由所有关系形成的范畴 **Rel**。本节将先专注于介绍范畴 **Fun** 上的工作，而与 **Rel** 相关的内容将在第 2.1.2.3 节中展开。

函子 函子 (记作 F 或 G) 是范畴上的同态。它将对象映射到对象，态射映射到态射，并在映射后保持单位态射与态射复合的一致性，如下所示：

$$\forall A, \mathsf{F}id_A = id_{\mathsf{F}A} \qquad \forall f, g, \mathsf{F}(f \cdot g) = \mathsf{F}f \cdot \mathsf{F}g$$

在程序演算中，最常见函子的是**多项式函子**，它们由以下基础函子组合而成。

- 单位函子 I 保持对象与态射不变。
- 给定对象 A，常函子 $!A$ 将任意对象映射到对象 A，任意态射映射到 id_A。
- 二元函子 $\times$ 表示集合的直积，其定义如下。

$$A \times B \coloneqq \{(a, b) | a \in A, b \in B\} \qquad (f \times g)(a, b) \coloneqq (f\ x, g\ x)$$

- 二元函子 $+$ 表示集合的直和，其定义如下。

$$A + B \coloneqq \{1\} \times A \cup \{2\} \times B \qquad (f_1 + f_2)\ (i, x) \coloneqq (i, f_i\ x)$$

递归数据结构 程序演算使用多项式函子抽象地定义数据结构[75–78]。给定函子 F，定义一个 F-代数为一个类型为 $A \leftarrow \mathsf{F}A$ 的态射，其中 A 可以是任意的对象。可以证明在范畴 **Fun** 中，对于每个多项式函子 F，都在同构意义下存在唯一的 F-代数 $in_{\mathsf{F}} :: T \leftarrow \mathsf{F}T$，称为初始代数，它可以准确地 “模拟” 任何其他的 F-代数[79]。具体而言，对于任意 F-代数 $f :: A \leftarrow \mathsf{F}A$，都存在态射 $g :: A \leftarrow T$ 满足：

$$f \cdot \mathsf{F}g = g \cdot in_{\mathsf{F}} \tag{2.1}$$

在上述公式中，态射 in_{F} 的目标对象 T 具有与直观上的数据结构相同的性质。其中，函数 g 可以被视作数据结构 T 上的递归函数，f 是递归过程中的函数体，A 是递归结果的类型，而 in_{F} 是数据结构的构造函数。此时，公式 2.1 构成了对递归函数 g 的定义。其左侧表示先递归到子结构，再用函数 f 合并所有递归结果；而其右侧表示先将子结构合并成完整的数据结构，再直接使用递归函数 g 计算最终结果。

以列表为例，考虑元素类型为 A 的列表，它在函数式语言中的定义一般如下所示。

$$\text{List } A = \text{Nil} \mid \text{Cons } (A, \text{List } A)$$

该数据结构与多项式函子 $\mathsf{F}_L = !\mathbf{1} + !A \times \mathsf{I}$ 对应，其中 $\mathbf{1}$ 表示一个大小为 1 的集合。直观地，构造函数 Nil 不需要任何参数，于是对应空元组上的常函子 $!\mathbf{1}$；而构造函数 Cons 接受一个 A 集合内的元素和一个子列表，于是对应函子 $!A \times \mathsf{I}$。

在初始代数 $in_{\mathsf{F}} :: T \leftarrow \mathsf{F}T$ 的基础上，定义态射 $out_{\mathsf{F}} :: \mathsf{F}T \leftarrow T$ 为 in_{F} 的逆函数，它会将一个递归数据结构展开到其子结构。

递归函数　程序演算将数据结构上的常见递归组件定义为一些特殊的态射。首先，最基本的递归组件由公式 2.1 导出。给定由多项式函子 F 定义的数据结构 T 与态射 $f : A \leftarrow \mathsf{F}A$，**折叠态射** $(\!|f|\!)_{\mathsf{F}} :: A \leftarrow T$ 是下列方程的唯一解[76,78]。

$$(\!|f|\!)_{\mathsf{F}} = f \cdot \mathsf{F}(\!|f|\!)_{\mathsf{F}} \cdot out_{\mathsf{F}}$$

该方程直接给出了 $(\!|f|\!)_{\mathsf{F}}$ 的递归形式：它通过 out_{F} 将输入的数据结构展开，递归到所有子结构，再通过函数 f 将递归结果合并。因此展示态射表示了数据结构上的递归计算。为了方便，当函子不存在歧义时，本文将隐去其下标并使用简写 $(\!|f|\!)$。

与折叠态射相反，**展开态射**表示数据结构的构造。给定由多项式函子 F 定义的数据结构 T 与态射 $f :: \mathsf{F}A \leftarrow A$，展开态射 $[\!(f)\!]_{\mathsf{F}} :: T \leftarrow A$ 是下列方程的唯一解[76]。

$$[\!(f)\!]_{\mathsf{F}} = in_{\mathsf{F}} \cdot \mathsf{F}[\!(f)\!]_{\mathsf{F}} \cdot f$$

该方程展示了 $[\!(f)\!]$ 对数据结构的构造过程，它递归地用函数 f 将输入值展开，并将所有结果合并成一个数据结构。同样，当函子不存在歧义时，本文将展开态射简写为 $[\!(f)\!]$。

将展开态射与折叠态射结合，可以得到一个更通用的递归组件——**重折叠态射**。给定由多项式函子 F 定义的数据结构 F 与态射 $\phi :: A \leftarrow \mathsf{F}A$ 和 $\psi :: \mathsf{F}B \leftarrow B$，重折叠态射 $[\![\phi, \psi]\!]_{\mathsf{F}} :: A \leftarrow B$ 为下列关于态射 h 的方程的最小不动点[80]：

$$h = \phi \cdot \mathsf{F}h \cdot \psi$$

该定义展示了重折叠态射的递归形式：它通过 ψ 将输入值展开，递归对子结构计算，再通过 ϕ 将子结果合并。因此，不难证明 $[\![\phi, \psi]\!] = (\!|\phi|\!) \cdot [\!(\psi)\!]$，即一个重折叠态射是一个折叠态射和一个展开态射的复合。重折叠态射的表达能力是十分强大的，它几乎可以描述实际中的所有递归函数[81]。

上述组件都会改变数据结构的类型，而**类型函子**可以在保持结构不变的情况下，变换数据结构中的元素。继续以列表为例。在上文中，本文展示了如何使用多项式函子定义元素类型为 A 的列表 List A。此时，List 可以被看做对象上的函数，它接受集合 A 并分别返回所有元素均在集合 A 内的列表集合。类型函子是对这一函数定义的扩充。

给定一个以对象 X 为参数的多项式函子 $\mathsf{F}(X)$，其类型函子 T 的定义如下：

- 对于对象 A，定义 $\mathsf{T}A$ 为由多项式函子 $\mathsf{F}(A)$ 定义的递归数据结构。
- 对于态射 $f :: A \leftarrow B$，定义 $\mathsf{T}f :: \mathsf{T}A \leftarrow \mathsf{T}B$ 为折叠态射 $(\!|in_{\mathsf{F}(B)} \cdot (apply_{\mathsf{F}}\ f)|\!)$，其中 $apply_{\mathsf{F}}\ f$ 表示将函数 f 应用到所有参数 X 的位置，并保持其他结构不变。

直观上，$\mathsf{T}A$ 表示了元素类型为 A 的数据结构，而 $\mathsf{T}f$ 将 f 应用到了其中的所有元素。

2.1.2.2 范畴论模型下的程序变换

本节将对上述组件间的变换规则进行介绍。

香蕉分离定理 。香蕉分离定理可以将两个作用于同一个递归数据结构的折叠态射融合成一个，从而减少对数据结构的遍历次数[82]。该定理也被称为元组化[83]。

定理 1 (香蕉分离定理[82])**.** 给定折叠态射 $(\!|f|\!)$ 和 $(\!|g|\!)$，如下等式总是成立。

$$(\!|f|\!) \vartriangle (\!|g|\!) = (\!|(f \cdot \mathsf{F}\pi_1) \vartriangle (g \cdot \mathsf{F}\pi_2)|\!) \quad \textbf{\textit{where}} \quad (f \vartriangle g)\ x \coloneqq (f\ x, g\ x)$$

霍纳法则及其推广。霍纳法则指代如下等式，它来源于加法乘法之间的分配率。

$$a_0 + a_1 \times x + a_2 \times x^2 + \cdots + a_n \times x^n = a_0 + (a_1 + (a_2 + \cdots (a_n + 0) \times x \cdots) \times x) \tag{2.2}$$

使用范畴论概念下的程序模型，霍纳法则可以被推广到任意数据结构和其他运算符上[84]。为了描述这一推广，首先定义组件 tri。给定一个带参数的多项式函子 $\mathsf{F}(x)$、对应的类型函子 T 和态射 $f :: A \leftarrow A$，$trif :: \mathsf{T}A \leftarrow \mathsf{T}A$ 被定义为折叠态射 $(\!|in \cdot \mathsf{F}\ (\mathsf{T}f)|\!)$。它将函数 f 应用到数据结构 $\mathsf{T}A$ 中的每一个元素上，且对深度为 d 的元素重复应用了 d 次。下列公式展示了组件 tri 在列表上的行为，其中 f^i 表示 i 个 f 的复合。

$$tri\ f\ [x_0, x_1, \ldots, x_n] = [x_0, f\ x_1, \ldots, f^i\ a_i, \ldots, f^n\ a_n]$$

定理 2 (推广后的霍纳法则[74])**.** 给定一个带参数的多项式函子 $\mathsf{F}(X)$ 和对应的类型函子 T，对于任意态射 $f :: A \leftarrow A, g :: A \leftarrow \mathsf{F}(A)\ A$，下列公式总是成立。

$$f \cdot g = g \cdot (apply_{\mathsf{F}}\ f) \cdot (\mathsf{F}\ f) \quad \rightarrow \quad (\!|g|\!) \cdot trif = (\!|g \cdot (\mathsf{F}\ f)|\!)$$

融合定理与酸雨定理 得到高效算法的一个常见方式是将原程序中的组件融合到一起[49]，因为这样可以减少构建与遍历中间数据结构的开销。因此，程序演算中存在着许多用于融合组件的规则。其中最基本的是针对折叠态射的融合定理，它给出了折叠态射与另一个态射融合的充分条件。

定理 3 (融合定理[74])**.** 给定多项式函子 F，对于任意态射 $h :: A \leftarrow B, f :: B \leftarrow \mathsf{F}B$, 与 $g :: A \leftarrow \mathsf{F}A$，下列公式始终成立。

$$h \cdot f = g \cdot \mathsf{F}h \quad \rightarrow \quad h \cdot (\!|f|\!) = (\!|g|\!)$$

而作为上述融合定理的特例，可以得到关于折叠态射与类型函子的融合引理。

引理 1. 给定一个带参数的多项式函子 $\mathsf{F}(X)$ 和对应的类型函子 T，对于任意态射 $f :: A \leftarrow B, g :: C \leftarrow \mathsf{F}(B, C)$，下列等式始终成立。

$$(\!|g|\!) \cdot \mathsf{T}f = (\!|g \cdot (apply_{\mathsf{F}}\ f)|\!)$$

而针对更一般的递归函数，酸雨定理[85] 给出了重折叠态射层面的融合规则。该定理依赖自然变换的概念与重折叠态射的三元组形式。

定义 2 (自然变换)**.** 给定函子 F 和 G，多态函数 $\phi_A :: \mathsf{F}A \leftarrow \mathsf{G}A$ 是一个 G 到 F 的自然变换（记作 $\phi :: \mathsf{F} \leftarrow \mathsf{G}$）当且仅当对于任意的态射 $f :: A \leftarrow B$，都有 $\phi_A \cdot \mathsf{G}f = \mathsf{F}f \cdot \phi_B$。

直观来说，G 到 F 的自然变换只是修改了数据的结构，并没有修改其内容，因此在 G 处的任意计算经过自然变换都可以被直接移交到 F 处执行。下列**移动引理**说明了在重折叠态射中，一个自然变换可以自然地在展开态射与折叠态射之间移动。

引理 2 (移动引理)**.** 对于任意的自然变换 $\eta :: \mathsf{F} \leftarrow \mathsf{G}$ 和态射 $\phi :: A \leftarrow \mathsf{F}A$，$\psi :: \mathsf{G}B \leftarrow B$，如下等式都成立。

$$[\![\phi \cdot \eta, \psi]\!]_{\mathsf{G}} = [\![\phi, \eta \cdot \psi]\!]_{\mathsf{F}}$$

根据引理 2，可以得到重折叠态射的三元组形式。给定态射 $\phi :: A \leftarrow \mathsf{F}A$，$\psi :: \mathsf{G}B \leftarrow B$ 和自然变换 $\eta :: \mathsf{F} \leftarrow \mathsf{G}$，定义 $[\![\phi, \eta, \psi]\!]_{\mathsf{F},\mathsf{G}} = [\![\phi \cdot \eta, \psi]\!]_{\mathsf{G}} = [\![\phi, \eta \cdot \psi]\!]_{\mathsf{F}}$。下面列举了关于重折叠态射三元组形式的一些显然事实。

$$[\![\phi, \eta, \psi]\!]_{\mathsf{F},\mathsf{G}} = [\![\phi \cdot \eta, id, \psi]\!]_{\mathsf{G},\mathsf{G}} = [\![\phi, id, \eta \cdot \psi]\!]_{\mathsf{F},\mathsf{F}}$$

$$(\!|f|\!)_{\mathsf{F}} = [\![in_{\mathsf{F}}, id, f]\!]_{\mathsf{F},\mathsf{F}} \quad (\!|f|\!)_{\mathsf{F}} = [\![f, id, out_{\mathsf{F}}]\!]_{\mathsf{F},\mathsf{F}}$$

酸雨定理可以将一部分特殊的重折叠态射与展开态射（折叠态射）融合[85]。

定理 4 (折叠态射-重折叠态射融合定理)**.** 对于任意的高阶多态函数 $\tau : \forall A, (\mathsf{F}A \to A) \to \mathsf{F}A \to A$，如下等式始终成立。

$$(\!|\phi|\!)_{\mathsf{F}} \cdot [\![\tau\ in_{\mathsf{F}}, \eta, \psi]\!]_{\mathsf{F},\mathsf{G}} = [\![\tau\ \phi, \eta, \psi]\!]_{\mathsf{F},\mathsf{G}}$$

定理 5 (重折叠态射-展开态射融合定理)**.** 对于任意的高阶多态函数 $\sigma_A : \forall A, (A \to \mathsf{F}A) \to A \to \mathsf{F}A$，如下等式始终成立。

$$[\![\phi, \eta, \sigma\ out_{\mathsf{F}}]\!]_{\mathsf{G},\mathsf{F}} \cdot (\!|\psi|\!)_{\mathsf{F}} = [\![\phi, \eta, \sigma\ \psi]\!]_{\mathsf{G},\mathsf{F}}$$

2.1.2.3 关系范畴和优化问题

一般情况下，程序在每个输入下只会产生唯一一个输出，范畴 **Fun** 为这些程序提供了抽象模型。然而，在考虑优化问题的时候，我们通常需要一到多的映射——因为在一个优化问题中往往存在多个合法解。为了定义优化问题的抽象模型，Bird 和 de Moor 等人引入了关系上的范畴 **Rel** 作为工具[74]。

给定集合 A 和集合 B，一个 B 到 A 的关系 R 可以被 $A \times B$ 的一个子集 S_R 描述，其中 $(a,b) \in S_R$ 表示 R 将 b 关联到 a，记作 aRb。为了与集合区分，本文将使用编号靠后的大写字母（例如 R, S, T）来表示关系。与范畴 **Fun** 相比，**Rel** 中的对象仍然是集合，但是 B 到 A 的所有态射从 B 到 A 的函数集合扩充到了 B 到 A 的关系集合。下面展示了与关系有关的基本操作。

- 给定 $R :: A \leftarrow B$，其逆关系 $R^\circ : B \leftarrow A$ 被定义为 $bR^\circ a \iff aRb$。
- 给定 $R, S :: A \leftarrow B$，R 是 S 的子集（记作 $R \subseteq S$）当且仅当 $aRb \Longrightarrow aSb$。类似地，我们也可以定义关系上的交操作 $R \cap S$ 与并操作 $R \cup S$。
- 给定 $R :: A \leftarrow B, S :: B \leftarrow C$，其复合 $R \cdot S$ 定义为 $a(R \cdot S)c \iff \exists b, aRb \wedge bSc$。

与 **Fun** 类似，在 **Rel** 中同样可以使用多项式函子定义数据结构。而折叠态射的概念可以从 **Fun** 中的折叠态射延拓得到。这一过程涉及到幂集的概念与相关操作。

- 对于集合 A，$\mathsf{P}A$ 表示其幂集，即 $\{A' | A' \subseteq A\}$。P 可以被定义为 **Rel** 上的一个函子，但是本文不会使用其作用在关系上的部分。
- 关系 $\in :: A \leftarrow \mathsf{P}A$ 将集合关联到其元素，即 $a \in A \iff aRA$。
- 给定关系 $R :: A \leftarrow B$，函数 $\Lambda R :: \mathsf{P}A \leftarrow B$ 返回和一个元素关联的元素集合，即 $\Lambda R\ b \coloneqq \{a | aRb\}$。

给定多项式函子 F 和关系 $R :: A \leftarrow \mathsf{F}A$，折叠态射 $(\!|R|\!)_\mathsf{F}$ 的定义为 $\in \cdot (\!|\Lambda(R \cdot \mathsf{F} \in)|\!)_\mathsf{F}$。因为 $\Lambda(R \cdot \mathsf{F} \in)$ 是一个函数，所以 $(\!|\Lambda(R \cdot \mathsf{F} \in)|\!)$ 可以沿用范畴 **Fun** 中的定义。

在使用 **Rel** 描述问题时，并不需要单独引入展开态射 $[\![R]\!]$，因为它等价于 $(\!|R^\circ|\!)^\circ$。对应地，可以定义 **Rel** 上的重折叠态射。给定函子 F，关系 $R :: A \leftarrow \mathsf{F}A$ 和 $S :: \mathsf{F}B \leftarrow B$，重折叠态射 $[\![R, S]\!]_\mathsf{F}$ 为满足方程 $X = R \cdot \mathsf{F}X \cdot S$ 的最小解。与范畴 **Fun** 的情况相同，可以证明重折叠态射是一个折叠态射和一个展开态射的复合，即 $[\![R, S]\!] = (\!|R|\!) \cdot (\!|S^\circ|\!)^\circ$。

使用关系范畴描述优化问题 优化问题的定义依赖于最优化操作符 min。给定描述大小顺序的关系 $R :: A \leftarrow A$，其中 aRb 表示 a 在当前问题中不劣于 b，最优化关系 $min\ R :: A \leftarrow \mathsf{P}A$ 将一个集合关联到其中的所有最优元素，它的定义如下。

$$a(min\ R)A \iff a \in A \wedge \forall b \in A, aRb$$

即 a 是集合 A 的一个最优元素当且仅当 a 在 A 中且 a 不劣于 A 中的任何一个元素。

一般性地，优化问题可以被看做从所有合法方案中选取最优方案的问题。它可以被两个关系描述。给定输入类型 A 和方案类型 B，关系 $S :: B \leftarrow A$ 将输入关联到所有合法方案，关系 $R :: B \leftarrow B$ 描述了方案间的优劣。此时，优化问题可以被定义为 $min\ R \cdot \Lambda S$，它先用函数 ΛS 得到所有合法方案的集合，再由 $min\ R$ 选出最优方案。

因为关系 S 通常可以被定义成一个重折叠态射，所以程序演算中有关优化问题的工作基本只考虑形如 $min\ R \cdot \Lambda[\![S, T]\!]$ 的优化问题[21,74]。特别地，当重折叠态射中的展开态射是平凡的时候，即 $min\ R \cdot \Lambda(\!|S|\!)$，对应的优化问题又被称为顺序决策过程[70,86,87]。这一名字来自于 $(\!|S|\!)$ 的行为：它构造合法解的决策过程严格按照输入的结构，并依次使用子结构的合法解构建上一层结构的合法解。

2.1.2.4　算法层面的变换规则

分治算法　为了描述列表分治，程序演算引入了特殊组件**列表同态**。给定具有单位元 e 的二元操作符 $\odot$ 和函数 f，$hom\ (\odot)\ f$ 表示满足如下方程的函数 h（如果存在）。

$$h\ [] = e \qquad h\ [a] = f\ a \qquad h\ (xs +\!\!+ ys) = (h\ xs) \odot (h\ ys)$$

分治可以被直接用于计算列表同态[20,88–91]。给定对应操作符 $\odot$ 的列表同态 h 和输入列表 xs，$h\ xs$ 的值可以通过三步计算得到：(1) 将 xs 拆分为 $ls +\!\!+ rs$，(2) 递归计算 $o_1 = h\ ls$ 和 $o_2 = h\ rs$，(3) 返回 $o_1 \odot o_2$。

同时，也有一系列关于列表同态的定理与变换规则得到了提出。其中最为重要的是**第三列表同态定理**[92].

定理 6 (第三列表同态定理)**.** *任何可以同时被从左到右计算和从右到左计算的函数都可以被实现为列表同态.*

Morihata 等人于 2009 年将第三列表同态定理 (定理 6) 推广到了任意数据结构[67]。本文将以二叉树 Tree 为例对推广后的第三列表同态定理进行介绍。

```
Tree = Node Int Tree Tree | Leaf
```

首先，若要将第三列表同态定理推广到树结构上，一个自然的形式是“任何可以同时被自顶向下计算和被自底向上计算的函数都可以被实现为树同态”。为此，需要先定义树上这三类计算的含义。Morihata 等人的工作使用树上的路径来定义这三类计算，自然地，自顶向下表示按照从上到下的顺序处理路径上的每一个节点，自底向上表示按照从下到上的顺序处理每一个节点，而树同态则是从路径中间的某一个位置切开，并行地计算树的两个部分。

该工作使用 Huet 压缩子 [93] 定义路径。对于二叉树，其压缩子 Zipper 是一个元素类型为 Either (Int, Tree) (Int, Tree) 的列表，其中 Either A B 的定义如下。

$$\text{Either } A\ B = L\ A \mid R\ B$$

每个压缩子代表了树上从根到叶子节点的一条路径。对于每一个元素，$L\ (n,t)$ 表示当前节点的值为 n，左子树为 t，而路径的下一步是右子树；$R\ (n,t)$ 表示当前节点的值为 n，右子树为 t，而路径的下一步是左子树。根据这一含义，可以定义压缩子到二叉树的转换函数 $z2t$: Tree ← Zipper 如下所示。

$$z2t\ [] = \text{Leaf} \quad z2t\ ([L\ (n,t)] +\!\!+ x) = \text{Node}\ n\ t\ (z2t\ x)$$
$$z2t\ ([R\ (n,t)] +\!\!+ x) = \text{Node}\ n\ (z2t\ x)\ t$$

在压缩子的基础上，可以进一步引入路径计算的概念，它表示在压缩子上完成原本在树上进行的计算。而自顶向下和自底向上的计算都可以被看作是路径计算的特例。

定义 3 (二叉树路径计算)**.** 给定函数 $h :: A \leftarrow Tree$，函数 $h' :: A \leftarrow Zipper$ 是对应 h 的一个路径计算当且仅当 $h \cdot z2t = h'$。

定义 4 (二叉树自顶向下计算)**.** 给定函数 $h :: A \leftarrow Tree$ 及其路径计算 $h' :: A \leftarrow Zipper$，如果存在 $\otimes :: A \leftarrow (A \times (Either\ (Int, A)\ (Int, A)))$ 满足如下等式，则称 h' 是自顶向下的。

$$h'\ (x +\!\!+ [L\ (n,t)]) = (h'\ x) \otimes \big(L\ (n, h\ t)\big) \quad h'\ (x +\!\!+ [R\ (n,t)]) = (h'\ x) \otimes \big(R\ (n, h\ t)\big)$$

定义 5 (二叉树自底向上计算)**.** 给定函数 $h :: A \leftarrow Tree$ 及其路径计算 $h' :: A \leftarrow Zipper$，如果存在 $\oplus :: A \leftarrow ((Either\ (Int, A)\ (Int, A)) \times A)$ 满足如下等式，则称 h' 是自底向上的。

$$h'\ ([L\ (n,t)] +\!\!+ x) = \big(L\ (n, h\ t)\big) \oplus (h'\ x) \quad h'\ ([R\ (n,t)] +\!\!+ x) = \big(R\ (n, h\ t)\big) \oplus (h'\ x)$$

自顶向下（自底向上）路径计算的定义相当于压缩子结构上的从左到右（从右到左）计算。相似地，树同态的定义相当于压缩子结构上的列表同态，它要求在任何位置切开压缩子的情况下都能够完成计算。

定义 6 (树同态)**.** 函数 $h :: A \leftarrow Tree$ 是一个树同态当且仅当存在函数 $\phi :: A \leftarrow Either\ (Int, A)\ (Int, A)$ 和操作符 $\odot :: A \leftarrow (A \times A)$，满足如下等式。

$$h'\ [] = e \quad h'\ [L\ (n,t)] = \phi\ (L\ (n, h\ t))$$
$$h'\ [R\ (n,t)] = \phi\ (R\ (n, h\ t)) \quad h'(xs +\!\!+ ys) = h'\ xs \odot h'\ ys$$

其中函数 h' 等于 $h \cdot z2t$，值 e 表示操作符 $\odot$ 的单位元。

树分治的算法模式可以被应用于任意的树同态[94]。Morihata 等人证明了在给定 p 个处理器的情况下，如果 ϕ 和 $\odot$ 都是常数时间的，则树同态的结果可以在 $O(n/p+\log p)$ 的时间内计算，其中 n 为树的大小[67]。

基于以上定义，二叉树上的第三同态定理的内容如下。

定理 7 (二叉树上的第三同态定理)**.** 给定函数 $h : A \leftarrow Tree$，如果它存在自顶向下且自底向上的路径计算 $h' : A \leftarrow Zipper$，则 h 一定可以被实现为树同态。

稀疏定理　为了描述组合优化问题上的最优化算法，de Moor 于 2005 年提出了稀疏定理[70]，用于描述对顺序决策过程 $min\ R \cdot (\!|S|\!)$ 的优化。该定理来源于一个自然的剪枝思路。许多情况下，在通过子结构的合法解构造上一层结构的合法解时，并不是所有合法解都是有用的。而对于那些显然不优的合法解，我们可以在子结构层面就将其删去，从而节约后续的构造时间。

为了形式化地描述这一思路，de Moor 引入了稀疏化算子 *thin*。给定关系 $Q :: A \leftarrow A$，稀疏化关系 $thin\ Q :: \mathsf{P}A \leftarrow \mathsf{P}A$ 的定义如下所示。

$$x(thin\ Q)y \quad \Longleftrightarrow \quad x \subseteq y \wedge \forall b \in y, \exists a \in x, aQb$$

当 A 表示优化问题中解的类型且关系 Q 描述了解之间的优劣关系时，关系 $thin\ Q$ 将一个解的集合关系到它的一些效果上等价的子集。具体而言，令 y 是一个解的集合，x 是它的一个满足 $x(thin\ Q)y$ 的子集，那么条件 $\forall b \in y, \exists a \in x, aQb$ 要求 y 中任何一个解的效果都被 x 中的某一个解覆盖。这意味着解集 x 可以直接替代解集 y。

在稀疏化算子 $thin\ Q$ 的基础上，下面展示了**稀疏定理**的内容。

定理 8 (稀疏定理)**.** 如果 $Q \subseteq R$ 且 $S \cdot \mathsf{F}Q^\circ \subseteq Q^\circ \cdot S$，则如下公式成立。

$$min\ R \cdot (\!|thin\ Q \cdot \Lambda(S \cdot \mathsf{F} \in)|\!) \subseteq min\ R \cdot \Lambda(\!|S|\!) \tag{2.3}$$

在该定理中，公式右侧是一个顺序决策过程，而左侧给出了一个更小的关系。它表示在只需要得到任意一个最优解时，可以用左侧的关系代替朴素的顺序决策过程。

在等式左侧，$(\!|thin\ Q \cdot \Lambda(S \cdot \mathsf{F} \in)|\!)$ 是一个从输入到方案集合的关系。与 $(\!|S|\!)$ 类似，该关系严格按照输入的结构顺序构造解的集合。其构造过程分为两个阶段。在第一阶段中，函数 $\Lambda(S \cdot \mathsf{F} \in)$ 输出一个包含所有可以从子结构上的解构造得到的解的集合。接着，在第二阶段中，关系 $thin\ Q$ 将这个解的集合关联到所有与之等效的子集，表示删去那些在关系 Q 下不优的解。

考虑定理的两个条件。首先，$Q \subseteq R$ 蕴含 $min\ R = min\ R \cdot thin\ Q$，即对于任意解的集合，$thin\ Q$ 都不会将其中（在 R 意义下的）最优解删去，从而保证左侧程序在通过 $thin\ Q$ 剪枝的过程中，永远不会将最优解删去。

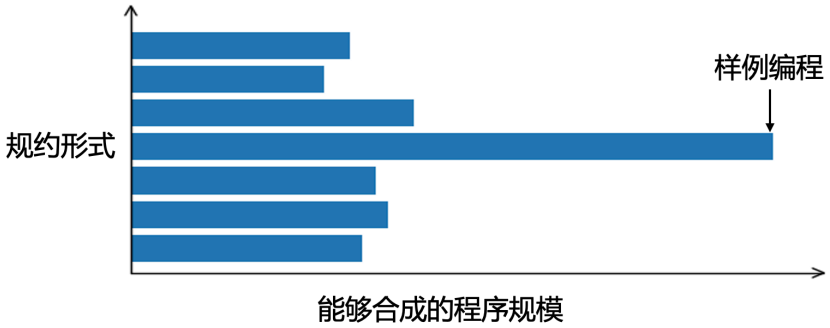

图 2.2　不同的规约形式下，已有的归纳程序合成方法的求解效率。

接着，考虑条件 $S \cdot \mathsf{F}Q^{\circ} \subseteq Q^{\circ} \cdot S$。该公式的两侧均为 $\mathsf{F}A$ 到 A 的关系，都将子结构上的解关联到上层结构的解。它等价于如下的公式。

$$\forall x, y \in \mathsf{F}A, \forall a \in A, (x \mathsf{F}Q y \wedge aSy \rightarrow \exists b \in A, bSx \wedge bQa) \tag{2.4}$$

它要求对于任意两个子结构的解 x, y，如果 x 在 Q 中优于 y (即 $x\mathsf{F}Qy$)，则对于任意 y 能产生的上层解 a，都存在 x 能产生的上层解 b，使得 b 在 Q 中优于 a（即 bQa)。该条件保证了每一步被 *thin* Q 排除的解都不可能在后续构造中产生更优的解。

后续的研究对稀疏定理提出了一系列改进，例如将该定理推广到更一般的优化问题[95]，将该定理与其他程序变换技术结合以进一步优化结果[72]。

2.2　归纳程序合成技术

归纳程序合成会在程序空间中搜索，直到找到满足给定规约的程序。其主要挑战在于搜索效率，因为程序空间通常非常大，甚至可能是无限的。为了解决这一挑战，现有研究对不同的程序空间与规约形式提出了许多针对性的高效方法。按照技术特点，现有方法可以被分为四类：基于枚举的合成方法[28,29]，基于约束求解的合成方法[30,31]，基于空间表示的合成方法[32–35] 和基于合一化的合成方法[36–38]。同时，近期也有一些工作尝试将这些合成方法与概率模型相结合，以进一步提升效率[39–41]。

然而，尽管已经存在大量的归纳程序合成方法，多数情况下的归纳合成仍然是低效的。如图2.2所示，现有方法只在一个名为**样例编程**的特例上，展现出了相对突出的求解能力。本节将先引入样例编程问题，再进一步介绍归纳程序合成的几类方法。

2.2.1　样例编程与反例制导的归纳程序合成

样例编程[96] 要求规约以输入输出样例的形式给出，例如 $f(1,2) = 3$。此外，输入输出形式的逻辑规约也可以通过反例制导的归纳程序合成框架（CEGIS）简化为样例

图 2.3　通过反例制导的程序合成将逻辑规约转为输入输出样例。

形式[43]。具体而言，给定规约 $\Phi = \exists f, \forall \overline{x}, f(\overline{x}) = ref(\overline{x})$，CEGIS 的运行流程如图2.3所示。它以迭代的方式进行程序合成，并在合成过程中维护一个输入输出样例的集合（初始为空）。在每一轮中，其合成过程如下所示。

- 首先，CEGIS 在当前的输入输出样例集合上调用样例编程求解器，并得到一个满足所有已知样例的候选程序。
- 接着，CEGIS 用一个验证器验证当前候选程序是否满足逻辑规约。如果满足，说明已经得到了一个正确程序，则将其作为结果返回。否则，CEGIS 要求验证器返回一个能拒绝当前程序的反例，将该反例加入样例集合，并开始下一轮迭代。

2.2.2　不同类别的归纳程序合成方法

接下来，本文将分类介绍现有的归纳程序合成方法。为了方便，本节只考虑样例编程问题，并假设程序空间由上下文无关文法定义。

基于枚举的合成方法　基于枚举的合成方法依次遍历程序空间中的程序，直到满足规约。现有方法一般按照从小到大、自底向上的顺序枚举[28]。对于每一个大小，该方法会尝试所有可能的产生式，并用较小的子程序构造当前大小的程序集合。

完全遍历所有程序往往会带来不必要的开销，因为程序空间中存在着等价程序，例如 $x+y$ 与 $y+x$。为了改进这一点，Abhishek 提出了剪枝策略观察等价法 [29]。对于每个枚举到的程序，观察等价法都会计算并记录该程序在每个样例上的输出，并跳过所有在样例集合上产生重复输出的程序。

基于约束求解的合成方法　给定上下文无关文法 G 与一系列输入输出样例 $I_i \mapsto O_i$，基于约束求解的合成方法将 G 中程序的语法与语义编码成约束，并使用 SMT 求解器解出满足所有输入输出样例的程序。具体而言，该方法需要编码两类约束。

- 结构约束 $\phi_s(\overline{x_s})$，其中 $\overline{x_s}$ 是一系列的结构变量。该约束要求 $\overline{x_s}$ 对应一个合法程序，且该程序被包含在程序空间中。
- 语义约束 $\phi_v(\overline{x_s}, \overline{x_i}, x_o, \overline{x_t})$，其中 $\overline{x_s}$ 是结构变量，$\overline{x_i}$ 是一系列代表输入的变量，x_o 是一个代表输出的变量，$\overline{x_t}$ 是一系列的临时变量。该约束要求 $\overline{x_s}$ 对应的程序在

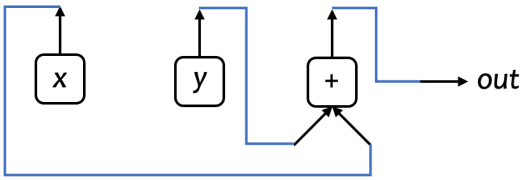

图 2.4　基于电路表示编码程序 $y+x$，其中黑色箭头代表每个组件的输入输出，蓝色线条代表电路连接。

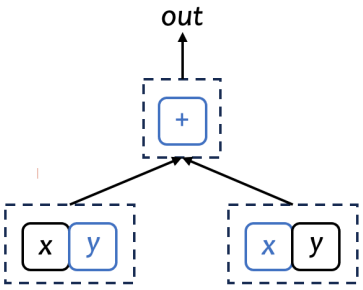

图 2.5　基于语法树编码程序 $y+x$，其中虚线矩形代表语法树的节点，蓝色矩形代表选中的组件。

输入 $\overline{x_i}$ 时的输出恰好是 x_o。

给定编码方法，满足下列约束的一组结构变量 $\overline{x_s}$ 对应着一个合法程序，它可以被 SMT 求解器解出。

$$\exists \overline{x_s}, \left(\phi_s(\overline{x_s}) \wedge \bigwedge_{I_i \mapsto O_i} \exists \overline{x_t^i}, \phi_v(\overline{x_s}, I_i, O_i, \overline{x_t^i}) \right)$$

目前常用的编码方法有两种，分别是基于**电路表示**的编码方法 [30] 和基于**语法树**的编码方法[31]。其中基于电路表示的编码方法由 Jha 等人于 2010 年提出。图 2.4 展示了该编码方法的一个实例。该编码方式假设用户给定了一系列可用的组件，包括所有的输入变量、可用的常量以及操作符，并将每一个程序视为连接这些组件的一个电路。在这一编码方式中，结构约束用于确保电路的合法性，包括电路无环，以及每个组件的输入（输出）都被唯一地连接到其他组件的输出（输入）；而语义约束则需要限制每个组件的行为，以保证全局输出与电路连接一致。

基于语法树的编码方法由 Mechtaev 等人于 2018 年提出[31]。该方法假设目标程序的语法树结构已经给定，并用结构变量表示每个节点使用的组件。图 2.5 展示了基于语法树编码程序的一个实例。在这种编码方式下，因为语法树的结构已经固定，所以结构约束只需要保证每个语法树节点恰有一个组件被选中；而语义约束则需要限制每个语法树节点的行为，以保证全局输出与语法树一致。

基于空间表示的合成方法　给定上下文无关文法 G 与一系列输入输出样例 $I_i \mapsto O_i$，基于空间表示的合成方法使用**变形代数空间** (VSA)[97] 表示 G 中满足输入输出样例的所有程序，并将该空间中的任意一个程序作为结果返回。

VSA 可以视为一种特殊的上下文无关文法，它包含了如下三类节点。

- 第一类节点直接表示了一个程序集合 $\{p_1, \ldots, p_k\}$。
- 第二类节点 $\mathbf{U}(N_1, \ldots, N_k)$ 表示了一些其他节点对应的程序集合的并。

- 第三类节点 $f(N_1, \ldots, N_k)$ 表示所有形如 $f(p_1, \ldots, p_k)$ 的程序形成的集合，其中 p_i 在节点 N_i 对应的程序集合中。

基于空间表示的合成方法要求问题中的上下文文法 G 符合 VSA 的形式。此时，G 中所有满足样例 e 的程序形成的 VSA（记作 $G[e]$）可以通过在节点上标记输出构造。具体而言，$G[e]$ 中的每一个节点都对应着 G 中的一个节点 S 和一个可能的输出 v，记作 $\langle S, v\rangle$，它对应的程序集合是 S 的一个子集，且恰好包含在样例 e 上输出为 v 的程序。

变形代数空间 $G[e]$ 的构造方法可以被分为自顶向下[33] 与自底向上[35] 两类。自顶向下的构造方法由 Polozov 等人于 2015 年提出[33]。该方法使用**见证函数**来将样例输出分解为可能的中间输出，并只对这些中间输出构造节点。

定义 7 (见证函数)**.** 给定全局输入 I 与操作符 $f(p_1, \ldots, p_k)$，见证函数 $\omega_{f,I}(v)$ 输出一个以 k 元组为元素的集合，满足如下条件。

$$\forall (v_1, \ldots, v_k) \in \omega_{f,I}(v). \qquad [\![f]\!]_I(v_1, \ldots, v_k) = v$$

其中 $[\![f]\!]_I(v_1, \ldots, v_k)$ 表示 f 在输入 $v_1, \ldots, v_k$ 且变量取值为 I 时的输出。

给定输入输出样例 $e = I \mapsto O$，自顶向下的方法会以如下方式构造节点 $\langle S, v\rangle$。

- 如果 $S = \{p_1, \ldots, p_k\}$，则 $\langle S, v\rangle \coloneqq \{p \in S \mid [\![p]\!](I) = v\}$，其中 $[\![p]\!](I)$ 表示程序 p 在输入为 I 时的输出。
- 如果 $S = \mathbf{U}(N_1, \ldots, N_k)$，则 $\langle S, v\rangle \coloneqq \mathbf{U}(\langle N_1, v\rangle, \ldots, \langle N_k, v\rangle)$。
- 如果 $S = f(N_1, \ldots, N_k)$，则计算见证函数 $\omega_{f,I}(v)$，为结果中的每个 k 元组构造对应的 VSA 节点，再将这些节点合并。

通过使用见证函数，自顶向下的方法只考虑可能会产生目标输出的中间结果，极大地减少了需要构造的 VSA 节点数量。

自底向上的构造方法由王新宇等人于 2017 年提出[35]。给定 VSA G 与输入输出样例 $e = I \mapsto O$，自底向上的方法通过如下方式构造 $G[e]$。

1. 对于 G 中的节点 $N = \{p_1, \ldots, p_n\}$，将集合 $\{p_i\}$ 并入节点 $\langle N, [\![p_i]\!](I)\rangle$。
2. 对于 G 中的节点 $N = \mathbf{U}(N_1, \ldots, N_k)$ 与每一个值 v，如果 $\langle N, v\rangle$ 未被创建且每一个 $\langle N_i, v\rangle$ 均非空，则创建节点 $\langle N, v\rangle \coloneqq \mathbf{U}(\langle N_1, v\rangle, \ldots, \langle N_k, v\rangle)$。
3. 对于 G 中的每一个节点 $N = f(N_1, \ldots, N_k)$ 与每一组值 $(v_1, \ldots, v_k)$，如果每一个 $\langle N_i, v_i\rangle$ 均非空，则将 $f\big(\langle N_1, v_1\rangle, \ldots, \langle N_k, v_k\rangle\big)$ 并入节点 $\langle N, [\![f]\!]_I(v_1, \ldots, v_k)\rangle$。
4. 重复步骤 2 和 3，直到 VSA 的结构不再改变或者迭代轮数到达上限。

基于上述构造方式，后续研究将抽象精化技术 [34] 与层次有限树自动机[98] 引入了构造过程，进一步提升了基于空间表示的程序合成方法的效率与泛用性。

基于合一化的合成方法 Kroening 等人于 2015 年提出了基于合一化的程序合成框架 (STUN)[36]。给定程序合成问题，该框架首先合成一些可能用于目标程序的组件，再使用特殊的合一操作符将这些组件合并成完整程序。常见的合一操作符包括分支操作符 *ite*、逻辑与 $\wedge$、逻辑或 $\vee$ 等。本文只对分支操作符 *ite* 进行介绍。

STUN 在合成分支程序时，要求文法 G 以 $S \to T \mid ite(C,T,T)$ 的形式给出，其中非终结符 C 与 T 分别代表分支条件与分支语句。STUN 框架下的程序合成器由语句合成器 $\mathfrak{T}$ 与合一器 $\mathfrak{U}$ 组成，其合成过程如下。

- 语句合成器 $\mathfrak{T}$ 生成一个能够覆盖所有样例的语句集合 T。
- 合一器 $\mathfrak{U}$ 合成合适的分支条件，并将 T 合并成一个满足所有样例的分支程序。

基于 STUN 框架，Alur 等人于 2017 年提出了一个高效的分支程序合成器 EUSolver [38]。它使用的语句合成器 $\mathfrak{T}_E$ 与合一器 $\mathfrak{U}_E$ 如下所示。

- $\mathfrak{T}_E$ 从小到大枚举可用的分支语句。对于每一个语句，如果它满足的样例集合与已有的语句均不相同，则把它加入到语句集合中。如果语句集合中的所有程序已经覆盖了所有样例，$\mathfrak{T}_E$ 便将它作为结果返回。
- $\mathfrak{U}_E$ 从小到大枚举可用的条件语句，并使用决策树学习算法 ID3[99] 来将语句集合合并成满足所有样例的分支程序。

与概率模型相结合的合成方法 随着机器学习技术的发展，也有一系列工作尝试使用概率模型加速已有的程序合成技术[39–41,100–102]。其中，Menon 等人于 2013 年使用概率上下文无关文法（PCFG）加速基于枚举的程序合成方法[101]。PCFG[103] 是一种经典的概率模型，它由一个上下文无关文法 G 与概率函数 γ 组成，其中 γ 为 G 中的每一条产生式赋予了一个概率值，表示其被选用的概率。为了保证概率是良定义的，γ 需要保证对 G 中的每一个非终结符 S，从 S 出发的所有产生式的概率和为 1。若程序 p 由产生式 $r_1, \ldots, r_n$ 得到，则 p 在 PCFG 中的概率为 $\prod_{i=1}^{n} \gamma(r_i)$。

概率上下文无关文法可以与自顶向下的枚举方法结合。在不引入 PCFG 时，自顶向下的枚举方法使用队列维护所有可能的语法展开结果，并每次将产生式应用于队首。而在引入概率模型后，则需要使用优先队列维护可能的展开结果，并总是将产生式应用于概率最大的情况。

Lee 等人于 2018 年对上述朴素方法作出了两点优化，并提出了程序合成器 Euphony[40]。首先，Euphony 使用 A* 算法枚举展开结果。对于每一个非终结符，定义其启发值为其对应程序中的概率最大值；对于每一个展开结果，定义其启发值为其概率乘以所有涉及的非终结符的启发值。可以证明，每一个展开结果的启发值一定是它能得到的程序的概率上界。利用这一点，Euphony 在使用优先队列维护展开结果时，会

将产生式应用于启发值最大的情况而非概率最大的情况。

其次，EUPHONY 使用了**概率高阶文法**来指导枚举过程。概率高阶文法 (PHOG)[104] 是一个由上下文无关文法 G、上下文函数 α 与概率函数 γ 组成的三元组。

- 给定包含非终结符的展开结果 p，$\alpha(p)$ 表示 p 中最左非终结符的上下文。
- 给定上下文 c 与产生式 r，$\gamma(c,r)$ 表示了产生式 r 在上下文为 c 时被选用的概率。

类似上下文无关文法，为了保证概率是良定义的，γ 需要对于任意上下文与任意 G 中的非终结符，从该非终结符所有产生式的概率和为 1。对于程序 p，若其最左展开使用的产生式为 $r_1,\dots,r_n$，且对应的上下文为 $c_1,\dots,c_n$，则 p 的概率为 $\prod_{i=1}^{n}\gamma(c_i,r_i)$。

PHOG 的表达能力严格强于 PCFG，因为后者是前者在上下文函数为常函数时的特例。因此，PHOG 能更精确地描述程序分布，从而更准确地引导枚举过程。实验结果表明，EUPHONY 在使用 PHOG 时比使用 PCFG 时多解出了 77.4% 的合成任务。

除了更改模型，选择适当的程序展开顺序同样可以提升概率模型的准确度，因为不同的展开顺序对应着不同的预测难度。基于这一观察，北京大学的熊英飞团队于 21 年提出了*扩展规则*的概念 [102]。在上下文无关文法的基础上，扩展规则显式地指定了展开顺序。具体而言，每个扩展规则由产生式 r 与位置符 p 组成，表示在已知产生式 r 中第 $(p+1)$ 个非终结符的情况下，可以扩展出产生式右侧的结果。当 $p=0$ 时，扩展规则与产生式等价，定义了一次自顶向下的展开；当 $p>0$ 时，它表示从子程序扩展到完整程序，定义了一次自底向上的展开；特殊地，$p=\perp$ 表示初始规则，它表示枚举的起点。实验表明，合适的扩展规则可以显著提升概率模型对程序合成的加速效果。

2.3　结合演绎与归纳的程序合成技术

尽管现有研究已经提出了大量演绎推理规则与归纳合成方法，但这些方法在求解算法问题时仍然存在其不足之处。

- 演绎推理的缺陷在于其**表达能力**。它优化程序的前提在于其变换规则足够表达从参考程序到目标程序的变换。然而，算法优化在很多情况下需要利用问题特定的性质，这些性质是难以预先预测的，因此也难以被预先定义的程序变换规则所覆盖。例如，目前最先进的演绎推理系统[105] 并不支持在变换过程中创造辅助值，但在本文收集的算法问题中，需要辅助值的任务占到了全部的 48%。
- 归纳程序合成的缺陷在于其**求解效率**。一方面，归纳程序合成是一个搜索过程，其时间开销会随着目标程序的变大而指数级上升；另一方面，算法问题中目标程序的规模往往很大，通常远远超出归纳合成技术的求解能力。因此，尽管目前已经研究尝试用归纳程序合成直接生成高效程序[16,17,106–108]，这些方法仍然只能支持线性结构的小规模程序（例如简单汇编程序），无法合成复杂算法。

```
def mts(xs):
  mts = 0
  for x in xs:
    mts = max(mts + x, 0)
  return mts
```

图 2.6　求解列表最大后缀和问题的从左到右计算。

```
def mts(xs):
  mts, tot = 0, 0
  for x in reversed(xs):
    mts = max(tot + x, mts)
    tot += x
  return mts
```

图 2.7　给定图 2.6 中的程序，由 Parsynt 推理出的从右到左计算。

为了克服这些缺陷，一些研究尝试结合演绎推理与归纳程序合成的优点，以高效地求解算法问题[10,25,27,62]。其中，演绎推理可以有效地化简程序合成问题，缓解归纳程序合成在效率上的挑战；而归纳程序合成可以在一个足够丰富的程序空间中搜索必要的组件，有效地提升推理系统的表达能力。

在这一框架下，多伦多大学的 Farzan 等人针对列表分治提出了 Parsynt 系统[10,62]。给定一个从左到右计算的参考程序，Parsynt 先用演绎变换将参考程序重写成从右到左计算的形式，并提取必要的辅助值；再使用归纳程序合成生成对应的合并操作。

以列表最大后缀和问题为例，图 2.6 展示了该问题上的一个从左到右计算。给定该参考程序，Parsynt 会枚举长度不同的变量列表，得到 *mts* 的符号输出，并尝试将该输出变形成从右到左计算的形式。以长度为 3 的列表为例，此时的变量列表为 $[a,b,c]$，而 *mts* 的符号输出为 $\max(\max(\max(a,0)+b,0)+c,0)$。为了得到从右到左计算，Parsynt 的变换目标是将这一符号输出拆分成关于 a 和 $mts([b,c])$ 的表达式，过程如下：

$$
\begin{aligned}
& \max(\max(\max(a,0)+b,0)+c,0) \\
\Longrightarrow\ & \max(a+b+c,b+c,c,0) \\
\Longrightarrow\ & \max(a+b+c,(\max(b,0)+c,0)) \\
\Longrightarrow\ & \max(a+\color{blue}{b+c},mts([b,c]))
\end{aligned}
$$

此时，拆分结果中除了 a 和 $mts([b,c])$，还包含了一个额外表达式 $b+c$。Parsynt 会将它泛化成列表求和函数 *sum* 在列表 $[b,c]$ 上的输出，并作为辅助值加入原程序。这样，Parsynt 可以成功地得到列表后缀和的一个从右到左计算，如图 2.7 所示。

因为 *mts* 在经过辅助值 *sum* 的扩充后既可以被从左到右计算也可以被从右到左计算，所以根据列表第三同态定理（定理 6），该扩充函数一定可以被分治计算，不需要更多的辅助值。因此，Parsynt 会使用归纳程序合成方法直接为 *mts* 与 *sum* 生成合并算子，从而规避了使用归纳程序合成寻找辅助值的难题。

使用类似的思路，北京大学的孙奕灿等人针对动态规划算法提出了 SynMem 系统[27]。该系统会使用演绎推理得到合适的动态规划状态，然后再使用归纳程序合成技

术生成对应的转移方程。

尽管这些系统已经可以合成相对复杂的程序，但它们仍然受限于演绎推理技术的表达能力缺陷，对使用场景有着严格的限制。Parsynt 需要一个从左到右计算作为输入，并不能支持其他形式的参考程序；而 SynMem 只能支持列表上的动态规划算法，难以被扩展到其他的数据结构。此外，这些系统在泛用性上也存在着局限性。它们的演绎推理系统均特化于各自的算法模式，很难被扩展到其他的算法模式上。

2.4　生成模型在算法问题上的应用

随着机器学习技术的发展，一些生成模型也可以被用于求解算法问题[63,64,109,110]。DeepMind 公司在 2022 年提出了自动编程模型 AlphaCode[109]，第一次在算法竞赛领域达到了人类选手的平均水平。该模型包含 410 亿参数，使用了基于 Transformer 的编码-解码架构。它首先使用了 GitHub 上的代码数据进行预训练，再在算法竞赛数据集上使用强化学习微调；最终在解题过程中，它会对候选程序聚类、用测试数据筛选，最终得到结果程序。实验结果表明，AlphaCode 可以在算法竞赛平台 Codeforces [111] 上击败接近 50% 的人类选手。

在此之后，随着大语言模型能力的提升，一些模型自然涌现出了对算法问题的求解能力 [63,64,112]。在这些模型的预训练数据中，包含了大量算法相关的材料，例如相关课程、教材、竞赛题目及题解等。因此，给定问题描述与时间复杂度需求作为提示，这些模型可以运行相关知识进行推理，并给出具有一定正确性与创新性的程序。

然而，相比于传统的程序合成方法，大语言模型的缺陷在于性质保障[113,114]。以正确性为例，因为大模型的推理基于自然语言而非严谨的逻辑系统，所以从原理上无法保证结果的正确性。即使在一些相对简单的编程任务上，它也可能会给出错误的程序。如图 2.8 所示，在将列表分治用于计算最大后缀和时，o3-mini 会将参考程序误认为是在求解最大子段和，从而返回完全错误的结果。

为了克服这一缺陷，已经有一些研究尝试将大语言模型与符号推理系统结合，由符号推理系统保障正确性，并由大语言模型加速搜索[115–117]。然而，这些研究目前还局限在几何题求解[115]、定理证明[116,117] 等特殊领域，无法被用于解决算法问题。

```python
def mts(xs):
  res = 0
  for i in range(len(xs)):
    now = 0
    for j in range(i, len(xs)):
      now += xs[j]
    res = max(res, now)
  return res
```

分治算法

线性时间

```python
def max_subarray(xs):
    def helper(lo, hi):
        # 处理区间 [lo, hi)
        # 若区间只有一个元素，返回对应信息
        if hi - lo == 1:
            x = xs[lo]
            return x, x, x, x  # (前缀, 后缀, 总和, 最大子数组和)
        mid = (lo + hi) // 2
        l_pre, l_post, l_total, l_best = helper(lo, mid)
        r_pre, r_post, r_total, r_best = helper(mid, hi)

        # 合并左右信息
        pre = max(l_pre, l_total + r_pre)  # 前缀最大和可能出现在左边或整个左段加右边的前缀
        post = max(r_post, r_total + l_post) # 后缀最大和可能出现在右边或整个右段加左边的后缀
        total = l_total + r_total          # 总和直接相加
        best = max(l_best, r_best, l_post + r_pre)  # 最大子数组和可能出现在左边、右边或跨越中点
        return pre, post, total, best

    if not xs:
        return 0  # 空数组特殊处理
    return helper(0, len(xs))[3]
```

图 2.8　在将分治用于求解最大后缀和时，O3-MINI 会错误地返回一个计算最大子段和的程序。

第三章　算法问题的归约方法

3.1　引言

使用程序合成求解算法问题时的一大挑战在于如何实现通用性。在算法设计领域存在着大量不同的算法模式，例如分治[20]、动态规划[21]、增量算法[9]、结构递归[22]、滑动窗口[23]、线段树[24] 等。因此，面向算法问题的程序合成方法必须具备足够的通用性，以支持多种不同模式下的程序合成。否则，若每次仅针对单一模式设计程序合成方法，则需要付出巨大的研究代价才能覆盖足够多的算法模式，难以实现广泛应用。

为了克服这一难题，本文提出了**提升问题**的概念，用以刻画不同算法模式在使用时的核心问题。具体而言，本文发现许多算法模式本质上都在描述对输入的遍历方式。这类算法模式存在平凡的应用方式：它先按照算法模式遍历输入，将完整的输入原样返回，再运行参考程序计算结果。作为例子，图 3.1 展示了应用列表分治时的平凡程序。它先用函数 `dac` 在分治模板的限制下还原了输入列表 `xs`，再在还原后的列表上运行了参考程序 `ref`，从而既使用了分治算法也保证了结果与参考程序一致。

然而，这一平凡程序并不能解决算法问题，因为它的运行效率严格比参考程序更差。而优化这一程序的关键在于如何改进对算法模式的使用，使函数 `dac` 在不完整返回输入的情况下，仍然能得到参考程序的输出。提升问题是对这一关键步骤的形式化，它定义了一个关于**中间数据结构消除**的程序合成问题。其目标是为一个数据结构合成一个高效形式，使得该数据结构上的一些特定计算可以被高效地完成。

在提升问题的基础上，本文进一步提出了 SuFu 语言，用于自动地将算法算法问题归约到提升问题。在使用这一语言时，用户需要提供算法问题的平凡程序（例如图 3.1）并在类型上标注需要被消除的中间数据结构。SuFu 会使用类型信息提取出所有与中间数据结构相关的计算，并抽取出对应的提升问题。图 3.2 展示了与图 3.1 对应的 SuFu 程序，其中 `Packed` 是用于标注中间数据结构的类型注释。

为了评估 SuFu 语言的能力，本文从现有研究中 [10,12,13,22,23,49,65,111,118–122] 收集了 290 个算法问题，构成一个多样化的数据集。通过使用 SuFu 语言对数据集中的所有问题进行描述，本文验证了该语言在不同算法模式上的通用性。同时，实验结果也证明了 SuFu 语言在归约时的有效性。它可以成功地将数据集中的所有问题归约到提升问题，平均用时仅为 0.02 秒。

综上所述，本章的主要创新点包括以下内容：

- 提出了一类新颖的程序合成问题，称为提升问题，以刻画不同算法模式在使用

```
def dac(xs, l, r):
  if r - l == 1:
    return [xs[l]]
  mid = (l + r) // 2
  lres = dac(xs, l, mid)
  rres = dac(xs, mid, r)
  return lres + rres

res = dac(xs, 0, len(xs))
return ref(res)
```

图 3.1　应用分治算法时的平凡解。

```
dac :: List -> Packed List
dac Single(x) = Single(x)
dac xs =
  let ls, rs = split xs in
  concat (dac ls) (dac rs)

main xs = ref (dac xs)
```

图 3.2　与图 3.1 对应的 SuFu 程序

时的核心问题（第 3.2 节）。

- 设计了一种用于描述算法问题的语言 SuFu（详见 第 3.3 节），同时提出了一种自动化方法，可将用 SuFu 描述的算法问题归约为提升问题（详见 第 3.4 节）。
- 构建了一个包含 290 个算法问题的数据集，并通过实验评估证明了 SuFu 语言的通用性与有效性（第 3.5 节）。

3.2　提升问题

3.2.1　问题定义

提升问题的定义依赖于多项式函子的概念（第 2.1.2.1 节），本文在此稍作回顾。

定义 8 (多项式函子)**.** 一个多项式函子 F 将类型映射到类型，函数映射到函数。它由四个基础函子通过特定文法组合而成，如下所示，其中 $\mathbb{F}$ 是文法中的非终结符，I 表示单位函子，$!T$ 表示对应类型 T 的常函子，而 $\times$ 与 $+$ 分别对应集合的直积与直和。

$$\mathbb{F} \coloneqq \mathsf{I} \mid !T \mid \mathbb{F} \times \mathbb{F} \mid \mathbb{F} + \mathbb{F}$$

多项式函子对应的类型映射与函数映射如下所示：

类型映射	$\mathsf{I}T$	$\coloneqq$	T	$(\mathsf{F}_1 \times \mathsf{F}_2)\,T$	$\coloneqq$	$\mathsf{F}_1 T \times \mathsf{F}_2 T$
	$(!T')\,T$	$\coloneqq$	T'	$(\mathsf{F}_1 + \mathsf{F}_2)\,T$	$\coloneqq$	$\left(\{1\} \times \mathsf{F}_1 T\right) \cup \left(\{2\} \times \mathsf{F}_2 T\right)$
函数映射	$\mathsf{I} f\ x$	$\coloneqq$	$f\ x$	$(\mathsf{F}_1 \times \mathsf{F}_2) f\ (x, y)$	$\coloneqq$	$\left(\mathsf{F}_1 f\ x, \mathsf{F}_2 f\ y\right)$
	$(!T') f\ x$	$\coloneqq$	x	$(\mathsf{F}_1 + \mathsf{F}_2) f\ (t,\ x)$	$\coloneqq$	$\mathsf{F}_t f\ x$

提升问题的定义如下所示。给定一个数据结构、该数据结构上的一系列操作，以及一个时间复杂度限制，提升问题的目标是为该数据结构合成一种高效形式，使得所有操作都可以在给定的时间复杂度内完成。

$$
\begin{array}{lcl}
\forall x : \texttt{Int} & & \\
comb_1\ x & = & repr\ [x] \\
\\
\forall xs, ys : \texttt{List} & & \\
comb_2\ (repr\ xs, repr\ ys) & = & repr\ (xs +\!\!+ ys) \\
\\
\forall xs : \texttt{List} & & \\
comb_3\ (repr\ xs) & = & ref\ xs
\end{array}
$$

```
def dac(xs, l, r):
  if r - l == 1:
    return comb1(xs[l])
  mid = (l + r) // 2
  lres = dac(xs, l, mid)
  rres = dac(xs, mid, r)
  return comb2(lres, rres)

res = dac(xs, 0, len(xs))
return comb3(res)
```

图 3.3　列表分治的提升问题以及对应的程序模板。

表 3.1　列表分治上提升问题的参数。

D	=	List	$\mathcal{T}$	=	$O(1)$			
F_1	=	!Int	G_1	=	I	$calc_1\ x$	=	$[x]$
F_2	=	$\mathsf{I} \times \mathsf{I}$	G_2	=	I	$calc_2\ (xs, ys)$	=	$xs +\!\!+ ys$
F_3	=	I	G_3	=	!Int	$calc_3\ xs$	=	$ref\ xs$

定义 9 (提升问题)**.** 给定数据结构 D，数据结构 D 上的一系列计算 $calc_i :: \mathsf{F}_i D \mapsto \mathsf{G}_i D$（其中 F_i 与 G_i 都是多项式函子），与时间复杂度限制 $\mathcal{T}$，提升问题的目标是合成重构函数 $repr :: D \mapsto D'$ 以及新数据结构 D' 上的一系列时间开销在 $\mathcal{T}$ 内的计算 $comb_i :: \mathsf{F}_i D' \mapsto \mathsf{G}_i D'$，使得 $comb_i$ 与 $calc_i$ 在 $repr$ 的映射下行为一致。其形式化规约如下所示：

$$\forall in \in \mathsf{F}_i D. \qquad comb_i\ (\mathsf{F}_i repr\ in) = \mathsf{G}_i repr\ (calc_i\ in) \tag{3.1}$$

例 4. 图 3.3 展示了将列表分治用于优化参考程序 ref 时的提升问题，其三行规约分别对应了分治的终止情况、对中间结果的合并、以及对最终输出的计算。将该问题的任意合法解填入右侧的程序模板，都能产生原算法问题的一个结果程序。表 3.1 列举了这一提升问题中各个参数的取值。

提升问题的规约（公式 3.1）考虑了输入空间 $\mathsf{F}_i D$ 中的所有输入。反例制导的归纳程序合成框架（CEGIS）可以将该规约简化为样例形式，其归约如下所示：

$$\forall (in \mapsto out) \in \mathbb{E}_i. \qquad comb_i\ (\mathsf{F}_i repr\ in) = \mathsf{G}_i repr\ out$$

其中 $\mathbb{E}_i$ 是关于 $calc_i$ 的一组输入输出样例，即 $\forall (in \mapsto out) \in \mathbb{E}_i, calc_i\ in = out$。

3.2.2　更多算法模式下的提升问题

```
state = comb1()
for v in xs:
    state = comb2(state, v)
return comb3(state)
```

图 3.4　流算法的程序模板。

$$
\begin{aligned}
&comb_1\ () && = && repr\ [] \\
&\forall xs : \texttt{List}.\ \forall v : \texttt{Int}. \\
&comb_2\ (repr\ xs, v) && = && repr\ (xs +\!\!+ [v]) \\
&\forall xs : \texttt{List} \\
&comb_3\ (repr\ xs) && = && ref\ xs
\end{aligned}
$$

图 3.5　流算法的提升问题。

表 3.2　流算法上提升问题的参数。

D	=	List	$\mathcal{T}$	=	$O(1)$			
F_1	=	!1	G_1	=	I	$calc_1\ x$	=	[]
F_2	=	I×!Int	G_2	=	I	$calc_2\ (xs, v)$	=	$xs +\!\!+ [v]$
F_3	=	I	G_3	=	!Int	$calc_3\ xs$	=	$ref\ xs$

流算法　流算法 [123] 是一个关于在线计算的算法模式，它在数据库和网络等领域有着广泛应用。图 3.4 展示了该模式的一个程序模板。给定输入列表，流算法会依次访问列表中的每个元素，在线地维护一个计算状态，并从最终状态中提取结果。

图 3.5 与表 3.2 分别展示了流算法对应的提升问题与该问题中各个参数的取值①。

- 表征函数 *repr* 定义了每个列表上的计算状态。
- 合并算子 $comb_1$ 计算空列表上的初始状态。
- 合并算子 $comb_2$ 的输入包括前缀列表 xs 上的计算状态以及当前元素 v，它需要计算长度增加 1 的列表上的计算状态。
- 合并算子 $comb_3$ 要从最终状态中提取参考程序的输出。

增量化算法　增量化算法 [9] 考虑了计算过程中对程序输入的修改，并建议在每次修改后直接更新原先的结果，而不是在新的输入上重新计算。不失一般性地，假设输入类型为 List，需要计算的函数为 $ref :: \texttt{List} \to \texttt{Int}$，而修改函数 *mod* 在接受一个类型为 M 的修改参数后，将原先的输入列表替换为一个新的输入。此时，增量化算法对应的提升问题如下所示，表 3.4 展示了该问题中各个参数的取值。

$$
\begin{aligned}
&\forall m : M.\ \forall xs : \texttt{List}. && comb_1\ (m, repr\ xs) && = && repr\ (mod\ m\ xs) \\
&\forall xs : \texttt{List}. && comb_2\ (repr\ xs) && = && ref\ xs
\end{aligned}
$$

- 表征函数 *repr* 定义了每个输入上需要被记录的信息。

① 在这些参数中，!1 代表一个大小为 1 的几何，其中唯一的元素是空元组 ()。

表 3.3　增量化算法上提升问题的参数。

D	=	List	$\mathcal{T}$	=	$O(1)$				
F_1	=	$!M \times \mathsf{I}$	G_1	=	I	$calc_1\ (m, xs)$	=	$mod\ m\ xs$
F_2	=	I	G_2	=	$!\mathtt{Int}$	$calc_2\ xs$	=	$ref\ xs$

```
info, res, l = comb1(), 0, 0
for r in range(len(xs)):
  info = comb2(info, xs[r])
  while l <= r and comb3(info):
    info = comb4(xs[l], info)
    l += 1
  res = max(res, r - l + 1)
return res
```

图 3.6　滑动窗口的程序模板

$$
\begin{aligned}
&comb_1\ () && = && repr\ [] \\
&\forall xs : \mathtt{List}.\ \forall v : \mathtt{Int}. \\
&comb_2\ (repr\ xs, v) && = && repr\ (xs +\!\!+ [v]) \\
&\forall xs : \mathtt{List}. \\
&comb_3\ (repr\ xs) && = && p\ xs \\
&\forall v : \mathtt{Int}.\ \forall xs : \mathtt{List}. \\
&v = head\ xs \rightarrow \\
&comb_4\ (v, repr\ xs) && = && repr\ (tail\ xs)
\end{aligned}
$$

图 3.7　滑动窗口的提升问题

- 合并算子 $comb_1$ 在接收修改参数后更新原先的信息。
- 合并算子 $comb_2$ 从目前记录的信息中提取出预期的计算结果。

最长子段问题　给定谓词 $p :: \mathtt{List} \rightarrow \mathtt{Bool}$ 和一个输入列表，最长子段问题的目标是在输入列表中选择一个连续子段，在满足谓词 p 的情况下最大化其长度。Zantema 在 1992 年研究了最长子段问题并提出了三种不同的算法模式[124]。这三种模式对应的算法问题都可以被规约到提升问题，此处本文将展示其中的第二种模式。

该算法模式又被称为**滑动窗口**，它以一种优化的方式枚举输入列表中的合法子段。图 3.6 展示了该模式的一个程序模板。

- 程序中包含了 4 个主要变量，其中 l 和 r 是当前子段的位置索引，它们对应着从第 l 个元素到第 r 个元素的连续子段；$info$ 记录着有关当前子段的信息；而 res 表示目前为止合法子段的最大长度。
- 该程序的枚举过程是一个双层循环。外循环将输入列表 xs 中的每个元素逐一追加到当前子段的末尾，内循环则重复删除当前子段的第一个元素，直到当前子段为空或已经满足谓词 p（由 $comb_3$ 判别）。
- 当谓词 p 是**前缀封闭**的时候，上述枚举过程一定能访问到最长的合法子段，其中前缀封闭要求 p 在接受一个列表的同时也接受该列表的所有前缀。

在求解关于滑动窗口的算法问题时，关键在于找到四个代码片段：$comb_1$ 提供空

表 3.4　滑动窗口上提升问题的参数。

D	=	List	$\mathcal{T}$	=	$O(1)$			
F_1	=	!1	G_1	=	I	$calc_1\ x$	=	[]
F_2	=	I×!Int	G_2	=	I	$calc_2\ (xs, v)$	=	$xs +\!\!+ [v]$
F_3	=	I	G_3	=	!Bool	$calc_3\ xs$	=	$p\ xs$
F_4	=	!Int × I	G_4	=	I	$calc_4\ (v, xs)$	=	$tail\ xs$

列表上的信息，$comb_2$ 和 $comb_4$ 分别在 r 和 l 增加 1 后更新子段的信息 $info$，而 $comb_3$ 则需要从当前的信息中判断谓词 p 是否成立。图 3.7 展示了滑动窗口上的提升问题，其中表征函数 $repr$ 定义了每个字段上需要被维护信息，而合并算子 $comb_1$ 至 $comb_4$ 需要与 $repr$ 保持一直①。表 3.4 列举了该提升问题中各个参数的取值。

线段树　线段树是一种经典的数据结构，用于处理列表上的子段修改与子段查询[122]。给定一个初始列表、一个查询函数 $query$、和一个修改函数 mod，在线性时间的预处理之后，线段树可以高效地计算 $query$ 在某子段上的值（例如计算从第 2 个元素到第 5000 个元素之间的第二小值）或者将修改函数 mod 应用于某个子段（例如将从第 2 个元素到第 5000 个元素中的每个元素都加 1），每个操作只需要花费对数的时间。

线段树的计算过程较为复杂，本文将其细节省略。简单地说，线段树使用分治算法响应查询，使用增量化算法响应修改。因此，线段树对应的提升问题可以被视为是分治算法提升问题与增量化算法提升问题的结合。该问题中最核心的几个合并算子如下所示，$comb_1$ 用于在分治时合并关于子段的信息，$comb_2$ 用于在修改时更新信息，而 $comb_3$ 则需要从子段信息中抽取出询问结果。

$$
\begin{array}{lll}
\forall xs, ys : \texttt{List}. & comb_1\ (repr\ xs, repr\ ys) & = repr\ (xs +\!\!+ ys) \\
\forall m : M.\ \forall xs : \texttt{List}. & comb_2\ (m, repr\ xs) & = repr\ (mod\ m\ xs) \\
\forall xs : \texttt{List}. & comb_3\ (repr\ xs) & = query\ xs
\end{array}
$$

3.3　SuFu 语言

虽然许多算法问题都能被归约到提升问题，但由于提升问题的定义涉及了许多抽象概念（例如多项式函子），这一过程对大多数开发者来说并不容易。为了解决这一问题，本文提出了 SuFu 语言来自动地将算法问题到样例形式的提升问题。

① 这儿 $comb_3$ 的规约与标准的提升问题略有不同，它包含了一个 $v = head\ xs$ 的前条件。然而，这一区别不会带来额外的挑战。该规约与标准的提升问题具有相同的样例形式，所以在 CEGIS 框架下它与提升问题完全等价。

```
dac :: List -> Packed List
dac Single(x) = Single(x)
dac xs =
  let ls, rs = split xs in
  concat (dac ls) (dac rs)

sndmin' xs = sndmin (dac xs)
```

图 3.8　SuFu 语言中的程序示例。

```
dac Single(x) = Single(x)
dac xs =
  let ls, rs = split xs in
  let lres = dac ls in
  let rres = dac rs in
  concat lres rres

sndmin' xs =
  let res = dac xs in
  sndmin res
```

图 3.9　给定图 3.8 中的程序，SuFu 的推理结果。

3.3.1　SuFu 语言概述

SuFu 语言的设计基于一个图灵完备的函数式语言，它使用算法模式的平凡应用来描述算法问题。图 3.8 展示了 SuFu 语言中的一个程序示例，它描述了将列表分治用于计算次小值的算法问题。在该程序中，`Single` 表示一个单元素列表，函数 `split` 将输入列表均分为左右两部分，而函数 `concat` 表示列表的拼接操作。

类型分析　在将算法问题归约到提升问题时，一个关键步骤在于识别所有与中间数据结构相关的计算。为了实现这一点，SuFu 引入了一个定制的类型系统，包含了与中间数据结构相关的类型构造与类型检查规则。在图 3.8 所示的程序中，函数 `dac` 的输出对应着需要被消除的中间数据结构。因此，其输出类型需要被 SuFu 提供的类型关键字 `Packed` 标记，从而指示 SuFu 产生关于消除该数据结构的提升问题。

给定包含 `Packed` 标记的描述程序，SuFu 会利用类型信息与约束求解系统自动地提取出程序中对 `Packed` 数据结构的所有操作。对于图 3.8 中的程序，SuFu 会提取出三处操作，如图 3.9 中的红色部分所示。按照自上而下的顺序，这三处操作分别对应了分治的边界情况，分治过程中对中间结果的合并，与分治结束时对列表次小值的计算。

SuFu 会将每个操作对应到一个合并算子，并通过类型分析得到每个合并算子对应的多项式函子。表 3.5 展示了 SuFu 的分析过程。以第一处操作为例，该操作只涉及变量 x，其类型为 `Int`，因此对应的输入函子 F_1 为 `!Int`。而该操作的输出类型是 `Packed List`，于是对应的输出函子 G_1 为单位函子 I。

样例收集　在确定合并算子的数量与类型后，构建样例形式提升问题的最后一步是获取这些合并算子的输入输出样例。为了完成这一步，SuFu 会产生一系列随机输入，在这些输入上运行原先的描述程序，跟踪所有操作片段的运行过程，并从中抽取样例。

例如，在随机输入 $[2, -1]$ 上运行图 3.9 中的描述程序时，SuFu 会收集到如下信息：

表 3.5　SuFu 对合并算子类型的推理过程

合并算子	操作片段	输入变量与类型	F_i	输出类型	G_i
$comb_1$	`Single(x)`	x : `Int`	`!Int`	`Packed List`	`!`
$comb_2$	`concat lres rres`	$lres, rres$: `Packed List`	`!`×`!`	`Packed List`	`!`
$comb_3$	`sndmin res`	res : `Packed List`	`!`	`Int`	`!Int`

- *dac* 的边界情况被被执行了两次，其中 x 分别为 2 和 -1，返回值分别为 [2] 和 $[-1]$。这些结果构成了关于合并算子 $comb_1$ 的两个样例，如下所示。

$$comb_1\ 2 = repr\ [2] \qquad comb_1\ -1 = repr\ [-1]$$

- *dac* 的中间情况被执行了一次，其中 *xs* 为 $[2,-1]$，递归结果 *lres* 与 *rres* 分别为 [2] 和 $[-1]$，最终返回值为 $[2,-1]$。这构成了关于 $comb_2$ 的一个样例，如下所示。

$$comb_2\ (repr\ [2], repr\ [-1]) = repr\ [2,-1]$$

- 最后，*sndmin* 在结果列表上被执行了一次，其中 *res* 为 $[2,-1]$，而 *sndmin* 的返回值为 1。这构成了关于 $calc_3$ 的一个输入输出样例，如下所示。

$$comb_3\ (repr\ [2,-1]) = 1$$

通过在大量随机输入上重复上述过程，SuFu 可以搜集到大量输入输出样例，从而得到样例形式的提升问题。

后续处理　SuFu 会将归约得到的提升问题交予后端的程序合成方法求解。在求解完成后，SuFu 将用合成结果中的合并算子 $comb_i$ 替换描述程序中对应的操作片段，从而将平凡的描述程序变换为算法问题的结果程序。

3.3.2　SuFu 的语言设计与类型系统

设计目标　给定包含 `Packed` 标注的描述程序，SuFu 首先需要提取所有操作 `Packed` 数据结构的程序片段。这些程序片段会被对应到提升问题中的合并算子，并在提升问题求解结束后被替换为相应的合成结果。直观上，这些程序片段需要满足如下条件：

- **正确性**：这些片段必须覆盖所有对 `Packed` 数据结构的构造与使用。因为我们的目标是消除所有的 `Packed` 数据结构，所以任何对 `Packed` 数据结构的构造与使用都不应当存在于结果程序，应当被合并算子所替换。
- **可行性**：这些片段的输出不能包含非 `Packed` 的数据结构。消除 `Packed` 数据结构

$$
\begin{array}{lcl}
t \in \texttt{Term} & := & \texttt{constants} \mid x \mid \texttt{app}(t_1, t_2) \mid \lambda(x, T, t) \mid \texttt{fix}(t) \mid \texttt{let}(x, t_1, t_2) \\
& \mid & \texttt{if}(t_1, t_2, t_3) \mid C(t) \mid \texttt{tuple}(\overline{t_i}) \mid \texttt{proj}(t, i) \mid \texttt{match}(t, \overline{C_i(x_i).t_i}) \\
& \mid & \texttt{rewrite(t)} \mid \texttt{label(t)} \mid \texttt{unlabel(t)} \\
T \in \texttt{Type} & := & B \mid T \to T \mid T_1 \times \cdots \times T_n \\
B \in \texttt{BaseType} & := & \texttt{Unit} \mid \texttt{Int} \mid \texttt{Bool} \mid B_1 \times \cdots \times B_n \mid \texttt{ind}(\overline{C_i}) \mid \texttt{Packed}(B) \\
E \in \texttt{EfficientType} & := & \texttt{Unit} \mid \texttt{Int} \mid \texttt{Bool} \mid E_1 \times \cdots \times E_n \mid \texttt{Packed}(B)
\end{array}
$$

图 3.10　中间语言 λ_{inter} 的完整语法，以抽象绑定树 [125] 的形式给出。

$\boxed{\Gamma \vdash_s t : T \text{ for } s \in \{\texttt{in}, \texttt{out}\}}$

(T-Rewrite)
$$\frac{\Gamma \vdash_{\texttt{in}} t : E \qquad E \in \texttt{EfficientType}}{\Gamma \vdash_s \texttt{rewrite}(t) : E}$$

(T-Label)
$$\frac{\Gamma \vdash_{\texttt{in}} t : B \qquad B \in \texttt{BaseType}}{\Gamma \vdash_{\texttt{in}} \texttt{label}(t) : \texttt{Packed}(B)}$$

(T-Unlabel)
$$\frac{\Gamma \vdash_{\texttt{in}} t : \texttt{Packed}(T)}{\Gamma \vdash_{\texttt{in}} \texttt{unlabel}(t) : T}$$

图 3.11　中间语言 λ_{inter} 中与中间数据结构相关的类型检查规则。

意味着将它们替换成一些标量值，这一过程通常会伴随着信息丢失。由于在归约得到的提升问题中，合并算子需要与操作片段保持一致的输入输出行为，因此如果存在着某个片段输出了不会被消除的数据结构，那么其合并算子将需要在丢失信息后仍然还原出一个完整的数据结构，这通常是不可能的。

- **最小性**：这些片段的总大小应当尽可能小。因为这些片段会被提升问题的合成结果替换，所以它们的总大小越小，需要被替换的部分也越小，那么对应的提升问题就越可能简单。

然而，从 SuFu 程序中直接提取满足以上条件的程序片段十分困难。这是因为 SuFu 语言混用了 Packed 数据结构与其他数据结构，导致难以区分与 Packed 相关的操作。

为了解决这一问题，本文额外设计了一个中间语言 λ_{inter}，其中所有对中间数据结构的构造与使用都需要通过语言给定的操作符显示地进行。通过设计该语言上的类型系统，本文保证了 λ_{inter} 中的任何一个类型良好的程序都对应了一组正确且可行的操作片段。此时，对操作片段的提取可以被视为一个翻译问题：需要将一个 SuFu 程序翻译到 λ_{inter}，并最小化操作片段的总大小。该问题可以被约束求解方法解决。

语言设计　图 3.10 展示了 SuFu 语言与中间语言 λ_{inter} 的语法，其中高亮部分是只属于 λ_{inter} 的语言构造，ind(·) 表示归纳数据类型，而 C 表示数据结构的构造函数。

λ_{inter} 在简单类型 λ 演算的基础上，扩充了以下有关中间数据结构的构造。

- 在类型层面，SuFu 加入了表示中间数据结构的类型构造 Packed。
- 在语句层面，SuFu 引入了三个新的操作符：label、unlabel 和 rewrite。其中，label 和 unlabel 是 Packed 类型的构造函数和析构函数；例如，label [1, 2] 的

类型为 Packed List 且 unlabel (label [1,2]) = [1,2]。而 rewrite 用于标记对 Packed 数据结构的操作片段；rewrite t 与 t 具有相同的类型和行为，但它会被视为 SuFu 的提取结果，并最终被替换为合并算子。

此外，λ_{inter} 区分了所有类型中的两个子类，其中基本类型 BaseType 包含了所有数据类型，即所有与函数无关的类型；而高效类型 EfficientType 是 BaseType 的一个子类，包含了所有满足可行性要求的输出类型。

图 3.11 展示了 λ_{inter} 中与 Packed 数据结构相关的类型规则。

- 为了保证**正确性**，λ_{inter} 在其类型判断 $\Gamma \vdash_s t : T$ 中加入了作用域 s，表示当前语句是否在 rewrite 的范围内。在此基础上，规则 T-Label 与 T-Unlabel 仅允许在 rewrite 的范围内使用 label 与 unlabel，从而确保被 rewrite 提取的程序片段覆盖了对 Packed 数据结构的所有构造与使用。
- 为了保证**可行性**，规则 T-Rewrite 将 rewrite 的输出类型限制在高效类型中，从而确保被提取的程序片段不会返回非 Packed 的数据结构。

例 5. 考虑图 3.8 中的 sndmin′ 程序。在这一程序中，用户将 *dac* 的输出类型标注为中间数据结构，从而在 λ_{inter} 中产生了一个类型错误：*dac* 的输出类型是 Packed List，但 sndmin 的输入类型是 List。SuFu 会分析这一类型错误，并在程序的合适位置插入操作符 label、unlabel 和 rewrite。考虑以下四种不同的方案。

```
sndmin (unlabel (dac xs))                                   (A)
sndmin (rewrite (unlabel (dac xs)))                         (B)
rewrite (sndmin (unlabel (dac xs)))                         (C)
let res = dac xs in rewrite (sndmin (unlabel res))          (D)
```

方案 *A* 插入了一个 unlabel，从而为 *sndmin* 提供正确的输入类型；然而，它仍然类型错误，因为这个 unlabel 并不在 rewrite 的范围内，违背了正确性与规则 T-Unlabel。

方案 *B* 确实用 rewrite 覆盖了 unlabel，但它仍然类型错误，因为其 rewrite 的类型为 List，违背了可行性要求与规则 T-Rewrite。它对应的提升问题要求从表征函数的输出（通常是一些整数）重建完整的列表，这是不可能的。

最后，方案 *C* 和 *D* 都是类型良好的，而 SuFu 最终会采用方案 *D*，因为该方案中 rewrite 片段大小更小。

代码翻译 给定只包含 Packed 标注的程序，SuFu 会自动地插入操作符 label, unlabel 与 rewrite，以得到一个类型良好的 λ_{inter} 程序，并最小化 rewrite 片段的总大小。在这些限制下，该问题等价于如下所示的代码翻译问题。

定义 10 (代码翻译问题)**.** 给定一个 SuFu 语言中的参考程序 p，代码翻译问题的目标是找到一个 λ_{inter} 中的程序 p'，满足：

1. p' 可以在 p 的基础上通过以下两类操作得到：(1) 在任意位置添加操作符 `label`，`unlabel` 和 `rewrite`，(2) 将任意表达式通过 `let` 绑定替换为变量。
2. p' 在 λ_{inter} 中类型正确。
3. p' 中被 `rewrite` 覆盖的程序片段的总大小尽可能地小。

值得注意的是，代码翻译问题中 SuFu 能进行的所有变换都不会改变程序语义，因此翻译结果一定与原程序具有相同的输入输出行为。

SuFu 会将上述翻译问题归约到 MaxSAT 问题，并调用现有的约束求解器求解。

- 首先，SuFu 会引入一系列的布尔变量，将搜索空间（上述条件 1）编码为一个符号程序。对参考程序中的每个位置，这些布尔变量控制了是否插入操作符、插入哪个操作符、以及是否将当前子表达式通过 `let` 绑定替换成变量。
- 接着，SuFu 会对符号程序进行类型检查，将类型约束（上述条件 2）编码为硬约束吗，并将最小化 `rewrite` 片段大小的目标编码为软约束。
- 在得到所有约束后，SuFu 直接使用 MaxSAT 求解器找到最优的翻译结果。

3.3.3　用 SuFu 编写算法模板

许多算法模式在 SuFu 中都可以被编写成一个模板函数，该函数接收参考程序和全局输入，并返回参考程序的输出。这些模板函数可以由算法专家编写，并作为一个算法库提供给用户；而用户在使用时只需要将参考程序传递给对应的模板函数，并不需要考虑任何与算法模式、SuFu 语言相关的细节。本文将以几个常见算法模式为例，展示它们在 SuFu 中的算法模板以及使用方式。

列表分治　图 3.12 展示了列表分治的算法模板以及 SuFu 的翻译结果。在模板中，函数 `dac'` 以列表分治的方式返回了完整的输入，其输出类型也被对应地标记为了中间数据结构。SuFu 可以正确地识别出该模板对中间数据结构的三处操作，其中前两处在分治过程中构建中间数据结构，而最后一处用于计算 `ref` 的输出。

利用该模板，用户可以直接通过如下程序描述将列表分治用于次小值的算法问题。

```
sndmin' xs = dac sndmin xs
```

流算法　图 3.13 展示了分治的算法模板以及翻译结果。在模板中，函数 `stream′` 将完整的前缀作为计算状态，从而把完整输入返回给参考程序 `ref` 。而因为这一计算状态

```
dac' :: List -> Packed List
dac' Single(x) = Single(x)
dac' xs =
  let ls, rs = split xs in
  let lres = dac' ls in
  let rres = dac' rs in
  concat lres rres

dac ref xs = ref (dac' xs)
```

```
dac' :: List -> Packed List
dac' Single(x) =
  let xs = Single(x) in
  rewrite (label xs)
dac' xs =
  let ls, rs = split xs in
  let lres = dac' ls in
  let rres = dac' rs in
  rewrite (label (concat (unlabel
      lres) (unlabel rres)))

dac ref xs =
  let res = dac' xs in
  rewrite (ref (unlabel res))
```

图 3.12　分治算法在 SuFu 中的算法模板（左侧）以及翻译后的 λ_{inter} 程序（右侧）。

```
stream' :: List -> Packed List
    -> Packed List
stream' Nil state = state
stream' Cons(h, t) state =
  let state' = append h state in
  stream' t state'

stream ref xs =
  let res = stream' xs Nil in
  ref res
```

```
stream' :: List -> Packed List ->
    Packed List
stream' Nil state = state
stream' Cons(h, t) state =
  let state' = rewrite (label (
      append h (unlabel state))) in
  stream' t state'

stream ref xs =
  let state = rewrite (label Nil) in
  let res = stream' xs state in
  rewrite (ref (unlabel res))
```

图 3.13　流算法在 SuFu 中的算法模板（左侧）以及以及翻译后的 λ_{inter} 程序（右侧）。

既会作为第二个输入传递给 stream′ 也会作为输出返回，所以 stream′ 的第二个输入类型与输出类型都需要被标记为中间数据结构。

滑动窗口　图 3.14 展示了滑动窗口在 SuFu 中的算法模板，它与图 3.6 严格对应。在该模板中，函数 sindows′ 接受四个输入，分别对应最长子段问题的谓词，从当前子段左端点开始的后缀，在当前子段右端点之后的后缀，以及当前子段上的信息。它会返回之后遍历的所有子段中，满足谓词 p 的最大长度。该模板将完整的子段作为信息记录在 sindows' 的第四个输入中，因此这一输入也被标记为了中间数据结构。

函数 swindow′ 的核心是它的第三种情况，即当前子段不是后缀的情况。此时，该函数会先计算子段的长度并判断它是否满足谓词 p。如果子段为空或 p 已被满足，那么它将会把当前子段的右端点向右移动一，即弹出第三个参数中的第一个元素，并把该元素放到当前子段的末尾。否则，函数 swindow′ 会把当前子段的左端点向右移动一，

```
swindow' :: (List -> Bool) -> List -> List -> Packed List -> Int
swindow' p Nil Nil seg = 0
swindow' p Cons(h, t) Nil seg =
  if p seg then len seg
  else swindow' p t Nil (tail seg)

swindow' p Cons(lh, lt) Cons(rh, rt) seg =
  let l = len seg in
  if (p seg) or (l == 0) then
    max l (swindow' p Cons(lh, lt) rt (append rh seg))
  else
    swindow' p lt Cons(rh, rt) (tail seg)

swindow p xs = swindow' p xs xs Nil
```

图 3.14　滑动窗口在 SuFu 中的算法模板。

即同时弹出第二个参数中的第一个元素与当前子段的第一个元素。

图 3.15 展示了 SuFu 对该模板的翻译结果。SuFu 可以正确地识别出所有对子段信息的操作，并提取出 7 处相关的操作片段。

3.4　算法问题到提升问题的自动规约方法

给定中间语言 λ_{inter} 中的描述程序，SuFu 会将每个 `rewrite` 片段对应到提升问题中的一个合并算子，并分两步得到样例形式的提升问题。

类型分析　为了收集合并算子对应的函子 F_i 与 G_i，SuFu 会跟踪描述程序的类型检查过程，并提取出所有对 `rewrite` 语句的类型判断，即 $\Gamma \vdash_s \texttt{rewrite}\ t : T$。SuFu 会将 Γ 中所有的非函数变量整理成一个元组[①]，并提取出该元组类型 T_{inp}。接着，SuFu 会在保证 T_{inp} 与 T 结构的情况下，将所有的 `Packed` 类型替换为单元函子 I，其他基本类型替换成对应的常函子，从而得到对应的多项式函子。更严谨地，F_i 与 G_i 为满足下列等式的多项式函子，其中 T_D 代表中间数据结构的实际类型。

$$\mathsf{F}_i(\texttt{Packed}\ T_D) = T_{inp} \qquad \mathsf{G}_i(\texttt{Packed}\ T_D) = T$$

以图 3.12 中的右侧程序为例，其第三个 `rewrite` 语句的类型判断如下所示。

$$\texttt{xs} : \texttt{List}, \texttt{res} : \texttt{Packed List} \vdash_{\text{out}} \texttt{rewrite (ref (unlabel res)) : Int}$$

其中输出类型 T 为 `Int`，输入类型 T_{inp} 按照 `(xs,res)` 的顺序可以被整理为 `List` ×

① 因为合成算法的局限性，本文目前不考虑将函数作为合并算子输入时的情况。

```
swindow' :: (List -> Bool) -> List -> List -> Packed List -> Int
swindow' p Nil Nil seg = 0
swindow' p Cons(h, t) Nil seg =
  if p seg then rewrite (len (unlabel seg))
  else let seg' = (rewrite (label (tail (unlabel seg)))) in
    swindow' p t Nil seg'

swindow' p Cons(lh, lt) Cons(rh, rt) seg =
  let l = rewrite (len (unlabel seg)) in
  if (rewrite (p (unlabel seg))) or (l == 0) then
    let seg' = rewrite (label (append rh (unlabel seg))) in
    max l (swindow' p Cons(lh, lt) rt seg')
  else
    let seg' = (rewrite (label (tail (unlabel seg)))) in
    swindow' p lt Cons(rh, rt) seg'

swindow p xs = swindow' p xs xs (rewrite (label Nil))
```

图 3.15　SuFu 对滑动窗口模板（图 3.14）的翻译结果。

$$\dfrac{\dfrac{\cdots}{(\texttt{xs} \mapsto [2,-1]) \vdash \texttt{dac}'\ \texttt{xs} \Downarrow \texttt{label}\ [2,-1]} \qquad \dfrac{\cdots}{(\texttt{xs} \mapsto [2,-1], \texttt{res} \mapsto \texttt{label}\ [2,-1]) \vdash \texttt{rewrite (ref (unlabel res))} \Downarrow 2}}{(\texttt{xs} \mapsto [2,-1]) \vdash \texttt{let res = dac}'\ \texttt{xs in rewrite (ref (unlabel res))} \Downarrow 2}$$

图 3.16　运行 dac sndmin $[1,-2]$ 时的部分过程。

(Packed List)。此时，F_3 与 G_3 分别等于 !List × I 与 !Int，对应如下方程。

$$\mathsf{F}_3(\texttt{Packed List}) = \texttt{List} \times (\texttt{Packed List}) \qquad \mathsf{G}_3(\texttt{Packed List}) = \texttt{Int}$$

样例收集　SuFu 通过追踪描述程序在随机输入上的运行收集合并算子的输入输出样例。这一过程基于 λ_{inter} 的大步环境语义 [126]。该语义的优点在于其执行过程严格遵循语法结构，我们可以从中直接获取 rewrite 语句的输入输出行为。

以图 3.12 右侧的描述程序为例，图 3.16 展示了 dac sndmin 在输入 $[2,-1]$ 上的部分执行过程。在大步环境语义中，程序执行的形式为 $E \vdash t \Downarrow v$，其中 E 是执行环境，它记录了每个自由变量的赋值，t 表示需要被执行的语句，而 v 是执行结果。

在这一执行过程中，右侧分支执行了程序 dac 中的第三个 rewrite 语句，它在环境 $(\texttt{xs} \mapsto [2,-1], \texttt{res} \mapsto \texttt{label}\ [2,-1])$ 下的执行结果为 2。该语句对应着提升问题中的第三个合并算子 $comb_3$，因此 SuFu 会将上述执行结果提取为样例 $([2,-1],[2,-1]) \mapsto 2$，并加入样例集合 $\mathbb{E}_3$。该样例表示了如下所示的归约。

$$comb_3\ ([2,-1], repr\ [2,-1]) = 2$$

表 3.6　本文数据集中 SuFu 程序的统计信息。

来源	# 任务数	程序大小		Packed 标注数量		
		均值	最大值	均值	最大值	不同的数据结构数量
融合	16	126.5	305	1.3	2	7
递归	178	157.5	484	1.1	3	35
其他	96	251.2	843	1.0	1	1
总计	290	186.8	843	1.1	3	37

更一般性地，对于每次执行，SuFu 都会提取出执行树中所有关于 `rewrite` 语句的部分，即 $E \vdash \texttt{rewrite}(t) \Downarrow v$。SuFu 会将环境 E 中的所有值按照顺序整理成一个元组 inp（变量顺序与类型分析时一致），再将 $inp \mapsto v$ 记录到对应的样例集合 $\mathbb{E}_i$ 中，其中 i 表示当前 `rewrite` 语句的编号。该样例在提升问题中表示了如下所示的约束：

$$comb_i\ (\mathsf{F}_i repr\ inp) = \mathsf{G}_i repr\ v$$

3.5　实验评估

为了评估 SuFu 语言的能力，本文设计了实验来回答以下的问题：

- **RQ1**: SuFu 的表达能力是否足够描述算法问题？
- **RQ2**: SuFu 是否能有效地将算法问题规约到样例形式的提升问题？

3.5.1　数据集构建

为了进行实验评估，本文从以下三个不同的来源收集到了 290 个算法问题。

- **融合**。融合问题的目标是消除低效程序中的中间数据结构，以达到加速的效果。尽管融合问题不属于传统意义上的算法问题，但是它们落在 SuFu 的能力范围内——只要用 `Packed` 关键字标注需要消除的中间数据结构即可。因此，本文从融合相关的文献中收集了 16 个任务 [49,65,118–120]，并将它们加入到了数据集中。
- **结构递归**。结构递归是一种常用于函数式编程的算法模式[22]。它要求按照给定的方式遍历输入，并在遍历中直接完成计算。本文从现有工作中收集了 178 个与结构递归相关的算法问题[22]，并将它们加入到了数据集中。
- **其他算法模式**。此外，本文还考虑了被现有工作关注的其他算法模式，例如分治 [10,12,121]、流算法 [13]、最长子段问题上的贪心算法 [23]、以及线段树 [122]。本文从这些相关工作以及算法竞赛平台 Codeforces [111] 处收集了 96 个与这些模式相关的算法问题，并纳入到数据集中。

该数据集具有显著的多样性。它涉及了 8 种不同的算法模式，并使用了包括列表、二叉树、抽象语法树在内的 37 种数据结构

表 3.7　SuFu 在将算法问题规约到提升问题时的统计信息。

来源	耗时	操作符数量	`rewrite` 语句大小
融合	0.02	8.56	16.00
递归	0.02	9.23	17.65
其他	0.04	9.55	16.39
总计	0.02	9.31	17.14

3.5.2　RQ1: SuFu 在描述算法问题时的表达能力

本文成功地用 SuFu 描述了数据集中的所有算法任务，验证了其表达能力。表 3.6 列举了这些 SuFu 程序的统计信息，其中“程序大小”展示了 SuFu 程序中的 AST 节点数量，而“`Packed` 标注数量”展示了被使用的 `Packed` 关键字数量。值得注意的是，在本文的数据集中，不同来源的任务呈现出了不同的挑战。

- 融合任务使用的 `Packed` 最多，代表它经常需要同时处理多个中间数据结构。
- 递归任务涉及 35 种数据结构，带来了不同数据结构上的通用性挑战。
- 其他算法模式上的任务具有最大的程序规模，带来了合成效率上的挑战。

3.5.3　RQ2: SuFu 在将算法问题规约到提升问题时的有效性

给定描述程序，SuFu 可以成功地将所有算法问题归约到提升问题。表 3.7 展示了归约过程中 SuFu 的具体表现，包括耗时、翻译时插入的操作符数量、以及被 `rewrite` 覆盖的程序片段总大小。值得注意的是，尽管 SuFu 的归约过程依赖 MaxSAT 求解器，在理论上具有指数级的时间复杂度，但是 SuFu 在实践中的归约速度极快，平均只需 0.02 秒。一个可能的解释是，实践中对中间数据结构的操作通常比较直接，使可行解的搜索空间远远小于其理论上界。

3.6　小结

本章主要关注求解算法问题时的通用性挑战。为了统一不同算法模式上的算法问题，本文提出了提升问题的概念，用于描述多个算法模式在使用时的核心问题，即如何为一个中间数据结构合成更加高效的数据表示，使得一些特定的计算可以被高效地完成。在提升问题的基础上，本文进一步地提出了 SuFu 语言，用于描述算法问题，并将它自动地归约到提升问题。这一归约过程分为两步，SuFu 首先会使用类型信息以及 MaxSAT 求解器将描述程序翻译到一个中间语言，从而识别出所有与中间数据结构相关的程序片段。接着，SuFu 会跟踪这些程序片段在随机输入上的运行行为，从而提取出提升问题所需的输入输出样例。

为了评估 SuFu 的有效性，本文从三个来源处收集了 290 个算法任务，并在这些

任务上检验了 SuFu 的表达能力与归约效率。实验结果表明，SuFu 可以成功描述数据集中的所有任务并高效地将它们归约到提升问题，平均耗时仅为 0.02 秒。

第四章　提升问题的分解方法

4.1　引言

程序合成方法在求解提升问题时的核心挑战在于求解效率不足。此类问题对应的目标程序往往规模庞大，远超现有程序合成方法的处理能力。为此，本文提出了一种基于**问题分解**的方法来应对这一挑战。该方法会将提升问题分解为一系列子任务，每个子任务仅需合成目标程序的一小部分，从而降低程序规模。

然而，通常来说，对程序合成任务进行有效分解是极具挑战性的。这是因为目标程序的不同子程序间往往高度依赖，难以针对特定子程序提取精确的规约。为了解决这一难题，本文使用了一种近似策略：在部分子任务中采用近似规约进行程序合成。该策略的核心在于，即使近似规约与精确规约间存在差异，只需要在最终阶段采用精确规约，便能保证整个合成过程的正确性；与此同时，在规约差异较小的情况下，此方法同样能够保证较高的合成完备性。

遵循这一思路，本文为提升问题提出了问题分解系统 AutoLifter。该系统包含两条针对性的分解规则：**组件消除**与**变量消除**。这两条规则分别利用了提升问题中的元组结构与函数复合结构，并通过近似规约打破子程序间的依赖关系。本文对这些近似规约进行了理论分析，并证明它们与精确规约足够接近，不会显著影响合成过程的完备性。此外，本文的实验结果也证实了 AutoLifter 在实践中的有效性。

综上所述，本章的主要创新点包括以下内容：

- 针对提升问题提出了两条分解规则，用于将提升问题分解为合成规模更小的子问题（第 4.3 节与第 4.4 节）。
- 分析了这些规则的理论性质，并证明了它们的正确性与近似完备性（第 4.5 节）。
- 实现了分解系统 AutoLifter 并通过实验评估证明了其有效性（第 4.6 节）。

4.2　分解过程概述

本节以列表分治计算次小值的算法问题为例，展示 AutoLifter 的分解过程。该问题的提升问题与目标程序如图 4.1 所示，其中三个合并算子分别对应分治的终止情况、对递归结果的合并、以及分治结束后对列表次小值的计算。同时，因为所有的输入输出样例都是从参考程序的运行过程中提取的，所以这些样例满足一些与分治相关的不变量。例如在终止情况中，输出列表 xs 一定是只包含输入元素 x 的单元素列表。

$$
\begin{aligned}
&\forall (x \mapsto xs) \in \mathbb{E}_1 \\
&comb_1\ x \quad = \quad repr\ xs \\
&\quad \textbf{where}\ xs = [x] \\
\\
&\forall \big((ls, rs) \mapsto xs\big) \in \mathbb{E}_2 \\
&comb_2\ (repr\ ls, repr\ rs) \quad = \quad repr\ xs \\
&\quad \textbf{where}\ xs = ls +\!\!+ rs \\
\\
&\forall (xs \mapsto v) \in \mathbb{E}_3 \\
&comb_3\ (repr\ xs) \quad = \quad v \\
&\quad \textbf{where}\ v = sndmin\ xs
\end{aligned}
$$

```
def repr(xs):
  return sndmin(xs), min(xs)

def comb1(x):
  return INF, x

def comb2(lres, rres):
  (l2, l1), (r2, r1) = lres, rres
  smin = min(l2, r2, max(l1, r1))
  return smin, min(l1, r1)

def comb3(res):
  return res[0]
```

图 4.1　对应将列表分治用于计算次小值的提升问题（左侧）与目标程序（右侧）。

组件消除　注意到目标的表征函数 *repr* 以及合并算子 $comb_1$ 与 $comb_2$ 都以一些组件的元组形式出现。表征函数 *repr* 包含了两个组件，分别计算列表次小值与列表最小值；对应地，合并算子 $comb_1$ 与 $comb_2$ 各包含了两个组件，分别在合并的过程中计算列表次小值与列表最小值。根据这一观察，组件消除规则的目标是将这些组件分解到不同的子问题，从而减少每次程序合成需要考虑的组件数量。

为了实现这一点，本文分析了表征函数 *repr* 需要满足的条件。

- 合并算子 $comb_1$ 要求表征函数 *repr* 在单元素列表上的输出可以被快速计算。它没有带来任何实质性的限制，可以被忽略。
- 合并算子 $comb_2$ 要求表征函数 *repr* 提供完善的信息，使得完整列表上 *repr* 的输出可以从子列表上的输出计算得到。
- 合并算子 $comb_3$ 要求表征函数 *repr* 提供足以计算列表次小值的信息。

值得注意的是，$comb_3$ 的条件允许 *repr* 输出任何冗余的信息。因此，我们可以先在只考虑 $comb_3$ 的情况下，找到表征函数中关于列表次小值的必要组件；再在这些组件的基础上考虑 $comb_2$ 中的条件，并补充必要的组件。

遵循这一思路，组件消除的第一个子问题将专注于 $comb_3$。该子问题的规约与目标程序如图 4.2 所示，其中 $repr_1$ 表示表征函数中的一部分组件，而 $comb_{3,1}$ 表示只考虑 $repr_1$ 情况下的合并算子。这一子问题会被 AutoLifter 中的第二条规则继续分解，详情见本节的后半部分。

在成功求解上述子问题后，我们可以得到 *repr* 的第一个组件以及 $comb_3$ 中只与这一组件相关的部分。接着，组件消除的第二个子问题会尝试将这些组件扩充成完整的 *repr* 与 $comb_3$。具体而言，在这些组件的基础上，最终的 *repr* 与 $comb_3$ 一定满足如下

$$\forall (xs \mapsto v) \in \mathbb{E}_3$$
$$comb_{3,1}\ (repr_1\ xs) \quad = \quad v$$
$$\textbf{where}\ v = sndmin\ xs$$

```
def repr1(xs):
  return sndmin(xs)

def comb3_1(res):
  return res
```

图 4.2　组件求解的第一个子问题（左）与目标程序（右）。

的形式，其中 $repr_2$ 表示 $repr$ 的剩余组件。

$$repr\ \ xs \coloneqq (repr_1\ xs, repr_2\ xs) \qquad comb_3\ res \coloneqq comb_{3,1}\ (fst\ res)$$

值得注意的是，因为 $comb_{3,1}$ 的输入只包含 $repr_1$ 的输出，所以它在 $comb_3$ 中只接收完整输入 res 的第一部分。

组件消除规则会将上述形式代入原先的提升问题（图 4.1），从而得到第二个子问题，其规约在化简后如下所示。

$$\forall\big((ls, rs) \mapsto xs\big) \in \mathbb{E}_2.$$
$$comb_2\ \big((repr_1\ ls, repr_2\ ls), (repr_1\ rs, repr_2\ rs)\big) \quad = \quad (repr_1\ xs, repr_2\ xs)$$
$$\forall (xs \mapsto v) \in \mathbb{E}_3. \qquad comb_{3,1}\ (repr_1\ xs) \ = \ v$$

其中关于 $comb_3$ 的部分已经不包含任何未知程序，可以被忽略；而关于 $comb_2$ 的部分则构成了一个新的提升问题，其样例集合 $\mathbb{E}_2'$ 与输入/输出函子 $(\mathsf{F}_2', \mathsf{G}_2')$ 如下所示。

$$\mathbb{E}_2' \coloneqq \big\{\big((repr_1\ ls, ls), (repr_1\ rs, rs)\big) \mapsto (repr_1\ xs, xs) \ \big|\ \big((ls, rs) \mapsto xs\big) \in \mathbb{E}_2\big\}$$
$$\mathsf{F}_2' \coloneqq (!\mathtt{Int} \times \mathsf{I}) \times (!\mathtt{Int} \times \mathsf{I}) \qquad \mathsf{G}_2' \coloneqq !\mathtt{Int} \times \mathsf{I}$$

因此，该规约仍然属于组件消除规则的适用范围，可以被递归地继续分解。

值得注意的是，组件消除的第一个子问题（图 4.2）使用的是一个近似规约。它只要求 $repr_1$ 为列表次小值提供足够的信息，却完全忽略了分治的计算过程。理论上，从该子问题中合成的 $repr_1$ 可能会输出一个古怪的、不能被分治高效计算的值，导致组件消除的第二个子问题无解。

变量消除　在递归地应用组件消除后，剩余的问题在于如何求解组件消除规则产生的第一类子问题（例如图 4.2）。变量消除规则会将这类问题进一步地分解为多个子问题，其中每个子问题只涉及唯一的一个未知程序。

以图 4.2 中的问题为例，为了解耦其规约中的两个未知程序 $repr_1$ 与 $comb_{3,1}$，变量消除会先推导得到一个只包含 $repr_1$ 的规约。直观地，一个合法的 $repr_1$ 应当保证存在

一个合并算子 $comb_{3,1}$ 可以满足 $\mathbb{E}_3$ 中的所有输入输出样例，其精确规约如下所示。

$$\exists comb_{3,1}.\, \forall (xs \mapsto v) \in \mathbb{E}_3.\quad comb_{3,1}\ (repr_1\ xs) = v \tag{4.1}$$

然而，这一规约并不能被直接用于合成 $repr_1$。它包含了一个作用在程序空间上的存在量词 $\exists comb_{3,1}$，但现有的程序合成方法均无法处理这样的复杂量词。

为了化简规约 4.1，注意到无论 $comb_{3,1}$ 是什么程序，它的输入输出行为一定是一个数学函数。因此，这一规约可以被近似为如下形式，其中 $\mathcal{F}$ 为所有函数形成的空间。

$$\exists f \in \mathcal{F}.\, \forall (xs \mapsto v) \in \mathbb{E}_3.\quad f\ (repr_1\ xs) = v$$

数学函数的优点在于它有一个简单且精确的定义：一个计算是函数当且仅当它在相同的输入下总是产生相同的输出。因此，上述近似可以被等价地改写为如下形式。

$$\forall (xs_1 \mapsto v_1), (xs_2 \mapsto v_2) \in \mathbb{E}_3.\quad repr_1\ xs_1 = repr_1\ xs_2 \to v_1 = v_2 \tag{4.2}$$

到此，原先难以处理的存在量词 $\exists comb_{3,1}$ 已经被完全消除。变量消除规则会将上述规约作为第一个子问题，从中合成 $repr_1$。在得到合成结果 $repr_1\ xs := sndmin\ xs$ 后，变量消除会将它代入原始问题（图 4.2），得到只涉及 $comb_{3,1}$ 的子问题，如下所示。

$$\forall (xs \mapsto v) \in \mathbb{E}_3.\quad comb_{3,1}\ (sndmin\ xs) = v \tag{4.3}$$

4.3 分解规则

4.3.1 组件消除

组件消除的输入是样例形式的提升问题，它包含一系列的规约，每条如下所示：

$$\forall (in \mapsto out) \in \mathbb{E}_i.\qquad comb_i\ (\mathsf{F}_i repr\ in) = \mathsf{G}_i repr\ out$$

不失一般性地，本文假设每条规约中的 G_i 只能是单位函子 G_i 或常函子 $!T$。它们分别对应了如下两种规约形式，本文将其称为**数据结构形式**与**常量形式**。

$$\forall (in \mapsto out) \in \mathbb{E}_i.\qquad comb_i\ (\mathsf{F}_i repr\ in) = repr\ out \tag{4.4}$$

$$\forall (in \mapsto out) \in \mathbb{E}_i.\qquad comb_i\ (\mathsf{F}_i repr\ in) = out \tag{4.5}$$

任何多项式函子 G_i 对应的规约都可以被拆分为这两种形式，如例 6 所示。

例 6. 当 $\mathsf{G} = \texttt{Int} + (\mathsf{I} \times !\texttt{Int})$ 时，其提升问题在展开函子 G 后如下所示。

$$\forall (in \mapsto out) \in \mathbb{E}.\qquad comb\ (\mathsf{F} repr\ in) = \begin{cases} v & \textbf{if}\quad out = (1, v) \\ (repr\ x, v) & \textbf{if}\quad out = (2, (x, v)) \end{cases}$$

该规约可以被拆分成以下四条子规约。第一条子规约判断输出属于 G 中的哪一条分支，第二条子规约处理左侧情况，最后两条规约处理右侧情况。在这些规约中，第一条第二条与第四条均符合样例形式，只有第三条符合数据结构形式。

$$\begin{array}{lll}
\forall (in \mapsto t) \in \mathbb{E}_1. & comb_1\ (\mathsf{F} repr\ in) = t & \textbf{where} \quad \mathbb{E}_1 \coloneqq \left\{ in \mapsto t \mid \left(in \mapsto (t, v)\right) \in \mathbb{E} \right\} \\
\forall (in \mapsto v) \in \mathbb{E}_2. & comb_2\ (\mathsf{F} repr\ in) = v & \textbf{where} \quad \mathbb{E}_2 \coloneqq \left\{ in \mapsto v \mid \left(in \mapsto (1, v)\right) \in \mathbb{E} \right\} \\
\forall (in \mapsto x) \in \mathbb{E}_3. & comb_3\ (\mathsf{F} repr\ in) = repr\ x & \textbf{where} \quad \mathbb{E}_3 \coloneqq \left\{ in \mapsto x \mid \left(in \mapsto (2, (x, v))\right) \in \mathbb{E} \right\} \\
\forall (in \mapsto v) \in \mathbb{E}_4. & comb_4\ (\mathsf{F} repr\ in) = v & \textbf{where} \quad \mathbb{E}_4 \coloneqq \left\{ in \mapsto v \mid \left(in \mapsto (2, (x, v))\right) \in \mathbb{E} \right\}
\end{array}$$

在求解得到 $comb_1, \ldots, comb_4$ 后，可以按照如下方式合并得到满足原规约的合并算子。

$$comb\ x \coloneqq \begin{cases} (1, comb_2\ x) & \textbf{if} \quad comb_1\ x = 1 \\ \left(2, (comb_3\ x, comb_4\ x)\right) & \textbf{if} \quad comb_1\ x = 2 \end{cases}$$

表征函数 $repr$ 在提升问题中的任务是为合并算子提供足够的信息，以完成预期的计算。组件消除规则利用的性质是，在提升问题中，目标的表征函数通常由两部分组成，即 $repr\ x \coloneqq (repr_1\ x, repr_2\ x)$，其中 $repr_1$ 为所有常量形式的规约提供足够的信息，而 $repr_2$ 为数据结构形式的规约补充信息，使得 $repr$ 的输出本身可以被计算。

于是，组件消除会按照如下方式拆分提升问题。

- 第一个子问题只考虑常量形式的规约，并合成一个能够提供足够信息的 $repr_1$。其规约如下所示，其中 $\mathbb{I}_v$ 表示所有常量形式规约的编号。

$$\forall i \in \mathbb{I}_v. \forall (in \mapsto out) \in \mathbb{E}_i. \qquad comb'_i\ (\mathsf{F}_i repr_1\ in) = out$$

- 在得到 $repr_1$ 后，组件消除会将 $repr\ x \coloneqq (repr_1\ x, repr_2\ x)$ 代入所有数据结构形式的规约，并得到第二个子问题。其规约如下所示，其中 $\mathbb{I}_d$ 表示所有数据结构形式规约的编号，操作符 $\vartriangle$ 的定义为 $(f \vartriangle g)\ x \coloneqq (f\ x, g\ x)$。

$$\forall i \in \mathbb{I}_d. \forall (in \mapsto out) \in \mathbb{E}_i. \qquad comb'_i\ \left(\mathsf{F}_i (repr_1 \vartriangle repr_2)\ in\right) = (repr_1\ out, repr_2\ out)$$

- 在解出这两个子问题后，它们的解可以按如下方式合并成原提升问题的解。

$$repr\ x \coloneqq (repr_1\ x, repr_2\ x) \qquad comb_i\ x \coloneqq \begin{cases} comb'_i\ (\mathsf{F}_i\ fst\ x) & \textbf{if} \quad i \in \mathbb{I}_v \\ comb'_i\ x & \textbf{if} \quad i \in \mathbb{I}_d \end{cases}$$

组件消除的第一个子问题会被变量消除规则继续分解，而其第二子问题仍然符合提升问题的形式，参数如下所示。其中 $\mathsf{F}[\mathsf{I} \mapsto \mathsf{F}']$ 表示把多项式函子 F 中所有的单位函

子 I 都替换成函子 F'，而 T_1 表示 $repr_1$ 的输出类型。

$$\mathbb{E}'_i \coloneqq \left\{ \mathsf{F}_i(repr_1 \vartriangle id)\ in \mapsto (repr_1\ out, out) \;\middle|\; (in \mapsto out) \in \mathbb{E}_i \right\}$$
$$\mathsf{F}'_i \coloneqq \mathsf{F}_i[\mathsf{I} \mapsto !T_1 \times \mathsf{I}] \quad \mathsf{G}_i \coloneqq !T_1 \times \mathsf{I}$$

于是，组件消除规则可以被递归地应用到第二个子问题上。这一递归过程会一直持续，直到 $repr_1$ 的求解结果为空，即 $repr_1\ x = ()$。这一结果表明原提升问题不再需要表征函数提供任何额外信息，因此它存在着一个平凡的解，如下所示：

$$repr\ x \coloneqq () \qquad comb_i\ x \coloneqq \begin{cases} comb'_i\ x & \textbf{if} \quad i \in \mathbb{I}_v \\ () & \textbf{if} \quad i \in \mathbb{I}_d \end{cases}$$

在这种况下，组件消除不会再继续产生第二个子问题，而是将这一平凡解直接返回。

4.3.2 变量消除

变量消除的输入是组件消除的第一类子问题，它由一系列常量形式的规约组成：

$$\forall i \in \mathbb{I}.\ \forall (in \mapsto out) \in \mathbb{E}_i. \qquad comb_i\ (\mathsf{F}_i repr\ in) = out$$

变量消除的目标是将 $repr$ 和 $comb_i$ 的合成分解到不同的子问题，拆分方式如下。

- 变量消除会用存在量词约束 $comb_i$，从而得到只包含 $repr$ 的规约，如下所示：

  $$\forall i \in \mathbb{I}.\ \exists comb.\ \forall (in \mapsto out) \in \mathbb{E}_i. \qquad comb\ (\mathsf{F}_i repr\ in) = out$$

  然而，因为现有程序合成技术无法处理程序空间上的存在量词，所以变量消除会对它做进一步地化简。该化简过程分为两步。首先，变量消除会用数学函数空间 $\mathcal{F}$ 上的存在量词替换合并算子空间上的存在量词，如下所示。

  $$\forall i \in \mathbb{I}.\ \exists f \in \mathcal{F}.\ \forall (in \mapsto out) \in \mathbb{E}_i. \qquad f\ (\mathsf{F}_i repr\ in) = out$$

  接着，变量消除利用函数定义（即相同输入下产生相同输出）进一步地消除了函数空间上的存在量词，从而得到了它的第一个子问题。

  $$\forall i \in \mathbb{I}.\ \forall (in \mapsto out), (in' \mapsto out') \in \mathbb{E}_i. \qquad \mathsf{F}_i repr\ in = \mathsf{F}_i repr\ in' \rightarrow out = out'$$

  注意到该规约的右侧完全已知，且只有在 $out \neq out'$ 时会对 $repr$ 产生约束。因

此，这一规约可以被简化为如下的样例形式。

$$\forall(\mathsf{F}, in, in') \in \mathbb{E}. \qquad \mathsf{F}repr\ in \neq \mathsf{F}repr\ in' \tag{4.6}$$

where

$$\mathbb{E} \coloneqq \left\{(\mathsf{F}_i, in, in') \;\middle|\; i \in \mathbb{I}, (in, out), (in', out') \in \mathbb{E}_i, out \neq out'\right\}$$

- 在求解得到 *repr* 后，样例消除会将它代入原始问题，从而得到其第二个子问题。

$$\forall i \in \mathbb{I}.\ \forall(in \mapsto out) \in \mathbb{E}_i. \qquad comb_i\ (\mathsf{F}_i repr\ in) = out \tag{4.7}$$

- 这两个子问题的结果不需要任何额外处理，直接形成了原始问题的解。

4.4　子问题的求解方法

AutoLifter 会分解产生两类叶子任务，分别对应于变量消除规则产生两个子任务（规约 4.6 和 4.7）。AutoLifter 使用了不同的归纳程序合成方法对它们进行求解。

4.4.1　合并算子的合成方法

首先考虑合并算子上的叶子任务（规约 4.7）。尽管它涉及多条规约与多个未知程序，但它的每一条规约都是完全独立的，可以被拆分为独立的 $|\mathbb{I}|$ 个问题，如下所示。

$$\forall(in \mapsto out) \in \mathbb{E}_i. \qquad comb_i\ (\mathsf{F}_i repr\ in) = out$$

这一问题符合样例编程的形式，可以被任意的样例编程求解器解决。

4.4.2　表征函数的合成方法

表征函数上的叶子任务由一系列的输入对 $\mathbb{E}$ 描述，它要求表征函数在这些输入对上始终产生不同的输出，如下所示。

$$\forall(\mathsf{F}, in, in') \in \mathbb{E}. \qquad \mathsf{F}repr\ in = \mathsf{F}repr\ in' \tag{4.8}$$

问题挑战　尽管表征函数 *repr* 的规模通常远小于合并算子，但求解这一子问题仍然面临着合成效率上的挑战。一方面，此问题的规约并不符合样例编程的形式，于是无法被样例编程求解器处理。另一方面，在求解复杂的提升问题时，表征函数通常需要输出多个值。此时，目标的 *repr* 仍然会有较大的规模，超出现有的通用合成方法（例如基于枚举的程序合成方法）的能力范围。

例 7 (最大子段和问题)**.** 在这个例子中，本文将用一个相对复杂的提升问题展示合成表征函数时的挑战。考虑如下的最大子段和问题。

- 给定一个整数列表，计算它的所有连续子段中最大的子段和。例如当输入列表为 $[-4, 1, 2, -5, 4]$ 时，最大的子段和为 3，它对应了连续子段 $[1, 2]$。

令 *mss* 为计算最大子段和的任意参考程序，并考虑将列表分治用于该程序的提升问题。在分解该提升问题的过程中，AutoLifter 会产生如下的、关于表征函数的子任务[①]，其目标是为分治的合并找到足以计算最大子段和的辅助信息。

$$\forall\big((ls, rs), (ls', rs')\big) \in \mathbb{E}. \qquad (repr\ ls, repr\ rs) \neq (repr\ ls', repr\ rs')$$

where

$$\mathbb{E} \subseteq \left\{ \big((ls, rs), (ls', rs')\big) \;\middle|\; \begin{array}{cc} ls, rs, ls', rs' \in \mathtt{List} & \wedge \\ (mss\ ls, mss\ rs) = (mss\ ls', mss\ rs') & \wedge \\ mss\ (ls +\!\!+ rs) \neq mss\ (ls' +\!\!+ rs') & \end{array} \right\}$$

此问题中的每个样例都对应了两组需要辅助信息的合并。它包含了两组分解后的列表，在这两组之间，左右列表上的最大子段和完全一致，但合并后的最大子段和不同。因此在不提供辅助值的情况，不存在任何数学函数能同时完成两组合并。

该问题上的目标程序需要同时返回列表的最大前缀和与最大后缀和作为辅助值，因为合并后的最大子段和可能来源于左侧的最大后缀和与右侧的最大前缀和的拼接。在本文使用的程序空间中，这一表征函数的实现如下所示[②]：

$$repr\ xs \quad := \quad \big(max\ (scanl\ (+)\ xs), max\ (scanr\ (+)\ xs)\big)$$

这一程序包含了 9 个操作符。在本文使用的文法上，这一规模已经超出了基于枚举的程序合成方法的能力范围。

求解框架 为了加速表征函数的合成，本文利用了如下的特殊性质。

- 复杂的表征函数往往会输出多个值，此时它的形式为一些较小组件形成的元组，即 $repr\ x := (repr_1\ x, \ldots, repr_k\ x)$。

根据这一结构，AutoLifter 使用的归纳合成方法分两步进行。它会先枚举一系列候选的组件，再尝试将这些组件组合成一个能满足所有样例的元组。

算法 1 展示了这一合成方法的伪代码，它不停地调用函数 `Extend`，直到找到一个满足所有样例的结果（第 16 行）。在每一轮中，它会先枚举下一个组件（第 10 行），再

① 此处本文对规约进行了化简，展开了函子并消除了 F*repr* 中的常量部分。

② 操作符 *scanl* 接受一个二元操作符和一个列表为输入。它会构建列表的所有非空前缀，并利用给定的二元操作符将每个前缀压缩成一个值。其具体定义为如下所示：

$$scanl\ (\oplus)\ [xs_1, \ldots, xs_n] := [xs_1, xs_1 \oplus xs_2, \ldots, xs_1 \oplus \cdots \oplus xs_n]$$

而 *scanr* 的定义类似，唯一的区别在于它考虑所有的非空后缀而不是前缀。

Algorithm 1: 针对表征函数 *repr* 的归纳合成器.

Input: 样例集合 $\mathbb{E} = \overline{(\mathsf{F}, in, in')}$ 和整数参数 lim_c。
Output: 满足所有样例的表征函数 $repr^*$.

```
1  inputs = {(F, i) | (F, in, in') ∈ 𝔼, i ∈ {in, in'}};
2  oe ← ObservationalEquivalenceSolver(𝔼);
3  ∀size ≥ 0, programs[size] ← [];  result ← ⊥;
4  Function IsCovered(prog, size):
5      return ∃size' ≤ size, ∃prog' ∈ programs[size'], prog' 满足所有被 prog 满足的样例;
6  Function Insert(prog, size):
7      if prog 满足所有样例 ∧result = ⊥ then result ← prog;
8      if ¬IsCovered(prog, size) then programs[size].Append(prog);
9  Function Extend():
10     component ← oe.Next(); Insert(component, 1)
11     prePrograms ← programs;
12     foreach size ∈ [1, ..., lim_c − 1] 且 prog ∈ prePrograms[size] do
13         Insert(prog △ component, size + 1);
14     end
15 Insert(null, 0);
16 while result = ⊥ do Extend();
17 return result;
```

尝试将这个组件插入到现有的元组程序中。

枚举过程　算法 1 使用观察等价法[29] 枚举候选的组件（第 2 行）。回顾第 2.2.1 节，观察等价法是一个基于枚举的程序合成方法，它自底向上按照从小到大的顺序遍历程序空间。观察等价法的核心是一个避免重复程序的剪枝策略。给定一系列输入，观察等价法会记录每个程序在每个输入上的输出，并跳过所有产生重复输出的程序。

在当前的合成任务中（规约 4.8），一个组件 *repr* 是否满足样例 (F, in, in') 只和它在 in 与 in' 上的输出有关。一旦有两个组件在所有的 (F, in) 与 (F, in') 上都产生相同的输出，那么它们在当前的合成任务上时完全等效的。基于这一性质，算法 1 使用了观察等价法来预先过滤掉所有重复的等价组件（第 1-2 行）。

例 8. 考虑包含如下两个样例的样例集合 $\mathbb{E}$。

$$e_1 := \{\mathsf{I}\times\mathsf{I}, ([1],[1]), ([1],[-1,1])\} \quad e_2 := \{\mathsf{I}\times\mathsf{I}, ([1],[1]), ([1,-1],[1])\}$$

在这一样例集合上，一个候选程序 *repr* 的表现只和以下三个输出有关。对于任意两个程序，只要它们在这三种情况下的输出相同，那么它们一定满足相同的样例集合。

$$(repr\ [1], repr\ [1]) \qquad (repr\ [1], repr\ [-1,1]) \qquad (repr\ [1,-1], repr\ [1])$$

合并过程　算法 1 使用数据结构 *programs* 存储枚举过程中构建的所有元组程序（第 3 行），其中 *programs*[*size*] 对应恰好使用 *size* 个组件的元组。在每一轮中，该算法会将

当前组件作为一个大小为 1 的元组加入 *programs*（第 10 行），并依次尝试将它并入已有的元组中（第 11-14 行）。为了限制合并的空间，算法 1 接受一个整数参数 lim_c 并至多考虑 lim_c 个组件形成的元组[①]（第 12 行）。

为了进一步加速合并的过程，本文使用了一个名为**观察覆盖法**的剪枝策略。这一策略继承了观察等价法的主要思想，即如果有两个程序在当前样例集合上的作用等价，那么就忽略其中较大的程序。观察覆盖法依赖如下引理。它表明在任何情况下，一个元组程序满足一个样例当且仅当它中间的任意组件满足了这个样例。

引理 3. *对于任意样例 (F, in, in') 与任意组件列表 $\overline{repr_i}$，下列公式永远成立。*

$$\mathsf{F}(repr_1 \vartriangle \ldots \vartriangle repr_k)\ in \neq \mathsf{F}(repr_1 \vartriangle \ldots \vartriangle repr_k)\ in' \ \leftrightarrow\ \exists i.\ repr_i\ in \neq repr_i\ in'$$

证明. 对多项式函子 F 的结构作归纳。当 F 是常函子或单位函子时，等式显然成立。

当 F 是两个函子的直积，即 $\mathsf{F}_1 \times \mathsf{F}_2$ 时，目标公式的推导过程如下，其中第二步利用了 F_1 与 F_2 上的归纳假设。

$$\begin{aligned}
&(\mathsf{F}_1 \times \mathsf{F}_2)(repr_1 \vartriangle \ldots \vartriangle repr_k)\ in \neq (\mathsf{F}_1 \times \mathsf{F}_2)(repr_1 \vartriangle \ldots \vartriangle repr_k)\ in' \\
\leftrightarrow &\bigvee_{k \in \{1,2\}} \mathsf{F}_k(repr_1 \vartriangle \ldots \vartriangle repr_k)\ in \neq \mathsf{F}_k(repr_1 \vartriangle \ldots \vartriangle repr_k)\ in' \\
\leftrightarrow &\left(\exists i.\mathsf{F}_1 repr_i\ in \neq \mathsf{F}_1 repr\ in'\right) \vee \left(\exists i.\mathsf{F}_2 repr_i\ in \neq \mathsf{F}_2 repr\ in'\right) \\
\leftrightarrow &\exists i.\left(\mathsf{F}_1 repr_i\ in \neq \mathsf{F}_1 repr_i\ in' \vee \mathsf{F}_2 repr_i\ in \neq \mathsf{F}_2 repr_i\ in'\right) \\
\leftrightarrow &\exists i.(\mathsf{F}_1 \times \mathsf{F}_2) repr_i\ in \neq (\mathsf{F}_1 \times \mathsf{F}_2) repr_i\ in'
\end{aligned}$$

当 F 是两个函子的直和，即 $\mathsf{F}_1 + \mathsf{F}_2$ 时，分两种情况讨论。首先，当 in 与 in' 来自于不同分支时，目标公式两侧均为真，因此成立。其次，当 in 与 in' 来自相同分支时，不妨假设它们都来自于第一个分支，分别为 $(1, v)$ 与 $(1, v')$。此时，目标公式的推导过程如下，其中第二步利用了 F_1 上的归纳假设。

$$\begin{aligned}
&(\mathsf{F}_1 + \mathsf{F}_2)(repr_1 \vartriangle \ldots \vartriangle repr_k)\ in \neq (\mathsf{F}_1 + \mathsf{F}_2)(repr_1 \vartriangle \ldots \vartriangle repr_k)\ in' \\
\leftrightarrow &\mathsf{F}_1(repr_1 \vartriangle \ldots \vartriangle repr_k)\ v \neq \mathsf{F}_1(repr_1 \vartriangle \ldots \vartriangle repr_k)\ v' \\
\leftrightarrow &\exists i.\mathsf{F}_1 repr_i\ v \neq \mathsf{F}_1 repr\ v' \\
\leftrightarrow &\exists i.(\mathsf{F}_1 + \mathsf{F}_2) repr_i\ in \neq (\mathsf{F}_1 + \mathsf{F}_2) repr_i\ in'
\end{aligned}$$

至此归纳成立，目标引理得证。 □

① 注意这一限制并不会影响算法 1 的完备性，即不会让它错失任何在程序空间中的合法程序，即使参数 lim_c 被设为 1 也是如此。这是因为算法 1 中的观察等价法是在完整的程序空间中枚举组件的，所以任意合法程序最终都会以一个单一组件的形式被找到。

基于这一引理，既然算法 1 的目标是找到合法的不超过 lim_c 个组件的元组，那么当下列条件被满足时，元组程序 *prog* 的效用会被元组程序 $prog'$ 完全覆盖。

- $prog'$ 使用的元组数量小于等于 *prog* 的数量。
- $prog'$ 满足所有被 *prog* 满足的样例。

利用这一性质，算法 1 会跳过所有被覆盖的元组程序——只有那些不被任何已有程序覆盖的元组程序会被加入到 *programs* 中（第 4-5 行与第 8 行）。

例 9. 在例 8 的样例集合上，程序 *max* （不满足任何样例）的效用会空程序 *null* （同样不满足任何样例）覆盖。这两个程序满足相同的样例集合且 *null* 使用了更少的组件，于是 *max* 可以被直接跳过。在任何时候，如果存在一个使用了 *max* 的程序 $max \triangle prog$ 满足所有样例，那么 $null \triangle prog$ 一定也满足所有样例。

4.5　系统性质

本节将讨论 AutoLifter 的理论性质，并证明它的分解过程始终保证正确性并在随机意义下以高概率保证完备性。

4.5.1　正确性

AutoLifter 的分解是保证正确性的。只要它的所有叶子问题都被正确地求解，那么它的最终结果也一定正确。

定理 9 (正确性)**.** 给定任何提升问题，如果在 AutoLifter 的分解过程中，所有叶子问题都被成功且正确地解出，那么 AutoLifter 返回的结果 $(repr, \overline{comb_i})$ 一定是原提升问题的合法解。

证明. 对分解树的结构进行归纳。叶子情况的正确性直接由定理前提给出。

对于非叶子情况，假设当前应用的规则是变量消除，它把合成问题 $\mathcal{P}$ 分解为了两个子问题 $\mathcal{P}_1$ 与 $\mathcal{P}_2$。根据归纳假设，$\mathcal{P}_1$ 与 $\mathcal{P}_2$ 的求解结果 *repr* 与 *comb* 都是正确的，即都满足对应的规约。注意到，$\mathcal{P}_2$ 直接由完整问题 $\mathcal{P}$ 在代入 $\mathcal{P}_1$ 的结果 *repr* 后得到，所以它在 *repr* 给定的情况下与 $\mathcal{P}$ 完全等价。因此，如果 $\mathcal{P}_2$ 被正确地求解，那么对应的结果 $(repr, comb)$ 一定是 $\mathcal{P}$ 的合法解。

应用组件消除的情况完全一致，因为组件消除的第二个子问题也是在原问题的基础上通过代入第一个子问题的解得到的。 □

4.5.2　完备性

因为 AutoLifter 在分解时使用了近似规约，所以在理论上它并不能保证完备性。

- 组件消除的第一个子问题只考虑为常量情况提供足够的信息，而并没有考虑这些信息在计算过程中能不能被快速地计算。理论上，第一个子问题可能会生成过度复杂的 $repr_1$，它的返回结果不可能被高效计算，导致第二个子问题无解。
- 类似地，变量消除的第一个子问题只要求存在一个数学函数 f 能够完成计算。然而，因为合并算子还受到计算效率的限制，所以理论上，第一个子问题的解 $repr$ 对应的函数可能无法被足够高效地计算，导致第二个子问题无解。

这些无解的子问题会导致 AutoLifter 在有解的提升问题上失败，从而损失完备性。

幸运的是，本文发现这些近似在实践中的影响微乎其微：在本文的实验中，AutoLifter 从来不会将有解的提升问题拆分到无解的子问题。这一现象的直接原因是算法 1 在合成表征函数时的优秀表现。回顾 AutoLifter 的分解过程，可以发现其中的每个子任务都只和已合成的表征函数组件有关。而在本文的实验中，算法 1 总是可以找到符合预期的表征函数，从而避免了无解的子任务。

在本节中，本文将对这一现象进行理论分析，并讨论算法 1 优秀表现的来源。

概率模型 为了将样例的来源考虑在内，本文在这里直接分析完整形式的提升问题，它可以被视为一个包含了足够多样例的样例形式。此外，为了方便，本文引入了几个简化条件：(1) 提升问题中只涉及一个合并算子，(2) 函子 F 为 I，即输入不包含额外常量，且 (3) 函子 G 为 $\mathsf{I}\times!V$，即输出包含一个数据结构与一个常量。这一简化后的规约在展开后如下所示，其中 $calc_1$ 的类型为 $D \to D$，而 $calc_2$ 的类型为 $D \to V$。本文在该规约上的所有分析都可以被泛化到一般情形的提升问题。

$$\forall in \in D. \qquad comb\ (repr\ in) = (repr\ (calc_1\ in), calc_2\ in) \tag{4.9}$$

为了分析 AutoLifter 在实践中的完备性损失，我们需要计算它有多大概率会将一个有解的提升问题分解为无解的子问题。然而，精确地计算这个概率是非常困难的。AutoLifter 的合成过程依赖 $repr$ 与 $comb$ 的程序空间，这些空间的大小通常无限，且其中每个程序的语义都十分复杂，使得精确计算难以进行。为了简化问题，本文对提升问题建立了一个概率模型，并在此基础上计算无解子问题的概率。

给定 $calc_1$ 与 $calc_2$，本文的概率模型 $\mathcal{M}[\overline{calc}]$ 假设 $repr$ 与 $comb$ 的程序空间（记作 $\mathcal{L}_{repr}$ 与 $\mathcal{L}_{comb}$）中的所有程序都是均匀随机函数。其具体定义如下所示：

- $\mathcal{M}[\overline{calc}]$ 依赖一系列参数，包括 $calc_i$ 的类型与语义，以及 $\mathcal{L}_{repr}$ 与 $\mathcal{L}_{comb}$ 中每个程序的语法与类型。模型中唯一随机的部分是程序空间中程序的语义。
- 为了简化描述，本文对 $\mathcal{M}[\overline{calc}]$ 中的参数作出了如下假设：
  - $\mathcal{L}_{aux}$ 中的所有程序都以组件元组形式出现（即 $repr_1 \vartriangle \ldots \vartriangle repr_k$），其中每个组件都只会返回一个辅助值。

  - 存在一个统一的值类型 V 作为每个组件 $repr_i$ 以及 $calc_2$ 的输出类型。
  - 类型 D 与 V 中不同的元素数量是有限的，分别记作 s_D 与 s_V[①]。
- $\mathcal{M}[calc]$ 均匀且独立地为程序空间中每个组件赋予一个函数作为语义。例如，对于一个类型为 $D \to V$ 的组件，共有 $S_V^{s_D}$ 个函数符合其类型，那么 $\mathcal{M}[calc]$ 会从中随机挑选一个函数作为语义，每个函数被选中的概率为 $s_V^{-s_D}$。

产生无解子问题的概率　。基于这一概率模型，本文的核心结果是证明了 AutoLifter 的**有界无解率**可以被提升问题的**错位因子**限制。在介绍具体的分析结果之前，本节将先引入一些必要的概念，从有界无解率开始。

定义 11 (有界无解率)**.** 给定概率模型 $\mathcal{M}[\overline{calc}]$ 与大小限制 lim_s，AutoLifter 的有界无解率，记作 $unreal\left(\overline{calc}, lim_s\right)$，表示 AutoLifter 在分解一个随机提升问题时，在已知该问题存在一个大小不超过 lim_s 的解的前提下，产生无解子问题的概率，如下所示：

$$\Pr_{\varphi \sim \mathcal{M}\left[\overline{calc}\right]}\Big[\text{AutoLifter 在分解提升问题 } \varphi \text{ 时产生了无解的子问题} \mid \exists (repr, comb), \left(size(repr, comb) \le lim_s \wedge (repr, comb) \text{ 是 } \varphi \text{ 的合法解}\right)\Big]$$

其中 $\varphi \sim \mathcal{M}[\overline{calc}]$ 表示从概率模型 $\mathcal{M}[\overline{calc}]$ 中随机采样一个提升问题，$size(repr, comb)$ 表示程序 $repr$ 与 $comb$ 的大小和。

这一定义通过引入大小限制 lim_s 将提升问题最小解的大小纳入了考虑范围。这一因子反映了提升问题 φ 的难度，同时也允许我们进行更加细粒度的分析。

第二个概念是提升问题的错误因子，其定义如下所示。

定义 12 (错位因子)**.** 给定参数为 $\overline{calc}$ 的提升问题和整数 t，我们称该提升问题的错位因子至少为 t，如果存在 t 对输入 $(in, in') \in D \times D$ 满足如下两个条件。

- 对于每一对输入 (in, in')，在不引入任何辅助值的情况下，合并算子 $comb$ 的预期输出不同，即 $calc_2\ in \neq calc_2\ in'$。
- 被这 t 个二元组包含的所有输入（共 $2t$ 个）两两不同。

错位因子的大小反映了提升问题对 $repr$ 的约束强度。为了说明这一点，考虑 AutoLifter 在分解原提升问题时产生的第一个关于 $repr$ 的叶子问题[②]：

$$\forall in, in' \in D.\quad calc_2\ in \neq calc_2\ in' \to repr\ in \neq repr\ in' \tag{4.10}$$

① 对于任何大小无限的类型，我们都可以通过取一个有限子集的方式在模型中近似。例如对于列表类型，我们可以只考虑在一个有限长度限制下的所有列表。
② 因为本节中的提升问题使用了所有样例，于是其子问题中的样例集合可以被改写为约束的前条件。

这一规约中，样例需要满足的条件与错位因子定义中的第一个条件相同。因此，提升因子为 t 意味着在上述子问题中至少存在着 t 对独立的需要被 *repr* 满足的样例。提升因子越大，这一子问题的规约就越强，那么一个错误的表征函数 *repr* 就越难满足近似规约，AutoLifter 也就越有可能找到正确的表征函数。

在这些概念的基础上，定理 10 展示了本文概率分析的结果。它表明 AutoLifter 的有界无解率被提升问题的错位因子、大小限制、以及输出域 V 的大小所限制。

定理 10. *给定参数为 $\overline{calc}$ 且错位因子至少为 t 的提升问题，AutoLifter 的有界无解率存在着一个如下所示的上界。*

$$unreal\left(\overline{calc}, lim_s\right) \leq 2^{lim_c \cdot lim_s} \exp\left(-\frac{t}{s_V^{lim_c}}\right)$$

证明. 为了简化符号系统，我们将用 $\mathcal{M}$ 指代 $\mathcal{M}[\overline{calc}]$，并不加区分地把一个程序合成任务 φ 当成谓词，其中 $\varphi(prog)$ 表示程序 *prog* 是合成问题 φ 的一个合法解。此外，这一证明还会用到如下的两个定义：

- 令 $\tilde{\varphi}$ 为在分解提升问题 φ 时，第一个关于表征函数的叶子问题（规约 4.10）。
- 令 $\mathbb{A}(\varphi) \subseteq \mathcal{L}_{repr}$ 为一个表征函数的集合，其中每个表征函数都被包含在了提升问题 φ 中某个大小不超过 lim_s 的解中。其形式化定义如下所示。

$$\mathbb{A}(\varphi) \coloneqq \left\{ repr \;\middle|\; \exists comb \in \mathcal{L}_{comb}. \left(\varphi(repr, comb) \wedge \text{size}(repr, comb) \leq lim_s\right)\right\}$$

使用这一符号，有界无解率定义中的条件（定义 11）可以被表示为 $|\mathbb{A}(\varphi)| > 0$。

步骤 1：一个充分条件 根据定义，$\mathbb{A}(\varphi)$ 中的任何程序都是子问题 $\tilde{\varphi}$ 的合法解。更进一步地，可以证明当 $\tilde{\varphi}$ 的合成结果在 $\mathbb{A}(\varphi)$ 中时，AutoLifter 一定不会产生任何无解的子问题。具体地，假设 $repr^* \in \mathbb{A}(\varphi)$ 是子问题 $\tilde{\varphi}$ 的合成结果。根据 $\mathbb{A}(\varphi)$ 的定义，一定存在合并算子 $comb^*$ 使得 $(repr^*, comb^*)$ 构成了原提升问题 φ 的合法解，如下所示。

$$\forall in \in D. \qquad comb^*\ (repr^*\ in) = (repr^*\ (calc_1\ in), calc_2\ in)$$

在得到 $repr^*$ 后，AutoLifter 只会再产生三个子问题，如下所示：

1. 第一个子问题是关于合并算子的叶子问题，它的目标是找到一个用于计算 $calc_2$ 输出的合并算子 $comb_2$，规约如下所示。

$$\forall in \in D. \qquad comb_2\ (repr^*\ in) = calc_2\ in$$

该问题存在一个合法解 $comb_2\ x \coloneqq snd\ (comb^*\ x)$。

2. 第二个子问题是关于表征函数的叶子问题，它的目标在 $repr^*$ 的基础上补充必要的辅助信息。因为 $repr^*$ 已经足够满足原提升问题，所以 AutoLifter 一定会返回空函数 *null* 作为该问题的合成结果。
3. 最后一个子问题是关于合并算子的叶子问题，它的目标是找到对应 $calc_1$ 的合并算子 $comb_1$，规约如下所示。

$$\forall in \in D. \qquad comb_1\ (repr^*\ in) = repr\ (calc_1\ in)$$

该问题存在一个合法解 $comb_1\ x \coloneqq fst\ (comb^*\ x)$。

根据以上分析，事件“子问题 $\tilde{\varphi}$ 的求解结果在 $\mathbb{A}(\varphi)$ 中”构成了 AutoLifter 成功求解的一个充分条件。于是，AutoLifter 的有界错误率不会大于如下的概率，其中 $\mathcal{S}_{repr}$ 表示用于合成表征函数的叶子求解器，即算法 1。

$$\Pr_{\varphi\sim\mathcal{M}}\left[\mathcal{S}_{repr}(\tilde{\varphi}) \notin \mathbb{A}(\varphi) \,\middle|\, |\mathbb{A}(\varphi)| > 0\right]$$

分析这一概率仍然是困难的，因为它涉及合成算法 $\mathcal{S}_{repr}$ 的具体行为以及 $\mathbb{A}(\varphi)$ 的定义。为了简化，本证明接下来三步的目标是从这一概率中消去 $\mathbb{A}(\varphi)$ 与 $\mathcal{S}_{repr}$。

步骤 2：消去左侧的 $\mathbb{A}(\varphi)$　本文按照如下方式变换这一条件概率。

$$\begin{aligned}
&\Pr_{\varphi\sim\mathcal{M}}\left[\mathcal{S}_{repr}(\tilde{\varphi}) \notin \mathbb{A}(\varphi) \,\middle|\, |\mathbb{A}(\varphi)| > 0\right] \\
=\ &\Pr_{\varphi\sim\mathcal{M}}\left[\mathcal{S}_{repr}(\tilde{\varphi}) \notin \mathbb{A}(\varphi) \wedge |\mathbb{A}(\varphi)| > 0\right] \Big/ \Pr_{\varphi\sim\mathcal{M}}\left[|\mathbb{A}(\varphi)| > 0\right] \\
=\ &\sum_{P\subseteq\mathcal{L}_{repr}} [|P| > 0] \Pr_{\varphi\sim\mathcal{M}}\left[\mathcal{S}_{repr}(\tilde{\varphi}) \notin P \wedge \mathbb{A}(\varphi) = P\right] \Big/ \sum_{P\subseteq\mathcal{L}_{repr}} [|P| > 0] \Pr_{\varphi\sim\mathcal{M}}\left[\mathbb{A}(\varphi) = P\right] \\
\le\ &\frac{\sum_{P\subseteq\mathcal{L}_{repr}} [|P| > 0] \left(\Pr_{\varphi\sim\mathcal{M}}\left[\mathcal{S}_{repr}(\tilde{\varphi}) \notin P \wedge \mathbb{A}(\varphi) = P\right] + (|P| - 1) \Pr_{\varphi\sim\mathcal{M}}\left[\mathbb{A}(\varphi) = P\right]\right)}{\sum_{P\subseteq\mathcal{L}_{repr}} |P| \Pr_{\varphi\sim\mathcal{M}}\left[\mathbb{A}(\varphi) = P\right]}
\end{aligned} \tag{4.11}$$

其中最后一步用到了不等式 $(a + c)/(b + c) \ge a/b$ （当 $a, b > 0, c \ge 0, a < b$ 时）。

在 $|P| > 0$ 的条件下，考虑如下的断言：

$$\begin{aligned}
&\Pr_{\varphi\sim\mathcal{M}}\left[\mathcal{S}_{repr}(\tilde{\varphi}) \notin P \wedge \mathbb{A}(\varphi) = P\right] + (|P| - 1) \Pr_{\varphi\sim\mathcal{M}}\left[\mathbb{A}(\varphi) = P\right] \\
&\qquad = \sum_{repr^*\in P} \Pr_{\varphi\in\mathcal{M}}\left[\mathcal{S}_{repr}(\tilde{\varphi}) \neq repr^* \wedge \mathbb{A}(\varphi) = P\right]
\end{aligned} \tag{4.12}$$

要证明这一断言，令 φ 是一个任意的满足 $\mathbb{A}(\varphi) = P$ 的提升问题。此时合成结果 $\mathcal{S}_{repr}(\tilde{\varphi})$ 存在两种情况。

- 当 $\mathcal{S}_{repr}(\tilde{\varphi}) \notin P$ 时，φ 的概率在断言的等式两侧都贡献了 $|P|$ 次。
- 当 $\mathcal{S}_{repr}(\tilde{\varphi}) \in P$ 时，φ 的概率在断言的等式两侧都贡献了 $|P|-1$ 次。

因此，断言 4.12 两侧的概率和一定相等。

通过将断言 4.12 应用到公式 4.11，我们可以继续进行如下的推导。推导的第二步用到了 $\mathbb{A}(\varphi)$ 中任意程序的大小都不超过 lim_s 的性质，并用 $\mathbb{L}_{\leq lim_s}$ 表示程序空间 $\mathcal{L}_{repr}$ 中所有大小不超过 lim_s 的表征函数形成的集合。

$$
\begin{aligned}
&= \sum_{P \subseteq \mathcal{L}_{repr}} \sum_{repr^* \in P} \Pr_{\varphi \in \mathcal{M}} \left[\mathcal{S}_{repr}(\tilde{\varphi}) \neq repr^* \wedge \mathbb{A}(\varphi) = P\right] \Big/ \sum_{P \subseteq \mathcal{L}_{repr}} \sum_{repr^* \in P} \Pr_{\varphi \sim \mathcal{M}} \left[\mathbb{A}(\varphi) = P\right] \\
&= \sum_{repr^* \in \mathbb{L}_{\leq lim_s}} \Pr_{\varphi \sim \mathcal{M}} \left[\mathcal{S}_{repr}(\tilde{\varphi}) \neq repr^* \wedge repr^* \in \mathbb{A}(\varphi)\right] \Big/ \sum_{repr^* \in \mathbb{L}_{\leq lim_s}} \Pr_{\varphi \sim \mathcal{M}} \left[repr^* \in \mathbb{A}(\varphi)\right] \\
&\leq \max_{repr^* \in \mathbb{L}_{\leq lim_s}} \left(\Pr_{\varphi \sim \mathcal{M}} \left[\mathcal{S}_{repr}(\tilde{\varphi}) \neq repr^* \wedge repr^* \in \mathbb{A}(\varphi)\right] \Big/ \Pr_{\varphi \sim \mathcal{M}} \left[repr^* \in \mathbb{A}(\varphi)\right] \right) \qquad (4.13) \\
&= \max_{repr^* \in \mathbb{L}_{\leq lim_s}} \Pr_{\varphi \sim \mathcal{M}} \left[\mathcal{S}_{repr}(\tilde{\varphi}) \neq repr^* \;\middle|\; repr^* \in \mathbb{A}(\varphi)\right] \qquad (4.14)
\end{aligned}
$$

公式 4.13 依赖于如下的不等式（其中 0/0 被定义为 0）。

$$
\forall a_i, b_i \geq 0, \sum_{i=1}^{n} a_i \Big/ \sum_{i=1}^{n} b_i \leq \max_{i=1}^{n} (a_i / b_i)
$$

到此，我们成功地消除了步骤一条件概率中左侧的 $\mathbb{A}(\varphi)$。

步骤 3：消去 $\mathcal{S}_{repr}$ 的调用 归纳合成器 $\mathcal{S}_{repr}$（算法 1）上一个值得注意的性质是，当 $\tilde{\varphi}$ 的最小解大小不超过 lim_s 时，$\mathcal{S}_{repr}(\tilde{\varphi})$ 的大小不会超过 $lim_c \cdot lim_s$。因为 $\mathcal{S}_{repr}$ 会按照从小到大的顺序枚举组件，且在枚举到最小解程序的时候会直接返回，所以 $\mathcal{S}_{repr}$ 的解程序中的每个组件的大小都不会超过 lim_s。而又因为 $\mathcal{S}_{repr}$ 最多使用 lim_c 个组件，所以其结果大小不会超过 $lim_c \cdot lim_s$。

于是，事件 "$\mathbb{L}_{\leq lim_c lim_s}$ 中除了 $repr^*$ 以外的程序都不满足 $\tilde{\varphi}$" 构成了 $\mathcal{S}_{repr}(\tilde{\varphi}) = repr^*$ 的一个必要条件。利用这一点，可以继续化简公式 4.14 ，并消除其中的 $\mathcal{S}_{repr}$。

$$
\begin{aligned}
&\Pr_{\varphi \sim \mathcal{M}} \left[\mathcal{S}_{repr}(\tilde{\varphi}) \neq repr^* \;\middle|\; repr^* \in \mathbb{A}(\varphi)\right] \\
&\leq \Pr_{\varphi \sim \mathcal{M}} \left[\exists repr \in \mathbb{L}_{\leq lim_c lim_s}, repr \neq repr^* \wedge \tilde{\varphi}(repr) \;\middle|\; repr^* \in \mathbb{A}(\varphi)\right] \\
&\leq \sum_{repr \in \mathbb{L}_{\leq lim_c lim_s}} [repr \neq repr^*] \Pr_{\varphi \sim \mathcal{M}} \left[\tilde{\varphi}(repr) \;\middle|\; repr^* \in \mathbb{A}(\varphi)\right] \\
&\leq 2^{lim_s lim_c} \max_{repr \in \mathbb{L}_{\leq lim_c lim_s}} [repr \neq repr^*] \Pr_{\varphi \sim \mathcal{M}} \left[\tilde{\varphi}(repr) \;\middle|\; repr^* \in \mathbb{A}(\varphi)\right] \qquad (4.15)
\end{aligned}
$$

其中最后一步用到的性质是大小不超过 k 的程序至多有 2^k 个[①]。

步骤 4：消除概率中的条件　利用 $\mathbb{A}(\varphi)$ 的定义展开公式 4.15 中的概率，如下所示：

$$\begin{aligned}&\Pr_{\varphi\sim\mathcal{M}}\left[\tilde{\varphi}(repr)\;\middle|\;repr^*\in\mathbb{A}(\varphi)\right]\\=&\Pr_{\varphi\sim\mathcal{M}}\left[\tilde{\varphi}(repr)\;\middle|\;\exists comb,\varphi(repr^*,comb)\wedge\mathrm{size}(repr^*,comb)\leq lim_s\right]\end{aligned}$$

注意到，该条件概率的事件是否成立只与 $repr$ 的随机语义有关，而其条件是否成立只与 $repr^*$ 的随机语义以及 $\mathcal{L}_{comb}$ 中所有程序的随意语义有关。因此，该概率中的事件与条件是完全独立的，我们可以安全地忽略其条件部分，如下所示：

$$\Pr_{\varphi\sim\mathcal{M}}\left[\tilde{\varphi}(repr)\;\middle|\;repr^*\in\mathbb{A}(\varphi)\right]=\Pr_{\varphi\sim\mathcal{M}}\left[\tilde{\varphi}(repr)\right]$$

步骤 5：使用错位因子估计 $\tilde{\varphi}(repr)$ 的概率　首先将概率 $\Pr\left[\tilde{\varphi}(repr)\right]$ 展开如下：

$$\Pr_{\varphi\sim\mathcal{M}}\left[\tilde{\varphi}(repr)\right]=\Pr_{repr}\left[\forall in,in'\in D.\,calc_2\;in\neq calc_2\;in'\rightarrow repr\;in\neq repr\;in'\right]$$

因为当前提升问题的错位因子至少是 t，所以存在 t 对完全无关的输入 $\overline{(in_i,in_i')}$ 满足 $calc_2\;in_i\neq calc_2\;in_i'$。代入上述概率，可以得到：

$$\begin{aligned}\Pr_{repr}\left[\tilde{\varphi}(repr)\right]&\leq\Pr_{repr}\left[\bigwedge_{i=1}^{t}repr\;in_i\neq repr\;in_i'\right]\\&=\prod_{i=1}^{t}\Pr_{repr}\left[repr\;in_i\neq repr\;in_i'\right]\\&\leq\left(1-s_V^{-lim_c}\right)^t\leq\exp\left(-\frac{t}{s_V^{lim_c}}\right)\end{aligned}$$

其中第二步利用了 in_i 与 in_i' 两两不同的性质，此时概率中的 t 个不等式完全独立；而第三部利用了 $repr$ 最多能输出 lim_c 个整数的性质，此时 $repr$ 输出域的大小不会超过 $s_V^{lim_c}$，两个随机输出不同的概率不会高于 $1-s_V^{-lim_c}$。

步骤 6：总结　目标定理可以通过拼接上述步骤的结果得到。首先，步骤 4 与 5 的结果可以导出如下的不等式：

$$\Pr_{\varphi\sim\mathcal{M}}\left[\tilde{\varphi}(repr)\;\middle|\;repr^*\in\mathbb{A}(\varphi)\right]\leq\exp\left(-\frac{t}{s_V^{lim_c}}\right)$$

① 此处我们假设大小的定义为某种二进制表示下使用的字节数。

在进一步考虑步骤 3 后，结果如下：

$$\Pr_{\varphi\sim\mathcal{M}}\left[\mathcal{S}_{repr}(\tilde{\varphi}) \neq repr^* \;\middle|\; repr^* \in \mathbb{A}(\varphi)\right] \leq 2^{lim_c \cdot lim_s} \exp\left(-\frac{t}{s_V^{lim_c}}\right)$$

最后，在加入步骤 1 与 2 后，我们知道上述不等式右侧的公式构成了 AutoLifter 有界无解率的一个上界，从而证明了目标定理。 □

压缩性质 定理 10 说明当错位因子足够大时，AutoLifter 的无解概率会足够小。幸运的是，实际中的提升问题通常都具有较大的错位因子。这一性质来源于提升问题的压缩性质。具体而言，因为提升问题的目标是为代价较高的中间数据结构找到更高效的表示，所以一般情况下，无论对于问题中描述常量输出的函数 $calc_2$ 还是对于程序空间 $\mathcal{L}_{repr}$ 中的候选程序，它们的输出空间通常远小于输入空间，于是它们的输入输出行为相当于将一个更大的集合压缩到了一个更小的集合。在本文使用的概率模型 $\mathcal{M}[\overline{calc}]$ 中，压缩性质可以被表示为形如 $s_V \ll s_D$ 的假设。

例 10. 在将分治函数应用于求解列表次小值时，参考程序 *sndmin* 与期望的辅助值 *min* 都具有压缩性质。它们的输入是一个整数列表，输出是列表中的一个元素。因此，在假设整数范围有界且列表长度趋于无穷的情况下，这两个函数输入空间与输出空间之间比值也会趋于无穷。

具有压缩性质的提升问题通常具有足够大的错位因子。具体地，考虑错误因子定义中的第一个条件，$calc_2\ in \neq calc_2\ in'$。如果近似地把 $calc_2$ 当做随机函数，那么对于任意一组不同的输入，上述条件被满足的概率是 $1 - 1/s_V > 1/2$。所以，几乎一定可以找到 $\Omega(s_D)$ 对不相关的合法输入，它们为错位因子提供了一个充分大的下界。

将上述分析与定理 10 中的结果结合，可以证明在假设压缩性质的情况下，AutoLifter 的有界错误率几乎一定很小，如下所示。

定理 11. 给定数据结构类型 D 与值类型 V，令 s_D 与 s_V 分别表示 D 与 V 中不同的元素数量。当 $calc_2$ 是类型为 $D \to V$ 的随机函数时，对于任意的常量 $\epsilon > 0$，当 s_D 与 $s_V^{lim_c}$ 的比值趋于无穷时，AutoLifter 的有界无解率大于 ϵ 的概率趋于 0。

证明. 容易验证在 $s_V = 1$ 时 AutoLifter 一定不会求解失败，所以不妨假设 $s_V > 1$。

首先分析错位因子在 $calc_2$ 随机时的大小。令 $k = \left\lfloor \frac{s_D}{2} \right\rfloor$，$\overline{(in_i, in_i')}$ 为任意 k 对不相关的输入，而 n 表示这 k 对中满足 $calc_2\ in \neq calc_2\ in'$ 的对数。一方面，根据错位因子的定义，n 构成了错位因子的一个下界。另一方面，因为每对输入满足条件的概率是 $1 - \frac{1}{s_V}$，所以 n 的期望是 $\left(1 - \frac{1}{s_V}\right) k \geq \frac{s_D}{4}$。

进一步地，令 δ 为任意大于 0 的常量。根据切诺夫界，可以得到如下的不等式。

$$\begin{aligned}\Pr_{calc_2}\left[n < (1-\delta)\frac{s_D}{4}\right] &\leq \Pr_{calc_2}\left[n < (1-\delta)\mathbb{E}[n]\right] \\ &< \exp\left(-\frac{\delta^2\mathbb{E}[n]}{2}\right) \\ &\leq \exp\left(-\frac{\delta^2 s_D}{8}\right)\end{aligned}$$

接着，根据定理 10 中的结果，对于任意的常量 ϵ，当错位因子 t 满足如下条件时，AutoLifter 有界无解率一定小于 ϵ，其中 $f(\cdot)$ 是一个细节不重要的函数。

$$2^{lim_c \cdot lim_s}\exp\left(-\frac{t}{s_V^{lim_c}}\right) < \epsilon \quad \Longleftarrow \quad t > s_V^{lim_c} f(\epsilon)$$

将两步分析结果整合，可以通过如下推导证明目标定理。

$$\begin{aligned}\Pr_{calc_2}\left[unreal\left(\overline{calc}, lim_s\right) > \epsilon\right] &< \Pr_{calc_2}\left[n < s_V^{lim_c} f(\epsilon)\right] \\ &= \Pr_{calc_2}\left[n < \frac{s_D}{8}\right] && \textbf{when } s_D \gg s_V^{lim_c} \\ &< \exp\left(-\frac{s_D}{32}\right) \\ &= 0 && \textbf{when } s_D \to \infty\end{aligned}$$

□

4.6 实验评估

为了检验 AutoLifter 的有效性，本文设计了实验来回答以下的问题：

- **RQ1**: AutoLifter 是否可以有效地将提升问题分解为更简单的子问题?
- **RQ2**: 针对表征函数设计的算法 1 是否能提升 AutoLifter 的求解效率?

4.6.1 实验设置

数据集　为了专注于检验 AutoLifter 在分解提升问题时的有效性，本实验只考虑本文数据集（第 3.5 节）中与其他算法模式相关的部分：这些任务相对困难，且一般需要复杂的表征函数。这一部分的数据集涉及四类算法模式，分别是分治算法 [10,12,121]、流算法 [13]、最长子段问题上的贪心算法 [23]、以及线段树上的区间操作算法 [122]。表 4.1 展示数据中与每一类算法相关的任务数量。

因为 AutoLifter 专注于求解提升问题，所以在本章的实验中，我们会先用 SuFu

表 4.1　数据集中与每类算法相关的任务数量。

算法	分治	流算法	最长子段	线段树	合计
任务数量	36	39	8	13	96

表 4.2　表征函数 *repr* 的程序空间。

起始非终结符	S	≔	$N_{\mathbb{Z}} \mid (S, S)$
整数表达式	$N_{\mathbb{Z}}$	≔	IntConst $\mid N_{\mathbb{Z}} \oplus N_{\mathbb{Z}} \mid sum\ N_{\mathbb{L}} \mid len\ N_{\mathbb{L}} \mid head\ N_{\mathbb{L}} \mid last\ N_{\mathbb{L}}$
		$\mid$	$access\ N_{\mathbb{Z}}\ N_{\mathbb{L}} \mid count\ F_{\mathbb{B}}\ N_{\mathbb{L}} \mid min\ N_{\mathbb{L}} \mid max\ N_{\mathbb{L}} \mid neg\ N_{\mathbb{Z}}$
列表表达式	$N_{\mathbb{L}}$	≔	Input $\mid take\ N_{\mathbb{Z}}\ N_{\mathbb{L}} \mid drop\ N_{\mathbb{Z}}\ N_{\mathbb{L}} \mid rev\ N_{\mathbb{L}} \mid map\ F_{\mathbb{Z}}\ N_{\mathbb{L}}$
		$\mid$	$filter\ F_{\mathbb{B}}\ N_{\mathbb{L}} \mid zip \oplus N_{\mathbb{L}}\ N_{\mathbb{L}} \mid sort\ N_{\mathbb{L}} \mid scanl \oplus N_{\mathbb{L}} \mid scanr \oplus N_{\mathbb{L}}$
二元运算符	$\oplus$	≔	$+ \mid - \mid \times \mid min \mid max$
整数函数	$F_{\mathbb{Z}}$	≔	$(+\ \text{IntConst}) \mid (-\ \text{IntConst}) \mid neg$
布尔函数	$F_{\mathbb{B}}$	≔	$(<0) \mid (>0) \mid odd \mid even$

将算法问题归约到提升问题，再评估 AutoLifter 在这些提升问题上的表现。

基准方法　本文考虑了两个可以被直接用于求解提升问题的通用程序合成方法，分别是 Enum [28] 与 Relish [98]。

- Enum [28] 是一个基于枚举的程序合成方法。给定一个提升问题，Enum 会按照从小到大的顺序枚举所有可能的 $(repr, comb)$，直到找到一个有效的解。
- Relish [98] 是一个基于空间表示法的合成方法。它首先使用**分层有限树自动机**排除许多无效程序，然后再在自动机的程序空间中搜索。

此外，为了评估 AutoLifter 中对表征函数的合成方法（算法 1），本文还考虑了一个 AutoLifter 的变种，称为 $\text{AutoLifter}_{\text{OE}}$。该变种与 AutoLifter 几乎相同，唯一的区别在于它直接使用观察等价法合成表征函数，而非使用本文中的算法 1。

其他配置。本文将每次求解的时间限制设为了 300 秒，内存限制设为了 8GB。

4.6.2　代码实现

在本章的实验中，本文针对整数列表实现了一个 AutoLifter 的原型。在给定相应的类型定义与程序空间的情况下，这一实现可以被简单地推广到其他情况。

领域特定语言　AuotLifter 依赖于两个程序空间 $\mathcal{L}_{aux}$ 与 $\mathcal{L}_{comb}$，分别对应候选的表征函数集合与合并算子集合。在本章的实现中，本文使用 DeepCoder [100] 中关于列表计算的领域特定语言作为表征函数的程序空间。它包含了 17 个与列表相关的运算符，包括常见的高阶函数（如 *map* 和 *filter*）以及在内部包含分支与递归计算的复杂操作符（如

表 4.3　合并算子 *comb* 的程序空间

起始非终结符	S	::=	$N_{\mathbb{Z}} \mid (S, S)$
整数表达式	$N_{\mathbb{Z}}$	::=	IntConst $\mid N_{\mathbb{Z}} \oplus N_{\mathbb{Z}} \mid \mathit{if}\ N_{\mathbb{B}}\ \mathit{then}\ N_{\mathbb{Z}}\ \mathit{else}\ N_{\mathbb{Z}} \mid N_{\mathbb{T}}.i$
布尔表达式	$N_{\mathbb{B}}$	::=	$\neg N_{\mathbb{B}} \mid N_{\mathbb{B}} \wedge N_{\mathbb{B}} \mid N_{\mathbb{B}} \vee N_{\mathbb{B}} \mid N_{\mathbb{Z}} \leq N_{\mathbb{Z}} \mid N_{\mathbb{Z}} = N_{\mathbb{Z}}$
元组表达式	$N_{\mathbb{T}}$	::=	Input $\mid N_{\mathbb{T}}.i$
二元运算符	$\oplus$	::=	$+ \mid - \mid \times \mid \mathit{div}$

count 和 *sort*）。此外，由于 DeepCoder 的语言不支持生成元组，本文在文法中添加了一个构造元组的运算符，以应对需要多个辅助值的情况。

而对于合并算子，本文使用程序合成竞赛 SyGuS-Comp [127] 中关于整数运算的领域特定语言作为程序空间。它包含基本的算术运算符（如 + 和 ×），比较运算符（如 = 和 ≤），布尔运算符（如 ¬ 和 ∧），以及分支运算符 *if-then-else*，并可以通过嵌套使用分支运算符来实现复杂计算。与表征函数的情况类似，本文添加了访问和构造元组的运算符（即 (S, S) 和 $N_{\mathbb{T}}.i$）来处理需要多个辅助值的情况。

这两个程序空间性质良好。首先，表征函数的程序空间满足压缩性质（第 4.5.2 节）。其中的每个程序都将一个的整数列表映射到一个固定大小的整数元组。因此，当列表长度足够大且每个整数的范围都被限制在固定范围时，这些程序的输入空间将会远大于其输出空间。此外，合并算子的程序空间足够高效。其中所有程序的时间复杂度均为 $O(1)$，从而保证了合成结果中合并算子的高效性。

其他配置　本文使用 EUSolver [38] 求解关于合并算子的叶子问题，它是现有最好的针对整数表达式的样例编程方法之一。

此外，在实现表征函数的合成方法（即算法 1）时，本文将参数 lim_c 设置为了 4，因为根据本文的观察，4 个辅助值就已经足以解决大多数已知的提升问题。

4.6.3　RQ1: AutoLifter 在求解提升问题时的有效性

表 4.4 展示了 AutoLifter 与 Enum 和 Relish 的比较结果。其中“求解数量”报告了每个求解器解决的任务数量；而“时间开销$_{\text{Base}}$”与“时间开销$_{\text{Ours}}$”分别展示了基准方法与 AutoLifter 的平均时间开销①。这些结果表明 AutoLifter 的求解能力显著优于基准方法。它不仅解决了更多的任务，而且时间开销显著更少。

然而，AutoLifter 的求解能力还有很大的提升空间。在本文的数据集上，它目前只能解决 96 个任务中的 58 个，仅占到约 60%。于是，本文分析了 AutoLifter 失败的 38 个任务，并归类得到以下两点原因：

① 在计算这些开销时，本文只考虑那些被基准方法与 AutoLifter 同时求解的任务。

表 4.4 AutoLifter 与 Enum 和 Relish 的比较结果。

求解器	分治			流算法		
	求解数量	时间开销$_{\text{Base}}$	时间开销$_{\text{Ours}}$	求解数量	时间开销$_{\text{Base}}$	时间开销$_{\text{Ours}}$
AutoLifter	**20**/36	42.8		**24**/39	1.84	
Enum	5/36	46.8	**0.23**	9/39	9.54	**0.30**
Relish	12/36	34.8	**6.57**	16/39	19.6	**0.35**
求解器	**最长子段**			**线段树**		
	求解数量	**时间开销$_{\text{Base}}$**	**时间开销$_{\text{Ours}}$**	**求解数量**	**时间开销$_{\text{Base}}$**	**时间开销$_{\text{Ours}}$**
AutoLifter	**5**/8	4.14		**9**/13	33.0	
Enum	1/8	15.0	**0.28**	4/13	115	**25.1**
Relish	3/8	4.23	**2.25**	7/13	86.5	**28.9**
求解器	**总计**					
	求解数量	**时间开销$_{\text{Base}}$**	**时间开销$_{\text{Ours}}$**			
AutoLifter	**58**/96	21.0				
Enum	19/96	31.3	**5.79**			
Relish	38/96	35.3	**7.38**			

- 首先，EUSolver 的求解能力显著不足。在这 38 个失败的任务中，AutoLifter 可以为 25 个任务合成符合预期的表征函数，但 EUSolver 并不能在时间限制能合成对应的合并算子。这是因为在复杂的提升问题中，合并算子的规模可能会非常大。它使用的操作符数量最大可达 559 ——这已经远远超出了 EUSolver 的求解能力。本文将在接下来的两章中重点解决这一问题。
- 其次，本文使用的默认程序空间的表达能力存在不足。剩下的 13 个任务都需要一些默认程序空间以外的特殊运算符，例如对整数列表的正则表达式匹配和整数的幂运算。在手动地将这些操作符加入程序空间后，AutoLifter 便能成功地为这 13 个任务合成正确的表征函数，并能完全解决其中的 7 个。

4.6.4 RQ2: AutoLifter 中表征函数合成方法的有效性

表 4.5 展示了 AutoLifter 与 AutoLifter$_{\text{OE}}$ 的比较结果，其组织方式与上个实验相同。该结果表明了算法 1 的有效性。在该算法的帮助下，AutoLifter 可以显著地解出更多的问题。注意到在共同解出的任务上，算法 1 会使用比朴素的观察等价法略多的时间，这是因为算法 1 中的优化带来了额外的常数开销。

此外，与表 4.4 中的数据对比，可以发现 AutoLifter$_{\text{OE}}$ 的表现仍然优于基准方法 Enum 与 Relish。这一结果进一步地展示了 AutoLifter 中的分解系统的有效性。

表 4.5　AutoLifter 与 $\text{AutoLifter}_{\text{OE}}$ 的比较结果。

求解器	分治			流算法		
	求解数量	时间开销$_{\text{Base}}$	时间开销$_{\text{Ours}}$	求解数量	时间开销$_{\text{Base}}$	时间开销$_{\text{Ours}}$
AutoLifter	**20**/36	42.8		**24**/39	1.84	
$\text{AutoLifter}_{\text{OE}}$	12/36	8.10	**5.93**	21/39	1.69	**1.60**
求解器	**最长子段**			**线段树**		
	求解数量	**时间开销$_{\text{Base}}$**	**时间开销$_{\text{Ours}}$**	**求解数量**	**时间开销$_{\text{Base}}$**	**时间开销$_{\text{Ours}}$**
AutoLifter	5/8	4.14		**9**/13	33.0	
$\text{AutoLifter}_{\text{OE}}$	5/8	**3.92**	4.14	7/13	**22.3**	28.7
求解器	**总计**					
	求解数量	**时间开销$_{\text{Base}}$**	**时间开销$_{\text{Ours}}$**			
AutoLifter	**58**/96	21.0				
$\text{AutoLifter}_{\text{OE}}$	45/96	**6.84**	7.25			

4.7　小结

为了解决合成效率不足的挑战，本章针对提升问题提出了分解系统 AutoLifter。它由两条分解规则组成，分别名为组件消除与变量消除。通过递归地应用这两条规则，AutoLifter 会将提升问题分解为一系列的子任务，每个子任务仅需合成目标程序的一小部分，从而显著减小了每一步合成的程序规模。

分解程序合成问题的挑战来源于子程序之间的依赖性。AutoLifter 使用近似规约打破依赖：组件消除规则会忽略一部分的组件，并优先满足另一部分组件上的规约；而样例消除规则会用数学函数的空间近似合并算子的程序空间，并优先合成表征函数。本文证明了这种近似规约并不会影响 AutoLifter 的正确性，同时通过在概率模型上进行数学分析，论证了这些近似不会对 AutoLifter 的完备性产生显著影响。

本文的实验结果证明了 AutoLifter 的有效性。与不分解问题的合成方法相比，AutoLifter 可以以显著更快的速度解出显著更多的提升问题。然而，AutoLifter 仍然无法解出数据集中的很大一部分问题。其中的主要原因在于，现有样例编程求解器难以合成提升问题中规模庞大的合并算子。为了改进这一点，本文接下来的两章将专注于提升样例编程的求解能力。

第五章　具有泛化能力保障的程序合成方法

5.1　引言

复杂的程序合成任务往往会涉及多个程序分支。分支程序包含众多分支语句和分支条件，其中分支语句需处理各种情况的输出，而分支条件则根据输入决定执行哪个分支语句。尽管单个分支语句和条件可能相对简单，但它们组合起来往往构成规模较大的分支程序，从而对程序合成的效率构成重大挑战。

为了高效地合成分治程序，Alur 等人于 2015 年提出了**基于合一化的程序合成框架**（STUN）。回顾第 2.2.1 节中的介绍，STUN 框架下的程序合成方法由两个组件组成：一个语句合成器和一个合一器。给定输入输出样例，语句合成器负责合成一组分支语句，确保每个样例至少被其中一个语句满足；随后，合一器为这些分支语句生成对应的分支条件，并将其整合为一个完整的分支程序。

然而，现有 STUN 求解器在处理算法问题时仍面临显著的效率瓶颈。即使是当前最先进的 STUN 求解器 EUSolver，在上一章的实验中也仅能成功求解约 60% 的合成任务。本文返现 EUSolver 的主要缺陷在于其**泛化能力**较弱。在基于 CEGIS 框架合成程序时，EUSolver 往往需要通过反复收集大量反例，才能逐步逼近并最终合成目标程序。例如，在合成与 $\max(x_1, x_2, x_4 + x_5)$ 等价的分支程序时，EUSolver 需要经过多达 393 轮迭代才能找到目标程序。如此高的迭代次数导致了显著的时间开销。

为了保障 STUN 求解器的泛化能力，本文应用了机器学习领域中的**奥卡姆学习理论** [44]，并在此基础上提出了**奥卡姆求解器**的概念。奥卡姆求解器遵循奥卡姆剃刀原理，偏好于合成较小的合法程序。它确保对于任意样例编程问题，其结果的大小不会超过最小合法程序的多项式级别，并在渐进意义上小于给定的样例数量。奥卡姆学习理论保证了奥卡姆求解器在接受多项式级别的样例时可以达到任何期望的准确度 [44]；而在实践中，基于奥卡姆学习理论的方法在不同领域中均表现出了良好的泛化能力 [128–132]。

在上述理论的指导下，本文证明了 EUSolver 不满足奥卡姆求解器的要求。因此，本文着手为 STUN 框架设计高效的奥卡姆求解器，并命名为 PolyGen。设计奥卡姆求解器的关键挑战在于如何在满足奥卡姆求解器条件的同时实现高效的程序合成。为了达到这一目标，PolyGen 将分支程序的合成任务分解为多个子任务，每个子任务仅涉及目标程序的一小部分。同时，本文将奥卡姆求解器的要求分解为一组适用于各子问题的充分条件。这些条件要求每个子任务要么合成一个较小的程序，要么合成一组总大小较小的程序。基于这些条件，本文为每一个子问题设计了相应的高效算法，以确保其能够满足各自的条件。

表 5.1　输入输出样例与对应的分支语句。

ID	输入	输出	语句	ID	输入	输出	语句	ID	输入	输出	语句
e_1	$(0,1,2)$	1		e_4	$(0,2,0)$	3		e_7	$(0,0,1)$	2	
e_2	$(1,0,2)$	2	$x+1$	e_5	$(-1,3,0)$	4	$y+1$	e_8	$(-3,3,-2)$	-1	$z+1$
e_3	$(-1,3,2)$	0		e_6	$(-1,1,-1)$	2		e_9	$(-1,0,4)$	5	

综上所述，本章的主要创新点包括以下内容：

- 基于奥卡姆学习理论提出了奥卡姆求解器的概念，并对 EUSolver 和 STUN 框架的泛化能力进行了理论分析（第 5.3 节）。
- 设计并实现了一个新颖的 STUN 求解器 PolyGen，该求解器在满足奥卡姆求解器要求的同时实现了高效的程序合成（第 5.4 节与 第 5.5 节）。
- 在条件整数运算领域对 PolyGen 进行了实验评估，验证了其在泛化能力和合成效率方面的优越性（第 5.6 节）。

5.2 概述

本节将用一个 SyGuS-Comp 中的任务①展示 PolyGen 的主要思想。该任务的目标程序 p^* 如下，其中 x,y,z 是三个整数输入。本节考虑该任务上的 9 个输入输出样例，如表 5.1 所示，其中“语句”展示了目标程序在每个样例上被运行的分支语句。

$$
\begin{aligned}
p^*(x,y,z) \coloneqq\ & \texttt{if } (x+y\geq 1) \texttt{ then } \{ \\
& \quad \texttt{if } (x+z\geq 1) \texttt{ then } \{x+1\} \texttt{ else } \{y+1\} \\
& \} \texttt{ else } \{ \\
& \quad \texttt{if } (y+z\geq 1) \texttt{ then } \{z+1\} \texttt{ else } \{y+1\} \\
& \}
\end{aligned}
$$

5.2.1 样例编程求解器 EUSolver

本章的目标是为分支程序设计一个奥卡姆求解器。如前文所述，奥卡姆求解器合成的程序大小应与目标程序的大小呈多项式关系，并在渐进意义下小于样例数量。本节将先分析现有的求解器 EUSolver [38]，并展示它不是奥卡姆求解器的直观原因。

EUSolver 遵循 STUN 框架，并在设计上也试图合成较小的程序。其语句合成器与合一器分别从小到大枚举分支语句和条件，然后尝试将最小的一些语句与条件组合成一个满足所有样例的程序。然而，尽管这种方法控制了单个语句与条件的大小，但它忽略了这些语句与条件的数量，因此经常会产生过大的结果。

① 任务名为 `mpg_ite.sl`。

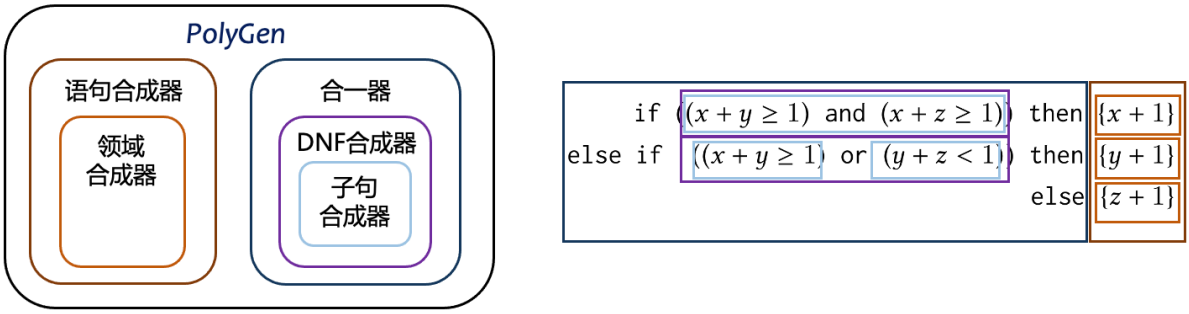

图 5.1　PolyGen 的结构，其中右图的每一个矩形表示了由左图对应颜色组件合成的部分。

语句合成器　EUSolver 的语句合成器按从小到大的顺序枚举所有可能的分支语句。当一个语句满足的样例集合与所有更小的语句都不同时，就将它加入结果集合，直到结果集合覆盖了所有样例为止。

这一策略的缺点在于，它经常会不必要地返回过多的小表达式，每个表达式只满足很少一部分样例。在本节的示例任务中，如果程序空间包含常量 $-1,\ldots,5$，EUSolver 的语句合成器会返回这些常量的集合 $\{-1,\ldots,5\}$ 而不是预期的 $\{x+1, y+1, z+1\}$。尽管这些语句都很小，但极端情况下，这些语句的数量会随着样例数量的增加而线性增加，从而违反奥卡姆求解器的要求。

合一器　EUSolver 的合一器将分支程序建模为一棵决策树，并采用传统的 ID3 算法构建合成结果。在这一过程中，样例被视为需要由决策树区分的集合，分支语句被用作决策树中的标签，通过枚举生成的一系列分支条件则作为决策树中可用的决策依据，而一个样例对应的合法标签是所有能够满足该样例的分支语句。

然而，ID3 的缺点在于它是为模糊分类问题涉及的，而在程序合成的场景下，它最大化信息熵的策略可能会对程序合成产生负面影响。例如，在 ID3 的评价标准下，条件 $x \geq 0$ 为解决本节的示例任务带来了大量的信息收益。在该任务的样例集合中（表 5.1），三个分支语句均匀分布，各满足三个样例；而分支条件 $x \geq 0$ 将这些样例划分为分布不均匀的两组，带来了信息收益：在第一组中有 50% 的样例对应 $x+1$，而在另一组中只有 20% 的样例对应 $x+1$。然而，对于程序合成来说，这两组样例仍然涉及到全部的三个分支语句，仍然需要找到条件来将它们区分。因此，选择 $x > 0$ 并不会为程序合成带来收益，反而会显著地增大了结果程序的大小。

综上，EUSolver 的核心缺陷在于只限制了每个分支语句与条件的大小，但没有限制其数量，于是仍然可能会返回过大的程序。

5.2.2 PolyGen 的求解思路

为了避免 EUSolver 的缺陷，PolyGen 不仅控制单个分支语句和分支条件的大小，还会控制它们的数量。PolyGen 的结构如图 5.1 所示。它由一系列的子合成器构成，每个只负责目标程序的一部分，从语句集合、条件集合，到单个语句、单个条件，再到每个条件中的每个子句。

语句合成器 PolyGen 的语句合成器对分支语句的数量设置了上限，并只考虑数量不超过该上限的语句集合。这一上限会在迭代过程中不断增加，直到找到一个大小不超过上限且满足所有样例的语句集合。每次迭代中，PolyGen 会使用一个随机化的合成方法。该方法保证如果在当前上限内存在合法的语句集合，那么至少以 $(1-\epsilon)$ 的概率找到一个合法结果。尽管这一随机化的方法存在合成失败的概率，但它在迭代的框架下仍然能保证一个小的结果：一旦当前上限内存在合法解，那么在经过额外的 n 此迭代后，合成失败的概率将降至 ϵ^n，而此时的上限也仅仅只增加了 n。

PolyGen 使用的随机化方法依赖于一个**领域合成器**，它需要是一个针对单一分支语句的奥卡姆合成器。为了方便说明，假设当前分支语句数量的上限为 3。此时，本章的示例任务中存在着一个合法的语句集合 $T = \{x+1, y+1, z+1\}$。

为了找到这个语句集合，PolyGen 会先对所有样例采样，每次得到一个子集，并调用领域求解器为这个子集合成分支语句。如果某个子集同时被 T 中的某个分支语句满足，领域求解器将有机会返回这个分支语句。例如，如果采样集合为 $\{e_1, e_2\}$，则根据奥卡姆求解器的泛化能力，领域求解器的合成结果将很有可能是 $x+1$。因此，在采样足够多的子集后，PolyGen 将可以正确地找到 T 中的某个分支语句。随后，它将会递归地处理剩余的输入输出样例，直到所有样例都满足。因为每轮 PolyGen 都能以高概率找到 T 中的某个语句，所以在递归后，它能以高概率找到完整的语句集合 T。

合一器 为了控制条件的数量，PolyGen 的合一器会合成**决策列表** [133] 而非决策树。在决策列表中，每个条件会将一个分支语句与其余语句完全区分，于是条件的数量将严格等于分支语句的数量减一，不会更多。图 5.1展示了 PolyGen 在本章示例任务上的合成结果，它在语义上完全等价于目标程序 p^*。

然而，因为决策列表的结构是线性的，所以它使用的条件会比决策树中的更大，带来了额外的效率挑战。为了解决这一挑战。本文令 PolyGen 按照析取范式（DNF）的形式合成条件，并对 DNF 中的不同部分设计了子求解器：给定一系列按照从小到大枚举得到的文字，**子句合成器**从这些文字中合成子句，**DNF 合成器**基于子句合成器合成分支条件，最后合一器基于 DNF 合成器合成最终的分支程序。

给定一组分支语句，合一器为每个语句 t 的条件创建一个子任务，其目标条件需要在**只**被 t 满足的样例上为真，并在不被 t 以及任何先前语句满足的样例上为假。例如，语句 $y+1$ 的条件需要在 $\{e_4, e_5, e_6\}$ 为真，在 $\{e_7, e_8, e_9\}$ 为假，而在 $\{e_1, e_2, e_3\}$ 上的输出不限，因为这些样例已经被上一条语句 $x+1$ 满足。这些子任务会被 DNF 求解器解决：如果该求解器是奥卡姆求解器，那么合一器同样能返回足够小的结果程序。

给定一系列正例（即输出为真的样例）与负例（即输出为假的样例），DNF 求解器需要合成一组子句，其中所有子句在所有负例上均为假，并且在每个正例上都至少有一个子句为真。注意到这个合成问题与分支语句求解器具有相似的形式：它们都需要找到一组程序（此处为一组子句）来覆盖一系列的样例（此处为所有负例）。于是，DNF 求解器采用了与分支语句求解器相同的算法，将问题拆分到了子句的合成。

最后，在合成子句时，PolyGen 会先按照从小到大的顺序枚举一系列文字，接着从这些文字中找到合适的子集，以构成满足所有样例的子句。PolyGen 将这个问题规约到了**加权集合覆盖问题**，并使用了一个标准的近似算法 [134] 对其求解。该近似算法不会返回过大的结果程序，从而确保了子句合成器满足奥卡姆求解器的条件。

5.3　奥卡姆求解器

本节将介绍奥卡姆求解器的定义，并分析 EUSolver 与 STUN 框架的可泛化性。

5.3.1　符号说明

本节将引入一些必要的符号，用于分析与讨论样例编程求解器的性质。

一个样例编程问题通常依赖于一个合成域，例如字符串处理领域 [135] 或者条件整数运算领域 [38]。本文将一个合成域定义为一个四元组 $\mathbb{D} = (\mathbb{P}, \mathbb{I}, \mathbb{O}, [\![.]\!]_{\mathbb{D}})$，其中 $\mathbb{P}, \mathbb{I}, \mathbb{O}$ 分别表示程序空间、输入空间和输出空间；而 $[\![.]\!]_{\mathbb{D}}$ 是一个解释器，$[\![p]\!]_{\mathbb{D}} :: \mathbb{I} \mapsto \mathbb{O}$ 将每个程序解释为一个输入空间到输出空间的函数。

为简单起见，本章对合成域做两个假设：(1) 存在一个对所有领域通用的解释器 $[\![.]\!]$；(2) 输出空间 $\mathbb{O}$ 总是由程序空间 $\mathbb{P}$、输入空间 $\mathbb{I}$ 和解释器 $[\![.]\!]$ 导出，即 $\mathbb{O} \coloneqq \{[\![p]\!](I) \mid p \in \mathbb{P}, I \in \mathbb{I}\}$。于是，本章会将合成域简写为 $(\mathbb{P}, \mathbb{I})$。此外，本文将使用符号 $\mathcal{F}$ 表示一族合成域，从而讨论样例编程求解器在不同合成域上的一般性质。

5.3.2　奥卡姆学习理论与奥卡姆求解器

在机器学习理论中，Anselm 提出了**奥卡姆学习理论** [44] 用于解释奥卡姆剃刀原则的有效性。在程序合成领域中，符合奥卡姆学习理论要求的样例求解器（称为**奥卡姆**

求解器）需要保证其合成结果的大小不会超过目标程序的多项式级别，并不会达到样例数量的线性级别。

定义 13 (奥卡姆求解器[①])**.** 对于常数 $\alpha \geq 1, 0 \leq \beta < 1$，如果存在常数 $c, \gamma > 0$ 使得对于任何程序域 $\mathbb{D} \in \mathcal{F}$，目标程序 $p^* \in \mathbb{P}$，输入集合 $\{I_1, \ldots, I_n\} \subseteq \mathbb{I}$，以及错误率 $\epsilon \in \left(0, \frac{1}{2}\right)$：

$$\Pr\left[size\left(\mathcal{S}\left(T(p^*, I_1, \ldots, I_n)\right)\right) > c\left(size(p^*)\right)^{\alpha} n^{\beta} \ln^{\gamma}\left(\frac{1}{\epsilon}\right)\right] \leq \epsilon$$

则 *PBE* 求解器 $\mathcal{S}$ 是族 $\mathcal{F}$ 上的 (α, β)-奥卡姆求解器，其中 $size(p)$ 是程序 p 的二进制表示的长度，$T(p^*, I_1, \ldots, I_n)$ 表示对应于目标程序 p^* 和输入 $I_1, \ldots, I_n$ 的 *PBE* 任务。

奥卡姆学习理论在概率近似正确框架下为奥卡姆求解器提供了可泛化性保障。

定理 12 (泛化能力保障[44])**.** 设 $\mathcal{S}$ 是域 $\mathbb{D}$ 上的 (α, β)-奥卡姆求解器。则存在多项式 $f(\cdot, \cdot)$ 使得对于任何 $0 < \epsilon, \delta < 1$，对于 $\mathbb{I}$ 上的任何分布 $\boldsymbol{D}$ 和任何目标程序 $p^* \in \mathbb{P}$，都满足：

$$\forall n > f\left(\frac{1}{\epsilon}, \ln\left(\frac{1}{\delta}\right)\right) \cdot \left(size(p^*)\right)^{\alpha/(1-\beta)},$$
$$\Pr_{I_i \sim D}\left[err_{D,p^*}\left(\mathcal{S}\left(T(p^*, I_1, \ldots, I_n)\right)\right) \geq \epsilon\right] \leq \delta$$

其中 $f(\cdot, \cdot)$ 是一个合适的多项式函数，而 $err_{D,p^*}(p)$ 表示当输入分布为 $\boldsymbol{D}$ 且目标程序为 p^* 时程序 p 的错误率，即 $err_{D,p^*}(p) \coloneqq \Pr_{I \sim D}\left[[\![p]\!](I) \neq [\![p^*]\!](I)\right]$。

当 ϵ 和 δ 固定时，定理12 表明着一个 $(\alpha, \beta)-$ 奥卡姆求解器只需要 $O\left(\text{size}(p^*)^{\alpha/(1-\beta)}\right)$ 个例子就能找到一个与目标程序 p^* 相似的程序。

奥卡姆求解器的定义可以反映样例编程求解器的实际泛化能力。以两个基本求解器 $\mathcal{S}_{min}$ 和 $\mathcal{S}_{rand}$ 为例，它们的定义如下，其中 T 是一个任意的样例编程任务，而 $\mathbb{P}(T) \subseteq \mathbb{P}$ 表示满足所有样例的合法程序的集合。

- $\mathcal{S}_{rand}$ 对结果质量没有任何保证，它会从 $\mathbb{P}(T)$ 中均匀地返回一个程序：

$$\forall p \in \mathbb{P}(T), \Pr\left[\mathcal{S}_{rand}(T) = p\right] = |\mathbb{P}(T)|^{-1}$$

- $\mathcal{S}_{min}$ 遵循奥卡姆剃刀原则，它总是返回 $\mathbb{P}(T)$ 中语法上最小的程序：

$$\mathcal{S}_{min}(T) \coloneqq \arg\min_{p \in \mathbb{P}(T)} size(p)$$

已有的大量实验结果表明，$\mathcal{S}_{min}$ 具有比 $\mathcal{S}_{rand}$ 更好的泛化能力。奥卡姆求解器的概念可以在理论上解释这一优势，因为 $\mathcal{S}_{min}$ 是一个奥卡姆求解器，而 $\mathcal{S}_{rand}$ 不是。

① 奥卡姆求解器的原始定义仅适用于确定性算法。在此本文将其定义自然地扩展到随机算法。

定理 13. 令 $\mathcal{F}_A$ 为所有合成域形成的族。$\mathcal{S}_{min}$ 是 $\mathcal{F}_A$ 上的 $(1,0)$-奥卡姆求解器，而 $\mathcal{S}_{rand}$ 不是 $\mathcal{F}_A$ 上的奥卡姆求解器。

证明. 令 p^* 表示目标程序，同时令 p 为 $\mathcal{S}_{min}$ 的合成结果。根据 $\mathcal{S}_{min}$ 的定义，$size(p) \leq size(p^*)$，于是 $\mathcal{S}_{min}$ 是一个 $(1,0)$-奥卡姆求解器。

对于 $\mathcal{S}_{rand}$，假设程序空间中的所有程序都满足给定的样例，且目标程序 p^* 是程序空间中最小的程序。那么，根据 $\mathcal{S}_{rand}$ 的定义，其合成结果 p 在程序空间上服从均匀分布。因此，$size(p)$ 可能比 $size(p^*)$ 大任意多倍，所以 $\mathcal{S}_{rand}$ 不是一个 Occam 求解器。 □

5.3.3　STUN 框架

回顾第 2.2.1 节中的介绍，STUN 框架专注于合成具有嵌套分支语句的程序。本节将引入它的形式化定义，以便于后续的理论分析。

STUN 要求程序空间可以被分解为两个子空间，分别对应分支语句与条件。

定义 14. 程序空间 $\mathbb{P}$ 是分支程序空间当且仅当存在着两个子空间 $\mathbb{P}_t$ 和 $\mathbb{P}_c$，使得 $\mathbb{P}$ 是满足以下条件的最小程序集合。

$$\mathbb{P} = \mathbb{P}_t \cup \left\{ \texttt{if } c \texttt{ then } p_1 \texttt{ else } p_2 \mid p_1, p_2 \in \mathbb{P}, c \in \mathbb{P}_c \right\}$$

本文使用二元组 $\langle \mathbb{P}_t, \mathbb{P}_c \rangle$ 表示从分支语句空间 $\mathbb{P}_t$ 和条件空间 $\mathbb{P}_c$ 派生的分支程序空间。此外，如果领域 $\mathbb{D}$ 使用的是分支程序空间，我们称该领域为分支领域。

STUN 通过两个步骤合成程序。它首先调用**分支语句合成器**合成一组分支语句 $\boldsymbol{P} \subseteq \mathbb{P}_t$，使得对于任何样例，$\boldsymbol{P}$ 中总有一个满足样例的程序；接着，它调用**合一器**从条件程序空间 $\langle \boldsymbol{P}, \mathbb{P}_c \rangle$ 中合成一个满足所有样例的分支程序。

定义 15 (分支语句合成器)**.** 给定样例编程任务 $\boldsymbol{T}$，分支语句合成器 $\mathcal{T}$ 需要返回一组覆盖所有样例的分支语句：

$$\forall (\boldsymbol{I} \mapsto \boldsymbol{O}) \in \boldsymbol{T},\ \exists p \in \mathcal{T}(\boldsymbol{T}), \quad [\![p]\!](\boldsymbol{I}) = \boldsymbol{O}$$

定义 16 (合一器)**.** 给定分支领域 $\mathbb{D} = \left(\langle \mathbb{P}_t, \mathbb{P}_c \rangle, \mathbb{I}\right)$，合一器 $\mathcal{U}$ 是一个函数，使得对于任何分支语句集 $\boldsymbol{P}$，$\mathcal{U}(\boldsymbol{P})$ 都是分支领域 $\left(\langle \boldsymbol{P}, \mathbb{P}_c \rangle, \mathbb{I}\right)$ 上的样例编程求解器。

一个 STUN 求解器由一个分支语句合成器 $\mathcal{T}$ 和一个合一器 $\mathcal{U}$ 构成。给定样例编程任务 $\boldsymbol{T}$，其合成结果为 $\mathcal{U}(\mathcal{T}(\boldsymbol{T}))(\boldsymbol{T})$。Alur 等人证明了当条件领域是**分支闭合**时，STUN 的合成过程是完备的[36]，即对于任何有解的样例编程任务 $\boldsymbol{T}$，STUN 中分支语句合成器与合一器对应的子问题都一定有解。

定义 17 (分支闭合)**.** 分支领域 $\mathbb{D}$ 是分支闭合的，如果：

$$\forall p_1, p_2 \in \mathbb{P}_t, \exists c \in \mathbb{P}_c, \forall I \in \mathbb{I}, \qquad [\![c]\!](I) \iff [\![p_1]\!](I) = [\![p_2]\!](I)$$

值得注意的事，任何包含相等判断操作符的分支领域都是分支闭合的，因为条件 c 可以被构造为条件 $p_1 = p_2$。本章默认假设分支程序空间是分支闭合的，并使用 $\mathcal{F}_C$ 表示所有分支闭合的合成域形成的族。

EUSolver [38] 是目前最先进的基于 STUN 框架的样例编程求解器。它将合成效率和泛化能力作为其设计目的，并在两者之间做出权衡。

EUSolver 中的分支语句合成器 $\mathcal{T}_E$ 受 $\mathcal{S}_{\min}$ 的启发，会按照从小到大的顺序枚举 $\mathbb{P}_t$ 中的分支语句。对于每个分支语句 t，如果没有更小的分支语句覆盖与 t 相同的样例集合，则 $\mathcal{T}_E$ 会将 t 加入结果集合。$\mathcal{T}_E$ 会在当前结果覆盖所有样例时将其返回。

EUSolver 中的合一器 $\mathcal{U}_E$ 分两步进行合成。首先，它会按照从小到大的顺序枚举 $\mathbb{P}_c$ 中的条件，并得到一个足够区分当前样例集合的条件集合 C。接着，$\mathcal{U}_E$ 会使用决策树学习算法 ID3 [99] 递归地合成分支程序。在每次递归时，$\mathcal{U}_E$ 会先尝试用一个分支语句覆盖所有剩余的样例，如果有，则将其作为结果返回；如果没有，$\mathcal{U}_E$ 会启发式地从 C 中选择一个条件，并根据该条件将当前样例分为两部分，递归地用于合成两个分支。

5.3.4 STUN 框架的可泛化性

本章的目标是在 STUN 框架下设计一个奥卡姆求解器。为了实现这一点，本文首先将奥卡姆求解器的条件拆分到了语句合成器与合一器，如下所示。

定义 18 (奥卡姆语句合成器)**.** 对于常数 $\alpha \geq 1, 0 \leq \beta < 1$，如果存在常数 $c, \gamma > 0$，使得对于任何领域 $\mathbb{D} \in \mathcal{F}_C$，目标程序 $p^* \in \mathbb{P}$，输入集合 $\{I_1, \ldots, I_n\} \subseteq \mathbb{I}$，以及错误率 $\epsilon \in \left(0, \frac{1}{2}\right)$，分支语句求解器 $\mathcal{T}$ 是 $\mathcal{F}_C$ 上的 (α, β)-奥卡姆语句合成器，如果它合成的所有分支语句的总大小几乎不会超过目标程序的多项式范围，如下所示：

$$\Pr\left[tsize\left(\mathcal{T}\big(T(p^*, I_1, \ldots, I_n)\big)\right) > c\left(size(p^*)\right)^{\alpha} n^{\beta} \ln^{\gamma}\left(\frac{1}{\epsilon}\right)\right] \leq \epsilon$$

其中 $tsize(P)$ 是分支语句集 P 中分支语句的总大小，即 $\sum_{t \in P} size(t)$。

定义 19 (奥卡姆合一器)**.** 对于常数 $\alpha \geq 1, 0 \leq \beta < 1$，如果存在常数 $c, \gamma > 0$，使得对于任何领域 $\mathbb{D} \in \mathcal{F}_C$，分支语句集合 $P \subseteq \mathbb{P}_t$，目标程序 $p^* \in (P, \mathbb{P}_t)$，输入集合 $\{I_1, \ldots, I_n\} \subseteq \mathbb{I}$，以及错误率 $\epsilon \in \left(0, \frac{1}{2}\right)$，合一器 $\mathcal{U}$ 是 $\mathcal{F}_C$ 上的 (α, β)-奥卡姆合一器，如果它合成结果的大小几乎不会超过目标程序大小以及分治语句总大小的多项式范围，如下所示：

$$\Pr\left[size\left(\mathcal{U}(P)\big(T(p^*, I_1, \ldots, I_n)\big)\right) > c\big(\max(size(p^*), tsize(P))\big)^{\alpha} n^{\beta} \ln^{\gamma}\left(\frac{1}{\epsilon}\right)\right] \leq \epsilon$$

本文证明了上述定义构成了 STUN 框架下奥卡姆求解器的一个充分条件，即任何满足上述定义的 STUN 组件一定可以构成一个奥卡姆求解器。这一理论结果使得我们可以独立地设计语句合成器与合一器，每一步只需要满足的各自的条件，而不再需要考虑全局的奥卡姆求解器的要求。

定理 14. 设 $\mathcal{T}$ 是 $\mathcal{F}_C$ 上的 (α_1, β_1)-奥卡姆语句合成器，$\mathcal{U}$ 是 $\mathcal{F}_C$ 上的 (α_2, β_2)-奥卡姆合一器，其中 $\beta_1\alpha_2 + \beta_2 < 1$。则由 $\mathcal{T}$ 和 $\mathcal{U}$ 组成的 *STUN* 求解器一定是 $\mathcal{F}_C$ 上的奥卡姆求解器，其参数如下所示：

$$\alpha \coloneqq (\alpha_1 + 1)\alpha_2 \qquad \beta \coloneqq \beta_1\alpha_2 + \beta_2$$

证明. 令 p^* 表示目标程序，令 $P \coloneqq \{t_1, \ldots, t_n\}$ 表示 $\mathcal{T}$ 合成的语句集合。根据奥卡姆求解器的定义，存在常量 c_1 与 γ_1 使得：

$$\forall \epsilon_1 \in \left(0, \frac{1}{2}\right), \Pr\left[tsize\,(P) > c_1\left(size(p^*)\right)^{\alpha_1} n^{\beta_1} \ln^{\gamma_1}\left(\frac{1}{\epsilon_1}\right)\right] \leq \epsilon_1$$

按照如下方式构造函数 $\varphi : \mathbb{P}_t \mapsto \langle P, \mathbb{P}_c\rangle$。它将语句空间中的每个候选语句都映射到 $\mathcal{U}$ 的程序空间中的一个等价程序。

$$\varphi(t) \coloneqq \texttt{if } (t = t_1) \texttt{ then } t_1 \texttt{ else } \ldots \texttt{ else if } (t = t_{n-1}) \texttt{ then } t_{n-1} \texttt{ else } t_n$$

根据函数 $size(p)$ 的定义（即程序 p 的二进制表示长度），可以得到如下所示的不等式：

$$size(\varphi(t)) \leq 2n\lceil \log_2 N\rceil + n \cdot size(t) + 2tsize(P) \leq c_3 size(t) tsize(P)$$

其中 N 表示语法中产生式的数量，c_3 是一个足够大的常量。

函数 φ 可以被扩展到完整的程序空间 $\langle \mathbb{P}_t, \mathbb{P}_c\rangle$ 上：$\varphi(p)$ 会将程序 p 中的所有分支语句 t 替换为对应的 $\varphi(")$。根据上述不等式，$size(\varphi(p))$ 不会超过 $c_3 size(p) tsize(P)$。

令 p_u 表示 $\mathcal{U}$ 的合成结果。因为 $\varphi(p^*)$ 是合一过程上的一个合法程序，所以根据奥卡姆合一器的定义，一定存在常数 c_2 与 γ_2 满足如下的不等式：

$$\forall \epsilon_2 \in \left(0, \frac{1}{2}\right), \Pr\left[size\left(p_u\right) > c_2\big(\max(size(\varphi(p^*)), tsize(P))\big)^{\alpha_2} n^{\beta_2} \ln^{\gamma_2}\left(\frac{1}{\epsilon_2}\right)\right] \leq \epsilon_2$$

对于任意的错误率 $\epsilon \in \left(0, \frac{1}{2}\right)$，通过将 ϵ_1 与 ϵ 取为 $\frac{1}{2}\epsilon$，可以进行如下所示推导：

$$\begin{aligned}
&\Pr\left[tsize\,(P) > c_1\left(size(p^*)\right)^{\alpha_1} n^{\beta_1} \ln^{\gamma_1}\left(\frac{2}{\epsilon}\right)\right] \leq \frac{\epsilon}{2} \bigwedge \\
&\Pr\left[size\left(p_u\right) > c_2\big(\max(size(\varphi(p^*)), tsize(P))\big)^{\alpha_2} n^{\beta_2} \ln^{\gamma_2}\left(\frac{2}{\epsilon}\right)\right] \leq \frac{\epsilon}{2} \\
\Longrightarrow\ &\Pr\left[size\left(p_u\right) > c_2\left(c_1 c_3\left(size(p^*)\right)^{\alpha_1+1} n^{\beta_1} \ln^{\gamma_1}\left(\frac{2}{\epsilon}\right)\right)^{\alpha_2} n^{\beta_2} \ln^{\gamma_2}\left(\frac{2}{\epsilon}\right)\right] \leq \epsilon \\
\Longrightarrow\ &\Pr\left[size\left(p_u\right) > c\left(size(p^*)\right)^{(\alpha_1+1)\alpha_2} n^{\beta_1\alpha_2+\beta_2} \ln^{\gamma}\left(\frac{1}{\epsilon}\right)\right] \leq \epsilon
\end{aligned}$$

其中 c 是一个足够大的常量，而 $\gamma \coloneqq \gamma_1\alpha_2 + \gamma_2$.

将上述结果对应到奥卡姆求解器的定义，便能得知由 $\mathcal{T}$ 与 $\mathcal{U}$ 组成的 STUN 求解器是一个 $((\alpha_1+1)\alpha_2, \beta_1\alpha_2+\beta_2)$-奥卡姆求解器。 □

5.3.5 EUSolver 的可泛化性

本节在最后分析 EUSolver 的泛化能力并证明它不满足奥卡姆求解器的要求。首先考虑语句求解器 $\mathcal{T}_E$，尽管 $\mathcal{T}_E$ 控制了每个分支语句的大小，但是因为它并不限制分支语句的数量，所以它返回的语句集合的总大小可能会非常大，如例 11 所示。

例 11. 考虑以下分支语句空间 $\mathbb{P}_t^n$，输入空间 $\mathbb{I}_t^n$ 和目标程序 p：

$$\mathbb{P}_t^n \coloneqq \{2, 3, \ldots, n+1, x+1\} \quad \mathbb{I}_t^n \coloneqq \{1, 2, \ldots, n\} \quad p \coloneqq x + 1$$

由于 p 是 $\mathbb{P}_t^n$ 中最大的分支语句，且对于 $\mathbb{I}_t^n$ 中的任何输入 x_0，总有一个更小的分支语句 c 与 p 表现相同，其中 c 是等于 $x_0 + 1$ 的常量。因此，无论样例编程任务是什么，$\mathcal{T}_E$ 总是返回 $P_A \coloneqq \{2, 3, \ldots, n+1\}$ 的一个子集，永远不会枚举到目标程序 p。

当 $\mathbb{I}_t^n$ 中的所有输入都包含在样例编程任务中时，$\mathcal{T}_E$ 合成的分支语句集合是 P_A。此时，P_A 的总大小是 $\Theta(n \log n)$，样例数量是 n，目标程序的大小是 $\Theta(\log n)$，不存在 $\alpha \geq 1, 0 < \beta < 1$ 和 $c > 0$ 使得 $tsize(P_A) \leq c\,(size(p))^{\alpha} n^{\beta}$ 对所有 n 都成立。因此，$\mathcal{T}_E$ 不是奥卡姆分支语句合成器。

此外，在语句集合是 P_A 时，EUSolver 的合一器会用到 P_A 中的所有语句，于是其结果大小不会小于 $tsize(P_A)$，这意外着 EUSolver 也不是一个奥卡姆求解器。

使用例 11 中的构造，我们可以证明如下定理。

定理 15. EUSolver 中的语句合成器 $\mathcal{T}_E$ 不是 $\mathcal{F}_C$ 上的奥卡姆分支语句合成器，且 EUSolver 本身也不是 $\mathcal{F}_C$ 上的奥卡姆求解器。

EUSolver 的合一器 $\mathcal{U}_E$ 的泛化能力与决策树学习算法 ID3 相关。然而，Hancock

等人证明除非 NP = RP，否则不存在多项式时间的对决策树的奥卡姆学习方法[136]。这一结论直接蕴含了如下定理。

定理 16. 除非 NP = RP，否则在 $\mathcal{F}_C$ 上不存在多项式时间奥卡姆合一器。

因为 EUSolver 中的合一器 $\mathcal{U}_E$ 是多项式时间的，所以上述定理表明 $\mathcal{U}_E$ 不太可能是一个奥卡姆合一器。

综上，本节的理论分析表明 EUSolver 本身不是一个奥卡姆求解器。同时，如果我们想要在定理 14 的是指导下设计奥卡姆求解器，那么 EUSolver 中的两个组件都不能被使用，我们需要从头设计奥卡姆语句合成器与合一器。

5.4 PolyGen 中的语句合成器

本节将介绍 PolyGen 中的语句合成器，记作 $\mathcal{T}_{poly}$。

5.4.1 概述

奥卡姆语句合成器的定义限制了语句集合的总大小。因此，如果一个语句合成器同时保证 (1) 合成的语句数量很少，且（2）合成的所有语句中的最大语句很小，那么该语句合成器必然是一个奥卡姆语句合成器。

引理 4. 对于常数 $\alpha_1, \alpha_2 \geq 0, 0 \leq \beta_1, \beta_2 < 1$，其中 $\beta_1 + \beta_2 < 1$，如果存在常数 $c, \gamma > 0$ 使得对于任何条件领域 $\mathbb{D} \in \mathcal{F}_C$，任何目标程序 $p^* \in \mathbb{P}$，以及任何输入集合 $\{I_1, \ldots, I_n\} \subseteq \mathbb{I}$：

1. 以高概率，$\mathcal{T}$ 返回的每个语句的大小都不会超过 $size(p^*)^{\alpha_1} n^{\beta_1}$。

$$\Pr\left[\max\left\{size(p) \;\middle|\; p \in \mathcal{T}\big(T(p^*, I_1, \ldots, I_n)\big)\right\} > c\left(size(p^*)\right)^{\alpha_1} n^{\beta_1} \ln^{\gamma}\left(\frac{1}{\epsilon}\right)\right] \leq \epsilon$$

2. 以高概率，$\mathcal{T}$ 返回的语句总数不会超过 $size(p^*)^{\alpha_2} n^{\beta_2}$。

$$\Pr\left[\left|\mathcal{T}\big(T(p^*, I_1, \ldots, I_n)\big)\right| > c\left(size(p^*)\right)^{\alpha_2} n^{\beta_2} \ln^{\gamma}\left(\frac{1}{\epsilon}\right)\right] \leq \epsilon$$

那么语句合成器 $\mathcal{T}$ 是 $\mathcal{F}_C$ 上的 $(\alpha_1 + \alpha_2, \beta_1 + \beta_2)$-奥卡姆语句合成器。

证明. 令 P 是 $\mathcal{T}$ 的合成结果。通过将两个前条件中的错误率取为 $\frac{1}{2}\epsilon$，可以得到下列公式一定以至少 $1 - \epsilon$ 的概率成立。

$$\max\left\{size(p) \;\middle|\; p \in \mathcal{P}\right\} \leq c\left(size(p^*)\right)^{\alpha_1} n^{\beta_1} \ln^{\gamma}\left(\frac{2}{\epsilon}\right) \bigwedge |P| \leq c\left(size(p^*)\right)^{\alpha_2} n^{\beta_2} \ln^{\gamma}\left(\frac{2}{\epsilon}\right)$$

因为 $tsize(P) \leq |P| \cdot \max\{size(p) \mid p \in P\}$，所以下列不等式至少以 $1-\epsilon$ 的概率成立。

$$\begin{aligned} tsize(P) &\leq \left(c\left(size(p^*)\right)^{\alpha_1} n^{\beta_1} \ln^{\gamma}\left(\frac{2}{\epsilon}\right)\right) \cdot \left(c\left(size(p^*)\right)^{\alpha_2} n^{\beta_2} \ln^{\gamma}\left(\frac{2}{\epsilon}\right)\right) \\ &\leq c'\left(size(p^*)\right)^{\alpha_1+\alpha_2} n^{\beta_1+\beta_2} \ln^{2\gamma}\left(\frac{1}{\epsilon}\right) \end{aligned}$$

其中 c' 是一个足够大的常量。因此，$\mathcal{T}$ 是一个 $(\alpha_1+\alpha_2, \beta_1+\beta_2)$-奥卡姆求解器。 □

注意到，上述引理中条件一的形式类似于奥卡姆求解器的定义。受此启发，本文将 $\mathcal{T}_{poly}$ 设计为了一个元求解器：它以一个语句空间上的奥卡姆求解器 $\mathcal{S}_t$ 为参数，并在满足上述引理中条件二的前提下，将问题分解到单一语句上的样例编程问题，并调用 $\mathcal{S}_t$ 对这些子问题进行求解。

这一过程的挑战在于不同的样例对应着不同的分支语句。$\mathcal{T}_{poly}$ 需要找到足够多的对应于同一分支语句的样例，才能使 $\mathcal{S}_t$ 合成一个目标语句。为了达成这一点，本文利用了如下性质：在目标集合中，一定存在一个语句覆盖了样例集合中的相当一部分。

引理 5. *对于任意样例编程任务 T 以及任意一个能覆盖 T 中所有样例的语句集合 P，P 一定存在一个语句 p 至少满足 $1/|P|$ 比例的样例，即：*

$$\left|\left\{(I \mapsto O) \in T \;\middle|\; [\![p]\!](I) = O\right\}\right| \geq |T|/|P|$$

证明. 令 $t_1, \ldots, t_n$ 表示 P 中的每个分支语句，且令 $w_1, \ldots, w_n$ 表示 t_i 满足的样例数量。因为 P 覆盖了 T 中的全部样例，所以 $\sum_i w_i$ 一定不小于 $|T|$。基于这一点，可以得到不等式 $\max w_i \geq |T|/|P|$。 □

给定样例编程任务 T，令其目标语句集合为 P^*，且 $t^* \in P^*$ 是其中覆盖最多样例的分支语句。根据上述引理，如果我们从 T 中随机采样 n_t 个样例，则语句 t^* 与所有随机样例一致的概率至少为 $|P^*|^{-n_t}$。利用这一保障，$\mathcal{T}_{poly}$ 会重复地从 T 中采样一小组随机样例，并在每组样例上调用 $\mathcal{S}_t$：根据 $\mathcal{S}_t$ 的可泛化性保障，当样例数量与采样轮数足够大时，$\mathcal{T}_{poly}$ 将以高概率找到一个在语义上与 t^* 相似的语句。

而对于引理 4 中的第二个条件，$\mathcal{T}_{poly}$ 假设分支语句的大小存在一个上界 k，并只考虑合成语句数量不超过 k 的集合。$\mathcal{T}_{poly}$ 保证在 k 超过一个适当的阈值时，能够以高概率找到一个满足所有样例的语句集合。在这一保证下，$\mathcal{T}_{poly}$ 会按照从小到大的顺序迭代地尝试所有 k，直到合成成功，此时的语句数量不会超过阈值太多，一定满足引理 4 的要求。

5.4.2 合成算法

算法 2 展示了 $\mathcal{T}_{poly}$ 的伪代码。除了一个样例编程任务，它还需要一个语句空间上的奥卡姆求解器 $\mathcal{S}_t$ 与一个常数 c，用于配置迭代过程中的阈值。此外，我们假设 $\mathcal{S}_t$ 能

Algorithm 2: PolyGen 中的语句合成器 $\mathcal{T}_{poly}$。

Input: 一个样例编程任务 T，语句空间 $\mathbb{P}_t$ 上的 (α, β)-奥卡姆求解器 $\mathcal{S}_t$ 和一个常数 c。
Output: 一个能覆盖所有样例的语句集合 P。

```
1  Function GetCandidatePrograms(examples, k, n_t, s):
2      result ← {};
3      foreach turn ∈ [1, n_t × k^{n_t}] do
4          从 examples 中均匀且独立地抽样 n_t 个示例 e_1, ..., e_{n_t};
5          p ← S_t(e_1, ..., e_{n_t});
6          if p ≠ ⊥ ∧ |Covered(p, examples)| ≥ |examples|/k ∧ size(p) ≤ c s^α n_t^β then
7              result ← result ∪ {p};
8          end
9      end
10     return result;
11 Function Search(examples, k, n_t, s):
12     if |examples| = 0 then return {};
13     if k = 0 或 (examples, k) 已被访问过 then return ⊥;
14     foreach p ∈ GetCandidatePrograms(examples, k, n_t, s) do
15         searchResult ← Search(examples − Covered(p, examples), k − 1, n_t, s);
16         if searchResult ≠ ⊥ then return {p} ∪ searchResult;
17     end
18     return ⊥;
19 s ← 1;
20 while True do
21     n_l ← c s^{α/(1−β)};  k_l ← c s ln |T|;
22     foreach (k, n_t) ∈ [1, k_l] × [1, n_l] do
23         if (k, n_t) 之前未访问过 then
24             P ← Search(T, k, n_t, s);
25             if P ≠ ⊥ then return P;
26         end
27     end
28     s ← s + 1;
29 end
```

处理无解的样例编程问题，并在无解时返回 $\perp$。

$\mathcal{T}_{poly}$ 算法由三部分组成。第一部分是函数 `GetCandidatePrograms`（缩写为 `Get`），它实现了前文讨论的随机采样过程。`Get` 接收四个输入：*examples* 是一组输入输出样例，k 是语句数量的上限，n_t 表示每次采样的样例数量，而 s 是限制每个语句大小的一个上界。在引理 5 的指导下，`Get` 会返回一组至少覆盖 k^{-1} 比例样例的语句。

`Get` 的实现是一个重复采样过程（第 3 行）。在每一轮中，它会随机采样 n_t 个样例（第 4 行），并调用求解器 $\mathcal{S}_t$ 从这些样例中合成一个候选语句（第 5 行）。`Get` 会收集所有的有效结果（第 7 行）并将它们返回给 `Search`（第 10 行）。值得注意的是，`Get` 只考虑那些大小最多为 $cs^{\alpha}n_t^{\beta}$ 的语句（第 6 行）。

这个上界来自奥卡姆求解器的定义（定义13）：当所有随机样例都对应于同一个目标语句，且该语句大小至多为 s 时，$\mathcal{S}_t$ 的合成结果以高概率不会超过 $cs^{\alpha}n_t^{\beta}$。然而，随

机样例也可能恰好对应某个不符合预期的语句，此时，$\mathcal{S}_t$ 的合成结果可能会比目标语句大得多。为了排除这些情况，$\mathcal{T}_{poly}$ 对语句大小设置了限制，并拒绝那些过大的程序。

算法 2 的第二部分是回溯搜索，即函数 Search （第 11-18 行）。给定一组输入输出样例 *examples* 和大小限制 k，Search 的搜索目标是一个至多包含 k 个语句且覆盖所有样例的集合。Search 首先调用函数 Get 以获取一组可能的语句（第 14 行），然后递归地尝试每一个语句，直到找到一个合法的集合（第 15-16 行）。

$\mathcal{T}_{poly}$ 的第三部分是一个迭代过程（第 19-29 行），它通过迭代的方式为参数 k, n_t 与 s 找到一个合适的值。在每一轮中，$\mathcal{T}_{poly}$ 考虑目标语句的数量及其大小都是 $O(s)$ 的情况，并按照如下方式选择样例数量 n_t 与语句数量 k。

- 根据定理12，当目标语句的大小为 $O(s)$ 时，$\mathcal{S}_t$ 需要 $\Omega\left(s^{\alpha/(1-\beta)}\right)$ 个样例来保证其准确性。因此，$\mathcal{T}_{poly}$ 将 n_t 的上界设为 $cs^{\alpha/(1-\beta)}$。
- 同样根据定理12，在给定 n_t 个样例的情况下，$\mathcal{S}_t$ 只能保证找到一个与目标语句相似的语句，它仍可能在常数比例的样例上产生与目标语句不同的结果。最坏情况下，$\mathcal{T}_{poly}$ 需要使用 $O(\ln |T|)$ 倍的语句才能覆盖 T 中的所有样例。于是，$\mathcal{T}_{poly}$ 将 k 的上界设为 $cs \ln |T|$。

由于 Get 和 Search 的时间开销会随着 k 和 n_t 的增加而迅速增长，所以 $\mathcal{T}_{poly}$ 会从小到大尝试 k 和 n_t 的所有选项（第 22-27 行），而不是直接使用最大可能的 k 和 n_t。

5.4.3 语句合成器的性质

最后，本节讨论 $\mathcal{T}_{poly}$ 的理论性质。首先，根据第 5.4.1 节中的理论分析，当 $\mathcal{S}_t$ 是语句空间上的奥卡姆求解器时，$\mathcal{T}_{poly}$ 一定是一个奥卡姆语句合成器。

定理 17. *给定分支领域形成的族 $\mathcal{F}_C$，令 $\mathcal{F}_T$ 为这些领域中分支语句对应的子领域形成的族。当 $\mathcal{S}_t$ 是 $\mathcal{F}_T$ 上的 (α, β)-奥卡姆求解器时，对于任意的 $\alpha' > \alpha, \beta < \beta' < 1$，$\mathcal{T}_{poly}$ 都是 $\mathcal{F}_C$ 上的 $(\alpha' + 1, \beta')$-奥卡姆语句合成器。*

证明. 令 s^* 为变量 s（第 19 行）在 $\mathcal{T}_{poly}$ 返回时的取值。令 p^* 表示目标程序，P^* 表示 p^* 中使用的语句集合，n 表示样例数量， P 表示 $\mathcal{T}_{poly}$ 的合成结果。根据算法 2，P 的总大小受到 s^* 的限制，如下所示：

$$tsize(P) \le |P| \cdot \max_{p \in P} size(p) \le cs^* \ln n \cdot c(s^*)^{\alpha} n^{\beta} \le c'(s^*)^{\alpha+1} n^{\beta''}$$

其中 β'' 是任意一个比 β 大的常量，而 c' 是一个足够大的常量。

给定样例列表 T' 与参数 k, n_t, s，我们称函数调用 Search(T', k, n_t, s) 是合法的当如下三个随机事件同时满足。

- $\mathcal{E}_1$：至少存在一轮**合法**采样。令 $t^* \in P^*$ 是所有目标语句中满足 T^* 内最多样例的语句。我们称一轮采样是合法的当且仅当 t^* 满足所有采样得到的样例。
- $\mathcal{E}_2$: 不等式 $size(t_0) \leq cs^{\alpha} n_t^{\beta}$ 成立，其中 t_0 是 $\mathcal{S}_t$ 在第一轮合法采样中的合成结果。
- $\mathcal{E}_3$: 在 T' 内所有被 t^* 满足的样例中，t_0 至少满足了一半的比例。

考虑 Search 的一条递归路径 RC，其中每一步的样例列表为 $T'_0 = T, \ldots, T'_{n_c} = \varnothing$。

- **断言:** 当 RC 中的所有函数调用均合法时，n_c 最多为 $2|P^*| \ln |T| + 1$。
- **证明:** 根据 t^* 的定义, 它在 $|T_i|$ 中满足的样例数量至少为 $|T_i|/|P^*|$。此时，$\mathcal{E}_3$ 的成立蕴含了不等式 $|T_{i+1}| \leq \left(1 - 1/(2|P^*|)\right)|T_i|$。因为样例集合的大小一定为整数，所以可以得到 $n_c \leq 2|P^*| \ln |T| + 1$。

假设目前 s 的取值已经大于 $c_1 size(p^*)^{\alpha'} n^{\beta'}$，其中 α' 是一个任意大于 1 的常量，β' 是一个任意大于 0 的常量，而 c_1 是一个足够大的常量。同时假设第 22 行中 k 与 n_t 的取值分别为 $cs \ln n$ 和 $cs^{\alpha/(1-\beta)}$，即达到了各自的上限。此时因为 $k_t = cs \log n > 2|P^*| \ln n$，所以上述递归路径满足阈值 k_t 的限制。这意味着 $\mathfrak{T}_{poly}$ 在当前 s 的取值下合成成功的概率不会低于 RC 中所有函数调用均合法的概率。

接下来让我们依次考虑合法函数调用定义中的三个随机事件。

- 对于 $\mathcal{E}_1$，因为 t^* 是覆盖最多样例的目标语句，所以它满足一个随机样例的概率不会小于 $1/|P^*|$。因此，一轮采样是合法的概率不会低于 $|P^*|^{-n_t}$。进一步地，因为每次函数调用都会进行 $n_t \cdot k^{n_t}$ 次随机采样，所以在 n_t 足够大的时候，$\mathcal{E}_1$ 发生的概率一定满足如下所示的不等式。

$$\Pr[\mathcal{E}_1] \geq 1 - \left(1 - |P^*|^{-n_t}\right)^{n_t k^{n_t}} \geq 1 - \exp(-n_t) > 1 - \frac{1}{12 size(p^*) \ln n}$$

- 对于 $\mathcal{E}_2$，因为 $\mathcal{S}_t$ 是一个 (α, β)-奥卡姆求解器，所以一定存在常量 c_2 与 γ 满足：

$$\forall \epsilon \in \left(0, \frac{1}{2}\right), \Pr\left[size(t_0) > c_2 (size(p^*))^{\alpha} n_t^{\beta} \ln^{\gamma}\left(\frac{1}{\epsilon}\right)\right] < \epsilon$$

因此，当 s 中的常量系数 c_1 足够大时，如下不等式至少以 $1 - 1/(12 size(p^*) \ln n))$ 的概率成立，它蕴含了 $\mathcal{E}_2$ 的发生。

$$size(t_0) \leq c_2 (size(p^*))^{\alpha} n_t^{\beta} \ln^{\gamma}(12 size(p^*) \ln n) < cs^{\alpha} n_t^{\beta}$$

- 对于 $\mathcal{E}_3$，令 $\mathbb{I}'$ 为 T' 中所有被 t^* 满足的样例的输入，分布 D 是 $\mathbb{I}'$ 上的均匀分布。因为 $\mathcal{S}_t$ 是一个 (α, β)-奥卡姆合成器，所以根据定理 12，当如下不等式成立时，$\mathcal{E}_3$ 至少以 $1 - 1/(12 size(p^*) \ln n)$ 的概率发生，其中 c_3 是一个固定的常量。

$$n_t > c_3 \left(\frac{1}{2} \ln\left(24 size(p^*) \ln n\right) + \left(\frac{(size(p^*))^{\alpha} \ln^{\gamma}\left(24 size(p^*) \ln n\right)}{\epsilon} \right)^{1/(1-\beta)} \right)$$

上述不等式在 n_t 中的常量系数 c_1 足够大时一定成立。

令随机事件 $\mathcal{E}_i^{RC}$ 表示 RC 中第 i 次调用是合法的情况，且令随机事件 $\mathcal{E}^{RC}$ 表示 RC 中的所有调用都是合法的情况。那么根据上述分析结果，可以得到：

$$\begin{aligned}\Pr\left[\neg\mathcal{E}^{RC}\right] &\leq \sum_{i=1}^{n_c}\Pr\left[\mathcal{E}_i^{RC}\right] \leq (2|P^*|\ln n+1)\big(\Pr[\neg\mathcal{E}_1]+\Pr[\neg\mathcal{E}_2]+\Pr[\neg\mathcal{E}_3]\big)\\ &\leq (2size(p^*)\ln n+1)\cdot\frac{1}{4size(p^*)\ln n}\leq\frac{3}{4}\end{aligned}$$

这一结果表明，当 s 不小于 $c_1 size(p^*)^{\alpha'}n^{\beta'}$ 时，$\mathcal{T}_{poly}$ 的每轮调用都有至少 1/4 的概率合成成功。因此，对于任意的 $\epsilon\in(0,1)$，下列不等式一定成立。

$$\begin{aligned}&\Pr\left[s^*\leq c_1 size(p^*)^{\alpha'}n^{\beta'}+c_4\ln\left(\frac{1}{\epsilon}\right)\right]<\epsilon\\ \Longrightarrow\ &\Pr\left[tsize(P^*)\leq c_5 size(p^*)^{\alpha'(1+\alpha)}n^{\beta''+(1+\alpha)\beta'}\ln\left(\frac{1}{\epsilon}\right)\right]<\epsilon\end{aligned}$$

其中 c_r 与 c_5 是足够大的常量。注意到因为 α' 是一个任意大于 1 的常量，且 β' 与 β'' 均为任意大于 0 的常量。所以对于任意的 $\alpha^*>1+\alpha,\beta^*>\beta$，$\mathcal{T}_{poly}$ 都是一个 (α^*,β^*)-奥卡姆语句合成器. □

接着讨论 $\mathcal{T}_{poly}$ 的时间开销。它以高概率只会调用多项式次 `Search`，但 `Search` 的开销并不是多项式的。根据算法 2，在递归树上深度为 i 的 `Search` 调用会进行 $n_t(k-i)^{n_t}$ 次随机采样。在最坏情况下，$\mathcal{S}_t$ 会为所有的采样结果合成候选语句，且这些语句两两不同。此时，`Search` 将递归到 $n_t(k-i)^{n_t}$ 个不同的分支，导致对于每个 (n_t,k)，求解器 $\mathcal{S}_t$ 在最坏情况下都会被调用 $n_t^k(k!)^{n_t}$ 次。

然而在实践中，本文发现 $\mathcal{T}_{poly}$ 的效率通常比最坏情况好得多，其主要原因如下：

- 当随机样例不对应任何目标语句时，$\mathcal{S}_t$ 通常会合成失败而不是返回一个错误的语句，因为语句空间 $\mathbb{P}_t$ 的表达能力通常有限。
- 而即便有一些语句被错误地合成，它们也很难满足语句大小和覆盖样例数量的要求（算法 2 中的第 6 行）。

在理想情况下，如果 `Get` 永远只返回目标程序中的分支语句，那么 $\mathcal{S}_t$ 最多只会被调用 $n_t 2^k k^{n_t}$ 次，远小于其理论上界。

5.5 PolyGen 中的合一器

本节将介绍 PolyGen 中的语句合成器，记作 $\mathcal{U}_{poly}$。

5.5.1 概述

$\mathcal{U}_{poly}$ 按照决策列表[133] 的形式合成分支条件并构建结果程序。给定语句集合 $P \coloneqq \{p_1, \ldots, p_m\}$，$\mathcal{U}_{poly}$ 会按照如下形式将它们组合。

$$\texttt{if}\ (c_1)\ \texttt{then}\ p_1\ \texttt{else if}\ (c_2)\ \texttt{then}\ p_2\ \texttt{else} \ldots \texttt{if}\ (c_{m-1})\ \texttt{then}\ p_{m-1}\ \texttt{else}\ p_m$$

其中 $c_1, \ldots, c_{m-1}$ 是由 $\mathbb{P}_c$ 中的条件组成的析取范式，本文将它们所属的程序空间记作 DNF($\mathbb{P}_c$)。这一空间与另外两个子空间 L($\mathbb{P}_c$) 和 CL($\mathbb{P}_c$) 一起定义，其中文字空间 L($\mathbb{P}_c$) 包含了 $\mathbb{P}_c$ 中的条件及其否定，子句空间 CL($\mathbb{P}_c$) 包含了 L($\mathbb{P}_c$) 中所有子集的合取，而 DNF 空间 DNF($\mathbb{P}_c$) 包包含了 CL($\mathbb{P}_c$) 中所有子集的析取。

本文使用符号 $\langle \mathbb{P}_t, \mathbb{P}_c \rangle_{\text{DL}}$ 表示所有由 $\mathbb{P}_t$ 中的语句与 $\mathbb{P}_c$ 中的条件组成的决策列表集合。显然，决策列表空间 $\langle \mathbb{P}_t, \mathbb{P}_c \rangle_{\text{DL}}$ 是分支空间 $\langle \mathbb{P}_t, \mathbb{P}_c \rangle$ 的子空间。下列引理表明，我们可以在设计奥卡姆合一器时只考虑子空间 $\langle \mathbb{P}_t, \mathbb{P}_c \rangle_{\text{DL}}$，因为对于 $\langle \mathbb{P}_t, \mathbb{P}_c \rangle$ 中的任何程序，$\langle \mathbb{P}_t, \mathbb{P}_c \rangle_{\text{DL}}$ 中一定存在一个语义等价且大小接近的程序。

引理 6. *对于任何条件领域 $\mathbb{D}$ 和任何程序 $p \in \langle \mathbb{P}_t, \mathbb{P}_c \rangle$，存在一个程序 $p' \in \langle \mathbb{P}_t, \mathbb{P}_c \rangle_{\text{DL}}$，使得 p' 在 $\mathbb{I}$ 上与 p 语义等价，且 $size(p') \leq 2size(p)^2$。*

证明. 令 $P \coloneqq \{t_1, \ldots, t_m\}$ 表示程序 p 中的语句集合。考虑符合如下形式的程序 p'，其中决策列表的第 i 个语句恰好是 p 使用的语句 t_i。

$$\texttt{if}\ (c_1)\ \texttt{then}\ t_1\ \texttt{else if}\ (c_2)\ \texttt{then}\ t_2\ \texttt{else} \ldots \texttt{if}\ (c_{m-1})\ \texttt{then}\ t_{m-1}\ \texttt{else}\ t_m$$

对于 p 中的任意一个分支，定义它的**分支路径**为序列 $(pc_1, k_1), \ldots, (pc_n, k_n)$，其中 $pc_i \in \mathbb{P}_c$ 表示限制该分支的每个条件，且 $k_i \in \{0, 1\}$ 表示当前分支属于条件的哪种情况，0 表示满足而 1 表示不满足。构造如下所示的函数 φ，它将每条分支路径映射到了一个子句，如下所示：

$$\varphi((pc_1, k_1), \ldots, (pc_n, k_n)) \quad \coloneqq \quad \left(\texttt{and}_{k_i=0}\ pc_i\right)\ \texttt{and}\ \left(\texttt{and}_{k_i=1}\ (\texttt{not}\ \ pc_i)\right)$$

使用该函数，我们构造决策列表 p' 中的每个条件 c_i。对于分治语句 t_i，令集合 X_i 包含 t_i 在 p 中所有出现位置的分支路径。那么我们将条件 c_i 构造为析取范式 $\texttt{or}_{x \in X}\ \varphi(x)$。

显然，p' 与 p 语义等价且属于程序空间 $\langle \mathbb{P}_t, \mathbb{P}_c \rangle_{DL}$。所以接下来只需要考虑 p' 与 p 的大小关系。令 $c'_1, \ldots, c'_{n_c}$ 为 p 中所有的分支条件，l_i 与 r_i 分别表示第 i 个条件两侧分支语句的数量，N 表示语法中产生式的数量。那么 $size(p')$ 可以通过如下方式计算：

- p' 中所有分支条件的总大小不超过如下公式:

$$\sum_{i=1}^{n_c} \left(l_i(size(c'_i) + \lceil \log_2 N \rceil) + r_i(size(c'_i) + 2\lceil \log_2 N \rceil)\right) - (m-1)\lceil \log_2 N \rceil$$

Algorithm 3: $\mathcal{U}_{poly}$ 的算法框架。

Input: 语句集合 $P \coloneqq \{p_1, \ldots, p_m\}$，样例编程任务 T 和 DNF 合成器 $\mathcal{C}$。
Output: $\langle P, \mathbb{P}_c \rangle_{\mathrm{DL}}$ 中的一个满足所有样例的程序。

1 $conditionList \leftarrow \{\}$;
2 **for** $i \leftarrow 1; i < m; i \leftarrow i + 1$ **do**
3 $\quad examples \leftarrow \{I \mapsto \texttt{false} \mid (I \mapsto O) \in T - \texttt{Covered}(p_i, T)\}$;
4 $\quad examples \leftarrow \left\{I \mapsto \texttt{true} \mid (I \mapsto O) \in \texttt{Covered}(p_i, T) - \bigcup_{j=i+1}^{m} \texttt{Covered}(p_j, T)\right\}$;
5 $\quad c_i \leftarrow \mathcal{C}(examples)$;
6 $\quad conditionList.\texttt{Append}(c_i)$; $T \leftarrow \{(I \mapsto O) \in T \mid \neg[\![c_i]\!](I)\}$;
7 **end**
8 $result \leftarrow p_m$;
9 **for** $i \leftarrow m - 1; i > 0; i \leftarrow i - 1$ **do**
10 $\quad result \leftarrow \big(\texttt{if } conditionList_i \texttt{ then } p_i \texttt{ else } result\big)$;
11 **end**
12 **return** *result*;

- 分支操作符的总大小是 $(m-1)\lceil \log_2 N \rceil$，分支语句的总大小是 $\sum_{i=1}^{m} size(t_i)$。

于是，我们可以进行如下推导：

$$
\begin{aligned}
size(p') &\leq \sum_{i=1}^{n_c} \Big(l_i (size(c_i') + \lceil \log_2 N \rceil) + r_i (size(c_i') + 2\lceil \log_2 N \rceil) \Big) \\
&\quad - (m-1)\lceil \log_2 N \rceil + (m-1)\lceil \log_2 N \rceil + \sum_{i=1}^{m} size(t_i) \\
&\leq \left(\sum_{i=1}^{n_c} (l_i + r_i) size(c_i') + \sum_{i=1}^{m} size(t_i) \right) + 2\lceil \log_2 N \rceil \sum_{i-1}^{n_c} (l_i + r_i) \\
&\leq size(p) \left(\sum_{i=1}^{n_c} size(c_i') + \sum_{i=1}^{m} size(t_i) \right) + size(p) \cdot 2n_c \lceil \log_2 N \rceil \\
&\leq 2size(p)^2
\end{aligned}
$$

其中最后一步不等式 $2n_c\lceil \log_2 N \rceil \leq \mathrm{size}(p)$ 是因为 p 中的每个分支条件一定会对程序大小产生至少 $2\lceil \log_2 N \rceil$ 的贡献，包括一个大小不为 0 的条件和一个分支操作符。 □

算法3展示了 $\mathcal{U}_{poly}$ 的算法框架。给定一个大小为 m 的语句集合，它会将合成任务分解为 $m-1$ 个样例编程任务，分别针对每个语句前的条件。它按照顺序考虑每个语句（第 2-7 行），对于当前语句 p_i 与每个剩余的样例 $e \coloneqq (I \mapsto O)$，有三种可能的情况：

- 如果 p_i 不满足 e，那么它的分支条件 c_i 在输入 I 上必须为假（第 3 行）。
- 如果在剩余的语句 $p_i, \ldots, p_n$ 中，p_i 是唯一一个满足 e 的语句，则 c_i 在输入 I 上的值必须为真（第 4 行）。
- 否则，c_i 的值无关紧要，于是 $\mathcal{U}_{poly}$ 将忽略此示例（第 4 行）。如果合成的 c_i 在

输入 I 上为假，则将该样例 e 将留给后续语句满足（第 6 行）。

通过这种方式，$\mathcal{U}_{poly}$ 搜集到了关于 c_i 的输入输出样例，并调用 DNF 求解器 $\mathcal{C}$ 来合成 c_i 第 5 行）。最后，$\mathcal{U}_{poly}$ 会用这些语句和条件构建一个决策列表（第 8-11 行）。

不难证明当 DNF 求解器 $\mathcal{C}$ 是 DNF($\mathbb{P}_c$) 上的奥卡姆求解器时，合一器 $\mathcal{U}_{poly}$ 一定是一个奥卡姆合一器。特殊地，当 $\mathcal{C}$ 不包含任何随机性时，$\mathcal{U}_{poly}$ 可以具有更好的参数。

引理 7. 给定分支领域形成的族 $\mathcal{F}_C$，令 $\mathcal{F}_{\text{DNF}}$ 为这些领域中的析取范式对应的子领域形成的族。当 $\mathcal{C}$ 是 $\mathcal{F}_{\text{DNF}}$ 上的 (α,β)-奥卡姆求解器时，$\mathcal{U}_{poly}$ 一定是 $\mathcal{F}_C$ 上的 $(4\alpha',\beta)$-奥卡姆合一器，其中 α' 是任意大于 α 的常数。

特殊地，当 $\mathcal{C}$ 不包含任何随机性时，$\mathcal{U}_{poly}$ 也是一个 $\mathcal{F}_C$ 上的 $(4\alpha,\beta)$-奥卡姆合一器。

证明. 令 $P \coloneqq \{t_1,\dots,t_m\}$ 表示给定的语句集合，$p^* \in \langle P,\mathbb{P}_c\rangle$ 表示目标程序，同时令 s 表示 $\max(tsize(P), size(p^*))$。通过与证明引理 6 时相同的方式，构造 $c_1^*,\dots,c_{m-1}^*$ 为语句 t_i 在决策列表中的条件。根据构造过程的细节，我们还能得到如下性质：

- $\sum_{i=1}^{m-1} size(c_i^*) \le O(s^2)$。
- $c_1^*,\dots,c_{m-1}^*$ 不会互相重叠，即 $\forall 1 < i < j < m, \forall I \in \mathbb{I}, \neg([\![c_i^*]\!](I) \wedge [\![c_j^*]\!](I))$。

令 T_i 表示所有样例中被语句 t_i 满足但不被 $t_{i+1},\dots,t_m$ 满足的子集。根据算法 3，用于合成 c_i^* 的正例一定是 T_i 的子集。注意在该算法中，对于先后两个语句 $t_i, t_j (i<j)$ 和一个在目标程序中对应于 t_i 的样例，如果该样例能同时被两个语句满足，它将有可能会被交给 t_j 的条件处理。因此，先前根据目标程序构造的条件 c_i^* 可能无法满足 T_i 中的所有正例。于是我们进一步构造了如下的分支条件 c_i' 用于保证满足 T_i 中的所有样例。

$$c_i' \quad \coloneqq \quad c_i^* \ \texttt{or}\ \left(\text{or}_{j=1}^{i-1}\left(\text{or}_{k=1}^{n_j}\left(c_{j,k}^* \ \texttt{and}\ \left(t_i = t_j\right)\right)\right)\right)$$

其中 $c_{i,1}^*,\dots,c_{i,n_i}^*$ 表示 DNF c_i^* 中的所有子句。显然，c_i' 仍然符合 DNF 的形式并且满足 T_i 中的所有样例。因此，在合成第 i 个分支条件时，c_i' 一定是一个满足所有样例的合法程序。与此同时，不难证明所有 c_i' 的总大小不会超过 $O(s^4)$。

令 $c_1,\dots,c_{m-1}$ 表示 $\mathcal{C}$ 的最终合成结果，s_i 表示合法条件 c_i' 的大小。那么当 $\mathcal{C}$ 是一个 (α,β)-奥卡姆求解器时，一定存在常量 c',γ' 满足如下公式。

$$\forall i \in [1, m-1], \quad \Pr\left[size(c_i) > c' s_i^{\alpha} |T|^{\beta} \ln^{\gamma'}\left(\frac{m-1}{\epsilon}\right)\right] \le \frac{\epsilon}{m-1}$$

因此，下列不等式至少以 $1-\epsilon$ 的概率成立。

$$\sum_{i=1}^{m-1} size(c_i) \le c'|T|^{\beta}\ln^{\gamma'}\left(\frac{m-1}{\epsilon}\right)\sum_{i=1}^{m-1} s_i^{\alpha} \le c'|T|^{\beta}\left(\ln(m-1)+\ln\left(\frac{1}{\epsilon}\right)\right)^{\gamma'}\left(\sum_{i=1}^{m-1} s_i\right)^{\alpha}$$

$$\le c''|T|^{\beta} s^{4\alpha}\ln(m-1)^{\gamma'}\ln\left(\frac{1}{\epsilon}\right)^{\gamma'} \le c s^{4\alpha'}|T|^{\beta}\ln\left(\frac{1}{\epsilon}\right)^{\gamma}$$

其中 c'' 与 c 都是足够大的常量，$\gamma = \gamma'$ 且 α' 是一个大于 α 的常量。因为 $\mathcal{U}_{poly}$ 合成结果的大小不超过 $\sum_{i=1}^{m-1} \text{size}(c_i) + s$，所以此时 $\mathcal{U}_{poly}$ 是一个 $(4\alpha', \beta)$-奥卡姆合一器。

此外，当 $\mathcal{C}$ 不包含任何随机性时，c_i 的大小上限可以被改进为如下的不等式：

$$\forall i \in [1, m-1], \quad size(c_i) \leq c' s_i^{\alpha} |T|^{\beta}$$

此时，它们的大小总和满足下列不等式，其中 c 是一个足够大的常量。

$$\sum_{i=1}^{m-1} size(c_i) \leq c' s_i^{\alpha} |T|^{\beta} \leq c' |T|^{\beta} \sum_{i=1}^{m-1} s_i^{\alpha} \leq c' |T|^{\beta} \left(\sum_{i=1}^{m-1} s_i \right)^{\alpha} \leq c s^{4\alpha} |T|^{\beta}$$

这意味着 $\mathcal{U}_{\text{poly}}$ 在 $\mathcal{C}$ 不包含随机性时是一个 $(4\alpha, \beta)$-奥卡姆合一器. □

根据引理7，唯一剩下的问题是为析取范式族 $\mathcal{F}_{\text{DNF}}$ 设计一个奥卡姆求解器。本文将在接下来的两个小节中逐步构建这样一个条件求解器。

为描述方便，本文先引入一些必要的符号：

- 不加区分地，本文将 DNF d 视作由一系列子句形成的集合，将子句 c 视为由一系列文字形成的集合。
- 给定输入空间 $\mathbb{I}$ 与条件 p，令 $P(\mathbb{I}, p)$ 表示 p 为真的所有输入形成的集合，$N(\mathbb{I}, p)$ 表示 p 为假的所有输入形成的集合。
- 对于样例编程任务 T，令 $\mathbb{I}(T)$、$\mathbb{I}_P(T)$ 和 $\mathbb{I}_N(T)$ 分别表示 T 中的所有输入、正例上的所有输入与负例上的所有输入。

给定关于条件的样例编程任务 T，一个合法的条件 p 应当满足如下两个要求。

- p 在 T 的所有正例上取值为真，即 $\mathbb{I}_P(T) \subseteq P(\mathbb{I}(T), p)$。
- p 在 T 的所有负例上取值为假，$\mathbb{I}_N(T) \subseteq N(\mathbb{I}(T), p)$。

5.5.2 子句合成器

因为析取范式是一系列子句的析取，所以我们先考虑为子句设计一个奥卡姆求解器 $\mathcal{C}_{\text{CL}}$，其伪代码如算法4所示。$\mathcal{C}_{\text{CL}}$ 从合法条件的第一个要求出发：$\mathbb{I}_P(T) \subseteq P(\mathbb{I}(T), c)$。根据子句的定义，对于任何子句 c 和任何文字 $l \in c$，子句 c 为真的输入集合 $P(\mathbb{I}(T), c)$ 一定是文字 l 为真的输入集合 $P(\mathbb{I}(T), l)$ 的子集。这意味着只有那些满足所有正例的文字才能被用于结果子句。$\mathcal{C}_{\text{CL}}$ 将所有这些文字收集为子句 c_u（第 8 行）。然后，c_u 的子集正是那些满足第一个条件的子句。

剩余的任务是找到 c_u 的一个子集 c^* 以满足第二个要求，即 $\mathbb{I}_N(T) \subseteq N(\mathbb{I}(T), c^*)$。同时，为了保证 $\mathcal{C}_{\text{CL}}$ 是奥卡姆求解器，c^* 的程序大小应尽可能小。这些要求构成了**加权集合覆盖**的一个实例：$\mathcal{C}_{\text{CL}}$ 需要从 $\{N(\mathbb{I}(T), l) \mid l \in c^*\}$ 中选择一些集合来覆盖 $\mathbb{I}_N(T)$。

Algorithm 4: 子句求解器 $\mathcal{C}_{\text{CL}}$。

Input: 条件空间 $\mathbb{P}_c$ 和样例编程任务 T。
Output: $\text{CL}(\mathbb{P}_c)$ 中的一个满足所有样例的程序或 $\perp$。

```
1  Function SimplifyClause(c_u, T):
2      remNeg ← 𝕀_N(T);  c* ← ∅;
3      while remNeg ≠ ∅ do
4          l* ← arg max_{l∈c_u} ( |N(𝕀(T), l) ∩ remNeg| / size(l) );
5          c* ← c* ∪ {l*};  remNeg ← remNeg − N(𝕀(T), l);
6      end
7      return c*.
8  c_u ← {l ∈ L(ℙ_c) | 𝕀_P(T) ⊆ P(𝕀(T), l)};
9  if 𝕀_N(T) ⊈ N(𝕀(T), c_u) then return ⊥;
10 return SimplifyClause(c_u, T);
```

于是，本文在 $\mathcal{C}_{\text{CL}}$ 中使用了一个加权集合覆盖问题上的标准贪心算法（第 1-7 行）。它具有多项式的时间复杂度，并保证找到一个至多比最优解差 $\ln|c^*|$ 倍的解。在运行过程中，$\mathcal{C}_{\text{CL}}$ 会维护集合 *remNeg*，表示尚未被覆盖的所有负例（第 3 行），并总是选择在单位大小内能覆盖最多负例的文字（第 5 行），将它加入到结果中（第 6 行）。

加权集合覆盖中贪心算法的近似比保证了 $\mathcal{C}_{\text{CL}}$ 是一个奥卡姆求解器，如下所示。

引理 8. *给定分支领域形成的族 $\mathcal{F}_C$，令 $\mathcal{F}_{CL}$ 为这些领域中的子句对应的子领域形成的族。对于任何的 $0<\beta<1$，$\mathcal{C}_{CL}$ 是 $\mathcal{F}_{CL}$ 上的 $(1,\beta)$-奥卡姆求解器。*

证明. 该引理直接由加权集合覆盖问题上贪心算法的近似比导出。 □

5.5.3　析取范式的条件合成

最后，本文在 $\mathcal{C}_{\text{CL}}$ 的基础上，为析取范式实现了一个奥卡姆求解器 $\mathcal{C}$。根据运算符 `or` 的语义可以得到一个在形式上类似于引理 5 的结论。

引理 9. *设 T 是一个样例编程任务，d 是一个满足 T 中所有样例的 DNF 公式，则：*

- *d 中的所有子句在 T 的所有负例上必须为假，即 $\forall c\in d, \mathbb{I}_N(T)\subseteq N(\mathbb{I}(T),c)$。*
- *d 中存在一个子句，该子句至少在 T 中 $1/|d|$ 比例的正例上为真，即：*

$$\exists c\in d, |P(\mathbb{I}(T),c)| \geq |d|^{-1}|\mathbb{I}_P(T)|$$

根据这个引理，我们按照与语句合成器 $\mathcal{T}_{poly}$ 类似的方式设计了 $\mathcal{C}$，如算法 5 所示。与 $\mathcal{T}_{poly}$ 中的对应部分相比，该算法有两个主要区别：

- `GetPossibleClause` 的目标是找到一组子句，这些子句在所有负例上都为假，并在至少 k^{-1} 比例的正例上为真（第 4 行）。相应地，搜索函数 `Search` 在每次递归中只排除被覆盖的正例，并保留所有反例（第 6 行）。

Algorithm 5: 析取范式求解器 $\mathcal{C}$

Input: 条件空间 $\mathbb{P}_c$，样例编程任务 T 和常数 c_0。
Output: DNF($\mathbb{P}_c$) 中的一个满足所有样例的 DNF 公式。

```
1  Function Search(literals, T, k, s):
2      if |𝕀_P(T)| = 0 then return {};
3      if k = 0 或状态 (literals, T, k) 已被访问过 then return ⊤;
4      foreach c ∈ GetPossibleClause(literals, T, k) do
5          if size(c) > c_0 s ln|T| then continue;
6          searchResult ← Search(literals, T − {(I ↦ true) ∈ T | ⟦c⟧(I) = true}, k − 1, s);
7          if searchResult ≠ ⊥ then return {c} ∪ searchResult;
8      end
9      return ⊤;
10 s ← 1;
11 while True do
12     k_l ← c_0 s;
13     foreach (k, s′) ∈ [1, k_l] × [1, s] do
14         if (k, s′) 之前未被访问 then
15             P_c ← GetConditionsWithSizeBound(ℙ_c, s′);
16             d ← Search(L(P_c), T, k, s);
17             if d ≠ ⊥ then return d;
18         end
19     end
20     s ← s + 1;
21 end
```

- 因为文字空间 L($\mathcal{P}_c$) 可能很大，为了提高子句合成器 $\mathcal{C}_{\text{CL}}$ 的效率，$\mathcal{C}$ 会迭代地选择参数 s'（第 13 行），并在每次迭代中只考虑大小不超过 s' 的文字。

尽管 `GetPossibleClause`（缩写为 `Get`）可以按照与 $\mathcal{T}_{poly}$ 相似的方法实现成一个随机采样过程，但因为在 $\mathcal{C}$ 中子句合成器 $\mathcal{C}_{\text{CL}}$ 是已知且固定的，所以本文针对 $\mathcal{C}_{\text{CL}}$ 对原先的采样算法作了进一步优化。

回顾算法4，$\mathcal{C}_{\text{CL}}$ 会分两步合成子句。它首先找到所有可用文字的集合 c_u，然后贪心对它化简。为了找到一个对 $\mathcal{C}$ 有用的子句，集合 c_u 应该满足所有负例并至少满足 k^{-1} 比例的正例。在实践中，满足这个条件的不同 c_u 数量通常很小，于是本文将 `Get` 实现为先找到所有可能的 c_u，再使用函数 `SimplifyClause` 对它们一一化简。

为了形式化这一过程，本文引入了**代表性子句**的概念。给定输入空间 $\mathbb{I}$ 和一组文字 L，定义子句空间 CL(L) 上的关系 $\sim_{\mathbb{I}}$ 为子句在输入空间 $\mathbb{I}$ 上的语义等价关系：

$$c_1 \sim_{\mathbb{I}} c_2 \iff \forall I \in \mathbb{I}, [\![c_1]\!](I) = [\![c_2]\!](I)$$

这一关系将 CL(L) 划分成了若干个等价类。本文将对应于子句 c 的类记作 $[c]_{\mathbb{I}}$。容易证明，每个类 $[c]_{\mathbb{I}}$ 包含一个全局最大子句，该子句是等价类中所有其他子句的并集，即 $\exists c' \in [c]_{\mathbb{I}}, c' = \left(\cup_x x \text{ for } x \in [c]_{\mathbb{I}}\right)$。在这一结论的基础上，本文定义**代表性子句**来表示

Algorithm 6: $\mathcal{C}$ 中函数 `GetPossibleClause()` 的实现。

Input: 一组文字 L，样例编程任务 T 和上界 k。
Output: 根据引理9的一组可能子句。

```
1  result ← {∅};
2  foreach l ∈ L do
3      foreach c ∈ result do
4          if |P(𝕀_P(T), c ∪ {l})| ≥ k^{-1} |𝕀_P(T)| then result.Insert(c ∪ {l});
5      end
6      foreach c ∈ result do
7          if ∃c′ ∈ result, (P(𝕀_P(T), c) = P(𝕀_P(T), c′) ∧ c ⊂ c′) then result.Delete(c);
8      end
9  end
10 return {SimplifyClause(c) | c ∈ result ∧ 𝕀_N(T) ⊆ N(𝕀(T), c)}
```

$\mathcal{C}_{\text{CL}}$ 的运行过程中所有可能的 c_u 形成的集合。

定义 20 (代表性子句)**.** 给定输入空间 $\mathbb{I}$，大小限制 k 和一组文字 L，代表集 $R(\mathbb{I}, k, L) \subseteq \text{CL}(L)$ 包含了所有满足以下条件的子句 c：*(1)* c 是 $[c]_{\mathbb{I}}$ 中最大的子句，*(2)* c 在至少 k^{-1} 比例的输入上取值为真，即 $|P(\mathbb{I}, c)| \geq k^{-1}|\mathbb{I}|$。

根据这一定义，$R(\mathbb{I}_P(T), k, L)$ 是当样例编程任务为 T、大小限制为 k 且可用文字集为 L 时的可能 c_u 集合。本文对 `Get` 的实现如算法6所示。它时刻维护集合 *result*，表示对应当前文字集合 L' 的 $R(\mathbb{I}_P(T), k, L')$。因为 L' 最初为空，所以 *result* 最开始只包含一个空子句（第 1 行）。接着，`Get` 会按顺序将所有文字插入到 L' 中并相应地更新 *result*（第 2-9 行）。在处理所有文字后，*result* 一定与 $R(\mathbb{I}_P(T), k, literals)$ 相等，于是 `Get` 会简化并返回其中的所有合法子句（第 10 行）。

与语句合成器 $\mathcal{T}_{poly}$，不难证明 DNF 求解器 $\mathcal{C}$ 是一个奥卡姆求解器，如下所示。

引理 10. 给定分支领域形成的族 $\mathcal{F}_C$，令 $\mathcal{F}_{\text{DNF}}$ 为这些领域中的析取范式对应的子领域形成的族。对于任何 $0 < \beta < 1$，$\mathcal{C}$ 是 $\mathcal{F}_{\text{DNF}}$ 上的 $(2, \beta)$-奥卡姆求解器。

证明. 令 s^* 是变量 s（算法 5 中第 10 行）在 $\mathcal{U}_{poly}$ 合成结束时的取值。令 d^* 表示目标的 DNF 公式。当 $s \geq 2size(d^*)$ 且参数 k 与 s' 均与 s 相同时，考虑如下断言：

- **断言:** 在函数调用 `Search`$(literals, T, k, s)$ 中，令 d' 表示 d^* 中子句的一个子集，其中每个子句至少满足 T 中的一个正例。如果 $k \geq |d'|$，那么一定存在一个子句 $c \in$ `Get`$(literals, T, k)$ 满足如下公式：

$$\exists c^* \in d', \qquad P(\mathbb{I}_P(T), c^*) \subseteq P(\mathbb{I}_P(T), c) \wedge \text{size}(p_c(c)) \leq 2s \ln |T|$$

- **证明:** 令 c^* 是 d' 中满足最多正例的子句，那么显然它至少满足 $1/|d'|$ 比例的正例，即 $|P(\mathbb{I}_P(T), c^*)| \geq |d'|^{-1}|\mathbb{I}_p(T)|$。因为 $k \geq d'$，所以 $[c^*]_{\mathbb{I}_P(T)}$ 中的代表性子

句一定被包含在 $R(\mathbb{I}_P(T), k, literals)$ 中，从而一定会被 $\texttt{Get}(literals, T, k)$ 找到。令 c 表示上述代表性子句在简化后的结果。那么根据简化算法的近似比，以下不等式必然成立，从而目标断言得证。

$$\text{size}(p_c(c)) < 2\text{size}(p_c(c^*))(\ln|T| + 1) \leq 2s \ln|T|$$

根据上述断言可知，当 $s > 2size(d^*)$ 时，搜索过程 `Search()` 一定可以合成成功。在此时的合成结果中，最多存在 $O(size(d^*))$ 个被 $\mathcal{C}$ 合成的子句，其中每个子句的大小不超过 $O(size(d^*)\ln|T|)$。因此，合成结果的大小不会超过 $O((size(p_d(d^*)))^2 \ln|T|)$，这表明 $\mathcal{C}$ 是一个 $(2, \beta)$-奥卡姆求解器，其中 β 是区间 $(0, 1)$ 中的任意常量。

□

将该引理与之前的理论结果结合，可以得到 $\mathcal{U}_{poly}$ 是一个奥卡姆合一器，且 PolyGen 是一个分支领域上的奥卡姆合成器。

定理 18. *对于任何 $0 < \beta < 1$，$\mathcal{U}_{poly}$ 是 $\mathcal{F}_C$ 上的 $(8, \beta)$-奥卡姆合一器。*

证明. 该引理是引理 7 与引理 10 的组合。 □

定理 19. *对于任意常数 $\alpha \geq 0, 0 \leq \beta < 1/8$，当 $\mathcal{S}_t$ 是 $T(\mathcal{F}_C)$ 上的 (α, β)-奥卡姆求解器时，PolyGen 是 $\mathcal{F}_C$ 上的一个 (α', β')-奥卡姆求解器，其参数如下所示：*

$$\alpha' > 8(\alpha' + 1) \qquad 8\beta < \beta' < 1$$

证明. 该定理是定理 14、定理 17以及定理 18 的组合。 □

5.6 实验评估

为了检验 PolyGen 的有效性，本文设计了实验来回答以下问题：

- **RQ1**: 相比于现有的 STUN 求解器，PolyGen 的泛化能力和求解效率是否提升？
- **RQ2**: PolyGen 中的语句合成器与合一器是否提升了 PolyGen 的性能？

5.6.1 代码实现

因为提升问题中的合并算子一般只进行整数操作，所以本文主要关注 PolyGen 在**条件线性算数领域**（CLIA）上的性能。

条件线性整数领域 不同的 CLIA 领域具有相似的程序空间，它们仅在输入的数量上有所不同。具有 n 个输入的 CLIA 领域 $\mathbb{D}_I = (\langle \mathbb{P}_t, \mathbb{P}_c \rangle, \mathbb{I})$ 的定义如下：

- 语句空间 $\mathbb{P}_t$ 包含输入变量 $x_1, \ldots, x_n$ 的所有线性整数表达式。

- 条件空间 $\mathbb{P}_c$ 包含线性表达式之间的所有算术比较及其布尔表达式，即 $\mathbb{P}_c$ 是满足以下等式的最小集合。

$$\begin{aligned}\mathbb{P}_c = &\{e_1 \circ e_2 \mid e_1, e_2 \in \mathbb{P}_t, \circ \in \{<, \leq, =\}\} \cup \\ &\{c_1 \circ c_2 \mid c_1, c_2 \in \mathbb{P}_c, \circ \in \{\texttt{and}, \texttt{or}\}\} \cup \\ &\{\texttt{not}\ c \mid c \in \mathbb{P}_c\}\end{aligned}$$

- 输入空间 $\mathbb{I}$ 包含对输入变量的整数赋值。为了方便，本文只考虑参数 INF 范围内的所有赋值，并在实现中将 INF 默认设置为 50。

不难验证 $\mathbb{D}_I$ 是一个分支闭合的条件领域，于是 PolyGen 的所有性质在 $\mathbb{D}_I$ 上均成立。

领域求解器　PolyGen 需要一个语句空间上的奥卡姆求解器 $\mathcal{S}_t$ 来合成单一语句。在 CLIA 领域上，本文将 $\mathcal{S}_t$ 实现为了一个总是合成最小合法程序的优化器。具体地，给定样例编程任务 T，$\mathcal{S}_t$ 通过求解如下的优化问题来合成程序 $c_0 + c_1 x_1 + \cdots + c_n x_n$:

$$\textbf{Minimize} \sum_{i=0}^{n} |c_i| \qquad \textbf{Subject to } \forall(\overline{w_i} \mapsto O) \in T, \textstyle\sum_{i=1}^{n} w_i c_i + c_0 = O$$

这个优化问题是整数线性规划的一个实例，于是 $\mathcal{S}_t$ 会将它交给 Gurobi [137]（一个最先进的整数规划求解器）求解。不难验证 $\mathcal{S}_t$ 是一个 $(1, 0)$-奥卡姆求解器。

其他参数　在实现 $\mathcal{T}_{poly}$ 和 $\mathcal{U}_{poly}$ 时，本文默认将参数 c 和 c_0 设置为 2。

5.6.2　实验设置

数据集　为了评估 PolyGen 的性能，本文从两个不同来源收集了 100 个 CLIA 领域上的程序合成问题。

1. 首先，本文从 SyGuS-Comp 中收集了 82 个 CLIA 领域上的合成问题。此处的每个问题都包含一个形式化规约，其目标是合成一个满足规约的程序。
2. 其次，本文从算法问题中抽取了 18 个合成问题。此处的每个问题都对应着一个真实的合并算子，其目标是找到一个与目标算子等价的程序。

样例来源　因为数据集中所有任务都只提供了形式化规约，而 PolyGen 是一个样例编程求解器，所以在求解这些任务时，PolyGen 还需要一个额外的样例生成方法。为了综合评估 PolyGen 的性能，本文同时考虑了两种最常见的样例生成方式。

- $\mathcal{O}_V$ 遵循着 CEGIS 框架。它迭代地维护了一个初始为空样例集合，每次调用样例编程求解器用当前的样例集合上合成程序，并验证。如果当前结果正确，则

直接返回，否则将当前程序的一个反例加入样例集合，并开始新的一轮迭代。在使用 $\mathcal{O}_V$ 生成样例时，本文用最终使用的样例数量（即迭代轮数）衡量样例编程求解器的泛化能力，用总时间开销衡量样例编程求解器的合成效率。

- $\mathcal{O}_R$ 随机产生样例。它对应于实践中那些不存在验证器的场景。在这些场景中，已有方法通常会在资源限制下生成尽可能多的样例，然后调用样例编程求解器从这些样例合成程序，并直接返回。

  为了评估样例编程求解器在与 $\mathcal{O}_R$ 结合时的性能，本文使用了与 CEGIS 类似的方式。从空的样例集合开始，每一在现有的样例上运行样例编程求解器并验证结果。如果当前结果不正确，则从 $\mathcal{O}_R$ 处请求一个新的随机样例，并开始新的一轮合成，直到找到正确的程序。在这一过程中，本文用最终使用的样例数量衡量泛化能力，并用最后一轮的运行时间衡量合成效率。

基准方法 本文将 PolyGen 与三个现有方法比较，分别是 Enum [28]、EUSolver [38] 和 Euphony [40]。它们代表了提高泛化能力的三种不同方法：

1. 第一种方法遵循奥卡姆剃刀的原则，总是合成最小的合法程序。在 CLIA 上，Enum 是遵循该方法的最佳求解器，它按大小递增的顺序枚举程序，并通过观察等价法跳过重复程序。
2. 第二种方法将奥卡姆剃刀原则与高效的合成技术启发式地结合，从而在泛化能力和效率之间做出权衡。EUSolver 是遵循这种方法的最佳求解器。
3. 第三种方法使用机器学习模型来指导合成。Euphony 是这一类别中的最先进的求解器，它在 EUSolver 的基础上使用了概率模型指导搜索。

此外，由于 Enum 在 CLIA 上的效率有限，本文还考虑了一个针对 CLIA 领域的改进版本，记作 Enum$^+$。该求解器在 Enum 的基础上添加了对分支操作符的特殊处理，并仍然保证合成最小的合法程序。在枚举过程中，每当需要用分支操作符构建程序时，Enum$^+$ 会首先选择一个现有的布尔表达式作为条件；然后，对于每个现有程序 p，Enum$^+$ 仅在 p 是满足某个样例子集的最小程序时，才会尝试将 p 作为分支语句。更具体地，设 E_t（E_f）是分支条件为真（假）的样例集合。只有当存在子集 $E' \subseteq E_t$（E_f）使得 p 是满足 E' 中所有样例的最小程序时，Enum$^+$ 才会将 p 作为分支语句。

其他设置 在实验中，本文使用 Z3 [138] 验证合成结果的正确性。对于每次运行，本文将时间限制设为 120 秒，内存限制设为 8 GB，样例数量的上限设为 10^4。此外，为了避免随机性的影响，本文会将所有实验重复 5 次，并只考虑平均结果。

表 5.2　PolyGen 与基准方法的比较结果。

求解器	$\mathcal{O}_V$			$\mathcal{O}_R$			
	求解数量	样例数量	时间开销	求解数量	样例数量	样例下界	时间开销
PolyGen	97	×1.00	×1.00	93	×1.00		×1.00
Enum	9	×0.97	×3.67	9	×1.07		×52.3
Enum$^+$	26	×0.91	×8.23	15	×0.71		×43.3
EUSolver	65	×2.33	×6.14	65	×1.64	×3.32	×12.8
Euphony	51	×2.27	×7.42	53	×1.12	×3.30	×15.1

5.6.3　RQ1: PolyGen 在求解样例编程问题时的有效性

表 5.2 展示了 PolyGen 与基准方法的比较结果。为了比较泛化能力，本文考虑所有被基线方法与 PolyGen 同时求解的任务，计算在每个任务上它们使用的样例数量的比值，并在“样例数量”列汇报了这些比值的几何平均值。这一列中的 ×1.000 意味着 PolyGen 与基准方法使用了相同数量的样例，而比值越大，意味着 PolyGen 的泛化能力相对越好。类似地，“时间开销”列展示了时间成本上的比率，该比值越大意味着 PolyGen 的合成效率相对越高。

与 Enum 和 Enum$^+$ 相比，PolyGen 达到了和它们接近的泛化能力。注意，本文使用的奥卡姆学习理论是对 Enum 和 Enum$^+$ 的近似：奥卡姆学习理论将寻找最小合法程序的要求放宽到寻找多项式范围内的合法程序，从而为设计高效合成方法带来了更多的空间。实验结果表明，本文使用的近似在实践中并不会过多影响泛化能力。与此同时，PolyGen 具有显著更高的合成效率：它解决了更多的合成任务，并在那些共同解决的任务上有显著的效率提升。

与 EUSolver 和 Euphony 相比，PolyGen 在泛化能力和效率方面都表现得更好。泛化能力上的实验结果与本文的理论分析一致：PolyGen 具有更好的泛化能力是因为它是一个奥卡姆求解器，而 EUSolver 与 Euphony 不是。一个有趣的现象是，PolyGen 的优势在与 $\mathcal{O}_V$ 结合的时候更加明显，而在与 $\mathcal{O}_R$ 结合时则没有那么显著。这是因为 $\mathcal{O}_R$ 可能需要许多随机样例才能排除一个近似正确的程序，此时，何时随机到一个合适的样例成为了样例数量的瓶颈，而求解器本身泛化能力的重要性则被削弱。例如，在合成任务 `qm_neg_1.sl` 中，随机样例很难区分如下的两个程序：

$$p^*(x) \coloneqq (\texttt{if}\ (x < 0)\ \texttt{then}\ 1\ \texttt{else}\ 0)$$

$$p'(x) \coloneqq (\texttt{if}\ (x \leq 0)\ \texttt{then}\ 1\ \texttt{else}\ 0)$$

当输入范围在 $[-50, 50]$ 之间时，随机输入区分它们的概率小于 1%。这一现象表明了样例质量同样会对样例编程的性能产生显著影响，本文将在下一章中对这一点展开讨论。

表 5.3 PolyGen 与其弱化版本的比较结果。

求解器	$\mathcal{O}_V$			$\mathcal{O}_R$		
	求解数量	样例数量	时间开销	求解数量	样例数量	时间开销
PolyGen	97	×1.00	×1.00	93	×1.00	×1.00
PolyGen$_{-T}$	73	×1.15	×2.00	75	×0.96	×2.73
PolyGen$_{-U}$	71	×1.64	×1.95	69	×1.29	×1.93

同样值得注意的是，“样例数量” 列的结果存在幸存者偏差。在许多只有 PolyGen 解决的任务上，PolyGen 可能具有显著更好的泛化能力，但由于基准方法的失败，这些任务上的性能并没有被计入 “样例数量” 列中的结果。

为了验证这一点，本文使用样例生成方法 $\mathcal{O}_R$ 进行了额外实验。本文在那些 PolyGen 成功但 EUSolver 或 Euphony 失败的任务上以 30 分钟的时间限制重新迭代地运行失败的基准方法。从仅有 1 个随机示例开始，如果基准方法合成了不正确的程序，本文会将当前的样例数量作为泛化能力的下界，然后将样例数量加倍进行下一次迭代，直到合成成功或者超时。表 5.2 中的 “样例下界” 列汇报了在考虑这些下界后的几何平均值。实验结果证明了幸存者偏差的存在，因为使用样例的比率从 $\times 1.115 - \times 1.649$ 增加到了 $\times 3.320 - \times 3.302$。

最后，在合成效率方面，PolyGen 几乎解决了所有合成任务。本文调查了那些 PolyGen 失败的任务，并总结了两个主要原因：

- 由于语句合成器 $\mathcal{T}_{\text{poly}}$ 的时间成本会随着分支语句数量的增加而快速增长，PolyGen 无法高效地处理那些需要大量分支语句的情况。例如，PolyGen 无法为任务 `array_serach_15.sl` 合成语句集合，因为它需要 16 个不同的分支语句。
- 由于合一器 $\mathcal{U}_{\text{poly}}$ 按大小递增的顺序考虑所有文字，PolyGen 无法高效地处理文字规模较大的情况。例如，PolyGen 在任务 `mpg_example3.sl` 上超时，因为它需要使用一个较大的文字 $2x + 2y - z - 7 \leq 0$。

5.6.4 RQ2: PolyGen 中语句合成器与合一器的有效性

为了评估 PolyGen 中语句合成器与合一器的性能，本文考虑了 PolyGen 的两个变种 PolyGen$_{-T}$ 与 PolyGen$_{-U}$，它们分别将 PolyGen 中的语句合成器与合一器替换为了 EUSolver 的对应组件。表 5.3 展示了 PolyGen 与这些变种的比较结果。

实验结果表明，合一器 $\mathcal{U}_{\text{poly}}$ 在效率和泛化能力方面相比于 EUSolver 的合一器有很大提升。相比之下，语句合成器 $\mathcal{T}_{\text{poly}}$ 的主要贡献在于效率，并没有显著改善泛化能力。这一现象的原因是合一器所需的样例数量通常远大于语句合成所需要的样例数量，因为每个样例为分支条件提供的信息（类型为布尔）通常远小于每个样例为分支语句

提供的信息（类型为整数）。

在泛化能力外，$\mathfrak{T}_{\text{poly}}$ 相比 $\mathfrak{T}_E$ 的一个重要优势是它可以通过约束求解快速找到复杂的分支语句。例如，在合成任务 `mpg_example4.sl` 上，$\mathfrak{T}_{\text{poly}}$ 能够找到一个极端复杂的语句集合，如下所示。它远远超出了 $\mathfrak{T}_E$ 的求解能力。

$$\left\{10x+20y+15z-99, 9y+25w-11, 11x+15y+30z+22w+11, 16x+18z+5w-55\right\}$$

5.7　小结

为了更高效地合成提升问题中的合并算子，本章讨论了如何为分支程序设计一个更加高效的样例编程求解器。通过分析现有求解器 EUSolver 的表现，本文发现其核心不足在于泛化能力有限。在 CEGIS 框架下，EUSolver 往往需要收集大量的反例才能找到目标程序，这带来了巨大的时间开销。

为了提升样例编程求解器的泛化能力，本文首先引入了奥卡姆求解器的概念，从而在理论上为度量泛化能力建立了标准。一个奥卡姆求解器需要保证其合成的程序规模不会超过最小合法程序的多项式级别，并在渐进意义下小于样例的数量。根据奥卡姆学习理论，任何满足这一条件的求解器在接受多项式级别的样例后，都能保证返回一个接近目标程序的结果。

接着，本文从奥卡姆求解器的定义出发，遵循基于合一化的程序合成框架，按照自顶向下分解的方式设计了一个针对分支语句的奥卡姆合成器 PolyGen。本文首先按照结构将分支程序分解为子程序，从完整的程序到语句集合、条件集合，再到单个语句、单个条件，再到条件中每个子句、每个问题。随后，本文按照自顶向下的顺序为每个部分设计合成方法，在将合成问题分解到更低层次的同时，也将奥卡姆合成器的要求传递到子任务上。通过这种方式，本文为每个子程序设计了高效的合成方法，并从理论上保证了这些合成方法的组合一定能形成一个奥卡姆求解器。

本文的实验结果表明，PolyGen 相比已有的样例编程求解器具有显著更好的泛化能力与更高的求解效率。此外，一个有趣的现象是不同的样例生成方式会显著地影响 PolyGen 的性能，本文将在下一章中围绕这一点展开详细讨论。

第六章　高效的样例选择方法

6.1　引言

在实践中，输入输出样例的质量也会显著影响样例编程求解器的性能。例如，本文上一章的实验结果（第 5.6 节）表明，当样例编程求解器 PolyGen 接受高质量的反例时，其泛化能力和合成效率显著优于仅接受低质量随机样例的情况。

受到这一结果的启发，本章以加速程序合成为目标探索了高质量样例的生成方法。该问题与交互式程序合成领域中的**询问选择问题** [45] 密切相关。在交互式程序合成中，程序合成器每一轮可以向用户提出询问以获取更多信息，直至找到让用户满意的程序。而询问选择问题的目标是在每一轮中选择最佳的询问，以最小化向用户的提问数量，从而减少用户的负担。对应到加速样例编程的场景，样例生成器需要在每一轮选择一个合适的输入，根据规约生成对应的输出，并尽可能减少样例编程求解器找到目标程序所需的样例数量，以降低时间开销。

目前已有不少研究工作对询问选择问题进行了深入讨论。这些工作对询问选择问题进行了理论分析，并将其近似为一个关于输入的优化问题。在该问题中，目标函数衡量的是当前输入在多大程度上区分了**剩余程序**，即程序空间中满足现有样例的程序集合。例如，当剩余程序为 $0, x, y$ 时，询问选择器应偏好输入 $\langle x = 1, y = 2\rangle$（使所有剩余程序的输出各不相同），而非输入 $\langle x = 0, y = 0\rangle$（所有剩余程序的输出仍然相同）。

然而，即使经过简化，解决询问选择问题仍然面临显著的效率挑战。若要计算某个输入的目标函数值，一种直接的计算方式是枚举所有剩余程序，计算每个程序的输出，并评估这些输出的多样性。然而，由于程序合成问题的程序空间通常极为庞大，这种方法的时间开销完全无法接受。

现有的方法主要通过近似程序空间来应对这一效率挑战 [45,139]。在计算目标函数时，这些方法会先从所有剩余程序中采样一小部分，并仅考虑这些采样结果上的输出多样性。尽管这些方法在交互式程序合成场景下表现良好，但如果以加速程序合成为目标，它们仍存在两个核心缺点：

- **时间开销**。理论上，从剩余程序中采样的难度高于样例编程本身：它需要找到多个满足所有样例的程序，而样例编程只需找到一个。因此，计算近似目标函数的时间开销通常远大于样例编程本身。
- **通用性**。剩余程序的采样依赖于高效的见证函数，但这些函数在大多数合成域中并不存在，例如 PolyGen 所关心的条件整数领域。

为了克服这些缺点，本文设计了一种更加高效的近似方法。与现有方法相比，本

文提出的近似方法无需对剩余程序进行采样，其时间开销远小于样例编程本身；同时，它仅依赖于程序的语法语义，因此适用于几乎所有的合成领域。

为了实现高效的近似，本文分析了原始的目标函数，并发现计算它的主要挑战来源于运算符的复杂行为。针对这一问题，本文的近似方法分为三步：首先，本文尝试使用简单的随机过程模拟运算符的行为，并将其形式化为了**统一等价模型**；接着，基于该模型给出的预测定义了一个目标函数的近似；最后，提出并实现了一个高效的动态规划算法用于计算这一近似，并将其集成到样例选择方法 LearnSy 中。

综上所述，本章的主要创新点包括以下内容：

- 提出了统一等价模型以近似运算符的复杂行为（第 6.4 节），并在该模型的基础上实现了一个高效的样例选择方法 LearnSy（第 6.5 节）。
- 对 LearnSy 进行了实验评估，验证了它在加速程序合成时的有效性（第 6.6 节）。

6.2 概述

本节将使用一个示例任务展示 LearnSy 的核心思路。给定由如下文法 G_e 定义的程序空间，该任务的目标是合成一个与 $x+y$ 等价的程序，即 $x+y$ 或 $y+x$。

$$S := 1 \mid x \mid y \mid T+T \quad T := 1 \mid x \mid y$$

该文法包含了 12 个程序，按照从小到大的顺序列举如下。为了简化问题，假设该问题中变量 x 与 y 的取值范围被限定为了 $\{0,1,2\}$，此时共有 9 种可能的输入。

$$1, x, y, 1+1, 1+x, 1+y, x+1, x+x, x+y, y+1, y+x, y+y \tag{6.1}$$

此外，本节将只考虑基于枚举的样例编程求解器，讨论如何为该求解器选择合适的样例。该求解器会按照列表 6.1 中的顺序进行枚举，并返回第一个满足所有样例的程序。

6.2.1 样例选择中的最少等价对策略

不同的样例选择方法会显著地影响样例编程求解器使用的样例数量。最简单的方法是选择当前候选程序的一个反例，从而保证每轮合成至少会排除一个错误程序；然而，这种方法可能会过度拟合当前的结果，导致最坏情况下，需要的样例数量与程序空间的大小相当。很多时候，更巧妙的样例选择可以大幅度减少样例数量。在本节的示例任务中，如果可以在前两轮迭代中分别选择输入 $\langle x=0, y=2\rangle$ 和 $\langle x=1, y=0\rangle$，则基于枚举的样例编程方法可以在第三轮就找到目标程序 $x+y$。

为了选择最合适的样例，现有工作将样例选择问题规约到了最优决策树问题，并证明了一系列贪心策略在样例选择问题上的有效性[45]。本文主要考虑其中的一种策略，

表 6.1　使用最少等价对策略的合成过程。

轮数	样例编程结果	剩余程序	候选输入及其目标值	选取的样例
1	1	G_e 中的所有程序	$\langle 0,0\rangle$[62], $\langle 0,1\rangle$[56], $\langle 0,2\rangle$[34] $\langle 1,0\rangle$[56], $\langle 1,1\rangle$[90], $\langle 1,2\rangle$[46] $\langle 2,0\rangle$[34], $\langle 2,1\rangle$[46], $\langle 2,2\rangle$[42]	$\langle 0,2\rangle \mapsto 2$
2	y	$y, 1+1, x+y, y+x$	$\langle 0,0\rangle$[10], $\langle 0,1\rangle$[10], $\langle 0,2\rangle$[16] $\langle 1,0\rangle$[6], $\langle 1,1\rangle$[10], $\langle 1,2\rangle$[8] $\langle 2,0\rangle$[10], $\langle 2,1\rangle$[6], $\langle 2,2\rangle$[8]	$\langle 1,0\rangle \mapsto 1$
3	$x+y$			

最少等价对策略 [48,140]。这一策略定义一个输入上的目标值为该输入上输出相同的剩余程序对数，并总是选择具有最小目标值的输入作为当前样例。

表 6.1 展示了最少等价对策略下的合成过程，其中 $\langle a,b\rangle[w]$ 表示具有目标值 w 的输入 $\langle x=a, y=b\rangle$。以第二轮合成为例。此时在给定第一轮样例 $\langle 0,2\rangle \mapsto 2$ 的情况下，只有四个剩余程序 $y, 1+1, x+y$ 和 $y+x$：它们在输入 $\langle 1,0\rangle$ 上的输出分别为 $0, 2, 1$ 和 1。因此，输入 $\langle 1,0\rangle$ 的目标值为 6，对应着四个剩余程序与自身组成的二元组，加上 $x+y$ 与 $y+x$ 组成的两个二元组。根据表 6.1，在最少相等对策略下，基于枚举的样例编程方法只需两个样例便可完成合成。不难验证这已经是最优的样例选择方法之一。

6.2.2　对最少等价对策略的近似

尽管最少等价对策略在样例选择问题中十分有效，但精确地应用这一策略是非常困难的。在该策略中，计算一个输入的目标值需要需要枚举所有剩余程序并计算它们的输出。然而程序空间通常极大，导致这一过程的开销在实践中时无法接受的。

为了解决上述问题，本文希望为最少等价对策略提出一个可以被高效计算的近似。

主要思路　程序合成问题中的程序空间通常由上下文无关文法指定。这样的程序空间具有**可组合性**：文法中的每个非终结符对应了一个子程序的空间，而产生式将这些空间组合成更大的子空间，直到构成完整的程序空间。

在具有可组合性的程序空间上，具有可组合性的函数均可被高效计算：我们可以先在每个子空间上计算出一个子结果，并沿着产生式将子结果不断地合并，最终得到完整的计算结果。这一计算过程只需要处理每个子程序空间（即每个非终结符）一次，从而保证了高效性。

然而，最少等价对策略的目标函数并不具有可组合性，其原因在于运算符的复杂行为。考虑 G_e 中的产生式 $S \to T+T$。在没有任何样例的情况下，该产生式对应的目标值是它对应的所有程序中，输出相同的程序对数；而其子空间上的结果是所有对应非终结符 (T,T) 的程序二元组中，输出完全相同的二元组对数。这一子结果并不足以

表 6.2　一个针对文法 G_e 与输入 $\langle 0,2\rangle$ 的统一等价模型。

self	
$S \to T+T$	5/36
$S \to x$	0
…	…

cross	$S \to T+T$	$S \to x$	…
$S \to T+T$	N/A	1/9	…
$S \to x$	1/9	N/A	…
…	…	…	…

得到目标值，因为两对输出不同的子程序在被加号组合后仍可能输出相同。例如当输入为 $\langle x=0, y=1\rangle$ 时，子程序对 (x,y) 与 (y,x) 具有不同的输出，因此不会被计入子空间 (T,T) 上的结果；但在用加号组合后，完整程序 $x+y$ 与 $y+x$ 的输出相同，需要被计入 $S \to T+T$ 上的目标值。

为了消除运算符的复杂行为对目标值的影响，本文尝试用简单的随机过程替换运算符的复杂行为，以得到一个可组合的近似。在上述例子中，本文会将 + 近似为一个随机函数，并假设对于任意两对不同的输入，该函数始终以一个固定的概率产生相同的输出，将此概率记作 c。在此基础上，本文将目标值近似为在随机运算符下输出相同的程序对的期望数量。该近似的可组合性由期望的线性性保证。当子空间 (T,T) 上的结果是 w 时，产生式 $S \to T+T$ 上的近似目标值可以被计算为 $w+c(81-w)$：

- 当子程序的输出对应相等时，完整程序的输出一定相等，对近似值的贡献为 w。
- 当子程序的输出不完全相等时，它们有 c 的概率被随机函数合并到相同的输出。因为子空间 (T,T) 中一共有 $9^2=81$ 个二元组，所以期望有 $81-c$ 对输出不完全相等的二元组，对近似值的贡献为 $c(81-w)$。

统一等价模型　为了形式化上述思路，本文引入了**统一等价模型**的概念。一个统一等价模型与一个文法绑定，它会将文法中所有操作符的行为替换成随机函数，并预测文法中的两个程序产生相同输出的概率。表 6.2 展示了一个针对文法 G_e 与输入 $\langle 0,2\rangle$ 的统一等价模型，该模型由两组参数组成：

- 对于每个产生式 r，$\mathit{self}(r)$ 表示任意两组输出不完全相同的子程序在经过 r 的组合后输出相同的概率。
- 对于每一对不同的产生式 (r,r')，$\mathit{cross}(r,r')$ 表示对任意的一个从 r 展开的程序和一个从 r' 展开的程序，它们的输出恰好相同的概率。

统一等价模型按照程序结构递归地进行预测。图 6.1 展示了对程序 $x+y$ 与 $x+x$ 的预测过程，其中每个矩形代表了一个子任务，$p_1 =_? p_2$ 表示当前的目标是预测 p_1 与 p_2 输出相等的概率，而蓝色表达式展示了预测过程。给定程序 $x+y$ 与 $x+x$，模型会首先对它们的子程序 (x,y) 与 (x,x) 预测输出完全相等的概率，结果为 0。因为操作符 + 对应的随机函数将有 $\mathit{self}(S \to T+T)=5/36$ 的概率将不同的输入合并到相同的输出，

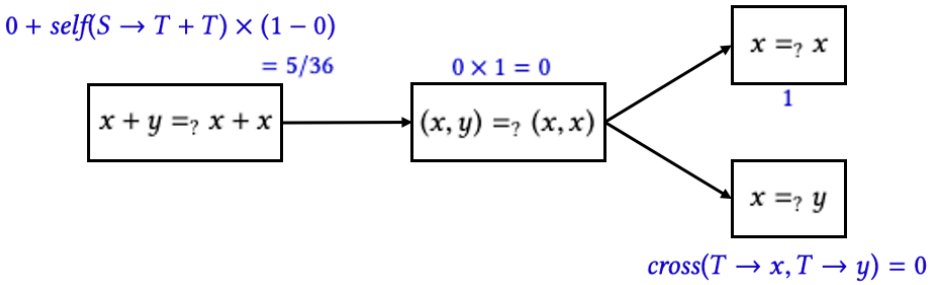

图 6.1　表 6.2 中的模型估计 $x + y$ 和 $x + x$ 是否输出相等的过程。

Algorithm 7: LearnSy 的运行过程。

Input: 上下文无关文法 G，输入空间 $\mathbb{I}$，当前的候选程序 p_c，以及现有样例集合 *examples*。
Output: 一个新的样例或表示接受 p_c 为最终合成结果的信号 $\perp$。

```
1 candidateInps ← GetCandidateInputs(𝕀, p_c);
2 bestObj ← +∞;  bestInp ← ⊥;
3 exampleModels ← {Learn(G, inp) | (inp ↦ oup) ∈ examples};
4 foreach inp ∈ candidateInps do
5     model ← Learn(G, inp);
6     obj ← Approximate(G, model, exampleModels);
7     if obj < bestObj then (bestObj, bestInp) ← (obj, inp);
8 end
9 return bestInp;
```

所以模型给出的最终预测是 $0 + 5/36 \times (1 - 0) = 5/36$。

在此基础上，本文对最少等价对策略的近似值是程序空间中相等程序对的期望数量，即统一等价模型为程序空间中的所有程序对给出的预测值之和。因为统一等价模型的预测严格按照程序结构进行，且程序空间本身是可组合的，所以该近似也是可组合的，能被高效地计算。本文将在第 6.5 节中对这一点展开详细讨论。

统一等价模型的学习方法　为了得到高质量的近似，关键在于如何为统一等价模型设置合适的参数。为了回答这一问题，本文首先刻画了一类在理论上具有良好性质的理想模型，称为**自然统一等价模型**；并在此基础上设计了一个基于采样的学习算法，以高效地产生接近理想情况的模型。本文将在第 6.4 节中介绍这一学习算法。

6.2.3　LearnSy 的运行过程

基于统一等价模型，本文提出了样例选择方法 LearnSy，如算法 7 所示。在选择样例时，LearnSy 会先生成一组候选输入（第 1 行），并从中选择近似值最小的输入返回（第 2-9 行）。具体而言，它会在**每个输入**上学习一个针对性的模型（第 3、5 行），计算每个候选输入的近似目标值（第 6 行），并选出具有最小近似值的输入（第 7 行）。

候选输入的生成过程取决于程序合成的场景。在 CEGIS 框架下，LearnSy 会用验证器产生一组反例作为候选输入；而在不存在验证器的情况下，LearnSy 会随机地从输入空间中采样得到候选输入。因为程序合成问题中的输入空间可能很大，所以 LearnSy 对候选输入的数量设置了上限，以控制时间成本。

6.3 最少等价对策略

6.3.1 符号说明

与第五章类似，本章使用二元组 $\mathbb{D} = (\mathbb{P}, \mathbb{I})$ 表示合成域，其中 $\mathbb{P}$ 与 $\mathbb{I}$ 分别表示程序空间与输入空间，并假设存在一个解释器 $[\![.]\!]$ 将程序解释为一个从输入到输出的函数。

本章假设程序空间 $\mathbb{P}$ 由一个**正则树文法** (RTG) 定义。该文法中的每个程序都严格按照其语法树展开，因此一定不存在歧义。此外，不失一般性地，本文只考虑有限的程序空间 $\mathbb{P}$，即由无环的 RTG 定义的程序空间。在程序合成的场景下，总是可以通过设置一个合适的大小限制，将无限的程序空间截断为一个有限空间。

定义 21 (正则树文法)**.** 一个正则树文法（*RTG*）是一个元组 $G = (N, \Sigma, s_0, R)$，其中：

- N 是一个有限的非终结符集合。s_0 是 N 中的一个元素，表示起始非终结符。
- Σ 是一个字母表，其中每个符号 f 都附有一个参数 k 表示参数数量，记作 $f^{(k)}$。
- R 是一个有限的产生式集合。R 中的每条产生式都具有 $s \to f^{(k)}(s_1, \ldots, s_k)$ 的形式，其中 $s, s_1, \ldots, s_k$ 是 N 中的非终结符，而 $f^{(k)}$ 是 Σ 中的符号。

如果一个程序可以通过有限数量的产生式从 s_0 展开得到，则该程序属于 G。

最后，本章将使用术语**等价性检查**表示形如 $[\![p]\!]I = [\![p']\!](I)$ 的谓词。它检查程序 p 和 p' 在输入 I 上的输出是否相同。

6.3.2 最少等价对策略

最少等价对策略 [48,140] 遵循着一个自然的想法：为了最小化程序合成需要的样例数量，每个样例都应当排除尽可能多的程序。最少等价对策略假设程序空间上存在一个概率分布，表示每个程序是目标程序的概率。在这一分布的基础上，最少等价对策略的目标是最小化一个随机程序在选择输入后仍然被保留的概率。

定义 22 (最少等价对)**.** 给定一个合成域 $(\mathbb{P}, \mathbb{I})$，一个程序空间 $\mathbb{P}$ 上的概率分布 φ，以及一组输入输出样例 E，最少等价对策略会通过最小化如下所示的目标函数来选择输入。

$$obj_0[\mathbb{P}, \varphi, E](I) \coloneqq \Pr_{p, p^* \sim \varphi} \left[p \in remain\big(\mathbb{P}, E \cup \{I \mapsto [\![p^*]\!](I)\}\big) \;\middle|\; p^* \in remain(\mathbb{P}, E) \right]$$

其中 $\Pr[C_1|C_2]$ 表示条件概率，p 表示一个随机的剩余程序，p^* 表示目标程序，$I \mapsto [\![p^*]\!](I)$ 表示当目标程序为 p^* 时输入 I 上的样例，$remain(\mathbb{P}, E)$ 表示满足 E 中样例的剩余程序集合。该函数计算在选择输入 I 后，随机剩余程序 p 仍然满足新样例的概率。

上述目标函数具有如下的等价形式，它也是“最少等价对”名字的来源，其中 $\varphi(p)$ 表示程序 p 在 φ 中的概率，而 $[\cdot]$ 是一个将 *true* 映射为 *1* 且将 *false* 映射为 0 的函数。

$$obj_1[\mathbb{P}, \varphi, E](I) \coloneqq \sum_{p,p' \in \mathbb{P}} \varphi(p)\varphi(p') \big[p, p' \in remain(\mathbb{P}, E)\big] \big[[\![p]\!](I) = [\![p']\!](I)\big]$$

在样例编程求解器给出候选程序 p_c 后，上述目标函数可以完全被等价性检查表示。容易证明，当 p_c 满足 E 中的所有样例时，下列函数等价于 obj_1，其中 $eq(p, p', I)$ 是 $[[\![p]\!](I) = [\![p']\!](I)]$ 的缩写，表示等价性检查的结果。

$$obj[\mathbb{P}, \varphi, E, p_c](I) \coloneqq \sum_{p,p' \in \mathbb{P}} \varphi(p)\varphi(p')eq(p, p', I) \prod_{(I' \mapsto O') \in E} \big(eq(p_c, p, I')eq(p, p', I')\big) \quad (6.2)$$

为了高效地评估一个程序的概率，本文进一步假设概率分布 φ 以**概率正则树文法**（PRTG）的形式给出。

定义 23 (概率正则树文法)**.** 一个概率正则树文法（*PRTG*）是一个 *RTG* (N, Σ, s_0, R) 与一个函数 $\gamma : R \mapsto [0, 1]$ 的二元组，该函数为每条产生式分配一个概率，并保证从同一非终结符开始的所有产生式的概率和恰好为 1。

对于由产生式 $r_1, \ldots, r_n$ 展开得到的程序，它在 *PRTG* 中的概率被定义为 $\prod_{i=1}^{n} \gamma(r_i)$。

例 12. 如下的 *PRTG* 为第 *6.2* 节中的文法 G_e 指定了一个均匀分布，其中 $r[w]$ 表示产生式 r 被选择的概率是 w。

$$S \coloneqq 1\ [1/12] \mid x\ [1/12] \mid y\ [1/12] \mid T + T\ [3/4] \quad T \coloneqq 1\ [1/3] \mid x\ [1/3] \mid y\ [1/3]$$

在上述 *PRTG* 中，程序 $x + y$ 的概率等于 $\gamma(S \to T + T)\gamma(T \to x)\gamma(T \to y) = 1/12$。

6.4　统一等价模型

6.4.1　模型定义

统一等价模型的目标是在**不运行程序**的情况下预测等价性检查的结果。给定程序 p, p' 与输入 I，下面展示了一个用于等价性检查的递归过程。

$$receq(p, p', I) \coloneqq \begin{cases} [\![p]\!](I) = [\![p']\!](I) & r \neq r' \\ \text{true} & r = r' \wedge \left(\bigwedge_{i=1}^{n} receq(p_i, p_i', I)\right) \\ [\![p]\!](I) = [\![p']\!](I) & \text{Otherwise} \end{cases}$$

其中 r 是 p 使用的第一条产生式；n 是 r 中的参数数量，$p_1,\dots,p_n$ 是 p 的子程序；而 $r',n',p'_1,\dots,p'_{n'}$ 是与程序 p' 相关的对应部分。

函数 *receq* 在其第二种情况下利用函数的性质避免了程序运行：由于每个操作符的输入输出行为必定是函数，所以当 p 和 p' 的所有子程序的输出都对应相同时，它们的输出也一定相同。然而，该函数的其余两种情况仍然依赖于程序运行。统一等价模型引入了两组参数（分别命名为 *self* 和 *cross*），将这两种情况近似为与程序运行无关的随机过程。这些参数分别表示 *receq* 在其第一种和第三种情况下返回 *true* 的概率。

定义 24 (统一等价模型)**.** 给定由文法 G 定义的程序空间，令 R 为 G 中的产生式集合。该程序空间上的统一等价模型 $\mathcal{M}$ 由两个函数 $self : R \mapsto [0,1]$ 和 $cross : R \times R \mapsto [0,1]$ 指定，它们分别为每条产生式和每对不同的产生式分配了概率。

模型 $\mathcal{M}$ 导出了一个预测函数 $\mathcal{M}(p,p')$，用于估计程序 p 和 p' 通过等价性检查的可能性。该函数的定义如下所示。

$$\mathcal{M}(p,p') \coloneqq \begin{cases} cross(r,r') & r \neq r' \\ w + (1-w)self(r) \quad \textbf{\textit{where}}\ w \coloneqq \prod_{i=1}^{n} \mathcal{M}(p_i,p'_i) & r = r' \end{cases}$$

其中 r 是 p 使用的第一条产生式；n 是 r 中的参数数量；$p_1,\dots,p_n$ 是 p 的子程序；而 $r',n',p'_1,\dots,p'_{n'}$ 是与程序 p' 相关的对应部分。在上述定义中，红色和绿色部分分别对应函数 *receq* 中的后两种情况，而 w 表示子程序输出相同的概率。

例 13. 回顾图 6.1，统一等价模型在对 $x+y$ 和 $x+x$ 预测时，其输出的符号形式如下：

$$w + (1-w)self(S \to T + T) \quad \textbf{\textit{where}}\ w \coloneqq cross(T \to x, T \to y)$$

6.4.2 统一等价模型的学习算法

统一等价模型的学习算法接受一个 PRTG 与一个程序输入，并返回一个针对给定分布与输入的统一等价模型。在设计这一学习算法时，本文首先引入了**自然统一等价模型**的概念：它是一类理想的统一等价模型，在预测精度上具有理论保障。接着，本文设计了一个高效的算法用于快速地生成一个与自然模型接近的模型。

自然统一等价模型 回顾统一等价模型的定义（定义 24），其中的每个参数在等价性检查时都具有明确的含义。

- $self(r)$ 对应两个程序输出相同的条件概率，其中条件为 (1) 两个程序都由产生式 r 展开，(2) 两个程序中子程序的输出不完全相同。
- $cross(r,r')$ 对应两个程序分别由产生式 r 和 r' 展开时，它们输出相同的概率。

在给定分布与具体输入的情况下，这些参数对应的概率是定义明确且可计算的。例如，参数 $cross(r_1, r_2)$ 对应的是事件 $[\![p_1]\!](I) = [\![p_2]\!](I)$ 发生的概率，其中 I 是给定输入，而 p_i 是根据给定分布从 r_i 的程序空间中采样的结果。因此，在学习统一等价模型时，一个自然的想法是将每个参数精确地设置为它对应的概率。本文将这个概率称为参数的**自然赋值**，而每个参数被都设置为其自然赋值的模型被称为**自然统一等价模型**。

定义 25 (自然统一等价模型)**.** 给定一个 *PRTG* φ 和一个输入 I，自然统一等价模型 $\mathcal{N}_\varphi^I$ 是在 φ 的程序空间上的一个统一等价模型，其参数如下。

$$self(r) \coloneqq \Pr_{p,p' \sim \varphi(r)} \left[[\![p]\!](I) = [\![p']\!](I) \mid \exists i, [\![p_i]\!](I) \neq [\![p'_i]\!](I) \right] \tag{6.3}$$

$$cross(r, r') \coloneqq \Pr_{p \sim \varphi(r), p' \sim \varphi(r')} \left[[\![p]\!](I) = [\![p']\!](I) \right] \tag{6.4}$$

其中 p_i 和 p'_i 分别表示 p 和 p' 的第 i 个子程序，$p \sim \varphi(r)$ 表示 p 是从 r 的程序空间中随机抽取的程序，其中每个程序被选中的概率与其由 φ 分配的概率成正比。

例 14. 表 6.2 对应着均匀分布（例 13 中的 *PRTG*）和输入 $\langle 2, 0 \rangle$ 上的自然统一等价模型。

考虑计算 $self(S \to T + T)$ 的自然赋值。该产生式的程序空间包含 9 个程序，它们的概率都相同。在输入 $\langle 2, 0 \rangle$ 上，这 9 个程序中分别有 $1, 2, 3, 2, 1$ 个程序输出 $0, 1, 2, 3, 4$，且这些程序子程序的输出两两不等，取值分别从 $(0, 0)$ 到 $(2, 2)$。于是，该程序空间中有 72 个程序对具有输出不完全相同的子程序，其中 10 对的最终输出相同。因此，$self(S \to T + T)$ 的自然赋值是 $10/72 = 5/36$。

自然统一等价模型能够精确地预测程序之间的整体等价性（引理11）。当等价性检查中的两个程序是随机的时，自然统一等价模型给出的期望估计始终等于基本事实，即两个随机程序在相应输入下输出相同的概率。

引理 11. 对于任何输入 I 和 *PRTG* φ，自然统一等价模型 $\mathcal{N}_\varphi^I$ 总是满足如下等式。

$$\mathop{\mathbb{E}}_{p,p' \sim \varphi} \left[\mathcal{N}_\varphi^I(p, p') \right] = \Pr_{p,p' \sim \varphi} \left[[\![p]\!](I) = [\![p']\!](I) \right]$$

证明. 让我们直接证明该引理的一个推广形式，如下所示：

$$\mathop{\mathbb{E}}_{p,p' \sim \varphi_s} \left[\mathcal{N}_\varphi^I(p, p') \right] = \Pr_{p,p' \sim \varphi_s} \left[[\![p]\!](I) = [\![p']\!](I) \right] \tag{6.5}$$

其中 s 是一个非终结符，φ_s 代表在 s 的程序空间上由分布 φ 导出的子分布，其中每个程序在 φ_s 中的概率正比于它在 φ 中的概率。显然，原引理是上述形式在 s 取起始非终结符时的特殊情况。

因为本章假设文法无环，所以非终结符间存在着一个拓扑排序 $s_1, \ldots, s_n$，它保证了文法中的每条产生式只会涉及更靠前的非终结符。本文通过对非终结符 s 在该拓扑排序中的下标作归纳的方式证明上述推广形式（公式 6.5）。

具体而言，公式 6.5 的等号两侧可以按照如下方式展开，其中 $R(s)$ 表示从非终结符 s 出发的产生式集合，φ_r 表示在 r 的程序空间上由 φ 导出的子分布。

$$\mathop{\mathbb{E}}_{p,p'\sim\varphi_s}\left[\mathcal{N}^I_\varphi(p,p')\right] = \sum_{r,r'\in R(s)} \gamma(r)\gamma(r') \mathop{\mathbb{E}}_{\substack{p\sim\varphi_r \\ p'\sim\varphi_{r'}}} \left[\mathcal{N}^I_\varphi(p,p')\right]$$

$$\Pr_{p,p'\sim\varphi_s}\left[[\![p]\!](I) = [\![p']\!](I)\right] = \sum_{r,r'\in R(s)} \gamma(r)\gamma(r') \Pr_{\substack{p\sim\varphi_r \\ p'\sim\varphi_{r'}}} \left[[\![p]\!](I) = [\![p']\!](I)\right]$$

我们只需要证明红色部分相等。此时根据 r 和 r' 是否相等存在着两种情况。

情况 1： $r \neq r'$　对于从 (r, r') 展开的任何一对程序 (p_x, p_y)，根据统一等价模型的定义，下列等式一定成立。将这一等式的左右两侧对 p_x 和 p_y 的取值在分布 φ_r 和 $\varphi_{r'}$ 上计算期望，便可得到红色部分的相等关系。

$$\mathcal{N}^I_\varphi(p_x,p_y) = cross\left[\mathcal{N}^I_\varphi\right](r,r') = \Pr_{\substack{p\sim\varphi_r \\ p'\sim\varphi_{r'}}} \left[[\![p]\!](I) = [\![p']\!](I)\right]$$

情况 2： $r = r'$　令产生式 r 的具体形式为 $s \to f(t_1, \ldots, t_k)$，其中 t_i 是在拓扑排序中比 s 更靠前的非终结符。根据该形式展开红色部分上侧的数学期望，如下所示。

$$\mathop{\mathbb{E}}_{p,p'\sim\varphi_r}\left[\mathcal{N}^I_\varphi(p,p')\right] = w + (1-w)self\left[\mathcal{N}^I_\varphi\right](r) \quad \textbf{where } w \coloneqq \mathop{\mathbb{E}}_{p_i,p_i'\sim\varphi_{t_i}}\left[\prod_{i=1}^k \mathcal{N}^I_\varphi(p_i,p_i')\right]$$

通过在非终结符 t_i 上应用归纳假设，可以将变量 w 的取值变换如下。

$$\begin{aligned}
\mathop{\mathbb{E}}_{p_i,p_i'\sim\varphi_{t_i}}\left[\prod_{i=1}^k \mathcal{N}^I_\varphi(p_i,p_i')\right] &= \prod_{i=1}^k \mathop{\mathbb{E}}_{p_i,p_i'\sim\varphi_{t_i}}\left[\mathcal{N}^I_\varphi(p_i,p_i')\right] \\
&= \prod_{i=1}^k \Pr_{p_i,p_i'\sim\varphi_{t_i}}\left[[\![p_i]\!](I) = [\![p_i']\!](I)\right] \\
&= \Pr_{p_i,p_i'\sim\varphi_{t_i}}\left[\bigwedge_{i=1}^k [\![p_i]\!](I) = [\![p_i']\!](I)\right]
\end{aligned}$$

Algorithm 8: 算法7中的学习过程 `Learn`。

Input: 包含产生式集合 R 的 PRTG φ，输入 I，以及两个常量 $n_s, w_{default}$。
Output: 针对分布 φ 和输入 I 的一个统一等价模型。

$self \leftarrow \{\}$; $cross \leftarrow \{\}$;
foreach $r \in R$ **do**
  $(p_1, \dots, p_{n_s}) \leftarrow \texttt{Sample}(r, \varphi, n_s)$; $(p'_1, \dots, p'_{n_s}) \leftarrow \texttt{Sample}(r, \varphi, n_s)$;
  $n_1 \leftarrow i \in \{1, \dots, n_s\}$ 中满足 $\texttt{IsSubDiff}(p_i, p'_i, I)$ 的 i 的数量;
  $n_2 \leftarrow i \in \{1, \dots, n_s\}$ 中满足 $\texttt{IsSubDiff}(p_i, p'_i, I)$ 且 $[\![p_i]\!](I) = [\![p_i]\!](I')$ 的 i 的数量;
  if $n_1 > 0$ **then** $self(r) \leftarrow n_2/n_1$ **else** $self(r) \leftarrow w_{default}$;
end
foreach $r, r' \in R$ **do**
  $(p_1, \dots, p_{n_s}) \leftarrow \texttt{Sample}(r, \varphi, n_s)$; $(p'_1, \dots, p'_{n_s}) \leftarrow \texttt{Sample}(r', \varphi, n_s)$;
  $cross(r, r') \leftarrow (i \in \{1, \dots, n_s\}$ 中满足 $[\![p_i]\!](I) = [\![p'_i]\!](I)$ 的 i 的数量$)/n_s$;
end
return $(self, cross)$;

将绿色部分记作事件 $\mathcal{E}$。我们可以通过如下方式推导得到红色部分的相等关系。

$$
\begin{aligned}
&\mathop{\mathbb{E}}_{p,p'\sim\varphi_r}\left[\mathcal{N}^I_\varphi(p,p')\right] \\
=\ & \Pr[\mathcal{E}] + (1-\Pr[\mathcal{E}])\mathit{self}\left[\mathcal{N}^I_\varphi\right](r) \\
=\ & \Pr_{p,p'\sim\varphi_r}\left[[\![p]\!](I) = [\![p']\!](I) \wedge \mathcal{E}\right] + (1-\Pr[\mathcal{E}]) \Pr_{p,p'\sim\varphi_r}\left[[\![p]\!](I) = [\![p']\!](I) \middle| \neg\mathcal{E}\right] \\
=\ & \Pr_{p,p'\sim\varphi_r}\left[[\![p]\!](I) = [\![p']\!](I) \wedge \mathcal{E}\right] + \Pr_{p,p'\sim\varphi_r}\left[[\![p]\!](I) = [\![p']\!](I) \wedge \neg\mathcal{E}\right] \\
=\ & \Pr_{p,p'\sim\varphi_r}\left[[\![p]\!](I) = [\![p']\!](I)\right]
\end{aligned}
$$

综上所述，归纳在两种情况下成立，于是形成了对目标引理的证明。

□

高效的学习算法　尽管自然统一等价模型具有良好的精度保障，但计算自然赋值的时间开销在实践中通常难以接受。根据公式 6.3 和 6.4，自然赋值总是依赖于所有相关程序的输出，因此其计算仍然面临由程序空间大小带来的效率问题。

为了解决这一问题，本文通过采样来近似地计算自然赋值，如算法 8 所示，其中 n_s 表示样本数量，$w_{default}$ 表示参数的默认值。该算法使用了两个自定义函数。

- $\texttt{Sample}(r, \varphi, n_s)$ 从产生 r 的程序空间中按照分布 φ 采样 n_s 个独立样本。
- $\texttt{IsSubDiff}(p, p', I)$ 检查 p 和 p' 的子程序在输入 I 下是否输出不同。

对于每个参数，算法 8 会生成 n_s 对随机程序样本，并用它们近似地计算自然赋值。该算法是高效的，其时间成本与文法大小成多项式关系，不会带来显著的额外开销。

上述学习算法可以为自然统一等价模型提供一个无偏估计。具体而言，该算法的

学习结果在期望意义下能给出与自然统一等价模型相同的预测结果。

定理 20. *对于任何输入 I，PRTG φ，以及任意程序 p, p'，下列等式总是成立*[①]*。*

$$\mathbb{E}[\mathcal{M}(p,p')] = \mathcal{N}_\varphi^I(p,p') \quad \textbf{\textit{where}}\ \mathcal{M} \coloneqq \texttt{Learn}(\varphi, I)$$

其中随机性来自 Learn 中使用的随机样本。

证明. $\mathcal{M}$ 中的参数可以按如下方式对应到自然统一等价模型 $\mathcal{N}_\varphi^I$ 中的参数，其中 (p, p_i) 表示 Learn 中的样本程序，φ_r 表示在产生式 r 的程序空间上由 φ 导出的子分布。

$$\begin{aligned}
&\mathop{\mathbb{E}}_{p_i,p_i'\sim\varphi_r}\left[self[\mathcal{M}](r)\right] \\
=&\mathop{\mathbb{E}}_{p_i,p_i'\sim\varphi_r}\left[\frac{size\left\{i \;\middle|\; [\![p_i]\!](I) = [\![p_i]\!](I') \wedge \texttt{IsSubDiff}(p_i,p_i',I)\right\}}{size\left\{i \;\middle|\; \texttt{IsSubDiff}(p_i,p_i',I)\right\}}\right] \\
=&\Pr_{p,p'\sim\varphi_r}\left[[\![p]\!](I) = [\![p']\!](I) \;\middle|\; \texttt{IsSubDiff}(p,p',I)\right] \\
=&\, self\left[\mathcal{N}_\varphi^I\right](r)
\end{aligned}$$

$$\begin{aligned}
\mathop{\mathbb{E}}_{p_i\sim\varphi_r,p_i'\sim\varphi_{r'}}\left[cross[\mathcal{M}](r,r')\right] &= \mathop{\mathbb{E}}_{p_i\sim\varphi_r,p_i'\sim\varphi_{r'}}\left[size\left\{i \;\middle|\; [\![p_i]\!](I) \neq [\![p_i']\!](I)\right\}\Big/ n_s\right] \\
&= \Pr_{p\sim\varphi_r,p'\sim\varphi_{r'}}\left[[\![p]\!](I) = [\![p']\!](I)\right] \\
&= cross\left[\mathcal{N}_\varphi^I\right](r,r')
\end{aligned}$$

根据这些对应关系，目标引理可以通过对 p 的程序结构归纳得到。 □

6.4.3 通过展开操作提升预测精度

尽管统一等价模型可以相对精确地预测程序空间上的整体等价关系，但在为特定程序对预测等价性时仍可能存在误差。为了减小这种误差，本文定义了一个针对 PRTG 的**展开**操作符，以提高统一等价模型的精度。该操作符会将特定的产生式展开成一系列常量，并同时确保每个程序的概率不变。

定义 26 (展开操作)**.** *给定在语法 G 上定义的 PRTG φ，展开操作接受 G 中的一条产生式 r^* 为参数，并通过如下方式对 φ 进行变换。*

1. *令 s 为产生式 r^* 所属的非终结符，P 为规则 r^* 对应的程序空间。*
2. *非终结符集合和起始非终结符保持不变。*
3. *对于每个程序 $p \in P$，添加一个不需要任何参数的符号 $\square_p$，其语义与 p 相同。*

① 该引理假设默认值 $w_{default}$ 从未被使用，这在样本数量充分大时几乎一定成立。

4. 移除产生式 r^*，并对于每个程序 $p \in \boldsymbol{P}$ 添加产生式 $s \to \square_p$。

5. 现有产生式的概率保持不变，而新产生式的概率等于相应程序在 φ 中的概率。

例 15. 考虑由以下语法 G 指定的程序空间。

$$S \coloneqq S_1 + S_1 \mid S_1 - S_1 \qquad S_1 \coloneqq x \mid y$$

通过对产生式 $S \to S_1 - S_1$ 展开，可以得到另一个语法 G'，如下所示。

$$S \coloneqq S_1 + S_1 \mid \square_{x-x} \mid \square_{x-y} \mid \square_{y-x} \mid \square_{y-y} \qquad S_1 \coloneqq x \mid y$$

考虑在输入 $\langle x = 0, y = 1 \rangle$ 上对程序 $x + y$ 和 $x - y$ 的等价性检查。这一检查的真实结果为假。下面展示了这两个文法上的统一等价模型给出的预测结果。

- 设 $\mathcal{N}_1$ 为对应语法 G 上的均匀分布与输入 $\langle x = 0, y = 1 \rangle$ 的自然统一等价模型。在该模型中，$cross(S \to S_1 + S_1, S \to S_1 - S_1)$ 的取值是 1/4，于是预测结果 $\mathcal{N}_1(x + y, x - y)$ 等于 1/4，绝对误差为 1/4。
- 设 $\mathcal{N}_2$ 为对应语法 G 上的均匀分布与输入 $\langle x = 0, y = 1 \rangle$ 的自然统一等价模型。在该模型中，$cross(S \to S_1 + S_1, S \to \square_{x-y})$ 的取值是 0，于是预测结果 $\mathcal{M}_2(x + y, x - y)$ 也是 0，这是准确的。

下列定理证明了展开操作对提升统一等效模型精度的积极作用。直观上，展开操作减少了每个程序使用的平均操作数。由于统一等效模型中的误差仅来自于对操作符的近似，因此展开操作减少了这种近似的数量，从而降低了误差累积。

定理 21. 给定两个程序 p 和 p'，输入 I 和统一等价模型 $\mathcal{M}$，定义随机事件 $\mathcal{E}(p, p', I, \mathcal{M})$ 表示 $\mathcal{M}(p, p')$ 的预测过程与对应的递归判断 $receq(p, p', I)$ 完全一致的情况，如下所示：

$$\mathcal{E}(p, p', I, \mathcal{M}) \coloneqq$$

$$\begin{cases} x_{cross} \leftrightarrow \left([\![p]\!](I) = [\![p']\!](I) \right) & r \neq r' \\ \left(x_{self} \leftrightarrow \left([\![p]\!](I) = [\![p']\!](I) \right) \right) \wedge \bigwedge_{i=1}^{n} \mathcal{E}(p_i, p_i', I, \mathcal{M}) & r = r' \wedge \texttt{IsSubDiff}(p, p', I) \\ \bigwedge_{i=1}^{n} \mathcal{E}(p_i, p_i', I, \mathcal{M}) & r = r' \wedge \neg \texttt{IsSubDiff}(p, p', I) \end{cases}$$

其中 r 表示 p 使用的第一条产生式，n 表示 p 的子程序数量，p_i 表示 p 的第 i 个子程序，而 r' 与 p_i' 是与程序 p' 相关的对应部分。此外，x_{cross} 与 x_{self} 是两个布尔类型的随机变量，它们为真的概率分别等于 $cross[\mathcal{M}](r, r')$ 和 $self[\mathcal{M}](r)$。

在此基础上，给定 *PRTG* φ，定义 $acc(\varphi, I)$ 为自然模型 $\mathcal{N}_\varphi^I$ 预测准确度：

$$acc(\varphi, I) \quad \coloneqq \quad \mathop{\mathbb{E}}_{p,p' \sim \varphi} \left[same\left(p, p', I, \mathcal{N}_\varphi^I\right) \right]$$

其中 $same(p, p', I, \mathcal{M})$ 表示事件 $\mathcal{E}(p, p', I, \mathcal{M})$ 发生的概率。

那么，对于任何输入 I，任何 *PRTG* φ 和任何产生式 r，令 φ^r 为在 φ 中展开产生式 r 的结果，那么 $acc(\varphi^r, I)$ 总是不小于 $acc(\varphi, I)$。

证明. 函数 *same* 的递归计算过程被事件 $\mathcal{E}$ 的定义导出，如下所示。

$$same(p, p', I, \mathcal{M}) \coloneqq \begin{cases} \left| cross_{\mathcal{M}}(r, r') + \left[[\![p]\!](I) = [\![p']\!](I) \right] - 1 \right| & r \neq r' \\ \left| w + (1-w) self_{\mathcal{M}}(r) + \left[[\![p]\!](I) = [\![p']\!](I) \right] - 1 \right| \prod_i same(p_i, p_i', I, \mathcal{M}) & r = r' \end{cases}$$

其中 w 对应了 `¬IsSubDiff(`p, p', I`)` 的结果，即 $\left[\bigwedge_i [\![p_i]\!](I) = [\![p_i']\!](I)\right]$。

考虑证明原定理的一个推广形式，如下所示：

$$\mathop{\mathbb{E}}_{p,p' \sim \varphi_s} \left[same(p, p', I, \mathcal{N}_\varphi^I) \right] \leq \mathop{\mathbb{E}}_{p,p' \sim \varphi_s^f} \left[same(p, p', I, \mathcal{N}_{\varphi^f}^I) \right] \tag{6.6}$$

其中 f 表示被展开的产生式，φ^f 是展开结果，s 是一个非终结符，φ_s 表示在 s 的程序空间上由 φ 导出的子分布。显然，原定理是上述形式在 s 取起始非终结符时的情况。

因为本章假设文法无环，所以非终结符间存在着一个拓扑排序 $s_1, \ldots, s_n$，它保证了文法中的每条产生式只会涉及更靠前的非终结符。本文通过对非终结符 s 在该拓扑排序中的下标作归纳的方式证明上述推广形式（公式 6.6）。

令 o 是产生式 f 所属的非终结符在拓扑序中的下标，s 是当前归纳步骤所考虑的非终结符，而 t 是 s 在拓扑序中的下标。

情况 1： $t < o$　此时 s 不会受展开的影响，所以公式 6.6 的左右两侧一定相等。

情况 2： $t = o$　令 R 是从非终结符 s 出发的产生式集合，R_0 是在从 R 中排除掉 f 后的集合。公式 6.6 的左侧公式可以被分解为如下三个部分。

$$\begin{aligned} \mathop{\mathbb{E}}_{p,p' \sim \varphi_s} \left[same(p, p', I, \mathcal{N}_\varphi^I) \right] &= \sum_{r,r' \in R} \gamma(r)\gamma(r') \mathop{\mathbb{E}}_{p \sim \varphi_r, p' \sim \varphi_{r'}} \left[same\left(p, p', I, \mathcal{N}_\varphi^I\right) \right] \\ &= \sum_{r,r' \in R_0} \gamma(r)\gamma(r') \mathop{\mathbb{E}}_{p \sim \varphi_r, p' \sim \varphi_{r'}} \left[same\left(p, p', I, \mathcal{N}_\varphi^I\right) \right] \\ &+ 2 \sum_{r \in R_0} \gamma(r)\gamma(f) \mathop{\mathbb{E}}_{p \sim \varphi_f, p' \sim \varphi_r} \left[same\left(p, p', I, \mathcal{N}_\varphi^I\right) \right] \\ &+ \gamma(f)^2 \mathop{\mathbb{E}}_{p,p' \sim \varphi_f} \left[same\left(p, p', I, \mathcal{N}_\varphi^I\right) \right] \end{aligned}$$

在这三个部分中，只有红色与绿色部分会受到展开操作的影响。于是我们只需要证明这两部分公式一定不大于它们在 φ^f 中的对应部分，如下所示，其中 R_f 表示由展开操

作引入的产生式集合，γ^f 表示 φ^f 中为每个产生式分配概率的函数。

$$\forall r \in R_0, \quad \gamma(f) \mathop{\mathbb{E}}_{\substack{p\sim\varphi_f \\ p'\sim\varphi_r}} \left[same\left(p,p',I,\mathcal{N}_\varphi^I\right)\right] \leq \sum_{f'\in R_f} \gamma^f(f') \mathop{\mathbb{E}}_{\substack{p\sim\varphi_{f'}^f \\ p'\sim\varphi_r^f}} \left[same\left(p,p',I,\mathcal{N}_{\varphi^f}^I\right)\right] \tag{6.7}$$

$$\gamma(f)^2 \mathop{\mathbb{E}}_{p,p'\sim\varphi_f} \left[same\left(p,p',I,\mathcal{N}_\varphi^I\right)\right] \leq \sum_{r,r'\in R_f} \gamma^f(r)\gamma^f(r') \mathop{\mathbb{E}}_{\substack{p\sim\varphi_r^f \\ p'\sim\varphi_{r'}^f}} \left[same\left(p,p',I,\mathcal{N}_{\varphi^f}^I\right)\right] \tag{6.8}$$

首先考虑公式 6.7。对于 R_0 中的任意一条产生式 r，定义辅助函数 $g_r(p)$ 如下：

$$g_r(p) \quad \coloneqq \quad \Pr_{p'\sim\varphi_r}\left[[\![p]\!](I) = [\![p']\!](I)\right]$$

此时，参数 $cross\left[\mathcal{N}_\varphi^I\right](f,r)$ 的取值可以被简化为 $\sum_{p\in P_f}\varphi(p)g_r(p)$。在此基础上，公式 6.7 的左侧可以被展开为如下形式，其中 P_f 表示产生式 f 在 φ 中的程序空间。

$$\begin{aligned}
&\gamma(f) \mathop{\mathbb{E}}_{p\sim\varphi_f,p'\sim\varphi_r} \left[same\left(p,p',I,\mathcal{N}_\varphi^I\right)\right] \\
=&\sum_{p\in P_f} \varphi(p)\left(g_r(p)cross\left[\mathcal{N}_\varphi^I\right](f,r) + (1-g_r(p))\left(1-cross\left[\mathcal{N}_\varphi^I\right](f,r)\right)\right) \\
=&1-2\sum_{p\in P_f}\varphi(p)g_r(p) + 2\left(\sum_{p\in P_f}\varphi(p)g_r(p)\right)^2
\end{aligned} \tag{6.9}$$

接着考虑公式 6.7 的右侧。对于任意被 f 展开的程序 $p\in P_f$，令 r_p 为它在 φ^f 中的对应产生式。因为 r_p 的程序空间只包含一个程序，所以不难验证 $cross\left[\mathcal{N}_{\varphi^f}^I\right](r,r_p) = g_r(p)$。利用这一结果，公式 6.7 的右侧可以被展开为如下形式。

$$\begin{aligned}
&\sum_{f'\in R_f}\gamma_f(f') \mathop{\mathbb{E}}_{p\sim\varphi_{f'}^f,p'\sim\varphi_r^f} \left[same\left(p,p',I,\mathcal{N}_{\varphi^f}^I\right)\right] \\
=&\sum_{p\in P_f}\varphi(p)\left(g_r(p)cross\left[\mathcal{N}_{\varphi^f}^I\right](r_p,r) + (1-g_r(p))\left(1-cross\left[\mathcal{N}_{\varphi^f}^I\right](r_p,r)\right)\right) \\
=&1-2\sum_{p\in P_f}\varphi(p)g_r(p) + 2\sum_{p\in P_f}\varphi(p)g_r(p)^2
\end{aligned} \tag{6.10}$$

根据 Cauchy-Buniakowsky-Schwarz 不等式，下列公式始终成立。

$$\left(\sum_{p\in P_f}\varphi(p)g_r(p)\right)^2 \leq \sum_{p\in P_f}\left(\varphi(p)g_r(p)\right)^2 \leq \sum_{p\in P_f}\varphi(p)g_r^2(p)$$

这意味着公式 6.9 不大于公式 6.10，于是公式 6.7 的左侧不大于其右侧，不等式成立。

而对于公式 6.8 中的不等式，其右侧可以通过如下方式化简。

$$\begin{aligned}
&\sum_{r,r'\in R_f} \gamma^f(r)\gamma^f(r') \mathop{\mathbb{E}}_{p\sim\varphi_r^f, p'\sim\varphi_{r'}^f} \left[same\left(p, p', I, \mathcal{N}_{\varphi^f}^I\right)\right] \\
=& \sum_{p,p'\in P_f} \varphi(p)\varphi(p') same\left(\square_p, \square_{p'}, I, \mathcal{N}_{\varphi^f}^I\right) \\
=& \sum_{p,p'\in P_f} \varphi(p)\varphi(p') \\
=& \gamma^2(f)
\end{aligned}$$

因为公式 6.7 的左侧一定不大于 $\gamma^2(f)$，所以其不等式一定成立。

到此，我们证明了归纳过程对于 $t = o$ 的情况成立。

情况 3： $t > o$　根据随机程序 p 和 p' 是否对应同一产生式，公式 6.6 的两侧均可以被分解为两部分。我们只需要证明该不等式左侧的两部分各自小于右侧的对应部分。

子情况 3.1：p 和 p' 对应着两个不同的产生式 r 和 r'。通过展开函数 *same* 的定义，此时的子目标可以被化简为如下形式：

$$\begin{aligned}
&\mathop{\mathbb{E}}_{p\sim\varphi_r, p'\sim\varphi_{r'}} \left[\left| cross\left[\mathcal{N}_\varphi^I\right](r, r') + \left[[\![p]\!](I) = [\![p']\!](I)\right] - 1\right|\right] \\
\leq& \mathop{\mathbb{E}}_{p\sim\varphi_r^f, p'\sim\varphi_{r'}^f} \left[\left| cross\left[\mathcal{N}_{\varphi^f}^I\right](r, r') + \left[[\![p]\!](I) = [\![p']\!](I)\right] - 1\right|\right]
\end{aligned}$$

因为展开操作符并不会改变程序集合以及每个程序的概率，所以参数 $cross\left[\mathcal{N}_\varphi^I\right](r, r')$ 与 $cross\left[\mathcal{N}_{\varphi^f}^I\right](r, r')$ 一定相等。因此上述不等式中的等号部分一定成立。

子情况 3.2：p 和 p' 对应着同一个产生式 r。同样通过展开函数 *same* 的定义，此时的子目标可以被化简为如下形式：

$$\begin{aligned}
&\mathop{\mathbb{E}}_{p,p'\sim\varphi_r} \left[\left| w + (1-w) self\left[\mathcal{N}_\varphi^I\right](r) + \left[[\![p]\!](I) = [\![p']\!](I)\right] - 1\right| \cdot \prod_i same\left(p_i, p_i', I, \mathcal{N}_\varphi^I\right)\right] \\
\leq& \mathop{\mathbb{E}}_{p,p'\sim\varphi_r^f} \left[\left| w + (1-w) self\left[\mathcal{N}_{\varphi^f}^I\right](r) + \left[[\![p]\!](I) = [\![p']\!](I)\right] - 1\right| \cdot \prod_i same\left(p_i, p_i', I, \mathcal{N}_{\varphi^f}^I\right)\right]
\end{aligned}$$

其中 w 对应了 $\neg$`IsSubDiff`(p, p', I) 的结果，即 $\left[\bigwedge_i [\![p_i]\!](I) = [\![p_i']\!](I)\right]$。

与上一个子情况类似，参数 $self\left[\mathcal{N}_\varphi^I\right](r)$ 与 $self\left[\mathcal{N}_{\varphi^f}^I\right](r)$ 一定相同。因此，上述公

式可以被进一步地化简为如下形式，其中非终结符 t_i 对应了产生式 r 的第 i 个参数。

$$\prod_i \mathop{\mathbb{E}}_{p,p'\sim\varphi_{t_i}} \left[same\left(p,p',I,\mathcal{N}_{\varphi}^{I}\right)\right] \leq \prod_i \mathop{\mathbb{E}}_{p,p'\sim\varphi_{t_i}^{f}} \left[same\left(p,p',I,\mathcal{N}_{\varphi^f}^{I}\right)\right]$$

根据归纳假设，该不等式左侧连乘中的每一项都小于等于右侧的对应项，因此不等关系一定成立。

综上所述，归纳过程对所有情况均成立，于是形成了对目标定理的证明。　□

6.5　对最少等价对策略的高效近似

使用统一等价模型，本文为最少等价对策略设计了一个可被高效计算的近似。

6.5.1　近似目标函数

在第 6.3.2 节中，本文介绍了最少等价对策略中的目标函数，并给出了该目标函数的一个只涉及等价性检查的形式（公式 6.2）。本文的近似目标函数在该形式的基础上，将每个等价性检查替换为对应的预测结果，例如等价性检查 $eq(p,p',I)$ 会被替换为 $\mathcal{M}_I(p,p')$，其中 $\mathcal{M}_I$ 表示针对输入 I 学习得到的统一等价模型。

定义 27 (近似目标函数)**.** 给定合成域 $(\mathbb{P},\mathbb{I})$，程序空间上的 *PRTG* φ，输入输出样例集合 E，以及满足所有样例的候选程序 p_c，近似目标函数 $appr[\mathbb{P},\varphi,E,p_c](I)$ 定义如下。

$$appr[\mathbb{P},\varphi,E,p_c](I) \coloneqq \sum_{p,p'\in\mathbb{P}} \varphi(p)\varphi(p')\mathcal{M}_I(p,p') \prod_{(I'\mapsto O')\in E} \left(\mathcal{M}_{I'}(p_c,p)\mathcal{M}_{I'}(p,p')\right)$$

其中 $\varphi(p)$ 表示程序 p 在分布 φ 中的概率，$\mathcal{M}_I$ 表示针对输入 I 的一个统一等价模型。

例 16. 考虑表 6.1 中的第二轮迭代。此时候选程序是 y，唯一存在的样例是 $\langle 0,2\rangle \mapsto 2$。于是在给定 *PRTG* φ 时，输入 $\langle 1,0\rangle$ 的近似目标值等于如下公式：

$$\sum_{p,p'\in G_e} \varphi(p)\varphi(p')\mathcal{M}_{\langle 1,0\rangle}(p,p')\mathcal{M}_{\langle 0,2\rangle}(y,p)\mathcal{M}_{\langle 0,2\rangle}(p,p')$$

6.5.2　近似目标函数的可组合性：两个案例分析

本文提出的近似目标函数具有可组合性。按照文法结构，其计算可以被递归地分解为子程序空间上的子任务，并且这些子任务上的结果可以合并得到完整空间上的结果。这一过程中不同的子任务数量很少，于是可以被动态规划算法高效地计算。

为了清晰地展示这一过程，本文将分析两个具体案例。在这些案例中，程序空间由第 6.2 节中的文法 G_e 指定，且 PRTG φ 为该文法上的均匀分布，其详细参数见例 13。

此外，为了方便，在这两个案例分析中，本文假设学习方法总是返回一个默认的统一等价模型 $\mathcal{M}$，其中所有参数的取值均为 1/2。

案例 1：第一次选择 在第一次样例选择时，因为不存在任何已有的样例，所以近似目标值可以被简化为 $\sum_{p,p'\in G_e}\varphi(p)\varphi(p')\mathcal{M}(p,p')$。为了计算这一表达式，本文根据 p 与 p' 使用的第一条产生式分类讨论，并将外层的求和展开为如下形式：

$$\varphi(1)\varphi(1)\mathcal{M}(1,1)+\cdots+\sum_{p_1+p_2,p_1'+p_2'\in G_e}\varphi(p_1+p_2)\varphi(p_1'+p_2')\mathcal{M}(p_1+p_2,p_1'+p_2')$$

其中第一项和最后一项分别对应 p 和 p' 从产生式 $S\to 1$ 和 $S\to T+T$ 展开的情况。

本文会计算展开后的每一项，再将结果加总。每一项的计算过程都分三步进行：首先展开求和，再提取子程序空间上的子任务并递归求解，最后合并子结果。以最后一项为例，其展开阶段分为四个步骤：

(1) 原先的求和范围是所有形如 p_1+p_2 与 $p_1'+p_2'$ 的完整程序，现将其拆分为关于子程序 p_1,p_2,p_1',p_2' 在非终结符 T 上的求和。

(2) 根据 PRTG 的定义，将每个程序的概率展开到关于子程序概率的表达式，例如 $\varphi(p_1+p_2)$ 的展开结果为 $\gamma(S\to T+T)\varphi(p_1)\varphi(p_2)$。

(3) 根据统一等价模型的定义，将等价性预测展开为关于子程序的预测。

这三个步骤后的结果如下所示，其中统一等价模型中的每个参数都被替换为了 1/2。

$$\sum_{p_1,p_2,p_1',p_2'\in T}\frac{9}{16}\varphi(p_1)\varphi(p_2)\varphi(p_1')\varphi(p_2')\left(\frac{1}{2}+\frac{1}{2}\mathcal{M}(p_1,p_1')\mathcal{M}(p_2,p_2')\right)$$

(4) 最后，将内层的加法（红色部分）展开为两个独立的求和，并按照如下方式重新组织。为简单起见，本文在此处忽略了一个等于 9/32 的全局系数。

$$S_1+S_2\quad\textbf{where } S_1:=\sum_{p_1,p_2,p_1',p_2'\in T}\varphi(p_1)\varphi(p_2)\varphi(p_1')\varphi(p_2')$$
$$S_2:=\left(\sum_{p_1,p_1'\in T}\varphi(p_1)\varphi(p_1')\mathcal{M}(p_1,p_1')\right)\left(\sum_{p_2,p_2'\in T}\varphi(p_2)\varphi(p_2')\mathcal{M}(p_2,p_2')\right)\tag{6.11}$$

根据 PRTG 的定义，S_1 的结果必然是 1。而 S_2 由两个与原始任务形式相同的子求和组成：每个子求和对应着一个非终结符，目标是对一个子程序空间计算近似目标值。因此，这些子求和是原始任务的子任务，可以被递归地计算。与此同时，因为这两个子任务对应相同的非终结符，所以我们可以只处理其中的一个任务并将结果复用。

案例 2：第二次选择　作为对案例 1 的补充，考虑一个更一般的情况，其中候选程序为 y，并存在着一个现有样例 $\langle 0,2\rangle \mapsto 2$。此时，输入 $\langle 1,0\rangle$ 上的近似目标值如下所示：

$$\sum_{p,p'\in G_e} \varphi(p)\varphi(p')\cdot \mathcal{M}_{\langle 1,0\rangle}(p,p')\cdot \mathcal{M}_{\langle 0,2\rangle}(y,p)\cdot \mathcal{M}_{\langle 0,2\rangle}(p,p')$$

与案例 1 类似，本文先将外层的求和按照产生式拆分，考虑对应产生式 $S \to T+T$ 的项，并对它展开。在经过展开的前三个步骤后，结果如下所示[①]：

$$\sum_{p_1,p_2,p_1',p_2'\in T} \varphi(p_1)\varphi(p_2)\varphi(p_1')\varphi(p_2') \left(\begin{array}{c} \left(1+\mathcal{M}_{\langle 1,0\rangle}(p_1,p_1')\mathcal{M}_{\langle 1,0\rangle}(p_2,p_2')\right) \\ \times \\ \left(1+\mathcal{M}_{\langle 0,2\rangle}(p_1,p_1')\mathcal{M}_{\langle 0,2\rangle}(p_2,p_2')\right) \end{array} \right)$$

此时，展开的最后一个步骤会将内部乘积（红色部分）通过分配率展开，得到四个子求和：一个不涉及任何模型，两个分别只涉及 $\mathcal{M}_{\langle 1,0\rangle}$ 与 $\mathcal{M}_{\langle 0,2\rangle}$，而最后一个同时涉及两个模型。这四个子求和都具有相同的形式，如下所示。其中 $\overline{\mathcal{M}}$ 表示涉及的模型列表，它在四个子求和中的取值分别为 $[], [\mathcal{M}_{\langle 1,0\rangle}], [\mathcal{M}_{\langle 0,2\rangle}]$ 和 $[\mathcal{M}_{\langle 1,0\rangle}, \mathcal{M}_{\langle 0,2\rangle}]$。

$$\left(\sum_{p_1,p_1'\in T} \varphi(p_1)\varphi(p_1') \prod_{\mathcal{M}\in\overline{\mathcal{M}}} \mathcal{M}(p_1,p_1')\right)\left(\sum_{p_2,p_2'\in T} \varphi(p_2)\varphi(p_2') \prod_{\mathcal{M}\in\overline{\mathcal{M}}} \mathcal{M}(p_2,p_2')\right) \tag{6.12}$$

通过扩展子任务的定义，我们可以统一上述公式中的两部分以及原始任务的形式。此时，每个子任务的参数包含一个非终结符 T、一个可选的候选程序 p_c 和一系列在具体程序 p_c 和两个程序变量 p, p' 之间的等价性预测；其目标是对子程序空间 T 中的所有程序对 (p,p')，计算所有等价性预测的乘积，并按照分布加权求和。在这一定义下，公式 6.12 中的两部分均为原始问题的子问题，可以被递归地求解。

6.5.3　计算近似目标函数的动态规划算法

通过推广上一节中的案例分析，可以得到计算近似目标函数的动态规划算法。

子任务　给定一个 PRTG φ，该算法中的每个子任务由一个非终结符 s、一个具体程序 p_c 和两个模型列表 $\overline{\mathcal{M}}_p$ 和 $\overline{\mathcal{M}}_c$ 指定。其目标是对子程序空间 s 中的所有程序对 (p,p') 求和，每对的贡献等于以下几项的乘积：(1) p 和 p' 的概率，(2) $\overline{\mathcal{M}}_p$ 中所有模型对 (p,p') 的等价性预测，以及 (3)$\overline{\mathcal{M}}_c$ 中所有模型对 (p_c,p) 的等价性预测。形式化地，该子任务

① 此处本文忽略了一个等于 9/128 的全局系数。

的目标是计算如下公式。

$$\sum_{p,p' \in s} \varphi(p)\varphi(p') \left(\prod_{\mathcal{M} \in \overline{\mathcal{M}}_p} \mathcal{M}(p, p') \right) \left(\prod_{\mathcal{M} \in \overline{\mathcal{M}}_c} \mathcal{M}(p_c, p) \right)$$

特别地，当 p_c 从未使用时（即 $\overline{\mathcal{M}}_c = []$ 时），本文将 p_c 记作缺省符号 $\perp$。

例 17. 让我们重新考虑前文中讨论过的任务，并将它们对应到上述子问题的定义。为了方便，本文将使用临时符号 $\mathcal{S}(s, p_c, \overline{\mathcal{M}}_p, \overline{\mathcal{M}}_c)$ 表示一个子任务。

- 上一节中第一个案例以及公式 6.11 中的子求和分别对应如下的子任务。

$$\mathcal{S}(\boldsymbol{S}, \perp, [\mathcal{M}], []) \qquad \mathcal{S}(\boldsymbol{T}, \perp, [\mathcal{M}], [])$$

- 上一节中第二个案例以及公式 6.12 中的子求和分别对应如下的子任务。

$$\mathcal{S}(\boldsymbol{S}, y, [\mathcal{M}_{\langle 1,0 \rangle}, \mathcal{M}_{\langle 0,2 \rangle}], [\mathcal{M}_{\langle 0,2 \rangle}]) \qquad \mathcal{S}(\boldsymbol{T}, \perp, \overline{\mathcal{M}}, [])$$

- 给定样例集合 $\boldsymbol{E}$ 与候选程序 p_c，输入 $\boldsymbol{I}$ 的近似目标值对应如下所示的子任务，其中 s_0 表示起始非终结符，$\mathcal{M}_I$ 表示输入 $\boldsymbol{I}$ 上的统一等价模型。

$$\mathcal{S}\left(s_0, p_c, [\mathcal{M}_I] +\!\!+ \overline{\mathcal{M}_E}, \overline{\mathcal{M}_E}\right) \quad \textbf{\textit{where}}\ \overline{\mathcal{M}_E} \coloneqq \left[\mathcal{M}_{I'} \mid (\boldsymbol{I}' \mapsto \boldsymbol{O}') \in \boldsymbol{E}\right]$$

计算过程 动态规划算法的计算过程与上一节中的案例分析几乎一致，所以本文在这里只作一个总结。对于每个子任务，本文会按照如下步骤进行计算，并记忆化所有已知的子任务结果。

1. 如果当前子任务的结果已知，则直接返回。
2. 根据 p 和 p' 使用的第一条展开式将求和拆分成若干项。
3. 对于每一项，将完整程序上的求和展开到子程序上的求和，根据 PRTG 的定义展开程序概率，并通过乘法分配律展开内部乘法。
4. 将所有子求和对应到子任务，并递归地求解。
5. 将子结果合并到完整的结果，将其存储到记忆表中，然后返回。

时间复杂度 动态规划的时间复杂度取决于子任务的数量与处理子任务的时间成本。

- **（子任务数量）**上述算法保证在每个子任务中，具体程序一定是初始的具体程序的子程序，并且两个模型列表都是原始列表的子列表。
- **（处理子任务的时间成本）**在处理子任务时，对求和的展开操作以及对子任务的合并操作都可以通过高维前缀和算法 [141] 高效地实现。

结合这两个因素，上述动态规划算法的时间复杂度是 $O(mn_r^2 s_p 2^m)$，其中 m 表示原始模型列表的总大小，n_r 表示产生式的数量，s_p 表示初始的具体程序的大小。因此，该算法的时间成本关于产生式数量是多项式的，它通常远小于程序空间的大小。

值得注意的是，在某些极端情况下，现有样例的数量可能非常大，这将导致上述算法需要同时处理大量的模型，从而产生指数级的计算开销。为了确保在这些情况下的计算效率，本文为样例数量设置了一个上限 lim_e，并在计算近似目标值时只考虑最新的 lim_e 个样例。这样的截断通常不会对近似结果产生太大影响，因为被忽略的样例在先前的样例选择问题中已经被重复考虑了许多次。

6.6 实验评估

为了检验 LearnSy 的有效性，本文设计了实验来回答以下问题：

- **RQ1**: LearnSy 是否能提升样例编程求解器的合成效率？
- **RQ2**: 相比于现有的样例选择方法，LearnSy 选择的样例质量如何？

6.6.1 代码实现

本文基于 SyGuS 框架 [28] 将 LearnSy 实现为了一个通用的样例选择方法。它接受一个 SyGuS 格式的程序合成问题、一系列输入输出样例、以及一个候选程序为输入，并根据近似后的的最小等价对策略返回一个新的样例。

候选输入　在选择样例前，LearnSy 会先生成一组候选输入（算法 7）。因为 SyGuS 框架中的程序规约总是以 SMT 公式的形式给出，所以本文使用 SMT 求解器产生的反例作为候选输入。具体而言，给定候选程序 p_c 与程序规约 $\varphi(p, I)$，在已有的候选输入集合 $\overline{I}$ 的基础上，LearnSy 会通过求解如下的 SMT 问题产生下一个候选输入：

$$\exists I \in \mathbb{I}. \qquad \neg\varphi(p_c, I) \wedge I \notin \overline{I}$$

本文使用 Z3 [138] 作为默认的 SMT 求解器，并将候选输入数量的上界设为了 5。

程序空间上的概率分布　本文的近似依赖于程序空间上的一个由 PRTG 定义的概率分布 φ。为了得到这一分布，本文采用了与现有研究相同的 PRTG 学习方法 [40,41]。给定一个 RTG 与一系列样本程序，该方法会统计每个非终结符与每条产生式的使用数量，并将产生式的概率设置为它与其非终结符的使用次数比值。

在实验评估中，本文通过二折叠交叉验证收集样本程序。每次实验中，本文会将数据集随机分为两半，以其中一半任务的目标程序作为样本学习 PRTG，并使用该 PRTG 求解另一半数据集中的任务。

展开操作 在学习模型前，LearnSy 会先迭代地应用展开操作以提高统一等价模型的准确度。它会不断地应用展开操作，每次贪心地选取包含程序数量最少的产生式，直到新引入的产生式总数超过一个预先设定的阈值 lim_f。该阈值的取值决定了 LearnSy 如何在精度和效率之间权衡。lim_f 越大，展开的产生式数量越多，统一等价模型的精度也就越高；但同时，展开后文法的规模也会越大，样例选择的时间开销也会越高。因为本章的目标是加速程序合成，所以本文将 lim_f 的默认值设为 100，一个较小的取值。

其他参数 在实现中，本文将学习模型时的样本程序数量（即算法 8 中的参数 n_s）设置为 10，并将计算近似目标值时的样例上限（即第 6.5.3 节中的参数 lim_e）设置为 3。

6.6.2 RQ1: LearnSy 对样例编程的加速能力

在本实验中，本文考虑了包括 PolyGen 在内的三种样例编程求解器，并评估了 LearnSy 对这些求解器的加速能力。

样例编程求解器 为了综合评估 LearnSy 的加速能力，本文考虑了三个基于不同技术路线的样例编程求解器，分别是基于枚举与合一化的求解器 EUSolver [38]，基于变形代数空间的求解器 MaxFlash [33]，与本文在第五章中提出的奥卡姆求解器 PolyGen。

基准方法 首先，本文考虑了 CEGIS 框架中的默认选择方法，记作 Default。该方法对所选输入没有偏好，总是会使用 SMT 求解器返回的第一个反例。

此外，一些现有的样例编程研究 [30,142] 针对特定领域提出了一些启发式的样例选择方法，本文同样将它们作为基准方法：

- **特殊输入法** [142]（记为 SigInp）是一个针对字符串处理的样例选择方法。它先根据字符串结构将所有可选的输入分类，再在每次样例选择时优先选择与现有样例类别不同的输入。
- **低位优先法** [30]（记作 Biased）是一个针对布尔向量的样例选择方法。它基于以下结论：对于大部分布尔向量上的函数，输入中的低位相比于高位对输出的影响更大 [143]。于是，该方法会优先选择与现有样例具有不同低位的输入。

数据集 本文考虑了 SyGuS-Comp 中的三个不同领域上的数据集 $\mathcal{C}_S$、$\mathcal{C}_I$ 和 $\mathcal{C}_B$。它们的具体信息如表 6.3 所示。

其他配置 对于每次运行，本文将时间限制设为了 120 秒，并将内存限制设置为了 8GB。

表 6.3　本文在 RQ1 中考虑的 SyGuS 数据集。

名称	大小	合成域	运算符数量	适用的样例编程求解器	适用的样例选择方法
$\mathcal{C}_S$	205	字符串	16	EUSolver、MaxFlash	Default、SigInp
$\mathcal{C}_I$	100	整数	10	EUSolver、PolyGen	Default
$\mathcal{C}_B$	450	位向量	10	EUSolver	Default、Biased

表 6.4　LearnSy 在加速样例编程方面的表现。

	求解器	选择方法	求解数量	使用样例数			时间开销		
				平均值	比较值		平均值	比较值	
$\mathcal{C}_S$	EUSolver	LearnSy	**146**	2.69			**14.4**		
		Default	145	2.82	0.13	4.56%	15.2	0.83	5.46%
		SigInp	**146**	**2.61**	−0.08	−3.24%	14.9	0.47	3.15%
	MaxFlash	LearnSy	**159**	**1.42**			**0.96**		
		Default	154	1.51	0.09	6.29%	1.41	0.45	31.9%
		SigInp	155	1.47	0.06	4.04%	1.57	0.61	38.8%
$\mathcal{C}_I$	EUSolver	LearnSy	**43**	**15.1**			**9.26**		
		Default	42	21.0	5.92	28.2%	12.3	3.06	24.9%
	PolyGen	LearnSy	**82**	**34.4**			**7.27**		
		Default	78	47.8	13.4	28.0%	12.7	5.48	43.0%
$\mathcal{C}_B$	EUSolver	LearnSy	**427**	**10.2**			**24.2**		
		Default	424	11.2	0.91	8.16%	27.5	3.32	12.1%
		Biased	424	10.9	0.69	6.34%	25.0	0.75	2.99%

实验结果　表 6.4 展示了本实验的结果。对于每个数据集、每个样例编程求解器和每个询问选择方法，“求解数量” 报告了成功求解的任务数量；“使用样例数” 报告了平均使用的样例数量，以及基准方法相比于 LearnSy 多使用的样例数量与比例；“时间开销” 报告了平均花费的合成时间，以及基准方法相比于 LearnSy 多花费的时间与比例。

根据这些结果，本文做出了如下观察。首先，与 CEGIS 的默认选择方法 Default 相比，LearnSy 总是可以进一步减少样例使用，并使样例编程方法变得更高效。特别地，LearnSy 可以将 PolyGen 的合成效率提升 43%。

其次，与启发式方法 Biased 和 SigInp 相比，LearnSy 在减少样例使用方面的能力相当，并且总是能够带来更多加速。这一结果进一步验证了 LearnSy 的有效性，因为 LearnSy 远比这些方法更通用。此外，在与 EUSolver 结合时，尽管 LearnSy 使用的样例数量略多于 SigInp，它带来的加速却显著更多。这是因为 SigInp 在样例选择时需要求解一个独立的合成任务，因此具有较大的时间开销；相比之下，LearnSy 在每一轮的平均时间开销仅为 0.03 秒，可以忽略不计。

表 6.5 LearnSy 在交互式程序合成中的表现。

选择方法	求解数量	使用样例数		
		平均值	比较值	
LearnSy	145	**3.245**		
RandomSy		3.756	0.510	13.58%
SampleSy		3.251	0.005	0.176%

6.6.3 RQ2: LearnSy 在提升样例质量方面的有效性

在本实验中，本文将 LearnSy 集成进了现有的交互式程序合成框架[45]，并将它和交互式程序合成中的样例选择方法进行了比较。

基准方法 本文考虑了现有样例选择方法。

- RandomSy [144] 对所选输入没有明确偏好。它不断地产生随机输入，直到当前输入能至少区分两个剩余程序为止。
- SampleSy [45] 目前在减少样例使用方面最先进的样例选择方法。它通过对剩余程序采样来近似最少等价对策略。

数据集 本文直接使用了现有交互式程序合成工作中的数据集 [45]。该数据集涉及两个合成域，分别包含了 150 个字符串处理领域的任务与 16 个程序修复领域的任务。

实验配置 对于每次运行，本文将总时间限制设为了 60 分钟，将内存限制设为了 8GB，并根据 SampleSy 的默认设置，为交互式程序合成的每一轮都设置了 2 分钟的时间上限——这对应了用户回答每个询问时的时间开销。此外，因为在交互式场景下样例选择的效率并不重要，于是本文将 LearnSy 中展开操作的阈值 lim_f 增加到了 3000，从而让其在时间允许的范围内提供尽可能高质量的样例。

实验结果 表 6.4 展示了本实验的结果，其中“求解数量”报告了成功求解的任务数量；而“使用样例数”报告了平均使用的样例数量，以及基准方法相比于 LearnSy 多使用的样例数量与比例。

实验结果表明，LearnSy 在减少样例使用方面的性能优于 RandomSy，并与 SampleSy 的表现相当。值得注意的是，LearnSy 的时间开销远小于 SampleSy。在每次样例选择时，LearnSy 的平均耗时仅为 0.91 秒，而 SampleSy 则需要两分钟：后者的时间成本达到了前者的百余倍。这表明 LearnSy 在大幅提升样例选择效率的同时，仍然可以提供与最先进的方法相近的样例质量。

6.7　小结

本章的目标是通过选择高质量的样例来减少程序合成时的样例使用数量，从而进一步提高程序合成的效率。这一目标与交互式程序合成中的样例选择问题十分接近，因此本文在已有工作的基础上展开了研究。现有工作已经将这一问题近似为一个关于输入的优化问题，其中一个理想的输入应当使剩余程序的输出尽可能不同，并在此优化问题的基础上构建了有效的样例选择方法。

然而，相比于交互式程序合成，本文中加速程序合成的目标为询问选择带来了更高的效率要求，使得现有方法不再适用。为了提升效率，本文提出了基于统一等价模型的近似方法。该方法通过用简单的随机过程模拟操作符的具体行为，避免了程序等价性判断中的复杂语义，从而得到了一个可以高效计算的近似目标函数。在此基础上，本文为统一等价模型提出了一种高效的基于采样的学习方法，证明了其理论性质；同时，为了进一步提升统一等价模型的预测精度，本文提出了程序空间上的展开操作，用于将程序空间转换为更容易被统一等价模型预测的形式。

本文的实验评估表明，LearnSy 可以成功地在三个不同的合成域上加速三种基于不同技术路线的程序合成方法。此外，本文还在交互式程序合成的场景下，将 LearnSy 与现有的询问选择方法进行了比较。结果同样证明了 LearnSy 的有效性：它能够在显著更小的时间开销下提供相近的样例质量。

第七章　系统实现与整体评估

本文将前文中的所有技术集成并实现为面向算法问题的程序合成系统 SuFu，并对该系统进行了实验评估。具体而言，本文设计了实验来回答以下研究问题：

- **RQ1**：SuFu 是否可以有效地求解算法问题?
- **RQ2**：相比于现有的程序合成方法，SuFu 对算法问题的求解能力如何?

7.1　系统实现

SuFu 集成了前文中的各项技术，包括 SuFu 语言（第三章）、分解系统 AutoLifter（第四章）、样例编程求解器 PolyGen（第五章）与询问选择方法 LearnSy（第六章）。图 7.1 展示了 SuFu 对算法问题的求解过程。它接受被 SuFu 描述的算法问题，并会通过如下步骤求解。

1. 使用 SuFu 语言中的自动规约方法将原问题规约到提升问题。
2. 使用 AutoLifter 将提升问题分解为子问题，并合成一个合适的表征函数 *repr*。
3. 将 PolyGen 与 LearnSy 按照 CEGIS 框架组合成程序合成器，并用它依次求解所有与合并算子相关的子问题。
4. 将合成结果填入最初的 SuFu 程序，从而得到对算法问题的合成结果。

本文已经将 SuFu 的代码实现在 GitHub 上开源①，并维护了一个可用的线上演示②。本节的剩余部分将介绍实现过程中的参数细节。

程序空间　SuFu 需要两个程序空间来分别定义表征函数与合并算子的搜索范围。在本文的实现中，这两个程序空间各由一些操作符指定：它们包含了由这些操作符构成的所有类型正确的程序。

- 表征函数的操作符包括参考程序中定义的所有函数，DeepCoder 引入的列表函数（表 4.2），以及所有相关数据结构的折叠态射，即第 2.1 节中定义的 $(\!|f|\!)$。这些折叠态射使得 SuFu 能够按需合成新的递归函数。
- 合并算子的操作符包括 SyGyS-Comp 中定义的整数运算操作符（见 表 4.3）。此外，为了支持元组与数据结构的操作，本文还引入了元组的构造函数、投影操作以及数据结构上的模式匹配运算符。其中，模式匹配运算符使 SuFu 能够按需从数据结构中提取所需的信息。例如，下图展示了列表上的模式匹配运算符，

① https://github.com/jiry17/sufu
② http://8.140.207.65/

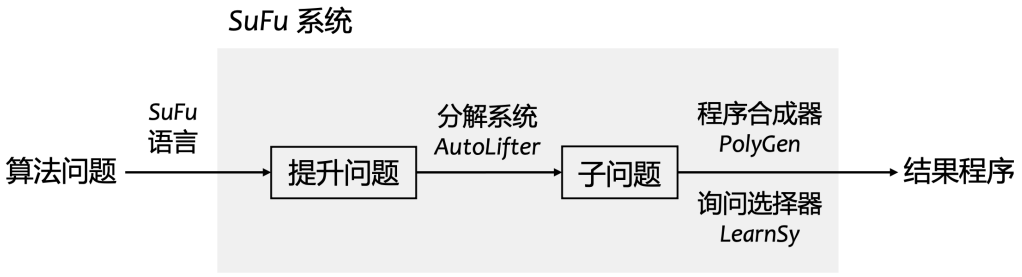

图 7.1　SuFu 对算法问题的求解流程。

它允许 SuFu 灵活地处理列表内容：

```
match ? with Nil -> ? | Cons(h,t) -> ?
```

随机输入生成器　在将算法问题归约到提升问题时，SuFu 需要生成一系列随机输入（第 3.4 节）。为了控制参考程序在随机输入上的运行时间，本文只考虑大小不超过 10 且整数范围在 $[-5,5]$ 内的输入，并从中均匀地采样至多 10^3 个用于归约。

此外，因为随机输入并不足以覆盖完整的输入空间，所以 SuFu 在其最外层引入了 CEGIS 框架用于保证合成结果的正确性。具体地，在得到合成结果后，SuFu 会验证当前结果与参考程序的等价性。如果这两个程序不等价，则 SuFu 会找到一个能够区分它们的输入，将该输入加入到用于归约的输入集合，并重新合成。与现有工作类似 [19,43,145]，本文在实现中使用有界验证检查合成结果的正确性。

其他配置　实现中的其他参数均与前几章实现中的默认参数一致。

7.2　实验设置

数据集　本文使用第 3.5 节中构建的数据集评估 SuFu 的有效性。该数据集包含 290 个算法问题，收集自三个不同的来源：融合问题 [49,65,118–120]、递归合成问题 [22] 以及其他算法模式上的合成问题 [10,12,13,23,111,121,122]。该数据集具有显著的多样性。它涉及 8 种不同的算法模式，使用了包括列表、二叉树、抽象语法树在内的 37 种数据结构，并涵盖了以下类型的算法问题：

- 教科书中的经典算法习题，例如快速检查一个括号序列是否匹配。
- 传统问题上的经典算法设计，例如最大子段和问题中的 Kadane 算法 [146]，以及内存分配中最坏适应法的高效实现 [147]。
- 算法竞赛中的高难度真题，例如 2021 年 Petrozavodsk 冬令营中的一道竞赛题。该训练营的参赛队伍代表了世界算法竞赛的最高水平，但在 243 支队伍中，仅

表 7.1　本章数据集上的统计信息。

来源	任务数	需要辅助值的任务数量		程序大小		Packed 数量	
				均值	最大值	均值	最大值
融合	16	5	31%	126.5	305	1.3	2
递归	178	49	28%	157.5	484	1.1	3
其他	96	86	90%	251.2	843	1.0	1
合计	290	140	48%	186.8	843	1.1	3

有 26 支在 5 小时的竞赛中成功解出了这道题目。

在第 3.5.2 节中，本文用 SuFu 语言描述了该数据集中的全部问题。于是在实验评估中，本文将用这些描述程序作为 SuFu 的输入，并在此基础上评估 SuFu 的性能。

表 7.1 列举了该数据集上的统计信息，其中“任务数”展示了每个来源的算法问题数量；“需要辅助值的任务数量”展示了需要创造辅助值的任务数量与比例——例如在将列表分治用于计算次小值时就需要引入列表最小值作为辅助值——该指标可以侧面反映算法问题的难度；“程序大小”列展示了参考程序中的 AST 节点数量，而“Packed 标注数量”展示了输入程序中的 Packed 关键字数量，它侧面展示了描述程序对中间数据结构操作的复杂程度。

基准方法　本文首先考虑了三个目前最先进的、针对不同算法模式的程序合成方法。

- Synduce [22] 是一个针对结构递归的程序合成方法。它的合成过程依赖于一个名为**部分程序展开**的演绎推理技术。由于这一技术的局限性，Synduce 限制了参考程序的形式，要求它只能恰好是两个结构递归的复合。
- Parsynt [10,25] 是一个针对分治的程序合成方法。它先使用一个特化的演绎推理系统从参考程序中抽取必要的辅助值，然后再使用归纳程序合成生成合并算子。受限于推理系统表达能力，Parsynt 要求参考程序符合流算法的形式。
- DPASyn [13] 是一个针对流算法的程序合成方法。它将流算法上的算法问题规约到了一个草图问题 [43] 并为其设计了一个特化的草图求解方法。

SuFu 的通用性显著优于这些基准方法。它可以被直接用于求解上述所有算法模式上的问题，而三个基准方法却无法泛化到其他算法模式上，因为它们均高度依赖于各自算法模式的特定性质。

其次，本文还考虑了一个草图求解方法 [148]。具体而言，算法问题可以被直接视为一个草图问题 [43]，其目标是填补算法模板中的空缺，并保证结果程序与参考程序语义等价。于是，本文基于目前最先进的草图问题求解器 Grisette [148] 实现了一个面向算法问题的程序合成方法，记作 Sketch，并将它作为基准方法。

表 7.2　SuFu 在完整数据集上的表现。

来源	求解数量		时间开销	合成结果大小	
				repr	*comb*
融合	14/16	88%	6.9	9.1	25.0
递归	170/178	96%	14.3	7.5	25.4
其他	80/96	83%	49.0	11.6	92.4
合计	264/290	91%	24.4	8.8	52.1

值得注意的是，本文在实验中没有考虑传统的演绎推理系统。据我们所知，所有现有的自动推理系统（包括最新的工作[105]）均无法创造原程序中不存在的辅助值，但这些辅助值对求解复杂算法问题是至关重要的。如表 4.1 所示，本文的数据集中有 48% 的任务需要创造额外的辅助值，这意味着现有的演绎推理系统最多只能求解 52% 的任务，能力十分有限。

其他配置　本文为每次求解设置了 10 分钟的时间限制与 8GB 的空间限制。

7.3　RQ1: SuFu 在求解算法问题时的有效性

表 7.2 展示了 SuFu 在本章数据集上的表现，包括它成功求解的任务数量，平均使用的合成时间（秒），以及合成结果中表征函数与合并算子的平均大小。总体上，SuFu 成功解决了数据集中 290 个任务中的 264 个（91%），平均耗时约 24 秒。经过统计，这 264 个任务中的 115 个都需要引入额外的辅助值，这证明了 SuFu 在求解复杂算法问题时的有效性，也展示了 SuFu 相比于传统演绎推理系统的优势。

合成结果的质量　由于本文实现中的验证器（有界验证）不能保证完全正确性，所以本文手动检查了 SuFu 的所有合成结果，并确认了它们与参考程序完全等价。

此外，本文还分析了所有合成结果的运行效率。

- 对于递归与其他算法模式上的任务，因为合并算子的程序空间中只包含常数时间的表达式，所以 SuFu 结果的时间复杂度具有理论保障，一定高效。
- 对于融合任务，本文手动检查了 SuFu 生成的 14 个程序，并将它们与原始论文中的融合结果比较。结果表明，SuFu 在其中 13 个任务上生成了与原始论文完全一致的程序，并为最后一个任务合成了效率更高的结果：它将列表 $[l,\dots,r]$ 上的求和计算融合为了常数时间的表达式 $(l+r)(r-l+1)/2$，消除了所有递归。

结果展示　表 7.3 展示了 SuFu 合成结果中的两个片段，其中用 红色 标注的部分是由 SuFu 生成的程序片段。第一个例子展示了 SuFu 的合成能力，它为该任务生成了一个

表 7.3　SuFu 的合成结果展示。

任务描述	程序片段

将分治用于计算一个整数列表（可能包含负数）的最大后缀乘积。	`( if r.1 == r.3 then l.1 * r.1`
	`  else if l.2 * r.3 > r.1 then l.2 * r.3 else r.1,`
	`if r.3 < 0 then l.1 * r.2`
	`  else if l.2 * r.3 < r.2 then l.2 * r.3 else r.2,`
	`l.3 * r.3 )`

给定两种布尔表达式语言 $\mathcal{L}_1$ 和 $\mathcal{L}_2$，从 $\mathcal{L}_2$ 的解释器合成 $\mathcal{L}_1$ 的解释器。	`| Lit b -> b`
	`| Neglit b -> not b`
	`| And (e1, e2) -> (eval e1) && (eval e2)`
	`| Or (e1, e2) -> (eval e1) || (eval e2)`

表 7.4　SuFu 与现有程序合成方法的比较结果。

子数据集	合成方法	求解数量	时间开销	子数据集	合成方法	求解数量	时间开销
递归	SuFu	**170**/178	11.4	分治	SuFu	**29**/36	28.7
	Synduce	125/178	**1.7**		Parsynt	24/36	**15.6**
流算法	SuFu	**33**/39	**9.2**	全部	SuFu	**264**/290	**11.1**
	DPASyn	15/39	10.3		Sketch	85/290	28.9

复杂的合并算子，而类似规模的合并算子在本文的数据集中普遍存在。第二个例子展示了 SuFu 对不同数据结构的支持能力，能够处理包括程序语法树在内的复杂结构。

7.4　RQ2: SuFu 与现有程序合成方法间的比较

表 7.4 展示了 SuFu 与现有程序合成方法的比较结果，其中子数据集“分治”与“流算法”分别包含了“其他”来源中关于分治和流算法的算法问题，而“时间开销”汇报了 SuFu 与基准方法的平均时间开销①。

与 Synduce 相比，SuFu 需要花费更多的时间，但能解出更多的任务。经过分析，本文发现 SuFu 在需要创造辅助值的任务上具有明显的优势。尽管 Synduce 支持发现新的辅助值，但受限于其推理系统的表达能力，它能够发现的辅助值范围有限；相比之下，SuFu 能从程序空间中搜索合适的表征函数，具备找到复杂辅助值的能力。

类似地，在与 Parsynt 的比较中，SuFu 的时间开销更大，但能解出更多任务。这是由于 Parsynt 对参考程序的形式要求为它提供了大量的额外信息。具体而言，Parsynt 要求参考程序符合流算法的形式，而许多流算法的实现也需要依赖额外辅助值的支持，例如计算列表次小值的流算法仍然需要列表最小值的帮助。Parsynt 可以直接从输入中获取这些辅助值，其比例约占所有辅助值的 40%。相比之下，SuFu 对输入的限制更少，获取的信息也更少，但仍然能够解决更多的问题，并实现相近的合成效率。

① 在计算平均开销时，本文只考虑那些同时被 SuFu 与基准方法解出的任务。

最后，SuFu 在与 DPASyn 和 Sketch 的比较中展现出显著优势。SuFu 能解决更多的问题，并且在共同解出的任务上，其合成效率与 DPASyn 相当，显著优于 Sketch。

第八章　结论与展望

8.1　本文工作小结

软件运行时的高效性作为软件开发的核心追求之一，一直以来受到广泛关注。为了指导高效软件的设计，过去的研究提出了诸多高效的算法模式（如分治算法、流算法、动态规划等）。然而，这些模式并不能被轻易地应用于实际开发中。一方面，由于大多数开发人员缺乏足够的算法能力，算法优化只能由少数专家在部分模块中进行；另一方面，算法层面的优化往往会大幅增加代码复杂度，带来代码缺陷的风险。

为了解决这些问题，本文的目标是提出一个程序合成系统，用于自动合成符合算法模式的程序。这样的自动化系统可以有效降低应用算法模式的难度，并提升应用结果的可靠性。然而，这样的合成系统面临着通用性和合成效率上的挑战。一方面，不同的算法模式在形式上存在巨大差异，因此一个有效的合成系统需要具备处理这些不同形式的通用能力；另一方面，高效算法通常规模庞大、结构复杂，这对程序合成的效率提出了极高的要求。

面对这些挑战，本文通过四个主要创新兼顾了通用性与合成的高效性。首先，为了解决通用性挑战，本文提出了 SuFu 语言，用于将不同的算法问题规约为形式固定的提升问题，从而消除了算法模式之间的形式差异。

接着，本文分三步实现了对提升问题的高效求解：

- 提出了分解系统 AutoLifter，用于将提升问题分解为规模更小的子问题。
- 提出了高效且具有泛化能力保障的程序合成方法 PolyGen，用于求解子问题。
- 提出了高效的样例选择方法 LearnSy，用于为 PolyGen 提供更高质量的样例，从而进一步提升合成效率。

本文将上述创新集成为了一个程序合成系统 SuFu 并对其进行了实验评估。本文从现有工作中收集了 290 个算法问题，涵盖了 8 种不同的算法模式与 37 种不同的数据结构，并在该数据集上将 SuFu 与现有的四种程序合成方法进行了比较。实验结果表明，SuFu 在时间限制内可以解决 264 个任务，平均开销仅为 24.4 秒，并且可以在更通用且输入更少的情况下，提供接近甚至优于现有方法的求解效率。

作为总结，本文面向算法问题提出了程序合成系统 SuFu。它通过语言设计、问题分解方法和程序合成方法上的创新，成功地解决了算法问题自动求解中的效率挑战与通用性挑战，显著地扩展了程序合成在求解算法问题时的能力边界。

8.2 未来工作展望

首先，算法作为计算机科学的重要组成部分，广泛存在于各种场景和问题中。因此，SuFu 作为一种自动求解算法问题的方法，具有被应用于下游任务的巨大潜力。例如，表 7.3 展示了将 SuFu 用于合成表达式语言解释器的案例，这是程序语义合成问题的一个特例 [149]。同时，已有研究表明，在程序等价性证明过程中，许多必要的辅助引理可以通过合成递归程序找到 [150]，这也属于 SuFu 的适用范围。探究 SuFu 在这些问题上的能力，以及为 SuFu 寻找更多的应用场景，都是非常有趣的研究方向。

此外，SuFu 的表达能力也存在提升的空间。具体而言，本文在求解提升问题时仅考虑输出为常量的表征函数和常数时间的合并算子。尽管在这一限制下，SuFu 已经能够解决许多复杂算法问题，但仍然存在一些情况需要更一般的表征函数与合并算子。例如，在使用列表分治计算逆序对数量时，目标的表征函数需要返回完整的排序后的列表作为辅助值；而对应的合并算子需要归并排序后的左右列表，其时间复杂度超过常量，达到了线性的量级。若要将 SuFu 扩展到这些情况，则仍有一些技术问题有待解决。例如如何扩展分解系统 AutoLifter 以支持不满足压缩性质的问题，如何将合并算子的合成方法从分支程序扩展到递归程序。这些都需要未来的研究。

更加值得期待的是 SuFu 与大模型相关技术的结合。尽管目前的大语言模型已经能够求解许多复杂的算法问题，但正如第 2.4 节中讨论的那样，大模型从原理上很难保证推理过程的完备性与推理结果的正确性。因此，即便在一些十分简单的编程任务上，大语言模型也可能表现出古怪的行为并给出错误的程序。而作为基于搜索的程序合成方法，SuFu 虽然可以更容易地提供性质保障，但在复杂问题上也会面临合成效率不足的问题。因此，一个可能的改进方案是将大语言模型用于提示 SuFu 的搜索过程，并由 SuFu 提供足够的性质保障，从而达到取长补短的效果。

参考文献

[1] Thomas H. Cormen et al. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. ISBN: 978-0-262-03384-8. URL: http://mitpress.mit.edu/books/introduction-algorithms.

[2] Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[3] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107. DOI: 10.1109/TSSC.1968.300136.

[4] Ana Belén Sánchez et al. "TANDEM: A Taxonomy and a Dataset of Real-World Performance Bugs". In: *IEEE Access* 8 (2020), pp. 107214–107228. DOI: 10.1109/ACCESS.2020.3000928. URL: https://doi.org/10.1109/ACCESS.2020.3000928.

[5] Yepang Liu, Chang Xu, and Shing-Chi Cheung. "Characterizing and detecting performance bugs for smartphone applications". In: *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*. Ed. by Pankaj Jalote, Lionel C. Briand, and André van der Hoek. ACM, 2014, pp. 1013–1024. DOI: 10.1145/2568225.2568229. URL: https://doi.org/10.1145/2568225.2568229.

[6] Ronnie Chaiken et al. "SCOPE: easy and efficient parallel processing of massive data sets". In: *Proc. VLDB Endow.* 1.2 (2008), pp. 1265–1276. DOI: 10.14778/1454159.1454166. URL: http://www.vldb.org/pvldb/vol1/1454166.pdf.

[7] Anne Ouyang et al. "KernelBench: Can LLMs Write Efficient GPU Kernels?" In: *CoRR* abs/2502.10517 (2025). DOI: 10.48550/ARXIV.2502.10517. arXiv: 2502.10517. URL: https://doi.org/10.48550/arXiv.2502.10517.

[8] Gerwin Klein et al. "seL4: formal verification of an OS kernel." In: *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*. Ed. by Jeanna Neefe Matthews and Thomas E. Anderson. ACM, 2009, pp. 207–220. DOI: 10.1145/1629575.1629596. URL: https://doi.org/10.1145/1629575.1629596.

[9] Umut A Acar et al. "Self-adjusting computation". PhD thesis. Carnegie Mellon University, 2005.

[10] Azadeh Farzan and Victor Nicolet. "Synthesis of divide and conquer parallelism for loops". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. Ed. by Albert Cohen and Martin T. Vechev. ACM, 2017, pp. 540–555. DOI: 10.1145/3062341.3062355. URL: https://doi.org/10.1145/3062341.3062355.

[11] Shu Lin, Na Meng, and Wenxin Li. "Optimizing Constraint Solving via Dynamic Programming". In: *IJCAI 2019, Macao, China, August 10-16, 2019*. ijcai.org, 2019, pp. 1146–1154. DOI: 10.24963/ijcai.2019/160. URL: https://doi.org/10.24963/ijcai.2019/160.

[12] Azadeh Farzan and Victor Nicolet. "Phased synthesis of divide and conquer programs". In: *PLDI 2021, Virtual Event, Canada, June 20-25, 2021*. ACM, 2021, pp. 974–986. DOI: 10.1145/3453483.3454089. URL: https://doi.org/10.1145/3453483.3454089.

[13] Yewen Pu, Rastislav Bodík, and Saurabh Srivastava. “Synthesis of first-order dynamic programming algorithms”. In: *OOPSLA*. ACM, 2011, pp. 83–98. DOI: 10.1145/2048066.2048076. URL: https://doi.org/10.1145/2048066.2048076.

[14] Tristan Knoth et al. “Resource-guided program synthesis”. In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*. Ed. by Kathryn S. McKinley and Kathleen Fisher. ACM, 2019, pp. 253–268. DOI: 10.1145/3314221.3314602. URL: https://doi.org/10.1145/3314221.3314602.

[15] Qinheping Hu et al. “Synthesis with Asymptotic Resource Bounds”. In: *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*. Ed. by Alexandra Silva and K. Rustan M. Leino. Vol. 12759. Lecture Notes in Computer Science. Springer, 2021, pp. 783–807. DOI: 10.1007/978-3-030-81685-8_37. URL: https://doi.org/10.1007/978-3-030-81685-8%5C_37.

[16] Phitchaya Mangpo Phothilimthana et al. “Scaling up Superoptimization”. In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS ’16, Atlanta, GA, USA, April 2-6, 2016*. 2016, pp. 297–310. DOI: 10.1145/2872362.2872387. URL: https://doi.org/10.1145/2872362.2872387.

[17] Eric Schkufza, Rahul Sharma, and Alex Aiken. “Stochastic superoptimization”. In: *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013*. Ed. by Vivek Sarkar and Rastislav Bodík. ACM, 2013, pp. 305–316. DOI: 10.1145/2451116.2451150. URL: https://doi.org/10.1145/2451116.2451150.

[18] Woosuk Lee and Hangyeol Cho. “Inductive Synthesis of Structurally Recursive Functional Programs from Non-recursive Expressions”. In: *Proc. ACM Program. Lang.* 7.POPL (2023), pp. 2048–2078. DOI: 10.1145/3571263. URL: https://doi.org/10.1145/3571263.

[19] Anders Miltner et al. “Bottom-up synthesis of recursive functional programs using angelic execution”. In: *Proc. ACM Program. Lang.* 6.POPL (2022), pp. 1–29. DOI: 10.1145/3498682. URL: https://doi.org/10.1145/3498682.

[20] Murray Cole. “Parallel Programming with List Homomorphisms”. In: *Parallel Process. Lett.* 5 (1995), pp. 191–203. DOI: 10.1142/S0129626495000175. URL: https://doi.org/10.1142/S0129626495000175.

[21] Oege de Moor. “Categories, Relations and Dynamic Programming”. In: *Math. Struct. Comput. Sci.* 4.1 (1994), pp. 33–69. DOI: 10.1017/S0960129500000360. URL: https://doi.org/10.1017/S0960129500000360.

[22] Azadeh Farzan, Danya Lette, and Victor Nicolet. “Recursion synthesis with unrealizability witnesses”. In: *PLDI ’22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*. Ed. by Ranjit Jhala and Isil Dillig. ACM, 2022, pp. 244–259. DOI: 10.1145/3519939.3523726. URL: https://doi.org/10.1145/3519939.3523726.

[23] Hans Zantema. “Longest Segment Problems”. In: *Sci. Comput. Program.* 18.1 (1992), pp. 39–66. DOI: 10.1016/0167-6423(92)90033-8. URL: https://doi.org/10.1016/0167-6423(92)90033-8.

[24] J. L. Bentley. “Solution to Klee”s Rectangle Problem”. In: (1977).

[25] Azadeh Farzan and Victor Nicolet. “Modular divide-and-conquer parallelization of nested loops”. In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*. Ed. by Kathryn S. McKinley and Kathleen Fisher. ACM, 2019, pp. 610–624. DOI: 10.1145/3314221.3314612. URL: https://doi.org/10.1145/3314221.3314612.

[26] Kazutaka Morita et al. “Automatic inversion generates divide-and-conquer parallel programs”. In: *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*. Ed. by Jeanne Ferrante and Kathryn S. McKinley. ACM, 2007, pp. 146–155. DOI: 10.1145/1250734.1250752. URL: https://doi.org/10.1145/1250734.1250752.

[27] Yican Sun, Xuanyu Peng, and Yingfei Xiong. “Synthesizing Efficient Memoization Algorithms”. In: *Proc. ACM Program. Lang.* 7.OOPSLA2 (2023), pp. 89–115. DOI: 10.1145/3622800. URL: https://doi.org/10.1145/3622800.

[28] Rajeev Alur et al. “Syntax-guided synthesis”. In: *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. IEEE, 2013, pp. 1–8. URL: https://ieeexplore.ieee.org/document/6679385/.

[29] Abhishek Udupa et al. “TRANSIT: specifying protocols with concolic snippets”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’13, Seattle, WA, USA, June 16-19, 2013*. Ed. by Hans-Juergen Boehm and Cormac Flanagan. ACM, 2013, pp. 287–296. DOI: 10.1145/2491956.2462174. URL: https://doi.org/10.1145/2491956.2462174.

[30] Susmit Jha et al. “Oracle-guided component-based program synthesis”. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*. Ed. by Jeff Kramer et al. ACM, 2010, pp. 215–224. DOI: 10.1145/1806799.1806833. URL: https://doi.org/10.1145/1806799.1806833.

[31] Sergey Mechtaev et al. “Symbolic execution with existential second-order constraints”. In: *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. Ed. by Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu. ACM, 2018, pp. 389–399. DOI: 10.1145/3236024.3236049. URL: https://doi.org/10.1145/3236024.3236049.

[32] William R. Harris and Sumit Gulwani. “Spreadsheet table transformations from examples”. In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. Ed. by Mary W. Hall and David A. Padua. ACM, 2011, pp. 317–328. DOI: 10.1145/1993498.1993536. URL: https://doi.org/10.1145/1993498.1993536.

[33] Oleksandr Polozov and Sumit Gulwani. “FlashMeta: a framework for inductive program synthesis”. In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA,*

USA, October 25-30, 2015. Ed. by Jonathan Aldrich and Patrick Eugster. ACM, 2015, pp. 107–126. DOI: 10.1145/2814270.2814310. URL: https://doi.org/10.1145/2814270.2814310.

[34] Xinyu Wang, Isil Dillig, and Rishabh Singh. "Program synthesis using abstraction refinement". In: *Proc. ACM Program. Lang.* 2.POPL (2018), 63:1–63:30. DOI: 10.1145/3158151. URL: https://doi.org/10.1145/3158151.

[35] Xinyu Wang, Isil Dillig, and Rishabh Singh. "Synthesis of data completion scripts using finite tree automata". In: *Proc. ACM Program. Lang.* 1.OOPSLA (2017), 62:1–62:26. DOI: 10.1145/3133886. URL: https://doi.org/10.1145/3133886.

[36] Rajeev Alur, Pavol Cerný, and Arjun Radhakrishna. "Synthesis Through Unification". In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*. Ed. by Daniel Kroening and Corina S. Pasareanu. Vol. 9207. Lecture Notes in Computer Science. Springer, 2015, pp. 163–179. DOI: 10.1007/978-3-319-21668-3_10. URL: https://doi.org/10.1007/978-3-319-21668-3%5C_10.

[37] Ruyi Ji et al. "Generalizable synthesis through unification". In: *Proc. ACM Program. Lang.* 5.OOPSLA (2021), pp. 1–28. DOI: 10.1145/3485544. URL: https://doi.org/10.1145/3485544.

[38] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. "Scaling Enumerative Program Synthesis via Divide and Conquer". In: *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*. Ed. by Axel Legay and Tiziana Margaria. Vol. 10205. Lecture Notes in Computer Science. 2017, pp. 319–336. DOI: 10.1007/978-3-662-54577-5_18. URL: https://doi.org/10.1007/978-3-662-54577-5%5C_18.

[39] Ashwin Kalyan et al. "Neural-Guided Deductive Search for Real-Time Program Synthesis from Examples". In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. URL: https://openreview.net/forum?id=rywDjg-RW.

[40] Woosuk Lee et al. "Accelerating search-based program synthesis using learned probabilistic models". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. Ed. by Jeffrey S. Foster and Dan Grossman. ACM, 2018, pp. 436–449. DOI: 10.1145/3192366.3192410. URL: https://doi.org/10.1145/3192366.3192410.

[41] Ruyi Ji et al. "Guiding dynamic programing via structural probability for accelerating programming by example". In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020), 224:1–224:29. DOI: 10.1145/3428292. URL: https://doi.org/10.1145/3428292.

[42] Susmit Jha and Sanjit A. Seshia. "A Theory of Formal Synthesis via Inductive Learning". In: *CoRR* abs/1505.03953 (2015). arXiv: 1505.03953. URL: http://arxiv.org/abs/1505.03953.

[43] Armando Solar-Lezama et al. "Combinatorial sketching for finite programs". In: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*. Ed. by John Paul Shen

and Margaret Martonosi. ACM, 2006, pp. 404–415. DOI: 10.1145/1168857.1168907. URL: https://doi.org/10.1145/1168857.1168907.

[44] Anselm Blumer et al. “Occam’s Razor”. In: *Inf. Process. Lett.* 24.6 (1987), pp. 377–380. DOI: 10.1016/0020-0190(87)90114-1. URL: https://doi.org/10.1016/0020-0190(87)90114-1.

[45] Ruyi Ji et al. “Question selection for interactive program synthesis”. In: *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*. Ed. by Alastair F. Donaldson and Emina Torlak. ACM, 2020, pp. 1143–1158. DOI: 10.1145/3385412.3386025. URL: https://doi.org/10.1145/3385412.3386025.

[46] Vu Le et al. “Interactive Program Synthesis”. In: *CoRR* abs/1703.03539 (2017). arXiv: 1703.03539. URL: http://arxiv.org/abs/1703.03539.

[47] Laurent Hyafil and Ronald L. Rivest. “Constructing Optimal Binary Decision Trees is NP-Complete”. In: *Inf. Process. Lett.* 5.1 (1976), pp. 15–17. DOI: 10.1016/0020-0190(76)90095-8. URL: https://doi.org/10.1016/0020-0190(76)90095-8.

[48] Micah Adler and Brent Heeringa. “Approximating Optimal Binary Decision Trees”. In: *Algorithmica* 62.3-4 (2012), pp. 1112–1121. DOI: 10.1007/s00453-011-9510-9. URL: https://doi.org/10.1007/s00453-011-9510-9.

[49] Philip Wadler. “Deforestation: Transforming Programs to Eliminate Trees”. In: *ESOP ’88, 2nd European Symposium on Programming, Nancy, France, March 21-24, 1988, Proceedings*. Ed. by Harald Ganzinger. Vol. 300. Lecture Notes in Computer Science. Springer, 1988, pp. 344–358. DOI: 10.1007/3-540-19027-9_23. URL: https://doi.org/10.1007/3-540-19027-9%5C_23.

[50] Martin Ward. “Proving program refinements and transformations”. PhD thesis. University of Oxford, UK, 1989. URL: https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.306712.

[51] Rod M. Burstall and John Darlington. “A Transformation System for Developing Recursive Programs”. In: *J. ACM* 24.1 (1977), pp. 44–67. DOI: 10.1145/321992.321996. URL: https://doi.org/10.1145/321992.321996.

[52] Laurent Kott. “Unfold/fold program transformations”. PhD thesis. INRIA, 1982.

[53] David Sands. “Total Correctness by Local Improvement in the Transformation of Functional Programs”. In: *ACM Trans. Program. Lang. Syst.* 18.2 (1996), pp. 175–234. DOI: 10.1145/227699.227716. URL: https://doi.org/10.1145/227699.227716.

[54] Philippa Gardner and John C. Shepherdson. “Unfold/Fold Transformations of Logic Programs”. In: *Computational Logic - Essays in Honor of Alan Robinson*. Ed. by Jean-Louis Lassez and Gordon D. Plotkin. The MIT Press, 1991, pp. 565–583.

[55] Taisuke Sato. “Equivalence-Preserving First-Order Unfold/Fold Transformation Systems”. In: *Theor. Comput. Sci.* 105.1 (1992), pp. 57–84. DOI: 10.1016/0304-3975(92)90287-P. URL: https://doi.org/10.1016/0304-3975(92)90287-P.

[56] Alberto Pettorossi, Maurizio Proietti, and Sophie Renault. “Reducing Nondeterminism while Specializing Logic Programs”. In: *Conference Record of POPL’97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris,*

France, 15-17 January 1997. Ed. by Peter Lee, Fritz Henglein, and Neil D. Jones. ACM Press, 1997, pp. 414–427. DOI: 10.1145/263699.263759. URL: https://doi.org/10.1145/263699.263759.

[57] Abhik Roychoudhury et al. "An unfold/fold transformation framework for definite logic programs". In: *ACM Trans. Program. Lang. Syst.* 26.3 (2004), pp. 464–509. DOI: 10.1145/982158.982160. URL: https://doi.org/10.1145/982158.982160.

[58] Michael J. Maher. "A Transformation System for Deductive Database Modules with Perfect Model Semantics". In: *Foundations of Software Technology and Theoretical Computer Science, Ninth Conference, Bangalore, India, December 19-21, 1989, Proceedings*. Ed. by C. E. Veni Madhavan. Vol. 405. Lecture Notes in Computer Science. Springer, 1989, pp. 89–98. DOI: 10.1007/3-540-52048-1_35. URL: https://doi.org/10.1007/3-540-52048-1%5C_35.

[59] Sandro Etalle and Maurizio Gabbrielli. "Transformations of CLP Modules". In: *Theor. Comput. Sci.* 166.1&2 (1996), pp. 101–146. DOI: 10.1016/0304-3975(95)00148-4. URL: https://doi.org/10.1016/0304-3975(95)00148-4.

[60] Nacéra Bensaou and Irène Guessarian. "Transforming Constraint Logic Programs". In: *STACS 94, 11th Annual Symposium on Theoretical Aspects of Computer Science, Caen, France, February 24-26, 1994, Proceedings*. Ed. by Patrice Enjalbert, Ernst W. Mayr, and Klaus W. Wagner. Vol. 775. Lecture Notes in Computer Science. Springer, 1994, pp. 33–46. DOI: 10.1007/3-540-57785-8_129. URL: https://doi.org/10.1007/3-540-57785-8%5C_129.

[61] Sandro Etalle, Maurizio Gabbrielli, and Maria Chiara Meo. "Transformations of CCP programs". In: *ACM Trans. Program. Lang. Syst.* 23.3 (2001), pp. 304–395. DOI: 10.1145/503502.503504. URL: https://doi.org/10.1145/503502.503504.

[62] Azadeh Farzan and Victor Nicolet. "Phased synthesis of divide and conquer programs". In: *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*. Ed. by Stephen N. Freund and Eran Yahav. ACM, 2021, pp. 974–986. DOI: 10.1145/3453483.3454089. URL: https://doi.org/10.1145/3453483.3454089.

[63] Daya Guo et al. "DeepSeek-Coder: When the Large Language Model Meets Programming - The Rise of Code Intelligence". In: *CoRR* abs/2401.14196 (2024). DOI: 10.48550/ARXIV.2401.14196. arXiv: 2401.14196. URL: https://doi.org/10.48550/arXiv.2401.14196.

[64] Xin Jin et al. "Binary Code Summarization: Benchmarking ChatGPT/GPT-4 and Other Large Language Models". In: *CoRR* abs/2312.09601 (2023). DOI: 10.48550/ARXIV.2312.09601. arXiv: 2312.09601. URL: https://doi.org/10.48550/arXiv.2312.09601.

[65] Zhenjiang Hu et al. "Tupling Calculation Eliminates Multiple Data Traversals". In: *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97), Amsterdam, The Netherlands, June 9-11, 1997*. Ed. by Simon L. Peyton Jones, Mads Tofte, and A. Michael Berman. ACM, 1997, pp. 164–175. DOI: 10.1145/258948.258964. URL: https://doi.org/10.1145/258948.258964.

[66] Maarten M Fokkinga. "Tupling and mutumorphisms". In: *Squiggolist* 1.4 (1989).

[67] Akimasa Morihata et al. “The third homomorphism theorem on trees: downward & upward lead to divide-and-conquer”. In: *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*. Ed. by Zhong Shao and Benjamin C. Pierce. ACM, 2009, pp. 177–185. DOI: 10.1145/1480881.1480905. URL: https://doi.org/10.1145/1480881.1480905.

[68] Richard S. Bird and Oege de Moor. “From Dynamic Programming to Greedy Algorithms”. In: *Formal Program Development - IFIP TC2/WG 2.1 State-of-the-Art Report*. Ed. by Bernhard Möller, Helmuth Partsch, and Stephen A. Schuman. Vol. 755. Lecture Notes in Computer Science. Springer, 1993, pp. 43–61. DOI: 10.1007/3-540-57499-9_16. URL: https://doi.org/10.1007/3-540-57499-9%5C_16.

[69] Paul Helman. *A theory of greedy structures based on k-ary dominance relations*. Department of Computer Science, College of Engineering, University of New Mexico, 1989.

[70] Oege de Moor. “A Generic Program for Sequential Decision Processes”. In: *Programming Languages: Implementations, Logics and Programs, 7th International Symposium, PLILP’95, Utrecht, The Netherlands, September 20-22, 1995, Proceedings*. Ed. by Manuel V. Hermenegildo and S. Doaitse Swierstra. Vol. 982. Lecture Notes in Computer Science. Springer, 1995, pp. 1–23. DOI: 10.1007/BFb0026809. URL: https://doi.org/10.1007/BFb0026809.

[71] Akimasa Morihata. “A Short Cut to Optimal Sequences”. In: *New Gener. Comput.* 29.1 (2011), pp. 31–59. DOI: 10.1007/s00354-010-0098-4. URL: https://doi.org/10.1007/s00354-010-0098-4.

[72] Akimasa Morihata, Masato Koishi, and Atsushi Ohori. “Dynamic Programming via Thinning and Incrementalization”. In: *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*. Ed. by Michael Codish and Eijiro Sumii. Vol. 8475. Lecture Notes in Computer Science. Springer, 2014, pp. 186–202. DOI: 10.1007/978-3-319-07151-0_12. URL: https://doi.org/10.1007/978-3-319-07151-0%5C_12.

[73] Shin-Cheng Mu. “Maximum segment sum is back: deriving algorithms for two segment problems with bounded lengths”. In: *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM 2008, San Francisco, California, USA, January 7-8, 2008*. Ed. by Robert Glück and Oege de Moor. ACM, 2008, pp. 31–39. DOI: 10.1145/1328408.1328414. URL: https://doi.org/10.1145/1328408.1328414.

[74] Richard S. Bird and Oege de Moor. “The algebra of programming”. In: *Proceedings of the NATO Advanced Study Institute on Deductive Program Design, Marktoberdorf, Germany*. Ed. by Manfred Broy. 1996, pp. 167–203.

[75] Tatsuya Hagino. “Category theoretic approach to data types”. PhD thesis. PhD thesis, University of Edinburgh, 1987.

[76] Erik Meijer, Maarten M. Fokkinga, and Ross Paterson. “Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire”. In: *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings*. Ed. by John Hughes. Vol. 523. Lecture Notes in Computer Science. Springer, 1991, pp. 124–144. DOI: 10.1007/3540543961_7. URL: https://doi.org/10.1007/3540543961%5C_7.

[77] Maarten M. Fokkinga. *Law and order in algorithmics*. Univ. Twente, 1992. ISBN: 978-90-9004816-1.

[78] Tim Sheard and Leonidas Fegaras. "A Fold for All Seasons". In: *Proceedings of the conference on Functional programming languages and computer architecture, FPCA 1993, Copenhagen, Denmark, June 9-11, 1993*. Ed. by John Williams. ACM, 1993, pp. 233–242. DOI: 10.1145/165180.165216. URL: https://doi.org/10.1145/165180.165216.

[79] Ernest G Manes and Michael A Arbib. *Algebraic approaches to program semantics*. Springer Science & Business Media, 2012.

[80] Akihiko Takano and Erik Meijer. "Shortcut deforestation in calculational form". In: *Proceedings of the seventh international conference on Functional programming languages and computer architecture*. 1995, pp. 306–313.

[81] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. "Deriving structural hylomorphisms from recursive definitions". In: *ACM Sigplan Notices* 31.6 (1996), pp. 73–82.

[82] Maarten M. Fokkinga. "Calculate Categorically!" In: *Formal Aspects Comput.* 4.6A (1992), pp. 673–692.

[83] Alberto Pettorossi. "Methodologies for transformations and memoing in applicative languages". PhD thesis. University of Edinburgh, UK, 1984. URL: http://hdl.handle.net/1842/15643.

[84] Geraint Jones and Mary Sheeran. "Designing Arithmetic Circuits by Refinement in Ruby". In: *Sci. Comput. Program.* 22.1-2 (1994), pp. 107–135. DOI: 10.1016/0167-6423(94)90009-4. URL: https://doi.org/10.1016/0167-6423(94)90009-4.

[85] Akihiko Takano and Erik Meijer. "Shortcut Deforestation in Calculational Form". In: *Proceedings of the seventh international conference on Functional programming languages and computer architecture, FPCA 1995, La Jolla, California, USA, June 25-28, 1995*. Ed. by John Williams. ACM, 1995, pp. 306–313. DOI: 10.1145/224164.224221. URL: https://doi.org/10.1145/224164.224221.

[86] Richard Bellman. "Some Applications of the Theory of Dynamic Programming - A Review". In: *Oper. Res.* 2.3 (1954), pp. 275–288. DOI: 10.1287/opre.2.3.275. URL: https://doi.org/10.1287/opre.2.3.275.

[87] Richard M. Karp and Michael Held. "Finite-state processes and dynamic programming". In: *SIAM Journal on Applied Mathematics* 15.3 (1967), pp. 693–718.

[88] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*. Ed. by Eric A. Brewer and Peter Chen. USENIX Association, 2004, pp. 137–150. URL: http://www.usenix.org/events/osdi04/tech/dean.html.

[89] Kiminori Matsuzaki et al. "A library of constructive skeletons for sequential style of parallel programming". In: *Proceedings of the 1st International Conference on Scalable Information Systems, Infoscale 2006, Hong Kong, May 30-June 1, 2006*. Ed. by Xiaohua Jia. Vol. 152. ACM International Conference Proceeding Series. ACM, 2006, p. 13. DOI: 10.1145/1146847.1146860. URL: https://doi.org/10.1145/1146847.1146860.

[90] Fethi A. Rabhi and Sergei Gorlatch, eds. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2003. ISBN: 978-1-85233-506-9. DOI: 10.1007/978-1-4471-0097-3. URL: https://doi.org/10.1007/978-1-4471-0097-3.

[91] Guy L. Steele Jr. "Parallel Programming and Parallel Abstractions in Fortress". In: *Functional and Logic Programming, 8th International Symposium, FLOPS 2006, Fuji-Susono, Japan, April 24-26, 2006, Proceedings*. Ed. by Masami Hagiya and Philip Wadler. Vol. 3945. Lecture Notes in Computer Science. Springer, 2006, p. 1. DOI: 10.1007/11737414_1. URL: https://doi.org/10.1007/11737414%5C_1.

[92] Jeremy Gibbons. "Functional pearls: The third homomorphism theorem". In: *Journal of Functional Programming* 6.4 (1996), pp. 657–665.

[93] Gérard Huet. "The zipper". In: *Journal of functional programming* 7.5 (1997), pp. 549–554.

[94] Karl R. Abrahamson et al. "A Simple Parallel Tree Contraction Algorithm". In: *J. Algorithms* 10.2 (1989), pp. 287–302. DOI: 10.1016/0196-6774(89)90017-5. URL: https://doi.org/10.1016/0196-6774(89)90017-5.

[95] Ruyi Ji et al. "Synthesizing Efficient Dynamic Programming Algorithms". In: *CoRR* abs/2202.12208 (2022). arXiv: 2202.12208. URL: https://arxiv.org/abs/2202.12208.

[96] Henry Lieberman. *Your wish is my command: Programming by example*. Morgan Kaufmann, 2001.

[97] Tom M. Mitchell. "Generalization as Search". In: *Artif. Intell.* 18.2 (1982), pp. 203–226. DOI: 10.1016/0004-3702(82)90040-6. URL: https://doi.org/10.1016/0004-3702(82)90040-6.

[98] Yuepeng Wang, Xinyu Wang, and Isil Dillig. "Relational program synthesis". In: *Proc. ACM Program. Lang.* 2.OOPSLA (2018), 155:1–155:27. DOI: 10.1145/3276525. URL: https://doi.org/10.1145/3276525.

[99] Quinlan J. Ross. "Induction of Decision Trees". In: *Mach. Learn.* 1.1 (1986), pp. 81–106. DOI: 10.1023/A:1022643204877. URL: https://doi.org/10.1023/A:1022643204877.

[100] Matej Balog et al. "DeepCoder: Learning to Write Programs". In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL: https://openreview.net/forum?id=ByldLrqlx.

[101] Aditya Krishna Menon et al. "A Machine Learning Framework for Programming by Example". In: *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*. Vol. 28. JMLR Workshop and Conference Proceedings. JMLR.org, 2013, pp. 187–195. URL: http://proceedings.mlr.press/v28/menon13.html.

[102] Yingfei Xiong and Bo Wang. "L2S: A framework for synthesizing the most probable program under a specification". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31.3 (2022), pp. 1–45.

[103] Alexander M. Rush and Michael Collins. "A Tutorial on Dual Decomposition and Lagrangian Relaxation for Inference in Natural Language Processing". In: *J. Artif. Intell. Res.* 45 (2012), pp. 305–362. DOI: 10.1613/jair.3680. URL: https://doi.org/10.1613/jair.3680.

[104] Pavol Bielik, Veselin Raychev, and Martin T. Vechev. "PHOG: Probabilistic Model for Code". In: *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York*

City, NY, USA, June 19-24, 2016. Ed. by Maria-Florina Balcan and Kilian Q. Weinberger. Vol. 48. JMLR Workshop and Conference Proceedings. JMLR.org, 2016, pp. 2933–2942. URL: http://proceedings.mlr.press/v48/bielik16.html.

[105] Ralf Hinze, Thomas Harper, and Daniel W. H. James. "Theory and Practice of Fusion". In: *Implementation and Application of Functional Languages - 22nd International Symposium, IFL 2010, Alphen aan den Rijn, The Netherlands, September 1-3, 2010, Revised Selected Papers*. Ed. by Jurriaan Hage and Marco T. Morazán. Vol. 6647. Lecture Notes in Computer Science. Springer, 2010, pp. 19–37. DOI: 10.1007/978-3-642-24276-2_2. URL: https://doi.org/10.1007/978-3-642-24276-2%5C_2.

[106] James Bornholt et al. "Optimizing synthesis with metasketches". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. 2016, pp. 775–788. DOI: 10.1145/2837614.2837666. URL: https://doi.org/10.1145/2837614.2837666.

[107] Phitchaya Mangpo Phothilimthana et al. "Chlorophyll: synthesis-aided compiler for low-power spatial architectures". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. Ed. by Michael F. P. O'Boyle and Keshav Pingali. ACM, 2014, pp. 396–407. DOI: 10.1145/2594291.2594339. URL: https://doi.org/10.1145/2594291.2594339.

[108] Rahul Sharma et al. "Conditionally correct superoptimization". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*. Ed. by Jonathan Aldrich and Patrick Eugster. ACM, 2015, pp. 147–162. DOI: 10.1145/2814270.2814278. URL: https://doi.org/10.1145/2814270.2814278.

[109] Yujia Li et al. "Competition-Level Code Generation with AlphaCode". In: *CoRR* abs/2203.07814 (2022). DOI: 10.48550/ARXIV.2203.07814. arXiv: 2203.07814. URL: https://doi.org/10.48550/arXiv.2203.07814.

[110] OpenAI. *o3*. URL: https://openai.com/ (visited on 03/06/2025).

[111] *Codeforces*. 2025. URL: https://codeforces.com (visited on 03/06/2025).

[112] DeepSeek-AI et al. "DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning". In: *CoRR* abs/2501.12948 (2025). DOI: 10.48550/ARXIV.2501.12948. arXiv: 2501.12948. URL: https://doi.org/10.48550/arXiv.2501.12948.

[113] Clinton J Wang et al. "EnigmaEval: A Benchmark of Long Multimodal Reasoning Challenges". In: *arXiv preprint arXiv:2502.08859* (2025).

[114] Seyed-Iman Mirzadeh et al. "GSM-Symbolic: Understanding the Limitations of Mathematical Reasoning in Large Language Models". In: *CoRR* abs/2410.05229 (2024). DOI: 10.48550/ARXIV.2410.05229. arXiv: 2410.05229. URL: https://doi.org/10.48550/arXiv.2410.05229.

[115] Trieu H. Trinh et al. "Solving olympiad geometry without human demonstrations". In: *Nat.* 625.7995 (2024), pp. 476–482. DOI: 10.1038/S41586-023-06747-5. URL: https://doi.org/10.1038/s41586-023-06747-5.

[116] DeepMind. *AI achieves silver-medal standard solving International Mathematical Olympiad problems*. 2024. URL: https://deepmind.google/discover/blog/ai-solves-imoproblems-at-silver-medal-level/ (visited on 10/10/2024).

[117] Huajian Xin et al. "DeepSeek-Prover: Advancing Theorem Proving in LLMs through Large-Scale Synthetic Data". In: *CoRR* abs/2405.14333 (2024). DOI: 10.48550/ARXIV.2405.14333. arXiv: 2405.14333. URL: https://doi.org/10.48550/arXiv.2405.14333.

[118] Andrew John Gill, John Launchbury, and Simon L. Peyton Jones. "A Short Cut to Deforestation". In: *Proceedings of the conference on Functional programming languages and computer architecture, FPCA 1993, Copenhagen, Denmark, June 9-11, 1993*. Ed. by John Williams. ACM, 1993, pp. 223–232. DOI: 10.1145/165180.165214. URL: https://doi.org/10.1145/165180.165214.

[119] Richard S. Bird. "Algebraic Identities for Program Calculation". In: *Comput. J.* 32.2 (1989), pp. 122–126. DOI: 10.1093/comjnl/32.2.122. URL: https://doi.org/10.1093/comjnl/32.2.122.

[120] Richard S. Bird and Oege de Moor. *Algebra of programming*. Prentice Hall International series in computer science. Prentice Hall, 1997. ISBN: 978-0-13-507245-5.

[121] Richard Bird. *Lecture notes in Theory of Lists*. Feb. 1989.

[122] Jon Louis Bentley. "Solutions to Klee's rectangle problems". In: *Unpublished manuscript* (1977), pp. 282–300.

[123] Nicole Schweikardt. "One-Pass Algorithm". In: *Encyclopedia of Database Systems*. Ed. by Ling Liu and M. Tamer Özsu. Springer US, 2009, pp. 1948–1949. DOI: 10.1007/978-0-387-39940-9_253. URL: https://doi.org/10.1007/978-0-387-39940-9%5C_253.

[124] Matias Martinez and Martin Monperrus. "ASTOR: A Program Repair Library for Java". In: *Proceedings of ISSTA*. 2016. DOI: 10.1145/2931037.2948705.

[125] Robert Harper. *Practical Foundations for Programming Languages (2nd. Ed.)* Cambridge University Press, 2016. ISBN: 9781107150300. URL: https://www.cs.cmu.edu/%5C%7Erwh/pfpl/index.html.

[126] Marc Dikotter. "Book review: The Definition of Standard ML by R. Milner, M. Torte, R. Harper". In: *SIGARCH Comput. Archit. News* 18.4 (1990), p. 91. DOI: 10.1145/121973.773545. URL: https://doi.org/10.1145/121973.773545.

[127] Rajeev Alur et al. "SyGuS-Comp 2018: Results and Analysis". In: *CoRR* abs/1904.07146 (2019). arXiv: 1904.07146. URL: http://arxiv.org/abs/1904.07146.

[128] Michael J. Kearns and Ming Li. "Learning in the Presence of Malicious Errors (Extended Abstract)". In: *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*. 1988, pp. 267–280. DOI: 10.1145/62212.62238. URL: https://doi.org/10.1145/62212.62238.

[129] Michael J. Kearns and Robert E. Schapire. "Efficient Distribution-Free Learning of Probabilistic Concepts". In: *J. Comput. Syst. Sci.* 48.3 (1994), pp. 464–497. DOI: 10.1016/S0022-0000(05)80062-5. URL: https://doi.org/10.1016/S0022-0000(05)80062-5.

[130] Dana Angluin and Philip D. Laird. "Learning From Noisy Examples". In: *Mach. Learn.* 2.4 (1987), pp. 343–370. DOI: 10.1007/BF00116829. URL: https://doi.org/10.1007/BF00116829.

[131] B. K. Natarajan. “Occam’s Razor for Functions”. In: *Proceedings of the Sixth Annual ACM Conference on Computational Learning Theory, COLT 1993, Santa Cruz, CA, USA, July 26-28, 1993*. 1993, pp. 370–376. DOI: 10.1145/168304.168380. URL: https://doi.org/10.1145/168304.168380.

[132] David J. Aldous and Umesh V. Vazirani. “A Markovian Extension of Valiant’s Learning Model”. In: *Inf. Comput.* 117.2 (1995), pp. 181–186. DOI: 10.1006/inco.1995.1037. URL: https://doi.org/10.1006/inco.1995.1037.

[133] Ronald L. Rivest. “Learning Decision Lists”. In: *Mach. Learn.* 2.3 (1987), pp. 229–246. DOI: 10.1007/BF00058680. URL: https://doi.org/10.1007/BF00058680.

[134] Vasek Chvátal. “A Greedy Heuristic for the Set-Covering Problem”. In: *Math. Oper. Res.* 4.3 (1979), pp. 233–235. DOI: 10.1287/moor.4.3.233. URL: https://doi.org/10.1287/moor.4.3.233.

[135] Sumit Gulwani. “Automating string processing in spreadsheets using input-output examples”. In: *POPL 2011, Austin, TX, USA, January 26-28, 2011*. 2011, pp. 317–330. DOI: 10.1145/1926385.1926423. URL: https://doi.org/10.1145/1926385.1926423.

[136] Thomas R. Hancock et al. “Lower Bounds on Learning Decision Lists and Trees (Extended Abstract)”. In: *STACS 95, 12th Annual Symposium on Theoretical Aspects of Computer Science, Munich, Germany, March 2-4, 1995, Proceedings*. 1995, pp. 527–538. DOI: 10.1007/3-540-59042-0_102. URL: https://doi.org/10.1007/3-540-59042-0%5C_102.

[137] LLC Gurobi Optimization. *Gurobi Optimizer Reference Manual*. 2021. URL: http://www.gurobi.com.

[138] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. “Z3: An Efficient SMT Solver”. In: *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*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24. URL: https://doi.org/10.1007/978-3-540-78800-3%5C_24.

[139] Ashish Tiwari et al. “Information-theoretic User Interaction: Significant Inputs for Program Synthesis”. In: *CoRR* abs/2006.12638 (2020). arXiv: 2006.12638. URL: https://arxiv.org/abs/2006.12638.

[140] Venkatesan T. Chakaravarthy et al. “Decision trees for entity identification: Approximation algorithms and hardness results”. In: *ACM Trans. Algorithms* 7.2 (2011), 15:1–15:22. DOI: 10.1145/1921659.1921661. URL: https://doi.org/10.1145/1921659.1921661.

[141] Vipin Kumar et al. *Introduction to parallel computing*. Vol. 110. Benjamin/Cummings Redwood City, CA, 1994.

[142] Saswat Padhi et al. “FlashProfile: a framework for synthesizing data profiles”. In: *PACMPL* 2.OOPSLA (2018), 150:1–150:28. DOI: 10.1145/3276520. URL: https://doi.org/10.1145/3276520.

[143] Henry S Warren. *Hacker’s Delight*. 2002.

[144] Mikaël Mayer et al. “User Interaction Models for Disambiguation in Programming by Example”. In: *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology, UIST 2015, Charlotte, NC, USA, November 8-11, 2015*. Ed. by Celine Latulipe, Bjoern Hartmann, and Tovi Grossman. ACM, 2015, pp. 291–301. DOI: 10.1145/2807442.2807459. URL: https://doi.org/10.1145/2807442.2807459.

[145] Emina Torlak and Rastislav Bodík. “A lightweight symbolic virtual machine for solver-aided host languages”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’14, Edinburgh, United Kingdom - June 09 - 11, 2014*. Ed. by Michael F. P. O’Boyle and Keshav Pingali. ACM, 2014, pp. 530–541. DOI: 10.1145/2594291.2594340. URL: https://doi.org/10.1145/2594291.2594340.

[146] Jon Bentley. “Programming Pearls: Algorithm Design Techniques”. In: *Communications of the ACM* 27.9 (1984), pp. 865–873.

[147] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts, 10th Edition*. Wiley, 2018. ISBN: 978-1-118-06333-0. URL: http://os-book.com/OS10/index.html.

[148] Sirui Lu and Rastislav Bodík. “Grisette: Symbolic Compilation as a Functional Programming Library”. In: *Proc. ACM Program. Lang.* 7.POPL (2023), pp. 455–487. DOI: 10.1145/3571209. URL: https://doi.org/10.1145/3571209.

[149] Jiangyi Liu et al. “Synthesizing Formal Semantics from Executable Interpreters”. In: *Proc. ACM Program. Lang.* 8.OOPSLA2 (2024), pp. 362–388. DOI: 10.1145/3689724. URL: https://doi.org/10.1145/3689724.

[150] Yican Sun et al. “Proving Functional Program Equivalence via Directed Lemma Synthesis”. In: *Formal Methods - 26th International Symposium, FM 2024, Milan, Italy, September 9-13, 2024, Proceedings, Part I*. Ed. by André Platzer et al. Vol. 14933. Lecture Notes in Computer Science. Springer, 2024, pp. 538–557. DOI: 10.1007/978-3-031-71162-6_28. URL: https://doi.org/10.1007/978-3-031-71162-6%5C_28.

攻读博士期间发表的论文及其他成果

[1] **Ruyi Ji**, Yuwei Zhao, Nadia Polikarpova, Yingfei Xiong, Zhenjiang Hu. **Superfusion: Eliminating Intermediate Data Structures via Inductive Synthesis.** PLDI'24: ACM SIGPLAN Conference on Programming Language Design and Implementation.（一作，CCF-A，对应第三章）

[2] **Ruyi Ji**, Yuwei Zhao, Yingfei Xiong, Di Wang, Lu Zhang, Zhenjiang Hu. **Decomposition-Based Synthesis for Applying Divide-and-Conquer-Like Algorithmic Paradigms.** TOPLAS: ACM Transactions on Programming Languages and Systems.（一作，CCF-A，对应第四章）

[3] **Ruyi Ji**, Chaozhe Kong, Yingfei Xiong, Zhenjiang Hu. **Improving Oracle-Guided Inductive Synthesis by Efficient Question Selection.** OOPSLA'23: Object Oriented Programming Languages, Systems and Applications.（一作，CCF-A，对应第六章）

[4] **Ruyi Ji**, Jingtao Xia, Yingfei Xiong, Zhenjiang Hu. **Generalizable Synthesis Through Unification.** OOPSLA'21: Object Oriented Programming Languages, Systems and Applications.（一作，CCF-A，对应第五章）

[5] **Ruyi Ji**, Yican Sun, Yingfei Xiong, Zhenjiang Hu. **Guiding Dynamic Programming via Structural Probability for Accelerating Programming by Example.** OOPSLA'20: Object-Oriented Programming, Systems, Languages, and Applications.（一作，CCF-A）

[6] Yican Sun, **Ruyi Ji**, Jian Fang, Xuanlin Jiang, Mingshuai Chen, Yingfei Xiong. **Proving Functional Program Equivalence via Directed Lemma Synthesis.** FM'24: 26th International Symposium on Formal Methods.（二作，CCF-A）

[7] Cole Kurashige, **Ruyi Ji**, Aditya Giridharan, Mark Barbone, Daniel Noor, Shachar Itzhaky, Ranjit Jhala, Nadia Polikarpova. **CCLemma: E-Graph Guided Lemma Discovery for Inductive Equational Proofs.** ICFP'24: ACM SIGPLAN International Conference on Functional Programming.（二作，CCF-B）

[8] Jingjing Liang, **Ruyi Ji**, Jiajun Jiang, Shurui Zhou, Yiling Lou, Yingfei Xiong, Gang Huang. **Interactive Patch Filtering as Debugging Aid.** ICSME'21: 37th International Conference on Software Maintenance and Evolution.（二作，CCF-B）

[9] Xiang Gao, Bo Wang, Gregory J. Duck, **Ruyi Ji**, Yingfei Xiong, Abhik Roychoudhury. **Beyond Tests: Program Vulnerability Repair via Crash Constraint Extraction**. TOSEM: ACM Transactions on Software Engineering and Methodology.（四作，CCF-A）

致谢

时光荏苒，2016 年我作为北大学子初次踏入燕园的经历仿佛就在昨天，转眼间九年的求学之旅即将画上句号。那时，28 号楼刚刚落成，三角地的老建筑还屹立在百讲边，小黄车才在北大崭露头角，《守望先锋》还是世界上最火的游戏；如今，新燕园校区都已颇具规模，燕园的旧街景早已消失的七七八八，ofo 总部的灯牌在南门外挂起又摘下，暴雪游戏从中国离开又回归。细数这些年的求学旅程，我受到了许多人的热诚帮助，在这里表达我心中由衷的感谢。

感谢杨芙清院士和梅宏院士，能够在由两位院士领导的北京大学软件工程研究所攻读博士学位，我感到无比荣幸。正是因为团队深厚的积累与卓越的研究实力，我才有机会向众多杰出的学术前辈们请教、学习，并自由探索学科前沿的问题。

感谢我的导师胡振江老师与熊英飞老师对我的悉心指导和支持。我与两位老师的第一次深入接触是在 2017 年《编程语言的设计原理》课堂上。当时的我怀揣着要比别人更早开始“科研”的幼稚想法，在不理解科研为何物时便联系了熊老师，并在对函数式语言一无所知的情况下强行选修了这门研究生课程。是两位老师用深入浅出的教学方式与丰富多样的课程内容，让我领略到了语言设计的魅力，引领我步入了程序语言研究的大门。在研究过程中，是胡老师的言传身教让我学会了如何从研究者的角度思考问题，也让我体会到了顶级学者对待学术研究坚持的态度与严谨的精神；是熊老师的耐心帮助让我从一个科研菜鸟慢慢地成长，逐渐成为一名能够独立面对挑战的研究者。而在我感到迷茫与挣扎的时候，他们总是用耐心的鼓励帮我排除杂念，坚定前行的道路。希望未来，两位老师能继续带领学生们做出更多影响深远的科研工作，将北大程序设计语言研究室变成世界级的程序语言研究殿堂。

感谢张路老师、张昕老师与王迪老师在合作中给予我的宝贵指导。无论是讨论项目进展还是论文写作，老师们总能敏锐地发现并指出我的疏漏与错误，耐心平等地与我探讨，使我受益匪浅。同样要感谢我在 UCSD 交流期间的导师 Nadia Polikarpova。是她的帮助与指导让我在大洋彼岸渡过了充实又美好的半年，也是她的鼓励让我有勇气在英文论文写作中追求卓越而非满足于平庸，真正体会到了科研写作的乐趣。

感谢我的父母亲人对我学业和生活的无私奉献。是他们的包容、付出与支持为我创造了这个如此平静专注的研究环境。感谢研究室里的王博学长、梁晶晶学姐、孙泽宇学长、朱琪豪学长和张星同学的帮助。他们分享的经验弥补了我个人视野的局限，帮我避免了许多弯路。同时感谢研究室里的孙奕灿、关智超、鄢振宇、唐丽娟、刘俊豪、张钊等同学在科研和生活上给予我的支持。我们一起喝酒、唱歌、爬山、玩桌游，为我

的博士生活增添了无数色彩，也在我情绪低落时给予了前进的力量。

最后，我也想感谢我自己。正是那份自己折腾自己的精神，让我拥有了如此丰富多彩的九年。回首过去，初入燕园时的憧憬、追求高绩点的执着、参加世界总决赛的紧张、负责全国比赛命题的激动、在百周年讲堂表演新年晚会节目的新鲜、创建算法协会时的热情、美团杯成功举办的自豪以及论文被连续拒稿后的失落与最终发表时的欣慰，这一切都是我对自己最好的馈赠。希望未来的我能继续保持这种心态，保持对这个世界的无限好奇和即刻行动的能力，度过无悔的一年又一年。