arXiv:2305.05661v1 [cs.GR] 9 May 2023

ShapeCoder：从非结构化基本图元中发现视觉程序的抽象

R. KENNY JONES，布朗大学，美国 PAUL GUERRERO，Adobe Research，英国 NILOY J. MITRA，伦敦大学学院和 Adobe Research，英国 DANIEL RITCHIE，布朗大学，美国

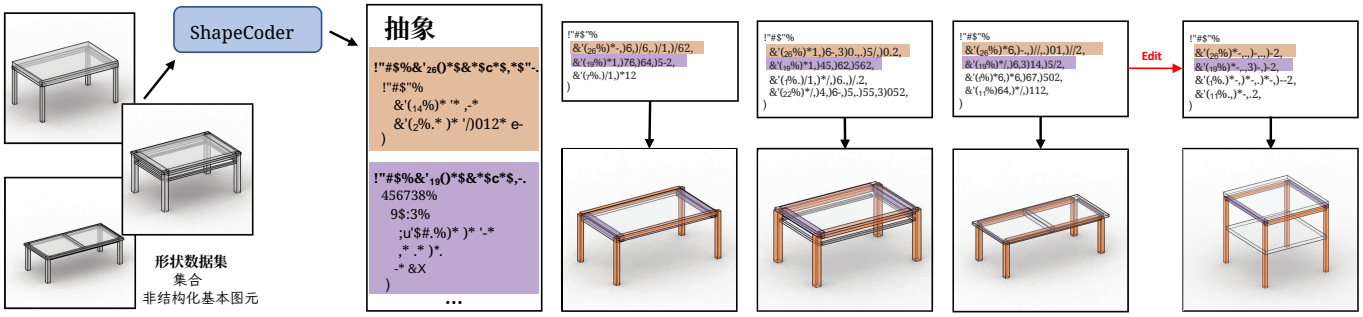

图 1. ShapeCoder 自动发现抽象函数，并推断使用这些抽象的视觉程序，以简洁地解释由非结构化基元表示的形状输入数据集。例如，橙色抽象仅使用五个参数，就能编码由相连水平支撑杆构成的四腿桌座分布。

我们介绍 ShapeCoder，这是首个能够接收由非结构化基元表示的形状数据集，并联合发现 (i) 有用的 抽象函数，以及 (ii) 用这些抽象解释输入形状的程序的系统。发现的抽象能够捕捉数据集中常见的模式（包括结构模式和参数模式），因此，使用这些抽象重写后的程序更加紧凑，也能抑制虚假的自由度。ShapeCoder 改进了以往的抽象发现方法，能够在输入假设限制更少的情况下，为更复杂的输入找到更好的抽象。这主要得益于两项方法上的进展：(a) 能够学习解决子问题的形状到程序识别网络；(b) 使用经条件重写方案增强的等式图，以可行的方式确定何时可以应用具有复杂参数表达式的抽象。我们在多个三维形状数据集上评估 ShapeCoder，其中基本图元分解要么解析自人工标注，要么由一种无监督长方体抽象方法生成。在所有领域中，ShapeCoder 都能发现捕捉高层关系、消除多余自由度的抽象库，并且与其他方法相比，实现更好的数据集压缩。最后，我们研究了重写为使用已发现抽象的程序如何有助于下游任务。

CCS 概念： • **计算方法** →**形状建模**。

附加关键词和短语：程序化建模、视觉程序、形状分析、形状抽象、库学习、e-图

作者地址：R. Kenny Jones，russell_jones@brown.edu，美国布朗大学；Paul Guerrero，guerrero@adobe.com，英国 Adobe Research；Niloy J. Mitra，n.mitra@cs.ucl.ac.uk，伦敦大学学院和英国 Adobe Research；Daniel Ritchie，daniel_ritchie@brown.edu，美国布朗大学。

© 2023 版权所有归所有者/作者所有。出版权已授权给 ACM。本文为作者版本，仅供个人使用。请勿再分发。最终定稿版本发表于*ACM*图形学汇刊，https://doi.org/10.1145/3592416。

ACM 参考格式：
R. Kenny Jones、Paul Guerrero、Niloy J. Mitra 和 Daniel Ritchie。2023 年。ShapeCoder：从非结构化基本图元中发现视觉程序的抽象。*ACM Transactions on Graphics* 42，4，文章1（2023年8月），17 页。https://doi.org/10.1145/3592416

1 引言

程序化模型是表示视觉数据的一种颇具吸引力的方式。视觉程序是执行后可生成视觉输出的表达式，相较于其他表示方式具有诸多优势，例如紧凑、易于理解和易于编辑 [Ritchie 等人 2023]。视觉程序涵盖了广泛的领域，而程序化工作流在建模软件中也日益普及 [Freiknecht 和 Effelsberg 2017年; SideFx 2014]。许多可视化编程领域有一个共同特点：其程序通常兼具结构多样性，且包含受复杂参数关系约束的变量。

通常，视觉程序是用面向特定视觉应用的领域专用语言（DSLs）编写的。并非所有视觉程序都同样有用。能够捕捉并约束其所表示视觉数据属性、结构良好的程序，通常有利于下游应用（例如编辑、生成、分析）。另一方面，编写不佳的程序则会失去这一优势。例如，对于由一组基本图元构成的输入视觉场景，将实例化的基本图元简单地合并起来的视觉程序或许能够实现完美重建，却会失去上述底层表示的所有优势。DSL 所包含的函数会影响其能够表示的程序类型，而要找到结构良好的程序，往往离不开一组“优质”函数。

抽象函数能够提取特定领域中结构和参数用法的共同模式，从而显著提升视觉程序的质量；但若没有专家进行人工设计，这类程序（及其抽象）很难获得。

自动抽象发现的理念已在通用编程领域得到研究 [Ellis et al. 2021]。也有一些方法专为视觉领域设计 [Jones et al. 2021],，其中的程序包含复杂的参数关系，使这项任务更加困难。尽管以往的方法在解决这项任务上已取得进展，但没有一种提供了完整的解决方案。限制这类方法应用的两个核心因素是：它们要么设计时未考虑视觉程序（因此无法找到有意义的参数关系），要么依赖难以满足的严格输入假设。

本文提出 ShapeCoder，这是一种能够在宽松假设下为视觉数据发现有用抽象的方法。ShapeCoder 接收一个基础领域特定语言，以及一个由形状构成的数据集；这些形状表示为基本图元的集合，不带任何额外标注。它会针对输入分布，在基础领域特定语言上发现一组抽象函数（即一个库）。ShapeCoder 利用发现的库，寻找能够解释数据集中形状的、包含抽象的程序（图 1）。我们的方法受到以往抽象发现方法的启发，并在此基础上加以改进。

与 DreamCoder [Ellis et al. 2021],（一种用于通用编程领域的库学习方法）类似，我们采用一种交错执行各阶段（梦境、唤醒、提议和整合）的迭代过程。这些阶段会反复运行，逐步发现一组能够最小化基于压缩的目标函数的抽象函数库。梦境阶段会训练一个识别网络，唤醒阶段则利用该网络推断出能够解释输入形状的视觉程序。关键的是，我们设计的识别网络能够为复杂的输入场景找到部分解。因此，即使输入数据集缺少由示例构成的课程（在基础领域特定语言下，有些输入很容易求解），ShapeCoder 仍然能够处理这些数据集；而这正是一些先前工作的局限 [Ellis et al. 2021]。

利用唤醒阶段的程序，提议阶段提出候选抽象，随后集成阶段对其进行推理，以寻找改进后的库版本。这些阶段的灵感来自 ShapeMOD [Jones 等人 2021],，一种专为视觉数据设计的抽象发现方法。ShapeMOD 的集成阶段依赖于对经过筛选的有限程序行序子集进行穷举搜索。ShapeMOD 假定可以访问包含层级部件标注的数据集，以减少行序的数量，但这种方法难以扩展。为克服这一局限，我们设计了一种集成阶段，利用 e-图 [Tate 等人，2011年；Willsey 等人 2021],。e-图是一种数据结构，可通过重写操作高效表示大量等价程序。这使我们能够在庞大的重构程序空间中搜索，并让 ShapeCoder 集成更复杂、与输入分布更匹配的抽象。为使 e-图扩展和提取在我们的问题设定中可处理，我们实现了一种新颖的条件重写方案：在应用抽象重写之前，惰性求值参数关系，从而避免 e-图规模膨胀。

我们在多个视觉领域中运行 ShapeCoder，并证明 ShapeCoder 在所有领域中都能发现有意义的参数关系和结构关系，从而大幅简化输入数据集。根据衡量输入数据集抽象程度的目标函数，我们发现 ShapeCoder 的表现显著优于 Shape-MOD（即使后者可以使用我们的唤醒阶段）和 DreamCoder（若没有任务课程，则无法收敛）。通过一系列消融实验，我们论证了本方法的设计决策，并展示了条件重写方案和自底向上识别网络的重要性。最后，我们探究将我们的方法与能够以无监督方式自动将三维形状转换为基本图元的方法相结合，从而直接从“野外”的三维网格中发现程序和抽象函数 [Chang et al. 2015]。在这种设置下，我们观察到，即使面对嘈杂且不一致的基元分解，ShapeCoder 仍能发现有趣的高层抽象。

总而言之，我们的贡献包括：(i) ShapeCoder，这是一种能够学习推断视觉程序的方法，这些程序利用自动发现的抽象来解释和简化由非结构化基本图元表示的一组形状；(ii) 一种识别网络，即使无法访问任务课程，也能够推断使用已发现抽象的视觉程序；(iii) 一种重构操作，通过条件重写方案扩充等式图，从而以可处理的方式识别复杂参数关系上的匹配项。

我们的代码可在 github.com/rkjones4/ShapeCoder/ 获取。

2 相关工作

我们的方法与许多先前工作相关，这些工作以不同方式涉及视觉程序：非视觉程序的抽象发现、视觉程序归纳（VPI）以及视觉程序生成。我们首先概述这些领域，然后在本节末尾详细讨论与我们最相关的两项工作：DreamCoder [Ellis 等人 2021] 和 ShapeMOD [Jones 等人 2021]。

程序抽象。 此前已有多种方法致力于在上下文无关语言中发现抽象。在立面语法 [Martinovic 和 Van Gool 2013] 或更一般的语法类型 [Hwang 等，2011；Ritchie 等，2018；Talton 等，2012]的背景下，这些语言只能对基本图元或子程序之间有限的一组关系进行建模。针对更一般的程序集合，研究者已在 Exploration-Compression 算法 [Dechter 等， 2013] 以及较新的 DreamCoder [Ellis 等， 2021]中探索了抽象发现。与我们的方法类似，这些方法通过多轮交替进行程序归纳和抽象发现来寻找抽象：在程序归纳阶段，针对给定问题集合发现程序；在抽象发现阶段，检查已发现的程序，以寻找反复出现的模式。我们将在本节末尾更详细地讨论 DreamCoder。最近，研究者针对 DreamCoder 的算法 [Bowers 等，2023]的抽象步骤提出了改进方法。与我们的工作最相关的是，Babble [Cao 等， 2023], 同样使用等式图 [Tate 等， 2009] 来识别抽象应用，但没有专门的机制来处理具有复杂参数化表达式的重写器，这使得 ShapeCoder 能够扩展到复杂的三维视觉领域。Babble 在等式图上采用反统一

来提出抽象；有趣的是，可以考虑如何将这一方案扩展到与我们的等式图公式一起使用，在该公式中，我们明确避免将参数化操作展开为 e-节点。另一个相关问题是在单个 程序中发现共同模式，而不是在一组程序中发现。这一问题已针对 L系统 [Guo 等， 2020] 或 CAD 程序 [Nandi 等，2020；Wu 等，2019]进行了探索。

视觉程序归纳。推断一个能够重建给定目标的视觉程序，是计算机图形学中长期存在的问题。早期方法主要关注植被 [Stava 等，2014年；Xu 和 Mould、建筑立面 [Müller 等，2007年；Wu 等， 2014], 以及城市景观 [Demir 等，2014年；Vanegas 等， 2012]。更全面的综述请参阅 [Aliaga 等， 2016] 。

在近期研究中，通常会在方法的关键部分采用神经网络组件。其中一些方法要求已有程序结构，并且只估计程序参数，使其与给定的三维形状 [Michel 和 Boubekeur，2021年；Pearl 等，2022] 或二维材质 [Shi 等，2020]相匹配。其他方法则试图联合推断程序参数和程序结构。可视化编程领域涵盖多种常见程序类型，例如 CSG 构造序列 [Du 等，2018年；Kania 等，2020年；Ren 等，2021、2022；Sharma 等，2018年；Yu 等， 2022], CAD 建模序列 [Ganin 等，2021年；Li 等，2020、2022；Seff 等，2022年；Xu 等， 2021], SVG 图形 [Reddy 等，2021a、b], 和 L 系统 [Guo 等， 2020], , 也包括自定义程序领域，例如在二维 [Ellis 等， 2018] 和三维 [Tian 等， 2019], 中使用循环和条件语句的基元声明、带线性变换的几何体实例化 [Deng 等， 2022] 以及用于材料的掩码程序化噪声模型 [Hu 等， 2022]。还有少数方法提出了适用于多种程序类型的推理方法 [Ellis 等，2019年；Jones 等， 2022]。与我们的程序领域最相关的是 ShapeAssembly [Jones 等， 2020] 和 ShapeMOD [Jones 等， 2021], , 它们输出用于排列长方体基元的程序。除 ShapeMOD 外，所有这些方法都假设使用包含完整算子集的领域特定语言。我们将在本节末尾单独讨论 ShapeMOD。

视觉程序生成。已有多种深度生成模型被提出，用于生成视觉程序。MatFormer [Guerrero 等人 2022] 生成材料的节点图；若干方法提出了用于生成可缩放矢量图形图像 [Carlier 等，2020；Reddy 等，2021a], 、CAD 草图 [Ganin 等，2021；Para 等，2021；Seff 等， 2022], 以及 3D CAD 建模序列 [Li 等，2022；Wu 等，2021；Xu 等， 2022]的生成模型。前文提到的 ShapeAssembly [Jones 等人 2020] 和 ShapeMOD [Jones 等人 2021] 方法也可以用作生成模型。与视觉程序归纳方法类似，除 ShapeMOD 外，这些方法都需要一个包含完整算子集的领域特定语言。

DreamCoder. 这项工作提出了一种系统，能够联合发现抽象，并在任意函数式编程语言中进行程序归纳 [Ellis et al. 2021]。DreamCoder 的核心是通过三个阶段完成这项艰巨任务。梦境阶段从库中抽样生成随机程序（也可以选择加入抽象）。唤醒阶段训练识别网络，根据梦境样本推断程序。抽象阶段检查唤醒阶段返回的程序语料库，并提出和整合能够改进目标函数的抽象。目标函数权衡程序在库下的似然与库的复杂度。

虽然 DreamCoder 的通用性使其能够有效扩展到各种程序推断任务，但它的抽象纯粹是结构性的，将实值程序组件视为离散化。这意味着它并不适用于形状（或其他视觉领域），因为理想情况下，抽象应同时捕捉复杂的参数关系和结构关系。将 DreamCoder 应用于形状程序的另一个挑战是，它的迭代过程依赖课程来解决任务：其所有阶段（做梦、唤醒、抽象）都依赖于一个假设，即在当前库函数下，至少有一些输入任务的解具有较高的概率。当输入任务构成一个课程时（例如，有些任务在基础领域特定语言下很容易解决），这一过程就能很好地发挥作用，逐步发现越来越多的抽象，从而解决日益复杂的视觉程序归纳任务。不幸的是，当课程假设不成立时，DreamCoder 可能无法为给定领域发现任何程序或抽象。基于这些特性，我们研究了 DreamCoder 在具有参数关系的简单语法上的表现，发现它无法发现 ShapeCoder 能够找到的那类抽象。详情见补充材料。

ShapeMOD。 与 DreamCoder 不同，ShapeMOD 是一个面向视觉数据集（如形状程序）设计的系统。研究表明，它能够发现抽象，从参数表达式和程序结构两方面提取有意义的关系。然而，它并未彻底解决这一问题。ShapeMOD 只能在相当严格的输入假设下找到这些抽象：它要求输入为一组命令式程序，并且已知所有可能的有效行重排。事实上，由于推理所有能够产生相同语义输出的行重排难以处理，因此研究者采用启发式方法，将行的顺序限制在一个非常小的集合内。应用这些启发式方法需要访问分层语义分割，以便将子部分视为彼此独立的子程序。ShapeMOD 的整合和提议阶段（类似于 DreamCoder 的抽象阶段）依赖这些受限的程序重排，既用于发现候选宏，也用于确定何时可以应用这些宏。

ShapeCoder 与 ShapeMOD 的目标相同，但旨在以弱得多的假设发现有用的抽象：它不假设能够访问真实程序、规范线条顺序或层次分解。相反，ShapeCoder 接收一个数据集，其中每个形状都表示为一组无序的基本图元。在这些假设下发现抽象，既需要开发推理逻辑，以推断能够解释输入形状的程序，也需要扩展抽象阶段，使其能够对推断程序的任意重排进行推理。我们通过使用等式图和条件重写方案来解决后一个问题。

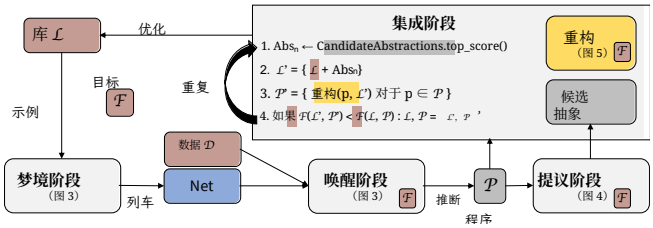

图 2. **概览**。 ShapeCoder 使用初始库 L、目标函数 F 和形状数据集 D（棕色方框）。算法的每一轮都会依次执行一系列阶段，逐步向 L 中添加抽象，以改进 F。**梦想** 阶段通过从 L 中采样来训练识别网络。**Wake** 阶段为 D 中的形状推断程序。**提议** 阶段生成候选抽象。**整合** 阶段使用重构 操作来决定何时将这些抽象添加到 L 中。

3 概述

ShapeCoder 会自动发现一组抽象函数库，这些函数专为形状输入数据集量身定制。它接收以下输入：描述函数式领域专用语言的库 L、形状数据集 D，以及目标函数 F。每个 $d \in D$ 都表示为一组非结构化基元，并且我们假设存在某种 L 的程序扩展 p，使得执行 p 即可重建 d。

ShapeCoder 的目标是最小化 F（第 3.1 节），它体现了 L 对 D 的适用程度（程序复杂度）与添加到 L 中的函数所包含的抽象数量（库复杂度）之间的权衡。我们将这一任务分解为多个步骤，每个步骤分别解决一个易于处理的子问题。图 2 展示了 ShapeCoder 的不同阶段。梦想 阶段（第4.2节）从 L 中采样场景，以训练程序识别网络。唤醒 阶段（第4.3节）利用该网络推断程序 P，以重新创建 D 中的形状。提议 阶段将 P 作为输入，并生成候选抽象函数。最后，集成 阶段会考察提出的候选抽象，并寻找 L 的修改版本以改进 F，随后可将其传递给后续的梦想 阶段。值得注意的是，集成阶段使用重构 函数（第 6 节），通过使用等式图和一种新颖的条件重写方案，以易于处理的方式，在不同库下寻找代价最小的等价程序。

在以下各节中，我们将逐步介绍这些不同阶段。文中的示例和图示采用了一种用于直线形状的玩具二维文法（附录 A）中的程序。更多实现细节见附录 B。

3.1 优化目标 F

ShapeCoder 的目标函数 F 接受两个参数：一个程序库 L，以及一个来自 L、与形状数据集 D 相对应的程序集合。F 衡量两个相互竞争的项之间的权衡：L 和 P 的复杂度。

每个 $p \in P$ 的复杂度根据奥卡姆剃刀原则计算：在其他条件相同的情况下，程序越短越好。我们使用程序词元的加权和来计算程序长度：如果 L 具有 T 种词元类型（例如布尔值、浮点数等），我们允许用户指定一个为每个 $\tau \in T$分配权重 λ。此外，ShapeCoder 采用几何误差函数 err，用于比较每个 $p \in P$执行后的几何形状与其对应的形状 $d \in D$。如果 $err(p, d)$ 返回的值高于用户定义的阈值，F 将返回 ∞。否则，该误差将以权重 λ_e 计入 F。

库复杂度可以通过追踪 L 中包含的函数数量来衡量。ShapeCoder 允许用户指定一个函数加权方案，ω。ω 从 L 中获取一个函数 f，并返回一个范围为 $(0, \infty)$ 的值。较低的 ω 值使得将 f 添加到 L 中更加容易。例如，我们发现，根据 f 所消耗的输入参数数量 f，增加 ω 是有益的，因为这通常表明某种模式过于通用。

借助这一机制，其中 $\tau(p)$表示 p 中类型为 τ的词元数量，我们可以将 ShapeCoder 的目标表示为：

$$\mathcal{F}(\mathcal{L}, \mathcal{P}) = \frac{1}{|\mathcal{P}|}\left(\sum_{p \in \mathcal{P}}\left(\sum_{\tau \in T} \lambda_\tau * \tau(p)\right) + \lambda_e * err(p, d)\right) + \sum_{f \in \mathcal{L}} \omega(f)\,.$$

4 推断视觉程序

虽然 ShapeCoder 将形状数据集 D 作为输入，但它并不知道给定库版本 L 中的哪些程序 P 最能表示 $d \in D$。为了解决这个问题，ShapeCoder 使用程序识别网络（第4.1节），该网络在从 L 中随机采样的程序上进行训练（梦境阶段，第4.2节），以推断使 F 最小化的 P（唤醒阶段，第4.3节）。

为简化这一搜索，我们的识别网络学习推断部分解：从 L 中找出能够重建部分输入基本图元的表达式。随后将找到的表达式组合起来，形成能够解释输入场景的完整程序。这种框架要求 L 包含一种组合子运算（例如并集）。为确保我们的搜索过程总能找到某个 解，我们假设可以使用一种解析方法，找出 L 中能够重建任意基本图元的表达式 d（例如，任意长方体都可以通过一系列缩放、旋转和平移来表示）。

4.1 识别网络

我们在图 3 左侧展示了 ShapeCoder 的识别网络。识别网络以几何基本图元组成的场景作为输入，旨在输出 L 中一个与部分输入基本图元相对应的表达式。我们将该网络实现为一个 Transformer [Vaswani 等人 2017] 提出的解码器，它以自回归方式预测 L 中的词元序列。该网络通过因果掩码，以输入基本图元的编码作为条件：如果每个 M 基本图元由 K个参数表示，那么网络会关注 $K \times M$ 个条件词元（图中示例为$M = 3$ 个和 $K = 4$ 个）。为将表达式转换为词元序列，我们为 L 中的离散元素分配唯一索引。对于实数值参数，我们采用一种简单的映射方法：给定一个输入场景，我们提取基本图元参数化中的所有实数值，通过四舍五入（保留 $K \times M$位小数）将它们分箱，再排序以生成词元映射（浅蓝色方框）。该映射用于构成条件词元，并将网络的预测结果转换回实数值。

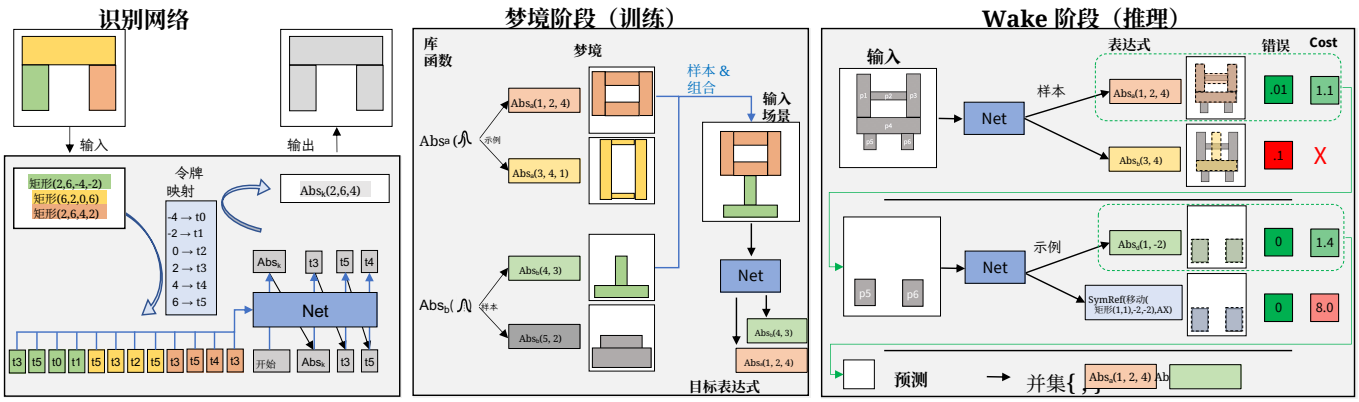

图 3. **梦想与唤醒阶段**。（左）ShapeCoder 的识别网络是一种 Transformer 解码器，它通过关注标记化输入图元，自回归地预测函数和参数化。（中）梦想阶段通过从库函数中采样表达式来训练识别网络，这些表达式经过随机组合，形成（输入、目标）训练对。（右）唤醒阶段利用识别网络寻找能够解释输入形状的程序。在一系列迭代步骤中，它对表达式进行采样，选择代价最优的表达式，并从输入画布中移除已覆盖的基本图元，直到画布为空。

4.2 梦境阶段

梦想阶段通过从 L 中随机采样示例场景来训练识别网络。图 3 中间的方框展示了这一过程。开始梦想阶段时，对于每个函数 $f \in \mathrm{L}$，ShapeCoder 会为 f 生成 N_D 个梦想。通过对 f 的每个参数槽进行随机实例化采样来生成梦想。我们采用拒绝采样，通过检查容易强制满足的属性，避免生成不良几何结构的梦想（几何结构超出场景边界、基本图元具有负尺寸、基本图元完全包含在其他基本图元中等）。

然而，由于 D 中的形状往往包含需要多个函数才能更好解释的场景，仅直接使用特定于函数的梦境进行训练是不够的。我们通过采样特定于函数的梦境，并组合其输出基本图元，形成复合场景来解决这一问题（蓝色箭头）。如果一个复合场景是由组合 K 个采样梦境形成的，那么我们就可以为识别网络导出 K 个配对训练样例：网络的输入是复合场景，而每个 K 个采样梦境都将作为目标输出。例如，给定图 3 中包含橙色和绿色基本图元的输入场景，我们会训练网络预测绿色和橙色表达式序列（即存在一对多映射）。准备好这些配对数据后，只需确保每个 $f \in \mathrm{L}$ 至少出现在 N_D 个目标序列中，即可通过最大似然更新以监督方式训练识别网络。

4.3 唤醒阶段

唤醒阶段接收输入形状 d，并旨在利用识别网络推断出一个使 F 最小化的程序 p。我们在图 3 右侧展示了这一过程。

首先，将场景初始化为包含 d 的基本图元。然后，唤醒阶段以迭代方式执行以下步骤。首先，利用输入场景对识别网络进行条件化处理；识别网络根据其输出概率，从 L 中采样大量表达式，直至达到超时时间（1 秒）。对于每个采样得到的表达式 e，我们记录其代价：e 在 F 下的程序复杂度，并按其所解释的基本图元数量进行归一化。请注意，如果 e 无法重建输入场景中的某个基本图元子集，它就会产生较大的几何误差，而 F 会返回 ∞（图中的红色 X）。唤醒阶段选择代价最低的 e^*（绿色虚线），并从输入场景中移除其覆盖的所有基本图元，然后将更新后的场景反馈给识别网络。重复这些步骤，直至画布为空。满足此条件后，通过对每个 e^* 应用 L 中的组合子运算，形成解释 d 的最终程序 p（例如，底行中橙色和绿色表达式的并集）。对于每个输入场景，都会将 L 中单个基本图元对应的“朴素方案”表达式添加到采样表达式集合中，从而确保能够找到有效解。

在每个 ShapeCoder 轮次中，唤醒阶段都会使用识别网络推断一组能够解释 D 的程序。但我们应该将这些预测独立处理吗？一种做法是在每次唤醒阶段开始前清空 P 中的所有程序条目。然而，这会导致 ShapeCoder“忘记”在先前轮次中发现的优质解。因此，我们采用以下方法：对于轮次 r，$r > 0$，如果 P 中包含先前发现的程序，并且 $\mathrm{P}r$ 包含在轮次 r 的唤醒阶段发现的程序，那么我们将 P 的每个条目设为 $combine(p, p_r)$ 的结果，其中 $combine$ 通过贪心替换搜索来优化 F。

5 提议与集成抽象

梦想与唤醒阶段共同训练并使用识别网络，以推断出一组程序 P，用来解释输入数据集 D 中的形状。提议阶段（第 5.1 节）对 P 进行推理，以提出候选抽象函数；集成阶段（第 5.2 节）则利用这些函数寻找能够改进 F 的库变体。

5.1 提议阶段

提议阶段的目标是在 P 中搜索抽象函数，若将其加入 L，就能改进 F。由于这种搜索

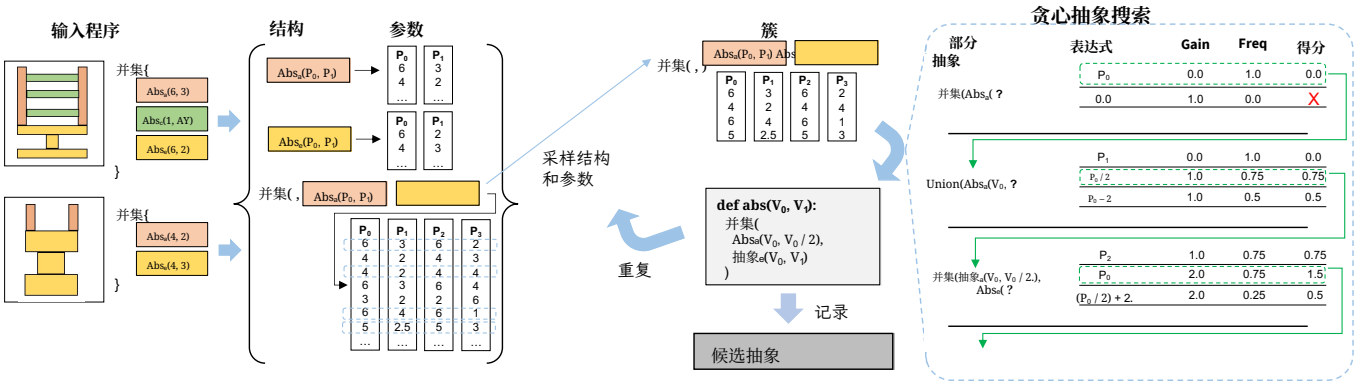

图 4. **提议阶段。** 提议阶段接收一组程序，并输出一组候选抽象。首先，从输入程序中记录可能的结构及其参数化方式。然后，通过采样一种结构和一部分参数化方式来形成簇。对于每个簇，贪心抽象搜索会生成一种可能的抽象，并将其记录下来。

在计算上无法通过全局搜索求解，ShapeCoder 的提议阶段转而求解更易处理的子问题（P 的子集），并汇总局部解。图 4 概述了这一过程。

识别结构和参数。 由于 L 是一种函数式语言，生成一个抽象 a 需要两个步骤：确定 a 的结构（它有哪些子函数），以及确定 a 的参数化方式（a 接收什么输入，以及这些输入如何映射到其子函数）。我们应该考虑哪些结构作为可能的抽象？每个程序 $p \in$ P 都是在唤醒阶段通过组合解决子任务的表达式而得到的，因此 p 不会有一致或规范的顺序。因此，我们希望通过考虑每个 $p \in$ P 中函数可能重排序所形成的结构变体，来消除表达式顺序的影响。然而，由于一般性解决方案难以处理，我们转而只考虑一组有限的潜在抽象结构：在 P 中发现的子表达式的单体及成对组合。我们将所有此类观察到的结构记录为键，并将这些结构的参数化方式记录为值（见图中括号所示的数据结构）。此外，我们发现，应用一个简单的过滤步骤很有用，该步骤会从此映射中移除在 P 中很少出现的结构（出现频率低于 P 的 5%）。

簇采样与搜索。 记录潜在结构及其观测到的参数化后，提议阶段便开始一个迭代过程。为了将全局问题转化为局部问题，随机采样一个结构及其部分参数化，形成一个簇。随后，在该簇上运行贪心搜索，以寻找能够优化 F 的抽象 a。生成的函数会被记录到一个候选抽象数据结构中，该结构跟踪 $p \in$ P 的覆盖集合，这些内容可通过应用 a 进行简化。此过程会重复多次；每当候选抽象数据结构收到一个先前观测到的抽象时，覆盖集合便会扩展。

贪心抽象搜索。 我们采用贪心搜索，为给定的簇寻找抽象 a（图 4 右侧）。该搜索由一个得分 函数引导，该函数提供启发式估计，用于评估将a加入 L 后其对 F 的改进程度。得分 由两个项的乘积构成：频率 和收益。频率 （图中的 Freq 列）是簇中a 能够重建的实例百分比（使用正确的参数化）。收益 表示：每当程序 p可以使用a重写（记作p_a）时，从程序中移除的参数数量。例如，图 4 中提出的抽象在每次能够应用时都会移除两个浮点型参数，这对应于簇中的槽位 P_1 和 P_3。使用 F 中的加权方式（第 3.1 节），我们得到：

$$gain(a) = \sum_{\tau \in T} \lambda_\tau * (\tau(p) - \tau(p_a)) .$$

所提议抽象中的函数序列由采样簇的结构决定，但参数槽位应如何填充？对于每个槽位，我们会考虑一组可能的表达式，计算每个选项的得分，并将得分最高的表达式加入部分抽象。如果频率 始终为零，则得分 会作废。对于浮点型参数槽位，ShapeCoder 会遍历可能参数关系的偏好排序，从而生成表达式。对于离散型参数槽位，可以重用先前实例化的参数，或指定一个静态值。此搜索始终会将定义一个新的自由参数（例如，使用采样簇中的参数化）作为选项（如每个步骤顶行所示）。

5.2 集成阶段

集成阶段接收一个库 L、一组程序 P，以及提议阶段生成的候选抽象。它会搜索 L 的修改版本，以便用于重构 P，从而提升 F。重构 操作（第 6 节）利用等式图，在不同的 L 变体下高效搜索代价最小的等价程序。

集成阶段首先记录起始目标值：F (L, P)。随后，它会迭代执行一系列步骤，尝试贪婪地提升该值。首先，通过采样一个候选抽象并将其加入 L，构建一个新的库变体 L'。选择 得分值最高的抽象，其中 频率 的概念从簇推广到了整个 P。然后，在 L' 下对每个 $p \in$ P 应用 重构 操作，形成新的程序集 P'。最后，如果 F (L', P') 优于 F (L, P)，则将 L 和 P 都替换为修改后的版本。

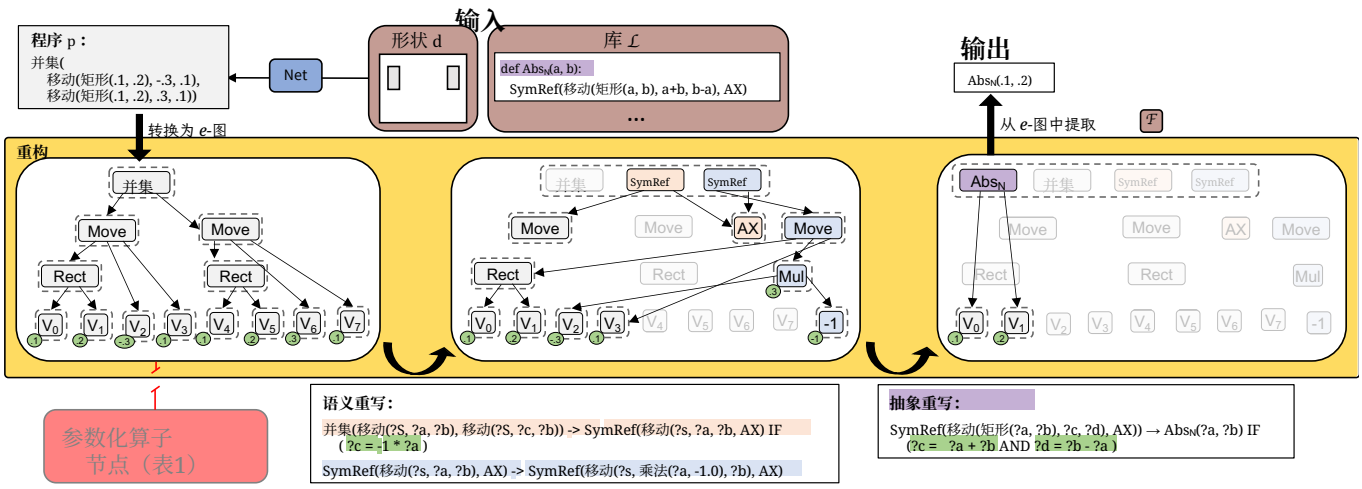

图5。**重构**。 重构操作利用等式图来识别何时可以应用抽象。输入程序会被转换为等式图，并通过语义重写和库特定重写进行扩展，从而发掘可提取的低代价等价表达式。我们设计了一种条件重写方案，能够在不为参数化算子添加过多 e-节点的情况下，对参数关系（绿色高亮部分）进行推理（红色框）。

评估修改后的库 L'成本较高，因为这需要对每个 $p \in$ P运行重构 操作，因此我们通常会在每个集成阶段考虑少量排名靠前的候选抽象，即 N_A。为了尽可能提高得分 启发式方法的准确性，每当将加法 a添加到库 L后，L'使F得到提升时，我们都会检查哪些 $p \in$ P对加法 a的频率 有所贡献，并降低覆盖集合中与之重叠的其他抽象的频率 。

除了这种贪心搜索之外，在集成阶段还会考虑另外两种库变体。当向 L 中添加a无法改善 F 时，我们会计算 P 与 P' 之间频率显著下降的函数集合；将该集合称为 f_{dec}。随后，我们将 $L_{dec} = \{ L + a - f_{dec} \}$ 视为一种库变体。该过程使贪心集成搜索能够避免陷入局部最小值：由于 L 中已经存在相似（但更差）的函数，a 原本不会被添加到 L 中。此外，为完成集成阶段，我们还会考虑逐一移除每个 $f \in$ L 的库变体。在所有比较中，如果某个库变体能够改善目标函数，就将其设为新的默认库。在集成阶段结束时，取得最佳 F 得分的 L 会被传递到后续的梦境阶段，以开始新一轮 ShapeCoder轮次。

6 使用等式图重构程序

ShapeCoder 的集成阶段会评估如何使用库变体紧凑地表示 P，但它如何知道何时可以应用抽象？为此，我们使用重构操作：它以程序 p 为输入，旨在找到 p^*，即与 p 等价且能使 F 最小化的程序。这是一个难以求解的搜索问题，我们通过使用等式图 [Tate 等人 2011] 以及条件重写方案，使其变得可处理。本节余下部分将简要介绍等式图，并通过一个示例（见图 5）说明它们在重构中的作用。

关于等式图的背景。等式图是一种专门的数据结构，能够高效地表示大量等价程序。

我们在图中左侧的引出框中展示了一个 e-图示例。e-图由 e-节点（实线框）和 e-类（虚线框）组成。每个 e-节点都与 L 中的一个项相关联，并且如果该项是一个函数，就会有指向其子 e-类的指针（箭头）。每个 e-类都包含一组等价的 e-节点。e-图的根是这样一个 e-类：它包含与输入表达式中最外层算子（图中的并集）相关联的 e-节点。

当这种表示与重写规则结合使用时，便能发挥作用。重写规则是特定于领域的、通过模式匹配实现的程序变换，同时保持语义等价。例如，对于任意 ?a 和 ?b：Union(?a, ?b) 与 Union (?b, ?a) 等价。通过迭代应用重写规则，等式图得以扩展，从而创建新的 e-类和新的 e-节点。这些新创建的结构引用现有的 e-类和 e-节点，使等式图能够以节省空间的方式表示大量等价程序。重要的是，等式图还支持快速找到起始表达式经过重写后代价最小的版本，方法是从根等价类开始运行贪心递归算法。

重构操作。 重构操作会接收来自唤醒阶段的输入程序 p。首先，它会将 p 转换为一个 e-图，如图5左侧的标注框所示。在此步骤中，每个浮点类型的令牌都会被独立变量（V_0 至 V_7）替换。

该操作还会将库 L 作为输入。它利用 L 提供两类重写操作。语义重写表达了基础领域特定语言函数上的领域知识，并作为语言定义的一部分提供。例如，蓝色重写表达了以下逻辑：将子表达式 ?s 移动到 xy 位置 (?a, ?b)，并关于 X轴进行反射，这等价于将 ?s 移动到 xy 位置 (-1 × ?a, ?b)，并关于 X轴进行反射。抽象重写与 L 中的抽象相对应，其中的重写表达了应用该抽象所需满足的条件。例如，输入库中的 Abs_N (top-middle) 创建了紫色高亮显示的抽象重写（右下方）。

ShapeCoder 通过迭代应用这些重写算子来扩展 e-图。在中间帧中，橙色重写首先将一个新的 AX 轴 e-节点引入新的 e-类，并将一个新的 SymRef e-节点引入根等价类。随后，可以在橙色 e-节点上进行匹配并应用蓝色重写，以添加蓝色高亮显示的 e-节点。此时，可以应用紫色抽象重写，并将一个新的 Abs_N e- 节点添加到根等价类中。重构操作将继续扩展 e- 图，直到其达到饱和状态（无法再添加任何内容）或达到超时时间。

重写扩展等式图后，我们可以在根等价类上运行提取过程，找出等式图中与起始程序 p 等价的代价最低的表达式 p^*。在这个例子中，p^* 将等于 $Abs_N(V_0, V_1)$，我们可以利用最初将程序转换为等式图时所用参数映射的逆映射，将其重写为 Abs_N (.1, .2)。

条件重写方案。 上述说明还缺少一个关键步骤：我们如何知道何时可以应用重写？等式图通常会搜索基于结构模式的匹配项，某些语义重写也可以纳入这一框架（例如蓝色重写）。然而，其他重写（例如紫色抽象重写）则需要同时进行结构匹配和参数匹配。例如，要应用 Abs_N，结构匹配要求找到某个由 e-类构成的子图，该子图与以下模式匹配：SymRef(Move(Rect(?a,?b),?c,?d),AX)，其中 ?a 到 ?d 均可由任意 e-类填充。除此之外，应用 Abs_N还需要进行参数匹配，其逻辑以绿色高亮标出：?c 所在位置必须等于 ?a 和 ?b 两个位置之和，而 ?d 所在位置必须等于 ?b 减去 ?a。

我们如何支持这种参数化匹配？一种朴素方案是将参数约束转换为结构约束：SymRef(Move(Rect(?a,?b),Add(?a,?b),Sub(?b,?a)),AX)。这种方法的问题在于，在尚不清楚这些 e-节点是否有用之前，就必须先将参数化操作（例如 Add 或 Sub）对应的 e-节点添加到 e-图中。当输入参数（V_i）很多时，这种朴素方案会使 e-图的规模急剧膨胀，导致重构操作失效。图中断开的红框展示了我们为避免这种膨胀所做的选择。

ShapeCoder 通过利用条件重写方案，解决了 e-图规模急剧膨胀的问题。条件重写是指先查找结构匹配项，但只有通过额外检查后才应用重写操作。这样，每种参数关系（重写操作中以绿色突出显示）都只会在识别出结构匹配项之后才进行惰性求值。

具体而言，在这个工作示例中，应用紫色重写将找到以下匹配项：?a 与 V_0 匹配，?b 与 V_1 匹配，?c 与 Mul (V_2,-1) 匹配，而 ?d 与 V_3 匹配。要检查参数化关系是否成立，我们需要知道与每个匹配的 e-类相关联的实数值。随后，要检查诸如 ?d = ?b - ?a 这样的关系，我们只需比较 V_3 and V_1 - V_0 之间的数值差异即可。此检查并不要求精确匹配，而是允许用户指定最大误差阈值，从而应用近似等价的重写；这通常是 e-图的一项限制。

表1. 比较我们的条件重写方案与朴素替代方案。即使对于包含许多参数的复杂输入表达式，条件方案也能快速使 e-图达到饱和（时间以秒为单位报告）。当复杂度过高时，朴素方案会超时。

重写方案	8 个参数	16 个参数	32 个参数
朴素方案	.22	2.6	X
条件方案	.01	0.04	2.1

对于某些 e-类，找出其相关联的实数值十分简单：对于每个与浮点类型参数 e-节点（V_0 to V_7）相关联的 e-类，我们记录 e- 类标识符与数值之间的映射。由于某些重写会创建新的浮点类型节点（例如蓝色 Mul e-类），这一过程变得更加复杂。我们通过在所有重写步骤期间动态更新 e-类到实值的映射来处理这种情况（以 e-类上的绿色高亮表示），这是一项常数时间操作。我们的条件重写步骤与非条件重写步骤同样快速，而且关键是避免使用不必要的参数化运算符 e-节点无谓地扩展 e-图。总之，对于 ShapeCoder 通常处理的重构问题，条件重写相比朴素方案可显著提升速度（见表1）。

7 结果与评估

我们在由非结构化基元集合表示的视觉形状分布上运行 ShapeCoder。我们在第 7.1 节中介绍这些领域。在第 7.2 节中，我们比较 ShapeCoder 所发现的抽象相较于其他方法在改善目标函数方面的效果。我们主要与 ShapeMOD [Jones et al. 2021]进行比较。在正文中，我们没有纳入与 DreamCoder [Ellis et al. 2021], 的比较，因为我们发现它在带有参数关系的玩具语法上表现不佳（见补充材料）。在第 7.3 节中，我们分析所发现抽象的属性，并通过事后推断程序研究其通用性。在第 7.4 节中，我们进行消融实验，以研究各种算法组件的重要性。在第 7.5 节中，我们展示了方法的另一种应用：仅给定一个以三维网格为数据的输入数据集，推断包含抽象的视觉程序；在此过程中，我们利用来自预训练无监督长方体分解方法的含噪基元 [Yang and Chen 2021]。最后，在第 7.6 节中，我们探讨 ShapeCoder 所发现的抽象如何惠及下游任务。

7.1 实验领域

在主要结果部分，我们研究三维形状领域。我们在补充材料中提供了二维形状玩具数据集上的实验结果。我们的实验使用来自 PartNet [Mo et al. 2019], 的制造物体，其中人工标注用于将每个三维物体转换为由长方体组成的无结构集合，这些长方体表示部件包围盒。我们遵循三维形状抽象发现领域以往的工作，并在 PartNet 的 椅子、桌子 和 储物家具 类别中的形状上进行实验。我们执行 [Jonesetal. 2020], 所述的长方体简化步骤，使得我们的起始基元集合与 ShapeMOD 使用的相同，但移除了所有层级结构和规范排序信息。

表 2。不同方法发现的抽象库的抽象发现性能，以目标函数 F 进行衡量。

类别	方法	$\mathcal{F}\Downarrow$	$\lvert\mathcal{L}\rvert$	结构数量	参数数量
椅子	输入图元	146.0	6	29	61
	ShapeMOD+输入	109.0	21	16	46
	ShapeMOD+Wake	83.0	21	12	36
	ShapeCoder	**63.6**	33	10	27
桌子	输入图元	125.0	6	25	51
	ShapeMOD+输入	84.2	25	11	34
	ShapeMOD+Wake	69.1	17	10	30
	ShapeCoder	**40.9**	37	8	18
储物家具	输入图元	154.0	6	30	62
	ShapeMOD+输入	119.0	16	20	48
	形状MOD+Wake	103.0	10	19	45
	ShapeCoder	**71.3**	31	11	33

我们用于实验的领域特定语言（附录 A）包含 4 个低层操作：(i) 实例化一个基元（长方体）；(ii) 移动一个形状（移动）；(iii) 旋转一个形状（旋转）；以及 (iv) 将两个形状合并（并集）。我们还在基础领域特定语言中提供了两个中层对称操作，分别对应 (v) 镜像对称和 (vi) 平移对称（SymRef 和 SymTrans）。

7.2 发现抽象

对于每个 PartNet 类别，我们在该类别的 400 个形状上运行 ShapeCoder 四轮。ShapeCoder 使用 Python 和 Rust 实现，采用 PyTorch 和 Egg（一种等式图库）[Willsey 等人 2021]。我们在配备 GeForce RTX 3090 Ti 图形处理器和 Intel i7-11700K CPU 的机器上运行 ShapeCoder，发现为单个类别完成抽象发现所需时间不到 24 小时（最多占用 4GB 图形处理器内存）。

发现能够改进目标函数的抽象。我们在表 2 中报告了从 ShapeCoder 中发现的抽象对我们所优化的目标函数的影响。从左到右，各列分别表示目标函数得分（F，越低越好）、库中包含的函数数量（|L|），以及使用利用所发现抽象的程序来表示形状输入数据集所需的平均操作数量（结构数量）和参数数量（参数数量）。

每个类别的顶部输入图元行展示了 ShapeCoder 的初始目标函数值。该行报告了使用“朴素方案”程序覆盖输入形状的基本图元所需的代价，其中每个基本图元都会在相应命令产生效果时进行旋转、移动和实例化（例如，移动距离为零的操作会被忽略）。ShapeCoder 在底部各行中找到的最终目标函数得分远优于这一初始值。对于椅子、桌子和储物家具，初始目标函数值分别降低了 56%、67% 和 53%。这一改进是通过添加抽象函数（第 2 列）实现的，这些函数移除了表示输入集形状所需的自由度（第 3 列和第 4 列）。

我们还比较了在这种设置下 ShapeCoder 与 ShapeMOD 的表现。ShapeMOD 算法需要以命令式程序数据集作为输入，以及程序中各行可能的排列方式，由于我们的任务设置缺少真实标注程序，我们将 ShapeCoder 与两个版本的 ShapeMOD 进行比较，这两个版本都尝试优化与 ShapeCoder 相同的目标函数：

- *ShapeMOD+*输入：我们将能够直接从基本图元输入集合中解析得到的“朴素方案”程序作为输入提供给 ShapeMOD。
- *ShapeMOD+*唤醒阶段：我们将 ShapeCoder 第一个唤醒阶段的输出作为 ShapeMOD 的输入。需要注意的是，在第一个唤醒阶段的库中，唯一的“非平凡”函数是对称操作，这大致相当于对“朴素方案”程序运行对称检测。

对于这两个程序数据集，我们都无法得知各种表达式（例如，通过并集组合的子形状）应当如何排序，因此我们将所有有效排序中的一个随机子集传递给 ShapeMOD；如果不限制排序集合，ShapeMOD 的运行时间会长到难以接受（见补充材料）。

比较表 2 中的 ShapeMOD 变体与 ShapeCoder，可以清楚地看出，ShapeCoder 找到的抽象显著改善了目标函数，优于 ShapeMOD 找到的抽象。尽管 ShapeCoder 的唤醒阶段比“朴素方案”程序提供了更好的起点，但无论哪种情况，在缺少规范排序和层级标注时，输入程序的复杂度都过高，ShapeMOD 难以有效处理。

我们还将 ShapeCoder 与针对单个程序运行的方法进行比较，例如 Szalinski [Nandi 等人 2020]。Szalinski 同样在视觉程序中使用等式图；尽管其固定的重写规则很适合简化针对机械物体启发式推断出的单个 CAD 程序，但我们发现，这些规则无法显著压缩我们领域中的形状程序：对于椅子，Szalinski 的重写使目标函数从 146 降至 131，而 ShapeCoder 达到了 63。

7.3 对发现的抽象进行分析

图6展示了 ShapeCoder 在 PartNet 形状上运行时发现的部分抽象。识别网络学会了如何利用这些抽象来解释输入数据集中的形状（前三列）。使用这些抽象重写后的程序可以经过编辑来创建新形状，如第四列所示。发现的抽象具有许多理想特性：它们能够表示多样的几何表达式，并通过引入参数化关系来约束许多冗余自由度。ShapeCoder 在后续轮次中发现的抽象可以在子函数调用中引用先前发现的抽象，从而形成抽象的嵌套层级。在极端情况下，ShapeCoder 甚至能够发现可以解释整个输入形状的单个抽象。例如，在顶行的第一列和第三列中，一个抽象函数接收五个输入参数，执行后便能输出一把完整的椅子。利用这类抽象甚至有助于对三维形状进行结构分析。例如，图中展示的桌子抽象（第二行）始终对应于同一语义部件（常规桌腿），尽管该部件可能具有范围很广的输出几何形状。对于每个抽象，我们还展示了部分随机参数化（即梦境），以便直观呈现每个函数所描述的可能输出空间。

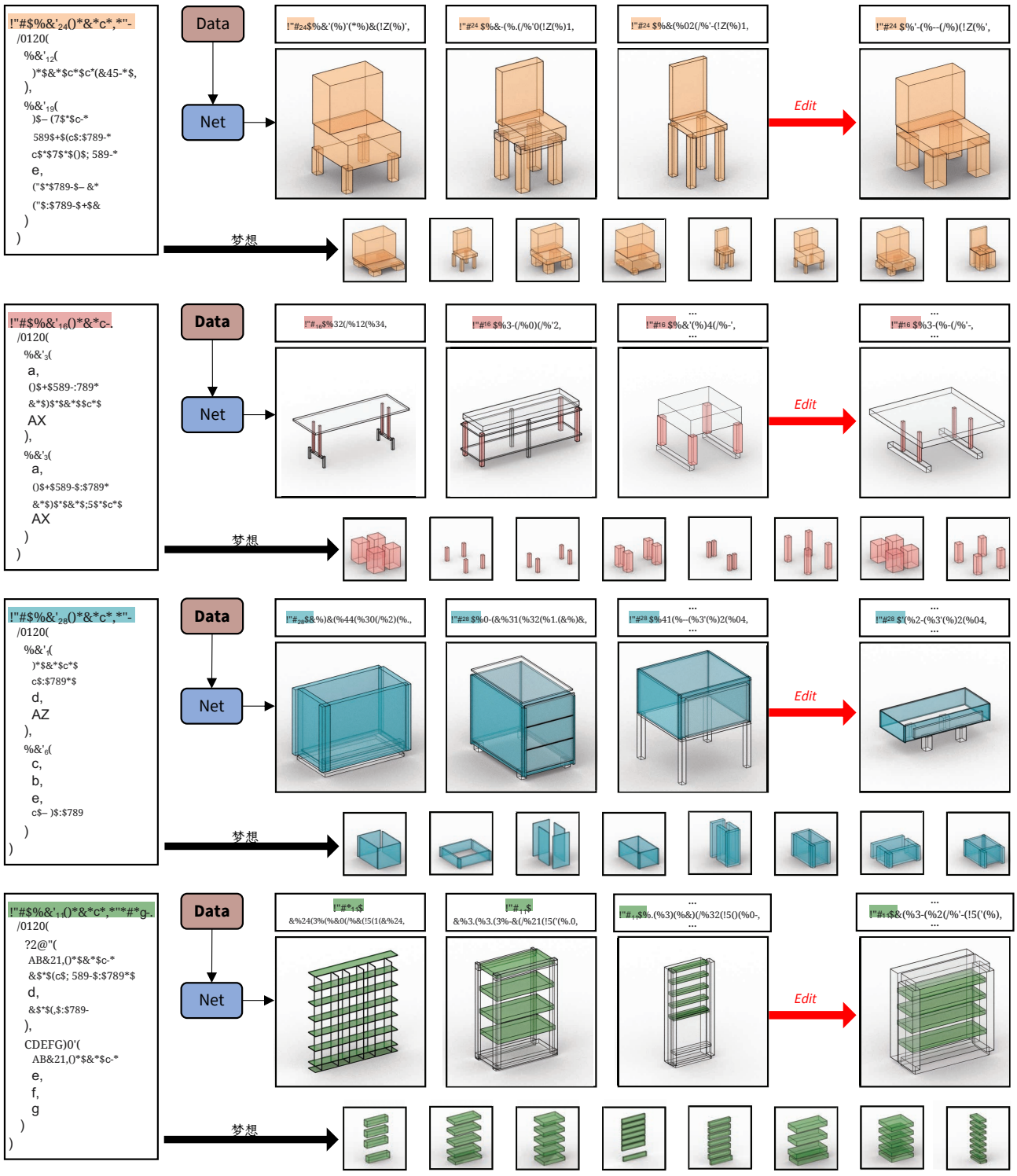

图6. 发现的抽象的定性示例。我们分别展示了一个椅子和桌子类别的抽象，以及两个储物家具类别的抽象。左侧展示抽象代码，其后是 ShapeCoder 在我们的形状数据集中发现的该抽象的三种不同用法。在最右侧一列中，我们手动编辑发现的程序，以创建新的形状。底部展示了随机采样的梦境。

表 3。我们通过比较 ShapeCoder 使用事后推断（PHI）压缩留出集（Val）中形状的效果（目标函数 F）与其正常运行期间发现的程序（顶行）的压缩效果，来衡量其所发现抽象的通用性。

形状集	推理方法	$\mathcal{F}$	抽象数量
训练	ShapeCoder	63.6	4.31
训练	PHI	67.5	4.67
Val	PHI	70.6	4.77

表 4. （左）通过跟踪目标函数的改进，对 ShapeCoder 的设计决策进行消融分析（条件详情见第 7.4 节）。我们的默认配置（底部）表现最佳。（右）测量在扰动（噪声水平）不断增大时，具有或不具有抽象的程序的输出执行有效性（使用弗雷歇距离）。抽象有助于在参数编辑后使形状保持在“分布内”。

条件	$\mathcal{F}\Downarrow$
无抽象	104.9
单次迭代	81.6
无梦想+Wake	99.0
无语义 Rws	75.2
无条件 Rws	100.0
无抽象偏好	70.7
ShapeCoder	**63.6**

噪声水平	无抽象	有抽象
0.1	**8**	**8**
0.2	18	**13**
0.3	40	**27**
0.4	88	**48**
0.5	157	**84**

事后推断。在抽象发现过程中，ShapeCoder 会找到利用抽象来解释其输入数据集中形状的程序。我们研究这些抽象能否推广到来自同一分布、但未纳入其优化过程的形状。我们利用 ShapeCoder 的识别网络，寻找能够解释未纳入抽象发现“训练”阶段的形状的程序。我们在这些形状上运行唤醒阶段，以寻找能够解释输入基本图元集的程序。随后，将这些程序传入重构操作，查看库重写规则能否进一步改进程序。

我们在表 3 中展示了对 PartNet 的椅子类别形状执行事后推断（PHI）过程的结果。表格的第一行显示了 ShapeCoder “训练”期间通过迭代构建的程序（例如，抽象发现）的目标函数值和平均抽象使用次数。中间一行中，我们对同一组形状“遗忘”在抽象发现过程中发现的程序，并运行 PHI 过程，尝试从头推断程序。最后一行中，我们对 ShapeCoder 此前从未见过的验证形状运行 PHI。虽然事后执行推理的效果略逊于经过多轮迭代发现程序，但对“训练”形状和“验证”形状运行 PHI 的结果差异相对较小。这一事实，加上抽象使用次数一栏始终较高的数值，表明 ShapeCoder 发现的许多抽象都能泛化到其所优化的数据集之外的形状。

7.4 ShapeCoder 消融实验

为了评估 ShapeCoder 背后的设计决策，我们进行了消融实验，追踪移除方法中的不同组件会如何影响我们发现的抽象类型，以及这些抽象会如何影响目标函数的优化。我们考虑以下消融条件：

- 无抽象：我们报告仅运行一次唤醒阶段、且不经过抽象阶段的结果。
- 单次迭代：我们仅对 ShapeCoder 运行一轮。
- 没有梦想+*Wake*：我们在无法访问识别网络的情况下运行 ShapeCoder 多轮次。取而代之的是，使用“朴素方案”程序来初始化算法。
- 无语义 *Rws*：我们在重构操作中移除了与基础领域特定语言相关的所有语义重写。
- 无条件 *Rws*：我们将条件重写方案替换为第 6 节中所述的“朴素方案”。
- 无抽象偏好：我们移除了偏好加权 ω，详见第 3.1 节。

我们使用 PartNet 的 椅子 类别中的形状，在表 4 左侧报告了这些不同变体的表现。所有消融条件都导致优化效果不如默认配置（底行）。如果没有抽象阶段，唤醒阶段返回的程序就无法利用高阶函数。仅进行一次 ShapeCoder 迭代，无法发现层次化抽象，而且唤醒阶段也无法学习更广泛地应用已发现的抽象。如果在没有梦想或唤醒阶段的情况下运行抽象阶段，该方法也会遇到类似问题：抽象可能未得到充分利用，也不会被整合到所有可用于表示的形状中。语义重写使等式图能够表示大量等价程序，我们在重构期间可对其进行高效搜索；如果不考虑这大量等价程序，我们就会再次对提出的抽象应用不足。无条件重写消融实验清楚地表明了条件重写方案的重要性：在分配给该消融实验的计算预算（3 天）内，使用“朴素方案”重写方案的 ShapeCoder 版本未能完成整个抽象阶段。因此，我们报告其在 3 天截止时的目标函数值。最后，我们的偏好加权方案有助于 ShapeCoder 避免局部极小值：主要是通过降低明显不佳（例如限制过多或过于宽泛）的候选抽象函数的权重。

7.5 从非结构化形状中发现抽象

作为 ShapeCoder 的一个示例应用，我们研究它在处理三维网格数据集时联合发现抽象函数库以及使用这些抽象的程序的能力。为了获取这类输入数据，我们采用一种对三维形状进行无监督长方体分解的方法 [Yang and Chen 2021]。具体而言，我们使用这种方法将一组 ShapeNet 网格转换为由非结构化、含噪基本图元组成的排列形式——一种 ShapeCoder 可以推理的数据格式。关于这一数据预处理的详细信息，请参阅附录 B.8

与第 7.2 节中的实验类似，我们构建了一个包含 400 个形状的数据集，其中的基本图元由这一无监督算法生成。我们在一个来自

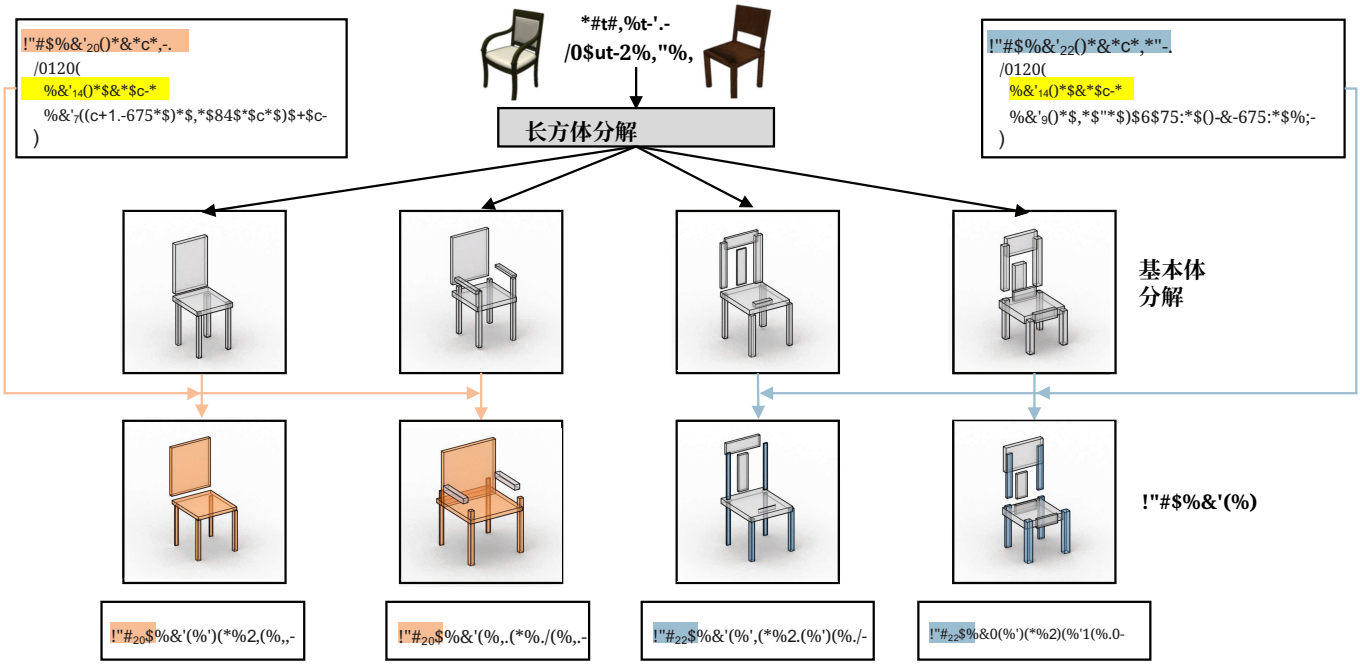

图 7. 我们采用一种无监督基元分解方法 [Yang and Chen 2021]，在三维网格数据集上运行 ShapeCoder。即使面对这些含噪基元分解，我们的方法仍能找到高层次且实用的抽象，以捕捉有意义的形状变化维度。有趣的是，我们展示的两个顶层抽象（分别以橙色和蓝色标示）都使用了同一个抽象子函数（以黄色突出显示）来创建一个四足底座。

ShapeNet [Chang 等人 2015] 的椅子数据集上运行 ShapeCoder 三轮，并在图7中展示了发现的部分抽象结果。尽管 ShapeCoder 接收到的基本体分解存在噪声且不规则，但它仍成功发现了一组有意义的抽象函数，这些函数揭示了更高阶的属性，并可应用于输入分布中的不同实例。例如，发现的 Abs_{20}捕捉到了与 ShapeCoder 在 PartNet 标注上运行时发现的相同基本椅子结构（Abs_{24}，图6）。事实上，经过 3 轮后，ShapeCoder 将目标函数得分提高了 61%（140 → 53.9），这与 ShapeCoder 处理干净的人工标注部件时观察到的量化提升相近。这些结果令人鼓舞，表明像 ShapeCoder 这样的系统能够发现复杂视觉现象的实用高级程序化表示，而无需依赖人工标注。

7.6 抽象的下游效益

本节通过两项实验，探究 ShapeCoder 发现的抽象如何使下游应用受益：在扰动下保持有效性，以及新颖形状合成。

在扰动下保持有效性。 由于我们的目标是发现能够去除多余自由度的抽象，因此可以通过扰动形状程序中的自由度，并检查它们是否“仍处于分布内”来评估效果。我们选取两个形状程序数据集，其中的程序分别使用或不使用抽象编写，并在不同噪声水平下扰动其参数。具体而言，噪声水平调节拟合到每个 DSL 函数的各个参数槽上的高斯噪声分布的标准差。对于每组经过扰动的程序，我们在预训练模型的特征空间中使用弗雷歇距离（FD），衡量其输出执行结果与验证集的相似度。我们在表 4 右侧报告了该实验的结果。我们发现，使用 ShapeCoder 发现的抽象重写程序，有助于在参数受到扰动时使形状保持“分布内”，这是目标导向编辑任务的一项重要属性。

新颖形状合成。 我们评估学习编写新颖形状程序的生成模型，是否能从使用已发现的抽象改写后的程序训练中获益。在这项实验中，我们使用事后推断过程（第 7.3 节）构建一个数据集，其中包含 3600 个采用 ShapeCoder 发现的抽象编写的椅子程序。我们使用该数据集训练一个自回归网络——Transformer 解码器，使其学习生成子程序；生成过程以画布为条件，该画布记录先前预测的程序部分的执行输出（附录 B.9）。为合成新颖形状，网络从空白画布开始，随后通过迭代采样表达式并将其输出添加到画布中，逐步构建复杂程序，直到预测出 STOP标记。

我们在图 8 中展示了该模型的输出。从定性角度看，我们发现该模型能够生成训练集中未出现过的新形状，同时明显保持在训练分布之内。从定量角度看，我们将该模型的输出与一个消融版本进行比较，后者在不含抽象的程序上训练。结果发现，通过抽象编写的程序进行学习，能将弗雷歇距离（相对于验证集）从 17.1 降至 13.8，提升了 19%



图 8. 从能够编写包含抽象的程序的生成模型中采样得到的程序（上），以及与之最相似的样本（下）。

。此外，基于视觉程序的生成模型若能通过抽象进行学习，则尤其具有吸引力，因为它们输出的程序冗余自由度更少，也更适合用于下游任务。

8 结论

我们介绍了 ShapeCoder，这是一个能够在由非结构化基本图元表示的一组形状中发现视觉程序抽象的系统。我们的方法不需要任何额外监督，例如真实程序、特定的程序操作顺序或程序课程。我们展示了 ShapeCoder 能够发现高级抽象，并在其他最先进的方法无法处理的领域中实现显著压缩。
ShapeCoder 还能找到利用这些抽象来简洁解释优化期间未观察到的形状的程序。最后，我们通过展示 ShapeCoder 在无监督方法生成的含噪基元分解上运行时，能够发现捕捉有意义自由度的实用抽象，证明了它的灵活性。

8.1 局限性与未来工作

虽然 ShapeCoder 是首个能够直接从非结构化基元中发现非平凡程序抽象的方法，但它也存在一些重要的局限性：

(i) 冗余的抽象。我们发现，多个抽象解释了同一个概念。虽然这些抽象可以看作同一语义概念的结构变体（例如柱脚椅底座和四腿椅底座），但对于下游任务而言，这些抽象后的程序可能显得冗余。这一点很难避免，因为目前我们不会“执行”这些程序来比较它们的几何输出。未来，我们希望探索“条件式”或“概率式”抽象。例如，椅子底座抽象可以根据离散输入参数，或根据为每种变体指定的概率，展开为柱脚底座或四腿底座。

(ii) 未饱和的等式图。对于复杂的输入表达式，由于等式图无法高效表示结合律-交换律约束，因此将其完全饱和在计算上可能不可行。虽然 ShapeCoder 并未直接解决这一问题，但我们采用条件重写，避免插入多余的参数化操作节点。这有助于缓解指数级膨胀，并使 ShapeCoder 能够探索比以往工作丰富得多的程序结构。尽管如此，在给定的计算预算内，我们并不总能将等式图完全饱和。这意味着一些可能有用的重写未能得到探索，也就无法加入抽象库。一种可能的做法是借助神经组件分摊集成阶段的成本：可以学习重写规则（例如，在强化学习设置中采用奖励结构），学习扩展等式图的哪些部分，或者将“大型”重写的任务交给学习模块，而不是等式图。不过，在无监督设置下训练此类模块仍有待进一步研究。

(iii) 自底向上的唤醒网络。 ShapeCoder 的识别网络（在唤醒阶段使用）会解决通过组合子操作拼接在一起的子问题。这种设计决策的一个缺点是，每当库版本发生变化时，都必须重新训练识别网络。此外，由于网络不会一次性预测完整程序，推理的运行成本可能很高，而且在整个数据集上推断程序的方式一致性较差。将这个自底向上网络替换为自顶向下网络会更具挑战性。不过，这也会带来其他可能性，例如不再需要将输入数据表示为基本图元的集合。

展望未来，我们相信 ShapeCoder 对许多其他可视化编程领域也会有所帮助，而不仅限于本报告中探讨的二维和三维形状文法。ShapeCoder 要求领域具备以下属性：*(a)* 语言是函数式的，*(b)* 包含组合子运算（例如并集），且 *(c)* 视觉输入可以分解为基本类型。事实上，属性 *(b)*和 *(c)* 仅在唤醒阶段才需要，因此可以通过使用接收“原始”视觉数据的程序推断网络来放宽这一要求。草图、构造实体几何、可缩放矢量图形，甚至着色器程序，都可能适合未来的探索。ShapeCoder 首次实现了直接从无结构的基本图元集合中发现程序抽象，从而减轻了收集、标注和归类形状类别的负担。

致谢

我们谨此感谢 Srinath Sridhar 和匿名审稿人提出的宝贵建议。形状程序的渲染图使用 Blender Cycles 渲染器生成。本研究部分由美国国家科学基金会资助项目 #1941808、布朗大学校长奖学金以及欧洲研究委员会资助项目（SmartGeometry）提供经费支持。Daniel Ritchie 是 Geopipe 的顾问，并持有该公司的股权。Geopipe 是一家初创公司，致力于开发三维技术，以构建真实世界的沉浸式虚拟副本，并将其应用于游戏、建筑等多个领域。

参考文献

Daniel G. Aliaga、Ilke Demir、Bedrich Benes 和 Michael Wand。2016 年。逆向过程虚拟世界三维模型的程序化建模。在*ACM SIGGRAPH 2016* 课程*(SIGGRAPH'16)*。文章16，共316页。https://doi.org/10.1145/2897826.2927323
Matthew Bowers、Theo X. Olausson、Lionel Wong、Gabriel Grand、Joshua B. Tenenbaum、Kevin Ellis 和 Armando Solar-Lezama。2023。面向库的自顶向下合成学习。*Proc.ACM*程序.*Lang.* 7，POPL，文章 41（2023 年 1 月），32 页。https://doi.org/10.1145/3571234
David Cao、Rose Kunkel、Chandrakana Nandi、Max Willsey、Zachary Tatlock，以及 Nadia Polikarpova. 2023. Babble：利用等式图和学习更好的抽象

反统一。《*ACM*程序设计语言汇刊》7，POPL，文章14（2023年1月），29页。https://doi.org/10.1145/3571207 Alexandre Carlier、Martin Danelljan、Alexandre Alahi 和 Radu Timofte。2020年。DeepSVG：用于矢量图形动画的分层生成网络。载于《神经信息处理系统进展》（*NeurIPS*），第33卷。16351–16361。Angel X Chang、Thomas Funkhouser、Leonidas Guibas、Pat Hanrahan、Qixing Huang、Zimo Li、Silvio Savarese、Manolis Savva、Shuran Song、Hao Su、Jianxiong Xiao、Li Yi 和 Fisher Yu。2015年。ShapeNet：信息丰富的三维模型库。*arXiv*预印本 *arXiv:1512.03012*（2015年）。Eyal Dechter、Jon Malmaud、Ryan P. Adams 和 Joshua B. Tenenbaum。2013年。通过模块化概念发现进行自举学习。载于第二十三届国际人工智能联合会议论文集（中国北京）（*IJCAI '13*）。AAAI出版社，1302–1309。Ilke Demir、Daniel G Aliaga 和 Bedrich Benes。2014年。城市尺度的建筑程序化。载于*2014*年第二届国际三维视觉会议，第1卷。IEEE，456–463。Boyang Deng、Sumith Kulal、Zhengyang Dong、Congyue Deng、Yonglong Tian 和 Ji ajun Wu。2022年。利用可重复隐式部件无监督学习形状程序。载于《神经信息处理系统进展》（*NeurIPS*）。Tao Du、Jeevana Priya Inala、Yewen Pu、Andrew Spielberg、Adriana Schulz、Daniela Rus、Armando Solar-Lezama 和 Wojciech Matusik。2018年。InverseCSG：将三维模型自动转换为构造实体几何树。载于亚洲计算机图形学与交互技术年会（*SIGGRAPH Asia*）。ACM。Kevin Ellis、Maxwell Nye、Yewen Pu、Felix Sosa、Joshua B. Tenenbaum 和 Armando Solar-Lezama。2019年。编写、执行、评估：借助读取—求值—打印循环进行程序综合。载于《神经信息处理系统进展》（*NeurIPS*）。Kevin Ellis、Daniel Ritchie、Armando Solar-Lezama 和 Josh Tenenbaum。2018年。学习从手绘图像推断图形程序。载于《神经信息处理系统进展》（*NeurIPS*）。Kevin Ellis、Catherine Wong、Maxwell Nye、Mathias Sablé-Meyer、Lucas Morales、Luke Hewitt、Luc Cary、Armando Solar-Lezama 和 Joshua B Tenenbaum。2021年。DreamCoder：通过醒眠式库学习自举归纳式程序综合。载于*ACM SIGPLAN*编程与软件新思想、新范式及反思国际研讨会（*SIGPLAN*）。835–850。Jonas Freiknecht 和 Wolfgang Effelsberg。2017年。虚拟世界程序化生成综述。《多模态技术与交互》1，4（2017年）。https://doi.org/10.3390/mti1040027 Yaroslav Ganin、Sergey Bartunov、Yujia Li、Ethan Keller 和 Stefano Saliceti。2021年。作为语言的计算机辅助设计。载于《神经信息处理系统进展》（*NeurIPS*）。Paul Guerrero、Milos Hasan、Kalyan Sunkavalli、Radomir Mech、Tamy Boubekeur 和 Niloy Mitra。2022年。MatFormer：用于程序化材质的生成模型。《*ACM*图形学汇刊》（*TOG*）41，4，文章46（2022年）。Jianwei Guo、Haiyong Jiang、Bedrich Benes、Oliver Deussen、Xiaopeng Zhang、Dani Lischinski 和 Hui Huang。2020年。通过推断L系统对分支结构进行逆向程序化建模。《*ACM*图形学汇刊》（*TOG*）39，5（2020年），1–13。Yiwei Hu、Chengan He、Valentin Deschaintre、Julie Dorsey 和 Holly Rushmeier。2022年。用于空间变化双向反射分布函数贴图的逆向程序化建模流程。《*ACM*图形学汇刊》（*TOG*）41，2（2022年），1–17。Irvin Hwang、Andreas Stuhlmüller 和 Noah D. Goodman。2011年。通过贝叶斯程序合并归纳概率程序。*CoRR* arXiv:1110.5667（2011年）。R. Kenny Jones、Theresa Barton、Xianghao Xu、Kai Wang、Ellen Jiang、Paul Guerrero、Niloy J. Mitra 和 Daniel Ritchie。2020年。ShapeAssembly：学习生成用于三维形状结构合成的程序。《*ACM*图形学汇刊》（*TOG*），*Siggraph Asia 2020*39，6（2020年），文章234。R. Kenny Jones、David Charatan、Paul Guerrero、Niloy J. Mitra 和 Daniel Ritchie。2021年。ShapeMOD：三维形状程序的宏操作发现。《*ACM*图形学汇刊》（*TOG*），*Siggraph 2021*40，4（2021年），文章153。R. Kenny Jones、Homer Walke 和 Daniel Ritchie。2022年。PLAD：利用伪标签和近似分布学习推断形状程序。载于*IEEE/CVF*计算机视觉与模式识别会议（*CVPR*）。Kacper Kania、Maciej Zieba 和 Tomasz Kajdanowicz。2020年。UCSG-NET——无监督发现构造实体几何树。载于《神经信息处理系统进展》（*NeurIPS*），第33卷。8776–8786。Changjian Li、Hao Pan、Adrien Bousseau 和 Niloy J. Mitra。2020年。Sketch2CAD：通过上下文草绘进行序列式CAD建模。《*ACM*图形学汇刊》（*TOG*）39，6（2020年），164:1–164:14。Changjian Li、Hao Pan、Adrien Bousseau 和 Niloy J. Mitra。2022年。Free2CAD：将徒手绘图解析为CAD命令。《*ACM*图形学汇刊》（*TOG*）41，4（2022年），93:1–93:16。Andelo Martinovic 和 Luc Van Gool。2013年。用于逆向程序化建模的贝叶斯语法学习。载于*2013*年*IEEE*计算机视觉与模式识别会议。201–208。https://doi.org/10.1109/CVPR.2013.33 Elie Michel 和 Tamy Boubekeur。2021年。用于参数化形状逆向控制的DAG修订。《*ACM*图形学汇刊》40，4（2021年），173:1–173:14。

Kaichun Mo、Shilin Zhu、Angel X. Chang、Li Yi、Subarna Tripathi、Leonidas J. Guibas 和 Hao Su。2019年。PartNet：用于细粒度和层次化部件级三维对象理解的大规模基准。在 *IEEE*计算机视觉与模式识别会议（*CVPR*）。Pascal Müller、Gang Zeng、Peter Wonka 和 Luc Van Gool。2007年。基于图像的立面程序化建模。*ACM Transactions on Graphics* 26，3（2007年），85。Chandrakana Nandi、Max Willsey、Adam Anderson、James R. Wilcox、Eva Darulova、Dan Grossman 和 Zachary Tatlock。2020年。利用等式饱和与逆变换合成结构化CAD模型。在 第*41*届 *ACM SIGPLAN*编程语言设计与实现会议论文集（英国伦敦）（*PLDI 2020*）。美国纽约州纽约市计算机协会，31–44。https://doi.org/10.1145/3385412.3386012 Wamiq Reyaz Para、Shariq Farooq Bhat、Paul Guerrero、Tom Kelly、Niloy Mitra、Leonidas Guibas 和 Peter Wonka。2021年。SketchGen：生成受约束的CAD 草图。在 神经信息处理系统进展（*NeurIPS*）。Ofek Pearl、Itai Lang、Yuhua Hu、Raymond A. Yeh 和 Rana Hanocka。2022年。GeoCode：可解释的形状程序。arXiv:2212.11715 [cs.GR] Pradyumna Reddy、Michael Gharbi、Michal Lukac 和 Niloy J Mitra。2021a年。Im2Vec：无需向量监督地合成矢量图形。在 *IEEE/CVF*计算机视觉与模式识别会议（*CVPR*）。7342–7351。Pradyumna Reddy、Zhifei Zhang、Matthew Fisher、Hailin Jin、Zhaowen Wang 和 Niloy J Mitra。2021b年。用于字体的多重隐式神经表示。在 神经信息处理系统进展（*NeurIPS*）。Daxuan Ren、Ji anmin Zheng、Jianfei Cai、Jiatong Li、Haiyong Jiang、Zhongang Cai、Junzhe Zhang、Liang Pan、Mingyuan Zhang、Haiyu Zhao 等。2021年。CSG-Stump：一种适用于学习的、类似构造实体几何的可解释形状解析表示。在 *IEEE/CVF*国际计算机视觉会议（*ICCV*）。12478–12487。Daxuan Ren、Jianmin Zheng、Jianfei Cai、Jiatong Li 和 Junzhe Zhang。2022年。ExtrudeNet：用于形状解析的无监督逆向草图-拉伸。在欧洲计算机视觉会议（*ECCV*）。Daniel Ritchie 、 Paul Guerrero 、 R. Kenny J ones、Niloy J. Mitra、Adriana Schulz、Karl D. D. Willis 和 Jiajun Wu。2023。面向计算机图形学的神经符号模型。*Computer Graphics Forum*（2023）。Daniel Ritchie、Sarah Jobalia 和 Anna Thomas。2018年。基于示例的程序化建模程序编写：结构与连续可变性。在 *EUROGRAPHICS*。Ari Seff、Wenda Zhou、Nick Richardson 和 Ryan P. Adams。2022年。Vitruvion：参数化CAD 草图的生成模型。在 国际机器学习表示会议（*ICLR*）。Gopal Sharma、Rishabh Goyal、Difan Liu、Evangelos Kalogerakis 和 Subhransu Maji。2018年。CSGNet：用于构造实体几何的神经形状解析器。在 *IEEE*计算机视觉与模式识别会议（*CVPR*）。Liang Shi、Beichen Li、Miloš Hašan、Kalyan Sunkavalli、Tamy Boubekeur、Radomir Mech 和 Wojciech Matusik。2020年。MATch：用于程序化材质捕获的可微材质图。*ACM Transactions on Graphics* 39，6（2020年12月），1–15。SideFx。2014年。程序化建模简介。https://www.sidefx.com/learn/ collections/introduction-to-procedural-modeling/。访问日期：2023-04-18。Ondrej Stava、Sören Pirk、Julian Kratt、Baoquan Chen、Radomír Měch、Oliver Deussen 和 Bedrich Benes。2014年。树木的逆向程序化建模。在*Computer Graphics Forum*，第33卷。Wiley Online Library，118–131。Jerry O. Talton、Lingfeng Yang、Ranjitha Kumar、Maxine Lim、Noah D. Goodman 和 Radomír Mech。2012年。通过贝叶斯语法归纳学习设计模式。在 *UIST*。Ross Tate、Michael Stepp、Zachary Tatlock 和 Sorin Lerner。2009年。等式饱和：一种新的优化方法 （*POPL '09*）。美国纽约州纽约市计算机协会，264–276。https://doi.org/10.1145/1480881.1480915 Ross Tate、Michael Stepp、Zachary Tatlock 和 Sorin Lerner。2011年。等式饱和：一种新的优化方法。*Logical Methods in Computer Science* 第7卷，第1期（2011年3月）。https://doi.org/10.2168/LMCS-7(1:10)2011 Yonglong Tian、Andrew Luo、Xingyuan Sun、Kevin Ellis、William T. Freeman、Joshua B. Tenenbaum 和 Jiajun Wu。2019年。学习推断并执行三维形状程序。在 国际机器学习表示会议（*I CLR*）。Carlos A Vanegas、Ignacio Garcia-Dorado、Daniel G Aliaga、Bedrich Benes 和 Paul Waddell。2012年。城市程序化模型的逆向设计。*ACM*图形学汇刊（*TOG*） 31，6（2012年），1–11。Ashish Vaswani、Noam Shazeer、Niki Parmar、Jakob Uszkoreit、Llion Jones、Aidan N Gomez、Ł ukasz Kaiser 和 Illia Polosukhin。2017年。注意力就是你所需要的一切。在 神经信息处理系统进展，I. Guyon、U. Von Luxburg、S. Bengio、H. Wallach、R. Fergus、S. Vishwanathan 和 R. Garnett（编），第30卷。Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf Max Willsey、Chandrakana Nandi、Yisu Remy Wang、Oliver Flatt、Zachary Tatlock 和 Pavel Panchekha。2021年。egg：快速且可扩展的等式饱和。*Proc. ACM Program. Lang.* 5，POPL，文章23（2021年1月），29页。https://doi.org/10.1145/ 3434304

Chenming Wu、Haisen Zhao、Chandrakana Nandi、Jeffrey I. Lipton、Zachary Tatlock 和 Adriana Schulz。2019年。Carpentry Compiler。*ACM Transactions on Graphics* 38，6，文章195（2019年11月），14页。https://doi.org/10.1145/3355089.3356518 Fuzhang Wu、Dong-Ming Yan、Weiming Dong、Xiaopeng Zhang 和 Peter Wonka。2014年。立面布局的逆向程序化建模。*ACM Transactions on Graphics* 33，4，文章121（2014年7月），10页。https://doi.org/10.1145/2601097.2601162 Rundi Wu、Chang Xiao 和 Changxi Zheng。2021年。DeepCAD：用于计算机辅助设计模型的深度生成网络。在 *IEEE/CVF* 国际计算机视觉会议（*ICCV*）。6772–6782。Ling Xu 和 David Mould。2015年。利用引导向量进行程序化树木建模。在 *Computer Graphics Forum*，第34卷。Wiley Online Library，47–56。Xianghao Xu、Wenzhe Peng、Chin-Yi Cheng、Karl D. D. Willis 和 Daniel Ritchie。2021年。利用区域图推断CAD建模序列。在 *IEEE/CVF*计算机视觉与模式识别会议（*CVPR*）。Xiang Xu、Karl DD Willis、Joseph G Lambourne、Chin-Yi Cheng、Pradeep Kumar Jayara- man 和 Yasutaka Furukawa。2022年。SkexGen：利用解耦码本自回归生成CAD构造序列。在 国际机器学习会议（*ICML*）。

Kaizhi Yang 和 Xuejin Chen . 2021. 长方体形状抽象的无监督学习 - 通过点云的联合分割进行构造。*ACM Transactions on Graphics* 40, 4, 文章 152（2021年7月），11页。https://doi.org/10.1145/3450626.3459873

Fenggen Yu、Zhiqin Chen、Manyi Li、Aditya Sanghi、Hooman Shayani、Ali Mahdavi - Amiri 和 Hao Zhang。2022年。CAPRI-Net：通过自适应基元组装学习紧凑CAD形状。发表于*IEEE/CVF*计算机视觉与模式识别会议（*CVPR*）。11768–11778。

一种形状文法

三维形状文法。下面我们将详细介绍我们的三维形状文法：

```
开始 →形状
形状 → 并集(形状,形状) |
       SymRef(形状, 轴) |
       SymTrans(形状,轴,整数,浮点数) |
       旋转(形状,轴,浮点数) |
       移动(形状,浮点数, 浮点数,浮点数) |
       长方体(浮点数, 浮点数,浮点数);
轴 → AX | AY |AZ;
整数 → [1, 6];
浮点数 → 原语ij | -1 | 0 | 1 | 2 |
       加法(浮点数，浮点数) |减法(浮点数，浮点数)
       乘法(浮点数，浮点数) |除法(浮点数，浮点数);
```

我们将语法中的所有非终结部分用斜体表示，并解释语言中终结算子（非斜体）的作用。Union 将两个子形状组合在一起。SymRef 表示沿某个轴进行对称反射。SymTrans 表示沿某个轴进行对称平移，创建指定数量的副本，且不超过指定距离。Rotate 指定围绕某个轴进行欧拉角旋转。Move 将长方体移动指定的距离。Cuboid 根据指定的尺寸实例化一个长方体。轴可以是 X、Y 或 Z轴。整数可以是 1 到 6 之间的整数。浮点数可以来源于输入场景的原始参数（Prim^{ij}），可以是常数，也可以是参数化运算的结果。

二维形状文法。 下面我们详细介绍二维形状文法：

```
开始 →形状
形状 → 并集(形状，形状) |
       SymRef(形状，轴) |
       SymTrans(形状，轴，整数，浮点数) |
       移动(形状，浮点数，浮点数) |
       矩形(浮点数，浮点数);
轴 → 轴 | AY;
整数 → [1, 4];
浮点数 → Primij | -1 | 0 | 1 | 2 |
       加法(浮点数, 浮点数) | 减法(浮点数,浮点数)
       乘法(浮点数, 浮点数) | 除法(浮点数,浮点数);
```

这是我们三维语法的简化版本，其中移除了旋转命令，并将所有三维参数化替换为二维参数化。

B 实现细节

下面介绍 ShapeCoder 的实现细节。对于第7节中的所有实验，我们设置 $N_A = 20$和 $N_D = 10000$。

B.1 目标函数权重

我们在 ShapeCoder 的目标函数（第 3.1 节）中使用以下权重：浮点数词元为 2.0，返回形状的函数词元为 1.0，返回浮点数的函数词元为 0.1（即参数化操作），类别词元（包括整数）为 0.5。此外，我们将几何误差权重 λ_e设为 10。

对于第 3.1 节中所述、并在第 7.4 节中进行消融实验的函数加权方案 ω，ShapeCoder采用以下逻辑。将一个新的抽象 f 加入 L 的基础代价为 0.25，但可根据 f的属性，在 0.125 到 0.5 的范围内调整。 f中存在参数表达式会降低 ω。 f中的输入参数过多会增加 ω：参数超过 6 个时开始产生惩罚，而输入参数超过 10 个的抽象会被直接拒绝。对于双子抽象（即使用多个子函数的抽象），我们会降低 ω；对于仅使用一个子函数的单子抽象，则会提高 ω。最后，如果发现 f在 P 中的使用频率极低，观测率低于 1%，我们也会直接拒绝 f。

B.2 几何误差函数

目标函数（第 3.1 节）使用几何误差函数 err，用于比较 L 中已执行表达式 e与目标形状 d的接近程度。由于此误差函数在唤醒阶段（第4.3节）中被广泛使用，因此它会检查部分解。假设执行 e 会生成一组基本图元 $prim_e$，而 d包含基本图元 $prim_d$。首先，我们的几何误差函数会为 $prim_e$中的基本图元找到到 $prim_d$中某个基本图元的最优映射。在具体实现上，我们构建一个大小为 $|prim_e| \times |prim_d|$的距离矩阵，用于计算每对输入基本图元与目标基本图元之间的领域特定距离度量（稍后说明）。对于距离超过用户定义的最大误差阈值的任意一对基本图元，我们将其配对距离设为一个任意大的值（10000）。我们使用匈牙利匹配算法，在此距离矩阵上找到最优匹配。如果 $prim_e$与 $prim_d$之间没有任何一对匹配项的距离超过 10000，那么该匹配

有效，并且 e for d 所产生的总误差仅为该最优匹配所涉及的距离矩阵中所有元素之和。

在集成阶段（第5.2节），我们可以修改这种方法，通过强制要求距离矩阵必须是方阵，来检查是否存在一个能够解释 d的程序 p。如果不满足这一条件，就意味着 p创建的基本图元数量与目标形状 d中预期的基本图元数量不匹配，因此 p 无效。

二维几何距离。 每个基本图元（矩形）由 4 个参数表示：宽度、高度、x 位置和 y 位置。要计算两个基本图元之间的距离，我们取各参数槽绝对差值的平均值。最大允许误差阈值设为 0.05。

三维几何距离。 每个基本图元（长方体）由 9 个参数表示：尺寸、位置和欧拉角旋转。要计算两个基本图元之间的距离，我们先计算每个长方体的角点位置，再记录两组点之间的豪斯多夫距离。最大允许误差阈值设为 0.1。

B.3 识别网络

我们的识别网络采用带因果掩码的 Transformer 解码器骨干架构。我们允许其以最多 16 个基本图元为条件（每个基本图元将贡献 K 词元），并将最大预测长度固定为 32。网络包含 2 个注意力块，每个块有 8 个注意力头，隐藏维度为 128。训练时批量大小为 64，丢弃率为 0.5，学习率为 .0001。每个梦境阶段（第4.2节）最多训练识别网络 300 个训练轮次，并在由留出梦境组成的验证集（占样本的 10%）上执行提前停止。

B.4 梦境创建

采样库函数。 在梦境阶段（第4.2节），ShapeCoder 会随机采样库函数的实例化，以训练识别网络。图6展示了一些梦境的可视化结果。对于参数化函数 f 所需的每个离散决策，我们会找出 L 中所有类型匹配的词元，并从该分布中均匀采样。浮点类型词元以高斯混合分布表示（最多包含 3 个混合成分）。这些分布旨在大致反映基础领域特定语言中某些参数槽位的合理取值。例如，‘Move’ 运算符的第一个浮点参数槽与 x轴定位相关，因此我们设计了一个三峰混合分布，其特性如下：一个以 0 为中心的主要成分，以及两个分别位于原点左侧和右侧的次要成分。这些分布不会显著改变识别模型的性能，因为模型能够在大量样本上进行训练；但它们确实能加快我们依据拒绝标准（见下文）找到有效梦境的速度。为抽象函数采样梦境时，抽象函数的参数输入会继承其子函数的分布。

梦境拒绝标准。 我们使用简单的检查来验证随机采样的梦境是否能生成有意义的训练数据，并拒绝任何不符合以下标准的梦境。所有基本图元都必须具有正尺寸。所有基本图元的角都必须必须位于分配的场景包围体 $[-1, 1]^n$ 内，允许 10% 的容差。每个基本图元至少有 50% 的区域必须可见（即未包含在另一个基本图元中）。每个基本图元都必须大于指定阈值：二维面积为 0.005，三维体积为 .00025。梦境不能包含超过 16 个基本图元。梦境不能使用冗余操作，例如连续应用两个移动命令。

构建复合场景。ShapeCoder 的识别网络在复合场景上进行训练，这些场景由采样特定函数的梦境并将它们组合而成。要构建一个复合场景，我们从 [1,4], 中随机采样一个整数 k，从所有尚未在 N_D 个梦境中表示过的库函数集合中采样 k 个函数，并从每个选定的函数中随机选择一个梦境。此外，我们有 50% 的概率向复合场景中添加干扰图元。干扰图元来自随机子采样的某个 $d \in$ D 中的基本图元。为使识别网络具备位置不变性，我们可以选择采样一个移动操作（频率为 50%），并将其应用于特定函数梦境生成的基本图元。请注意，此移动操作不包含在目标表达式中，因此识别网络必须对目标基本图元在复合场景中的出现位置保持不变。

B.5 组合唤醒程序

如第3.4节所述，在第 r 轮的唤醒阶段中发现的程序需要与之前各轮 r中发现的程序组合。这里我们详细说明 组合的实现方式。

假设我们处于第 r 轮 $r > 0$ 的唤醒阶段。对于某个 $d \in$D，P 中当前有一个程序条目 p_c。借助 *split* 函数——它会递归地从程序中移除组合子操作——我们可以将 p_c 转换为 L 中的一组表达式：

$split(p_c) = E_c = \{e_c^0、\ldots\ldots、e_c^{|E_c|}\}$。执行时，每个 e_c^i 都会生成一组基本图元 $prim_c^i$，它是 d 中基本图元的子集。ShapeCoder 会使用一个数据结构 Q_d，跟踪与 d 关联的所有此类先前表达式；这些表达式来源于唤醒阶段或集成阶段。

唤醒推理过程使用识别网络，在第 r 轮为 d预测一个新程序 p_r。我们通过构造 4 个程序变体，并保留使 F 最小的变体，来决定应保留哪个程序 p 于 P 中。我们考虑的变体如下。(i) 使用 p_c。(ii) 使用 p_r（注意，如果 $r = 0$，则始终会选择此变体）。(iii) 将 p_r 贪心地合并到 p_c 中。为此，我们首先计算 $split(p_r) = E_r = \{e_r^0, \ldots, e_r^{|E_c|}\}$。然后，对于每个 e_r^i，我们寻找 $prim_r^i$，并检查在 E_c、M中是否存在一组匹配实例，使得 $prim_r^i = \{prim_c^j$ 对于 $j \in M\}$。如果存在 M，则比较 e_r^i在 F 下的代价与各个 e_c^j之和（包含 $|M| - 1$次组合子调用）的代价：如果 e_r^i能改善 F，则从 E_c中移除每个 e_c^j，并将 e_r^i添加到 E_c中。(iv) 从 Q_d贪心地构造一个全新的程序。首先，将 E_r添加到 Q_d中。然后，Q_d通过初始化 E_n（为空）并重复以下步骤来贪心地创建一个新程序：计算 Q_d中每个 e的代价，取代价最小的表达式 e^*并将其添加到 E_n中，然后暂时移除 Q_d中与 $prim_e^*$存在非零重叠的所有其他条目。重复此过程，直到 E_n包含能够覆盖 d中所有基本图元的表达式。

创建这四个程序变体后（其中在*(iii)* 和 *(iv)* 中，组合子操作分别作用于 E_c 和 E_n），保留 F 下得分最低的变体，并将其放入 P 中。最后，我们注意到，需要一些额外的逻辑来确保 Q_d 和 p_c 保持最新。每当集成阶段尝试从 L 中移除一个函数 f 时，Q_d 中使用 f 的所有表达式都会被暂时移除。此外，如果 f 出现在 p_c 中，则使用 *(iv)* 中的贪心搜索为 p_c 寻找替代表达式。

B.6 参数关系的偏好排序

提议阶段（第 5.1 节）使用贪心搜索生成候选抽象。这些候选抽象包含参数表达式。下面详细介绍我们如何根据采样簇进行偏好排序，以搜索匹配的参数表达式。

要提议哪一个参数表达式，始终是在某个簇的上下文中做出的选择；该簇包含一种结构和一组参数化。在为候选抽象填充参数槽时，我们可能已经实例化了先前参数槽中使用的自由变量。为了找到当前参数槽的可能表达式，我们会根据先前实例化的自由变量进行推理。我们按照偏好排序进行迭代，依次考虑由先前变量构成、复杂度逐渐增加的参数表达式：仅含常量的表达式、含一个变量的表达式、含两个变量的表达式，最后是含三个变量的表达式。L 下所有包含 n 个变量的表达式集合，可以通过计算以下两者的笛卡尔积得到：(i) 所有需要 n 个变量的参数化算子组合；(ii) 所有由 n 个先前实例化变量构成的有序序列。为避免过拟合，我们限制所考虑的常量（对于我们的形状文法，仅使用 0）。对于每个表达式，我们检查簇中的哪些成员能够被该表达式覆盖。一旦找到一组能够共同覆盖簇中所有实例的表达式，就提前退出循环。该过程会生成大量可能的表达式（如图 4 所示），然后根据得分 函数从中选出一个。

B.7 等式图

我们的重构操作（第 6 节）使用 Egg 库 [Willsey 等人 2021] 实现了 e-图。Egg 支持定义领域特定语言、重写操作和代价函数，提取操作可以使用代价函数。Egg 提供了一个接口，用于定义能够推理条件逻辑的重写，但这些重写无法直接应用于我们的用例。我们的条件重写要求每个重写都能访问共享的 e-类到实值映射，因此我们扩展了这一功能。维护此映射需要虚拟重写操作：这些操作会检查各种参数化操作的结构匹配项，并更新映射，而不改变 e-图的结构。首次实例化 e-图时，我们会应用虚拟重写，匹配每个浮点变量 V_i，并在映射中为 V_i 添加一项。随后，在每一轮重写中，应用完所有语义重写和抽象重写后，我们会应用所有虚拟重写，以确保映射保持最新（这会处理图5中的蓝色 Mul e-类）。对于每个领域，我们都会为 Egg 提供一组语义重写，用以表达特定领域且保留语义的变换。三维领域有 25 个此类重写，二维领域有 16 个。我们在消融实验（第7.4节）中通过消融评估了纳入这些语义重写的重要性。

B.8 无监督原语分解

如第 7.5 节所述，我们采用一种无监督长方体分解方法，以便将 ShapeCoder 应用于仅包含网格的数据集中的形状。我们使用 [Yang and Chen 2021]，描述的方法及其发布的预训练模型，预测测试集中椅子的长方体分解结果。我们汇集了一个包含 400 个此类预测结果的数据集，并将这些预测结果解析为与我们的方法兼容的原语表示。此转换过程包含几个小型过滤步骤：剔除包含超过 12 个长方体的场景（我们发现这类预测通常带有噪声），并在长方体的欧拉角与 0 或 2π 的差值小于 0.05 时，将其调整为与轴对齐。

B.9 程序生成模型

我们在此详细介绍第 7.6 节所述的生成模型。该模型能够合成新的三维形状。我们将生成模型实现为带有因果掩码的 Transformer 解码器。它使用卷积神经网络将形状的体素化结果编码为嵌入向量，并以此作为 Transformer 的条件，使其自回归地预测 L 中的词元。网络从一个空白场景开始，迭代预测一个来自 L 的表达式 e，并将其添加回场景中（该场景将在下一时间步由卷积神经网络编码）。重复此过程，直至预测出特殊的“STOP”词元。

我们通过对一个包含 3600 把椅子的数据集运行事后推断程序（第 7.3 节）来获取该模型的训练数据，从而构建程序数据集 P。对于每个训练轮次，我们通过对每个 $p \in$ P 应用 拆分 （B.5），将表达式的排序随机化：打乱由 拆分得到的表达式，并将每个（先前的表达式、下一表达式）元组视为独立的训练样例。我们使用教师强制和最大似然更新来训练生成模型。我们训练该模型 4000 个训练轮次。模型包含 8 个 Transformer 层、16 个注意力头，隐藏层维度为 256。训练时，我们使用批量大小为 64、丢弃率为 0.1、学习率为 0.0005。在推理时，我们使用核采样（top90%）根据网络概率预测表达式。作为对比的“无抽象”版本采用完全相同的设置，唯一的区别是，事后推断程序使用的是初始 L 版本运行的（而不是 ShapeCoder 发现的版本）。