Abstract of “Neurosymbolic Methods for Shape Analysis and Generation”
by R. Kenny Jones, Ph.D., Brown University, May 2025.

Shape analysis and generation methods are critical to many visual computing applications. Stakeholders often want to populate physical and artificial spaces with high-quality, structured assets that support interaction and manipulation. Different shape representations support these desiderata to varying degrees. Programmatic representations (e.g. procedural models) are a popular choice with many benefits, but also come with inherent limitations: they are expensive to author, have limited output variety, and typically require a thoughtfully designed domain-specific language (DSL).

This dissertation explores a suite of neurosymbolic systems that combine learning with programmatic representations to aid in shape analysis and generation. When datasets of procedural assets are available, we can train generative models that synthesize novel shapes by learning to write programs. When we lack a dataset of procedural assets, we can train networks to search for programs that explain visual inputs with a bootstrapped, self-supervised learning paradigm. We show performance can be improved by reframing this program synthesis task as a program editing task, and also that this paradigm can be extended to infer stochastic programs capable of capturing a distribution of visual inputs. Finally, we investigate ways to discover better DSLs with little or no expert intervention. We propose two bottom-up library learning works that augment a starting DSL with automatically proposed functions that improve a data-driven compression objective, starting from shape datasets of either imperative programs or unstructured primitives. We also explore an alternative top-down framing, where we task a Large Language Model with authoring a library of shape abstraction functions from two forms of user design intent: text descriptions of functions to include in the library and a seed set of exemplar shapes. Together, these works demonstrate that the limitations of the procedural representation can be successfully mitigated through the application of hybrid neurosymbolic methods that learn to synthesize, infer, and abstract visual programs.

Neurosymbolic Methods for Shape Analysis and Generation

by

R. Kenny Jones

B. A. Williams College, 2017

Sc. M., Brown University, 2021

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island

May 2025

This dissertation by R. Kenny Jones is accepted in its present form
by the Department of Computer Science as satisfying the
dissertation requirement for the degree of Doctor of Philosophy.

Date _____ _____
Daniel Ritchie, Advisor

Recommended to the Graduate Council

Date _____ _____
James Tompkin, Reader

Date _____ _____
George Konidaris, Reader

Approved by the Graduate Council

Date _____ _____

Dean of the Graduate School

Vita

Kenny Jones grew up in sunny Palo Alto, California. He matriculated to the wonderful Williams College, nestled in the Berkshires of western Massachusetts, graduating in 2017 with bachelor's degrees in computer science and English literature. He spent one semester of his undergraduate studies abroad in the hauntingly beautiful Edinburgh, Scotland. After graduation, he returned to the bustling Bay Area, working for two years at Facebook as a software engineer on the Charitable Giving and Facebook AI teams. In 2019, he traveled back to the East Coast to begin a Ph.D in computer science at the buoyant Brown University in Rhode Island, receiving a M.Sc. in 2021. Kenny's research explores neurosymbolic methods to better understand and represent visual data. This work sits at the intersection of computer graphics, vision, and machine learning, and has been presented at leading conferences including CVPR, ICML, NeurIPS, and ACM SIGGRAPH. During his Ph.D, he was supported by a Brown University Presidential Fellowship and an internship at Adobe Research.

Acknowledgements

As I near the destination marked by this dissertation, I am filled with a deep sense of gratitude for the remarkable individuals who have shown me wisdom and kindness throughout this journey. Research, often, is inane when treated as a process unto itself. Its trials and tribulations, so narrowly focused, summon visions of Sisyphus and his daunting task. Thankfully, we find salvation in community, in its sense of purpose and camaraderie, which provides perspective and meaning to our shared endeavor – for after all, with clear purpose, one must imagine Sisyphus happy. Stepping onto this road of research, though some would say a dangerous business, has been a truly rewarding experience, made all the more meaningful by those who've walked alongside me, helping me keep my feet with their gracious guidance. So, if we are to put journey before destination, let me take this time to acknowledge just some of the many, many wonderful people to whom I am deeply indebted.

First, I would like to sincerely thank my advisor Daniel Ritchie. Daniel is a fantastic scholar who has shaped my aptitude, attitude, and appetite for research. His infectious optimism provided invaluable motivation, while his concrete guidance helped transform my half-formed ideas into fully-fledged plans of action. More importantly, Daniel's abundant patience, kindness, and thoughtfulness fostered a welcoming environment, one that I've been proud to call my academic home. The chief piece of advice I give to prospective Ph.D. students is to weigh the fit with a potential advisor above all else; in finding an advisor like Daniel, I count myself extremely fortunate. He has been instrumental in helping me to become the researcher I am today, and for that I will be forever grateful.

The work presented in this dissertation was supported through the generous guidance of numerous research mentors. At the start of my Ph.D, I was very lucky to begin work on a 'part-graph-programs' project with Paul Guerrero and Niloy Mitra. After almost six years of weekly project meetings, this direction has produced four papers (the Shape-X series), which form a substantial portion of this dissertation's contribution. Like these projects, I benefited tremendously from Paul and Niloy's expertise and perspective. Rana

Hanocka was a tremendous help in getting me started with 3D learning methods while working on the deceptively named 'mesh-gen' project. I am also grateful for Siddhartha Chaudhuri's insightful questions and advice about research and broader topics during my internship.

I've been fortunate to receive support from many members of the Brown community: I am especially indebted to my committee members, James Tompkin and George Konidaris, whose thoughtful approaches to research have strengthened the work in this dissertation. My very first course at Brown was James' research seminar, which proved foundational in shaping my view of what makes for good research. Beyond that, I've always appreciated James' ability to cut to the heart of hard problems with simple, insightful questions. George's research vision is inspiring, and though much of this dissertation lies outside his primary interests, lacking both robots and reinforcement learning, his advice on both high-level direction and low-level details has been immensely helpful. I would also like to thank Srinath Sridhar, Chen Sun, David Laidlaw and Stephen Bach, for thoughtful discussions, constructive feedback, and kind words of encouragement on various research projects and presentations.

Despite what Covid restriction guidelines at one point suggested, this journey was not undertaken in isolation, but rather in the wonderful company of peers and colleagues. To fellow group mates: Kai Wang, Theresa Barton, Xianghao Xu, Aditya Ganeshan, Arman Maesumi, Yuanbo Li, and Maxim Gumin; office mates: Qian Zhang, Rao Fu, Sudarshan Harithas; and lab mates, including Mikhail Okunev and Brandon Woodard, thank you for making the visual computing community at Brown so welcoming and supportive. I'll miss the collaborations, discussions, snacks, and board games, not to mention the timely help with resetting my computer. Throughout my research projects, I also had the privilege of collaborating with many brilliant and inspiring undergraduate and Master's students: Brian Oppenheim, Dylan Tian, Ellen Jiang, Homer Walke, Caleb Trotz, Bryce Blinn, David Charatan, Alex Ding, Aalia Habib, Alana White, Sarah Roberts, Norman Zhang, Anh Truong, Paul Biberstein, Junyu Liu, Rio Aguina-Kang, Stewart Morris, Brian Han, Jean Yoo, and Vivian Lu. I would also like to thank all of the amazing members of the A-staff and T-staff who were a tremendous help with resolving technical, logistical, and bureaucratic issues, including Lauren Clarke, John Tracey-Ursprung, Dawn Reed, Lori Agresti, Genie DeGouveia, Danielle Beatrice, John Bazik, Paul Vars, and Kathy Billings.

I first discovered my love for computer science and research at Williams College. A special thanks to Derrick Bonafilia who first got me interested in coding and machine learning at a time when I wanted to avoid anything with even a hint of engineering. Challenging and deeply rewarding courses and research with Morgan McGuire helped to foster a love of graphics and procedural representations. Andrea Danyluk (in memoriam) was a gracious mentor, who inspired me to investigate generative modeling techniques and

taught me how to approach independent research with thought and care. That the work of my dissertation would so naturally fall at the intersection of these two branches, first planted during my time at Williams, never crossed my mind at the time, yet in hindsight seems to have always been inevitable. Truly, in some cases, “the wheel weaves as the wheel wills”.

This dissertation would not have been possible without the love and support of my family and friends. Despite my many failings in elucidating even remotely comprehensible answers to simple questions like, “so what are you working on?”, they have unconditionally celebrated, cheered, consoled, and commiserated. Even while they imagine me Wyndle, cultivating gardens of chairs from thin air, with no discernible purpose, they have always believed in me. To my parents, Liz and Russ, my sisters, Maddy and Ella, my grandparents Charles and Gloria (in memoriam), thank you for everything, I love you all.

Finally, I would like to dedicate this dissertation to Juliana Veira: my wife, and better half. There is not a day that goes by where I am not blown away by how lucky I am to have found you – your wisdom, grace, laughter, determination, empathy, and love provide continual joy, and there is no one else I would rather have by my side. I’m forever grateful that we’ve had this chance to grow together and shape our lives in tandem. Since we began our grad school journeys six years ago, we’ve gotten married, traveled the globe, and adopted the best cats in the entire world: Alby (Albus Beano Dumbledore) and Dany (Daenerys Newton Targaryen). Though you beat me to the ‘doctoral’ destination, I am so excited for our larger journey together, and I can’t wait to see what comes next.

Contents

List of Figures

Chapter 1

Introduction

Visual reasoning plays a critical role in how people interact with and understand the physical world. The field of visual computing is concerned with how to best endow computing systems with the requisite skills to effectively analyze, synthesize, manipulate, and interact with visual data. A myriad of stakeholders have created a growing demand in this space. Applications within computer graphics require high-quality 3D assets for visual effects, animation, entertainment, and games. Augmented and virtual reality experiences desire 3D shapes and scenes that support intuitive manipulation and interaction. Artificial intelligence systems rely on synthetic visual data to train large data-driven models [175, 241, 103] and robotic agents in simulation [110, 224, 182, 1].

The question of representation is central to visual computing: *how* visual data is represented affects *what* downstream applications are supported. Unlike regular pixel grids for capturing 2D visual data, there is no *standard* representation for 3D geometry. 3D shapes can be expressed in a variety of representations, each with their own strengths and weaknesses. Voxels are the direct extension of pixels to 3D space, but they suffer from the curse of dimensionality. Point clouds are easy to obtain from the real world via depth sensors, but they lose surface connectivity and detail. Surface meshes are well-supported by rendering and simulation packages, but their irregular topologies cause difficulties for learning and optimization. Recently, there has been growing interest centered around approaches that represent 3D shapes 'neurally', including occupancy/SDF networks [26, 157], Neural Radiance Fields [138], and 3D Gaussian splatting [105]. While these exciting developments are capable of producing high-quality outputs, they often hallucinate implausible geometry, do not expose interpretable interfaces, and are hard to interact with or manipulate.

In this proposal, we study programmatic representations of visual data. Such representations have traditionally been known as *procedural models*: structured computer programs that produce visual data when executed. Procedural models have many compelling properties for downstream applications. Well-written procedural models produce high-quality geometry from compact representations. Finding symbolic representations that effectively capture visual phenomenon allows for shape and scene analysis: sometimes referred to as 'analysis by synthesis' [237] or 'inverse graphics' [8]. Procedural shape representations are interpretable to users with some programming background, and they often expose high-level parameters that can be manipulated to change attributes of the visual data they generate. Further, randomization of these parameters allows a single procedural model to generate a wide variety of different visual outputs—this is useful for rapidly exploring their design space or for populating large virtual worlds with non-repetitive content.

Despite these numerous benefits, there are a few main limitations that beleaguer procedural models. Foremost among these is availability: high-quality procedural models are hard to acquire. Typically, these programs must be authored by domain experts, which is an expensive and time consuming process that does not scale. Beyond this, while some procedural models are capable of producing a family of visual outputs, the range of output variety is usually fairly limited. It would be almost impossible, for instance, to design a single procedural model capable of producing all types of cars. Finally, the usefulness of any given procedural model is dependent on the language in which it is written. Well-designed procedural models typically leverage domain-specific languages (DSLs) that expose task-specific functionality to produce interesting structures while maintaining a compact programmatic form. When using a poorly constructed DSL, a procedural model might not be able to realize the potential benefits of this powerful representation.

This dissertation focuses on how these limitations can be mitigated through these use of neurosymbolic methods that integrate learning and programmatic representations. For instance, given a collection of programmatic shape representations, we can leverage deep generative models that learn to synthesize novel programs that can be executed to produce new geometry. When we lack human-authored procedural models, we can employ self-supervised bootstrapped learning algorithms that infer visual programs that recreate input data when executed. Given a base DSL, we can use optimization algorithms to discover new abstraction functions that improve the DSL for a particular modeling task. In summary, we find that neurosymbolic methods, that marry learning-based systems with symbolic representations, draw from the strengths of each of these disparate techniques, and often achieve a 'best of both worlds' solution.

1.1 Contributions

This dissertation introduces a series of neurosymbolic methods that aid in shape analysis and generation tasks. This content is based largely on seven previous publications [91, 98, 99, 93, 92, 94, 95], which are naturally grouped into three thematic directions.

1. **Generating shapes by learning to synthesize programs:** When datasets of annotated assets are available, we can train generative networks that learn to author novel shape programs. We explore the benefits of such a hybrid neural-procedural paradigm in our ShapeAssembly system [91]. We introduce the ShapeAssembly language and its differentiable interpreter, allowing the procedural specification of shape structures represented as connected part assemblies. We design a deep generative autoregressive model for ShapeAssembly programs, coupling the ease-of-training and variability of neural networks with the precision and editability of procedural representations. We demonstrate that training networks that learn how to author shape programs improves performance over other structured modeling alternatives.

2. **Unsupervised visual program induction:** For many domains of interest, we lack annotated program datasets, but we would still like to find programs that explain visual datum: this is the task of visual program induction (VPI). We propose PLAD, a method that trains a recognition network for VPI without access to labeled data [98]. PLAD introduces a conceptual framework that groups and generalizes a family of related self-supervised learning techniques. We run experiments across multiple VPI domains, and find that PLAD training outperforms previous state-of-the-art alternatives such as policy gradient reinforcement learning. In a follow-up work, we explore an extension of PLAD, VPI-Edit, that introduces networks that learn how to edit visual programs [99]. Given an initial program and a visual target, these networks predict local edit operations that can be applied to the input program to improve its similarity to a target. We show that this paradigm is more effective at solving VPI tasks compared with networks that try to author an entire program in 'one-shot'. In another extension of PLAD, we introduce Template Programs [93], partial programs that can explain groups of related visual inputs. We propose a neurosymbolic method that learns how to infer these stochastic procedural models in an unsupervised fashion. In experiments across VPI domains, we demonstrate that this framework supports multiple concept-related tasks, including cosegmentation, few-shot generation, and novel concept synthesis.

3. **Discovering better domain-specific languages:** Obtaining a 'good' procedural model requires access to a 'good' modeling language, where notions of 'good' are often task-specific. We explore how base domain-specific languages can be improved automatically with methods that search for abstraction functions that improve a data-driven compression-based objective. These library learning techniques are provided with a shape dataset, and try to discover a concise set of functions that abstract out common structural and parametric patterns: removing extraneous degrees of freedom from the underlying shape collection. ShapeMOD [92] assumes that the input shape dataset has associated imperative programs. ShapeCoder [94] relaxes this assumption, operating over collections of shapes represented with unstructured primitives. We experimentally demonstrate on collections of manufactured objects that learning over programs that use these discovered abstractions leads to better performance on important downstream tasks such as novel shape generation, directed shape manipulation, and inferring shape structures from unstructured geometry. As an alternative to these bottom-up approaches, we propose the ShapeLib [95] method, which leverages the priors of Large Language Models to design a library of shape abstraction functions in a top-down fashion. This system accepts two forms of user-provided design intent: text descriptions of functions to include in the library and a small seed set of exemplar shapes. Across multiple categories of manufactured 3D shapes, it is able to discover procedural abstractions that match this design intent, expose semantically aligned parametric handles, and generalize beyond the seed set.

Together, these contributions demonstrate how neurosymbolic methods can be used to mitigate the traditional limitations of procedural modeling, while maintaining a programmatic representation of visual data. When available, we include links to our open sourced code in the respective chapters for each method.

1.2 Document Overview

The rest of this dissertation is organized as follows: Chapter 2 first provides high-level background overviews of the fields of procedural modeling and program synthesis. It then discusses relevant related works on the topics of generative models of visual data, visual program synthesis, and abstraction discovery. Next we discuss the technical contributions of this proposal. In Chapter 3, we present ShapeAssembly, a generative model of shape programs. We then describe a series of works for unsupervised visual program induction, starting with the PLAD framework (Chapter 4), followed by extensions that train networks that learn how to edit programs (Chapter 5) and infer programs that capture a collection of visual inputs (Chapter 6). Next, we

introduce a series of works for automatic abstraction discovery: starting from a dataset of shape programs (ShapeMOD, Chapter 7), starting from a dateset of shapes represented with primitives (ShapeCoder, Chapter 8), and guiding a LLM to author functions that match an input design intent (ShapeLIB, Chapter 9). We conclude the dissertation in Chapter 10, with a summary of our main contributions and a discussion on the future of neurosymbolic methods for shape analysis and generation.

Chapter 2

Background

In this chapter, we overview the most relevant background material and related works to this dissertation. We first briefly survey the topics of procedural modeling, programs that generate visual outputs, and program synthesis, the problem of finding a program that meets a specification. We then discuss the most relevant related methods to the work of this dissertation in the topics of: generative modeling for visual data, visual program synthesis, and abstraction discovery.

2.1 Procedural Modeling

In procedural modeling, a symbolic program describes graphics content, such that when the program is executed it produces visual data as its output. Procedural models have a storied history that dates back to the origins of computer graphics as a field, when Ivan Sutherland used constraint programs to produce engineering sketches in the SketchPad system [39]. While procedural approaches have permeated almost all facets of the graphics pipeline, procedural models are most widely used to represent geometry and appearance of 3D objects. For instance, trees and vegetation have commonly been modeled in a procedural fashion, where context-free grammars, such as rewrite-based L-systems, are used to represent the fractal nature of these objects [161, 86, 160, 117]. Building facades and entire cityscapes are also a common interest of procedural modeling approaches [142, 154]. Beyond 3D geometry, procedural models are also common in texture modeling [3, 27, 6], and have even seen use in more varied application such as influencing the behaviors of virtual characters in a crowd [134].

Most relevant to the contents of this dissertation, is the use of procedural models to represent shapes

[49]. Shape programs, or procedural models that create shapes when executed, are often written in a domain-specific language, or DSL. The design process of complex procedural shape models often takes place within proprietary software development environments, where graphical node-based programming is commonly employed [6, 9, 227]. One representative approach is Constructive Solid Geometry (CSG), which builds complex shapes by combining primitives together with Boolean set operations (union, difference, intersection) [56]. Computer Aided Design (CAD) software typically constructs such 3D primitives by lifting 2D sketch profiles into 3D volumes through extrusion [30, 5].

Most procedural models are manually designed by domain-experts, an expensive and time-consuming process. While some procedural models are deterministic (they always produce the same output when executed), others are stochastic (they can produce different outputs when executed). Typically, these models use a single consistent structure that includes calls to functions that return random variables, optionally exposing these variable parameters as controllable 'handles'. For instance, a procedural program of a flower might expose an interface that allows a user to vary the length of the stem or the number of petals. This paradigm facilitates exploration over a wide-range of possible shape realization, allowing users to choose which of the outputs best fits their intended purpose. Unfortunately, designing a stochastic model is arguably harder than designing a deterministic procedural model.

In the following content of this dissertation, we discuss and explore works that attempt to automatically synthesize procedural models of shapes, with little or no human intervention.

2.2 Program Synthesis

Program synthesis is a broad field that has employed many techniques throughout its history. The typical problem framing presents a program synthesizer with two inputs: a programming language (often domain-specific) and a specification. The goal of the synthesizer is to return a program from the language that meets the specification. Ideas of automatic code generation have intrigued researchers since the inception of Artificial Intelligence; this problem has even been dubbed the "holy grail" of AI [62]. Despite this ambitious framing, progress in program synthesis had been slow until the past few decades, when advances in computing power and constraint-solving, along with novel enumeration techniques, started to produce useful systems capable of solving synthesis tasks beyond toy-domains [191, 155, 207, 61].

Program synthesis systems can be primarily divided along three central axes of design decisions: intent specification (e.g. how a user communicates the goals of the desired program), search space (e.g. the

programming language over which the synthesizer will search), and search technique (e.g. how the synthesizer will search for a program that meets the specification). While program synthesis methods have explored a wide-range of search algorithms (including enumeration, constraint-based, probabilistic search, etc.), more recently the field has focused on learning-based techniques. Typically methods within this paradigm task a neural network with guiding how the search should proceed over the space of possible programs [37, 220, 156, 197, 13, 22]. Since programs can be represented as sequences of discrete tokens, many learning-based methods use auto-regressive models, where each token is generated conditionally based on previously generated tokens [209]. This paradigm has become especially popular following the rapid adoption and success of Large Language Models (LLMs). Frontier LLMs are often trained on massive datasets of human-written code as part of their pretraining, and have demonstrated the ability to program in a way that generalizes across tasks and programming languages [19].

Some systems model this autoregressive process not only at the token level, but instead across program versions. A number of program synthesis methods have been proposed that learn how to repair or fix programs for domains where ground-truth programs are available. SED interleaves a series of synthesis, execution and debugging steps in order to improve synthesis of Karel programs from input/output examples [66]. Related approaches have explored learning how to 'fix' programs end-to-end by manipulating latent-space encodings of programs under a fixed decoder for the RobustFill domain [7]. A number of recent works have explored how LLMs can be used to fix programs when prompted with mismatching input/output examples [189, 21, 130, 148]. Though differing in details, the typical formulation these methods take involves presenting an LLM with a previous program version, and asking it to either (i) debug exceptions or (ii) modify program behavior in light of input/output mismatches. While these initial forays show promise, performance gains of code-editing LLMs are not always definitive when adjusted for inference budgets [152].

This dissertation introduces a series of methods for visual program induction, a sub-problem of program synthesis where the input specification is visual data, and the goal is to find a program whose output execution recreates the input. We discuss relevant related works that study this problem in Section 2.4.

2.3 Related work on Generative Models of Visual Data

Learning generative models of visual data is a rapidly growing field that has received a great deal of interest. Fueled by deep learning, and access to massive datasets, the capabilities of these systems have increased dramatically in recent years. Deep generative models learn to represent the probability distribution over an

input domain X (e.g. a collection of visual data). This probability distribution can then be sampled to synthesize novel instances from X (e.g. new visual data). There are a multitude of deep generative modeling paradigms, all with different strengths and weaknesses, including generative adversarial networks (GANs), variational autoencoders (VAEs), auto-regressive models, normalizing flows, and diffusion models [10, 28]. These deep generative modeling paradigms have been applied across many visual domains including image synthesis, processing, and manipulation [87, 245, 55, 104, 169, 181, 178], scenes [213], material and texture modeling [50, 36, 72, 64], and 2D drawing and sketching [68, 174, 210]. In principle, these models are easy to create: just provide training data, and a learning algorithm takes care of the rest. What's more, they are quite general: the same model architecture (and sometimes even the same trained model) can represent a huge variety of different kinds of visual data (e.g. the space of all human faces). However, these 'neural' methods are not without limitations. The representations these models learn are usually opaque and uninterpretable, making them hard for users to edit or manipulate. Additionally, as machine learning methods produce statistical *approximations* of the true function implied by their training data, such models may generate outputs that exhibit artifacts or fail to generalize beyond their training distribution.

For deep generative models that learn to synthesize 3D shapes, representation is a central design decision. Some of the earliest approaches generated shapes as 3D occupancy grids [223, 221], while later work has explored how to generate point clouds [44, 121], meshes [59, 146], and implicit representations [26, 157, 136, 24, 84, 78]. Most of these aforementioned generative models of 3D shapes create geometry directly. In contrast, *structure-aware* models learn to generate objects as arrangements of their component parts [18, 139]. These include approaches for iteratively adding parts to partially-complete shapes [198], generating symmetry hierarchies [120], composing parts from two different shapes [244], and generating hierarchical connectivity graphs [140].

This dissertation discusses structured generative models for visual data, where a neural network learns to model a distribution over shape programs. The ShapeAssembly system, introduced in Chapter 3, offers one of the first realizations of such a neurosymbolic generative model for 3D shape structure synthesis. This paradigm has also demonstrated success across visual domains and tasks [176], including node graphs for materials [60], scalable vector graphics [170, 15], CAD sketches [153, 184, 54], and 3D modeling sequences [222, 229, 217].

2.4 Related work on Visual Program Synthesis

Visual program induction is a sub-problem of program synthesis. Typically the input is a single visual entity, such as an image or a 3D shape, and the goal is to find a program whose execution would recreate this input. This problem framing is particularly appropriate for manufactured objects, as such shapes typically *originate* as CAD programs of some form (e.g. connected assemblies of parts, the geometry of which may be specified by lower-level instructions). Some methods for this task use non-learning based approaches. These typically rely on heuristics and are specialized for particular domains and/or tasks. Recent work includes reverse-engineering CAD programs from 3D shapes [38, 144, 228] and shape program manipulation [71].

As the design space of visual programs is often prohibitively large for exhaustive search, and insightful heuristics are hard to identify, neurally-guided approaches have become increasingly preferred. Unfortunately, while shape data is increasingly available in large quantities [16, 32], these shapes do not usually come with paired program annotations, so supervised learning cannot be used directly. To mitigate this lack of data, some approaches require an expert designed program structure as input, and then search for a parametrization of this program structure to match a given target [137, 159]. Other approaches aim to jointly infer both program parameters and structure, and have demonstrated success across a range of visual programming domains, including CAD modeling sequences [118, 54, 184, 119], SVG shapes [170, 171], and even custom DSLs with learned neural primitives [35]. These works leverage task-specific learning or architecture modifications that make the search space tractable.

Methods that infer shape programs in a general way (capable of learning across VPI domains), typically employ a two step learning process to circumvent the lack of paired data. First, these methods will generate *synthetic* data by sampling random programs under the input DSL and pairing them with the shapes they output. This pretrained network can then be fine-tuned towards a target shape distribution that lacks annotated ground-truth programs. In Chapter 4, we introduce PLAD, a fine-tuning technique that works across shape program inference domains, and discuss alternative fine-tuning paradigms in Section 4.1. While this approach, like many other VPI methods, attempts to author a complete program from only visual conditioning (e.g. a latent encoding), some approaches employ an an execution-aware search procedure. For instance, some methods will reason over partial program executions [151] that guide a more complex outer search [41]. Others use executor-gradients to guide inner-loop optimization [53, 236]. In Chapter 5, we propose an extension of PLAD that trains networks that learn how to edit programs towards a visual target in an execution-aware manner.

Another line of research has studied inverse methods for producing procedural models capable of generating a *distribution* of visual outputs. A typical framing such approaches take is to induce a grammar with a bottom-up procedure, e.g. through Bayesian merging [85]. These techniques have demonstrated success across many visual domains, including plants [193, 63] and buildings [150, 132, 149, 33], and some can even induce more general probabilistic programs [177]. Though these methods achieve impressive results, they lack generality and flexibility, struggling to induce grammars outside of their specialized domain, and often requiring highly structured input data. In Chapter 6, we introduce a method that extends a single-shape VPI method to search for procedural models that can explain a collection of related visual inputs. Our approach generalizes across visual programming domains, doesn't require structurally-annotated data, and supports a range of downstream tasks.

As Large Language Models (LLMs) have exploded in popularity, reshaping the field of computer science, initial investigations have explored their ability for producing and reasoning over programmatic descriptions of visual content [12]. Up to now, frontier LLMs and Vision-Language models (VLMs) have struggled to perform these generation and analysis tasks for non-toy shape modeling domains, though task-specific finetuning can improve performance to a degree [113, 164]. Some approaches have found success in using LLMs to guide searches over symbolic re-parameterizations of procedural models for 3D shape editing and manipulation tasks [52, 109, 82]. A related research direction of nascent interest has explored how well LLMs can author programs that describe the arrangements of objects within a scene [231, 47, 240, 126, 81, 4]. Unfortunately, directly prompting LLMs to generate programs that produce compelling 3D shapes has so far proved a harder task.

2.5 Related work on Abstraction Discovery

While procedural models offer an attractive representation for visual data, not all visual programs are equally useful. The usefulness of a procedural model is dependent on the programming language in which it is written; for instance, in representing the wheels of an office chair, it would be harder to author a 'good' procedural model without access to a language operator that created a rotational symmetry group. The functions of the DSL should be designed for the modeling task at hand, where specialized high-level functions (e.g. macros, abstractions, helper functions) are required to produce the 'most useful' procedural models. As illustrative examples, consider that in urban procedural modeling, a macro might capture how primitives combine to make a particular class of railing; in furniture modeling, a macro might be used to model a shelving pattern

that could be instantiated within different types and sizes of furniture; in plant modeling, a macro might be used to instantiate examples of petal structures across a family of flowers. Typically such abstraction functions must be carefully crafted by humans, but some prior work has investigated if these 'better' languages can be found automatically; typically by augmenting the functions of a base DSL with a library of additional abstraction functions.

Several prior methods aim to discover abstractions in context-free languages, where only a reduced set of relations between primitives or sub-programs can be modeled, in the context of façade grammars [133] or more general grammar types [200, 85, 177]. Recently, another line of work has investigated common abstraction discovery for general functional programs, under the framing of library learning: the Exploration-Compression (EC) algorithm [31] and its successor, DreamCoder [42]. EC operates by switching between two algorithmic phases: "exploration" (trying to find programs that solve input problems) and "compression" (finding abstractions common to these programs). DreamCoder extends the ideas of EC, replacing exploration with a "wake" phase (with similar goals), and compression with two new phases: a "sleep-dream" phase that fantasizes new tasks and a "sleep-abstraction" that mirrors EC's "compression" phase. We discuss DreamCoder, and how it relates to abstraction discovery methods for 3D shapes proposed in this dissertation (ShapeMOD and ShapeCoder), in more detail in Chapter 8. Recently, alternatives have been proposed for the abstraction step of a DreamCoder-like system, either by using a top-down search, like the STITCH algorithm [11], or by employing anti-unification over an equality preserving data-structure, like in the Babble algorithm [14]. LILO [58], is another recent method that uses a LLM to automatically document the abstractions discovered by STITCH. In Chapter 9, we introduce an alternative framework that guides a LLM through the process of implementing procedural abstraction functions that match a provided design intent.

A related problem studies how to discover common patterns in a *single* program, as opposed to a set of programs. This has been explored for L-Systems [63] and there has also been prior work on this problem in the realm of shape modeling languages. The Szalinski system takes a low-level CAD program as input and searches for a more compact, higher-level program which produces the same output geometry [145]. The Carpentry Compiler is a program optimizer that finds rewrites of low-level instructions to maintain high-level semantics while optimizing to reduce manufacturing cost [219]. Such approaches can also be seen as related to systems for more general program-rewriting, such as optimizing compilers [203].

Chapter 3

Learning to Generate Programs for Shape Structure Synthesis

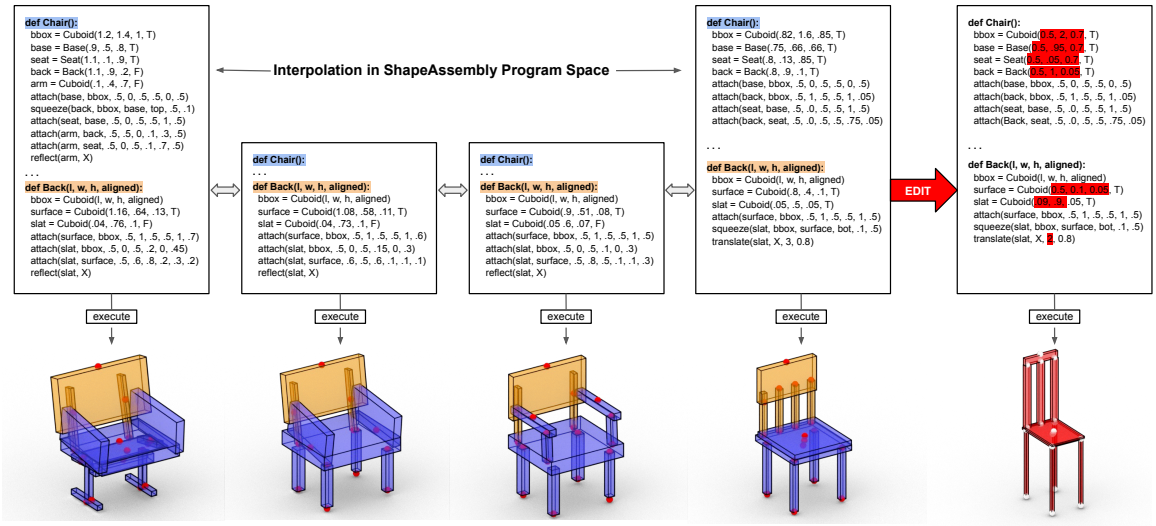

Figure 3.1: We present a deep generative model which learns to write novel programs in SHAPEASSEMBLY, a domain-specific language for modeling 3D shape structures. Executing a SHAPEASSEMBLY program produces a shape composed of a hierarchical connected assembly of part proxies cuboids. Our method develops a well-formed latent space that supports interpolations between programs. Above, we show one such interpolation, and also visualize the geometry these programs produce when executed. In the last column, we manually edit the continuous parameters of a generated program, in order to produce a variant geometric structure with new topology.

3D models of human-made objects are more in-demand than ever. Despite the growing demand, the craft of 3D modeling largely remains as difficult and time-consuming as it has ever been. The time and expertise

required to create 3D content by hand will not scale to these demands.

One promising way out of this conundrum is the development of *generative models* of 3D shapes, i.e. procedures which can be executed to generate novel shapes within some class [161, 142, 154]. An ideal generative model would produce plausible output geometry, capture a wide range of shape variations, and use an interpretable representation which a user could subsequently manipulate and edit. Unfortunately, no existing shape generative model achieves all of these properties.

In this chapter, we capitalize on our insight that procedural models and deep generative models have complementary strengths. Deep generative models are efficient to create and excel at broad-scale variability, and procedural models produce high-quality geometry by construction and better facilitate editing for fine-scale variability. We take a first step toward achieving the best of both worlds by integrating these two approaches into a single pipeline: a deep generative model that *learns to write programs*, which, when executed, themselves output 3D geometry. We hypothesize that going through this intermediate program representation produces a generative model with a smoother latent space, whose outputs are more likely to be physically valid, compact, and editable.

As the motivating applications mentioned earlier demand 3D models of human-made objects, we focus on generating novel part-based shape structures. We introduce SHAPEASSEMBLY, an "assembly language" for 3D shape structures. In SHAPEASSEMBLY, shape structures are represented by hierarchical assemblies of connected parts, where leaf-level parts are approximated by a bounding cuboid (a similar representation as the ones used by PartNet [141] and StructureNet [140]); these hierarchical cuboid structures can then be used to condition the generation of shape surface geometry in the form of e.g. point clouds. A SHAPEASSEMBLY program constructs a shape by declaring cuboids, iteratively attaching them to one another, and specifying symmetric repetitions of connected cuboid assemblies. The dimensions of these cuboids and the positions of these attachments are a program's parameters; manipulating them allows for exploring a family of related shapes. Furthermore, our interpreter for executing SHAPEASSEMBLY programs is fully differentiable, meaning it is possible to compute gradients of a program's output geometry with respect to its continuous parameters. Figure 3.1 shows some example hierarchical SHAPEASSEMBLY programs and the output shapes they produce.

While SHAPEASSEMBLY programs produce valid geometry under a range of parameter values, they do not exhibit *structural* variability, and authoring them from scratch still takes time. Thus, we train a neural network to write a variety of SHAPEASSEMBLY programs for us. Using programs we extract from a shape dataset, we train a hierarchical sequence VAE which outputs hierarchical SHAPEASSEMBLY programs. Each

node in the hierarchy uses a recurrent language model to generate the program text at that level, and to decide which cuboids should be expanded into subroutine calls. Furthermore, the well-defined semantics of SHAPEASSEMBLY allow us to identify semantically-invalid programs and modify the generator such that it never produces them. The programs shown in Figure 3.1 were written by our generative model, by decoding code vectors along a straight line in its latent space. We show that this generative model indeed learns to generate plausible, novel shape programs that were never seen its the training set. Note that one could consider solving our problem of novel shape program generation by first generating novel 3D *shapes* with an existing shape generative model and then using a VPI-like system to infer a program describing that shape. However, as we will later show, the programs produced by such a process are less clean and editable than ones generated by our model; furthermore, training to generate programs rather than shapes directly actually produces a better-structured latent space.

We evaluate our approach by comparing it to other recently-proposed generative models of 3D shape structure along several axes including plausibility, diversity, complexity, and physical validity. We find that our generated shapes are both more plausible and more physically-valid than those of other methods. Additionally, we assess the latent spaces of these models, and find that ours is better structured and produces smoother interpolations, both in terms of geometric and structural continuity. As a bonus, we also show that SHAPEASSEMBLY's decoder does a better job of fitting programs to unstructured point clouds while also maintaining physical validity, and that this performance difference is magnified by optimizing the program fit via our differentiable interpreter.

We provide code for our method at https://github.com/rkjones4/ShapeAssembly .

3.1 Approach

Figure 3.2 shows our overall pipeline. Our approach is divided into the following stages:

Input Our pipeline takes as input a large dataset of *hierarchical 3D part graphs* [141, 140]. This is a shape representation in which each node represents a part in a shape consisting of an assembly of parts. Nodes are connected via edges that denote physical part attachments. They can also be connected via parent-child edges that denote hierarchy relationships (i.e., that one part is composed of several other smaller parts). At the leaf level of this hierarchy, atomic parts are represented by cuboid proxy geometry (typically computed from minimum-volume bounding boxes of more detailed part meshes).

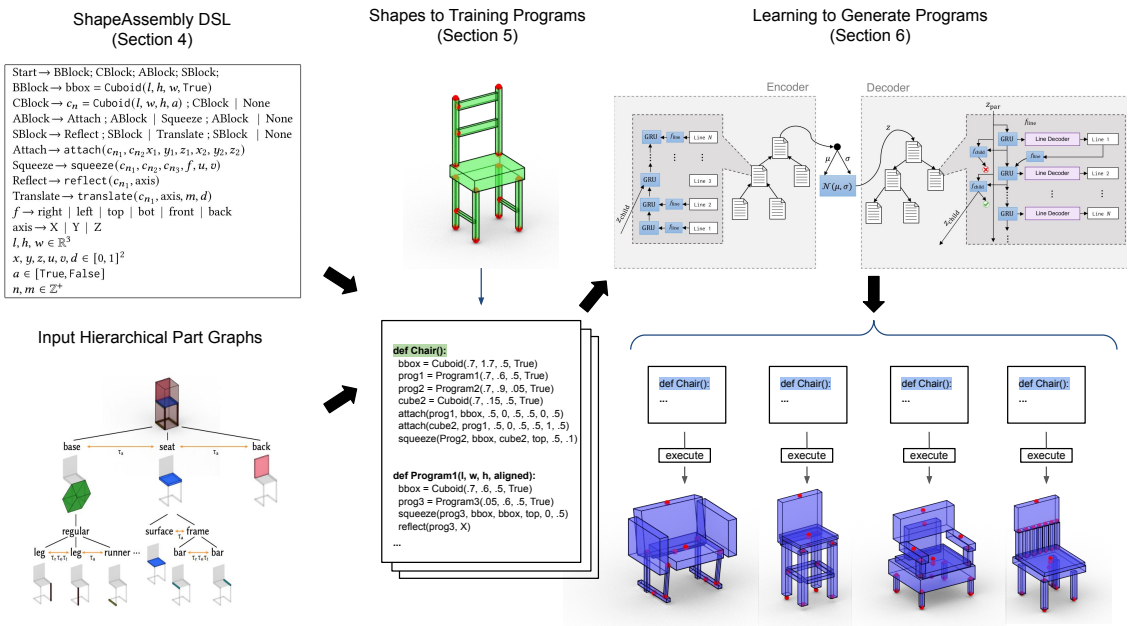

Figure 3.2: Our pipeline for generating 3D shape structure programs. We first define a DSL language for 3D shapes, SHAPEASSEMBLY . Then, given a dataset of hierarchical part graphs, we extract SHAPEASSEMBLY programs from them. Finally, we use these programs as training data for a deep generative model. Our method learns to generate novel program instances that can be executed to produce complex and interesting 3D shape structures.

Defining a DSL for connected, hierarchical shapes To represent shapes as programs, we introduce a domain-specific language (DSL). Since our input shapes are characterized by graphs of parts, where graph edges denote physical part connections, we introduce a DSL based around declaring parts and then attaching them to one another. We call this language SHAPEASSEMBLY (as in, an "assembly language" for shapes). Section 3.2 describes the language.

Creating a dataset of shape-program pairs Given the language described above, we present a method for finding programs that represent the shapes in our dataset. In our procedure, we first extract the program content based on a combination of data cleaning and geometric analysis. Then, we create canonical programs through a series of ordering and filter steps. Section 3.3 describes this procedure in more detail.

Learning to generate programs Finally, we treat the programs extracted from each shape as training data for a generative model. Section 3.4 describes our deep generative model's architecture, the procedure we use to train it, and how we sample from it to synthesize new programs, which when executed produce novel shape structures.

3.2 An Assembly Language for Shapes

Our goal in this section is to define a domain-specific language for shapes which are specified as connected assemblies of parts. As we focus on the problem of shape structure synthesis, cuboids, serving as part proxy geometry, are the only data type in our language. In Section 3.5, we show how to use other existing techniques to convert these proxies into surface geometry.

The primary operation in the language is attaching these cuboids together. Attachment turns out to be a very powerful and flexible operation. In fact, our language does not include any operations for explicitly positioning or orienting cuboids: all of this is accomplished via attachment operations. Additionally, the language includes higher-level *macros* that capture more complex spatial relationships, such as symmetry. At execution time, each macro is expanded into a series of cuboid declarations and attachment operations.

We call this DSL SHAPEASSEMBLY, because it is an "assembly language for shapes": a low-level language for creating shapes, in which shapes are created by assembling parts. Table 3.1 shows the grammar for SHAPEASSEMBLY, and Figure 3.3 shows an annotated program along with its output shape.

A SHAPEASSEMBLY program consists of four main blocks:

- **BBlock:** Declares a non-visible bounding volume of the overall shape. This bounding volume is treated

```
1.  def Chair():
2.    bbox = Cuboid(1, 1.5, .8, True)
3.    base = Base(.8, .5, .8, True)
4.    cube1 = Cuboid(.8, .1, .8, True)
5.    back = Back(.9, .8, .07, True)
6.    attach(base, bbox, .5, 0, .5, .5, 0, .5)
7.    attach(cube1, base, .5, 0, .5, .5, 1, .5)
8.    squeeze(back, bbox, cube1, top, .5, .05)

9.  def Base(l, w, h, aligned):
10.   bbox = Cuboid(l, w, h, aligned)
11.   cube0 = Cuboid(.2, .5, .2, True)
12    cube1 = Cuboid(.2, .5, .2, True)
13.   squeeze(cube0, bbox, bbox, top, .1, .1)
14.   squeeze(cube1, bbox, bbox, top, .1, .8)
15.   reflect(cube0, X)
      reflect(cube1, X)

17. def Back(l, w, h, aligned):
18.   bbox = Cuboid(l, w, h, aligned)
19.   cube0 = Cuboid(.9, .4, .07, True)
20.   cube1 = Cuboid(.1, .4, .05, True)
21.   attach(cube0, bbox, .5, 1, .5, .5, 1, .5)
22.   squeeze(cube1, bbox, cube0, bot, .3, .5)
23.   translate(cube1, X, 2, .5)
```

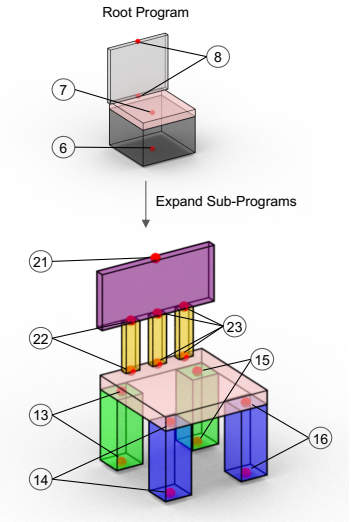

Figure 3.3: An example SHAPEASSEMBLY program and the shape that it generates. Parts are colored according to the line of the program which instantiates them, and attachment points are numbered accordingly. In the top shape, we show the executed Chair program without hierarchy. In the bottom shape, we show the Chair program executed hierarchically with its sub-programs (Base and Back). For instance, the light grey back part is expanded into the purple back surface and gold slats.

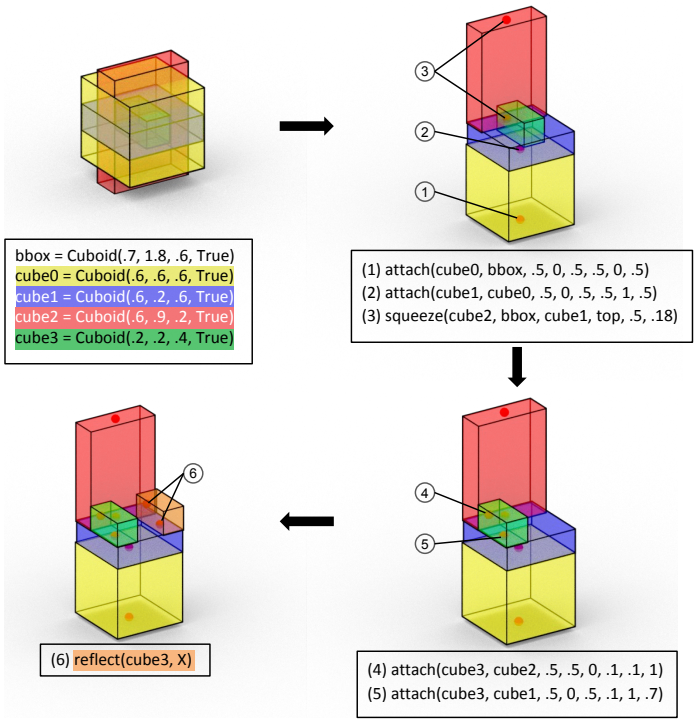

Figure 3.4: An illustration of how the SHAPEASSEMBLY interpreter incrementally constructs shapes by imperatively executing program commands. Cuboids are instantiated at the origin and are moved through attachment. Notice how the `reflect` command in line 6 acts as a macro function, creating a new cuboid and two new attachments.

as a physical entity to which other parts can be connected.

- **CBlock:** Declares all the cuboid part proxies that will be used by the remainder of the program. The `Cuboid` command takes in l, w, h parameters that control the starting dimensions of the part, and an aligned flag a that specifies if the part has the same orientation as its bounding volume.
- **ABlock:** Connects cuboids by iteratively attaching them to one another. The `attach` command takes in two cuboids, c_{n1}, c_{n2}, and attaches the point (x_1, y_1, z_1) in the local coordinate frame of c_{n1} with the point (x_2, y_2, z_2) in the local coordinate frame of c_{n2}. The `squeeze` macro expands into two `attach` statements, such that c_{n1} is placed in-between c_{n2} and c_{n3} along the specified face f at the face-coordinate position (u, v).
- **SBlock:** Generates symmetry groups by instantiating additional `Cuboid` and attach commands. The `reflect` macro reflects cuboid c_n over axis $axis$ of the bounding volume. The `translate` macro creates a translational symmetry group starting at c_n with m additional members along axis a of the bounding volume that ends distance d away.

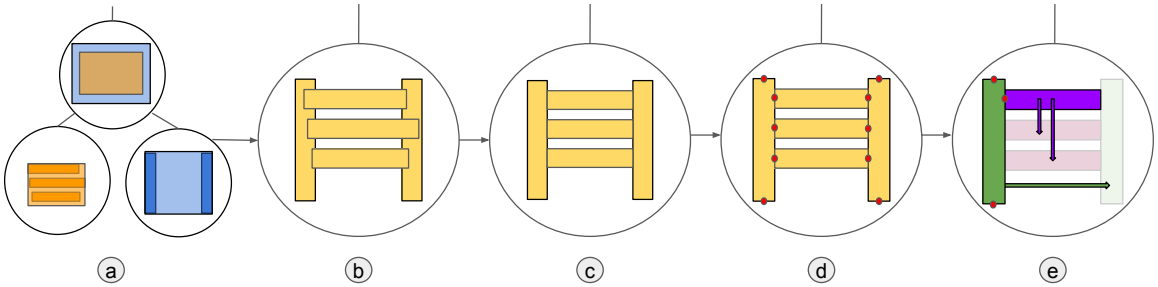

Figure 3.5: The steps of our program extraction pipeline. *(a)* Fragment of an input hierarchical part graph showing chair back (parent node), chair back frame (blue child), and chair back surface (orange child). *(b)* Locally flattening the hierarchy so that physically interacting leaf parts become siblings. *(c)* Shortening leaf parts that intersect other leaf parts. *(d)* Locating attachment points between parts. *(e)* Forming leaf parts into symmetry groups.

Semantics SHAPEASSEMBLY has imperative semantics: every line of the program immediately takes effect and alters the state of the shape being constructed. Figure 3.4 shows an example of a simple shape being imperatively constructed. Declaring a cuboid instantiates a new piece of cuboid geometry with the requested dimensions, centered at the origin. Invoking the `attach` command alters the cuboid, potentially translating, rotating, or resizing it in order to satisfy the attachment (see Appendix A.1 for details). Higher-level macros expand into two or more `Cuboid` or `attach` lines, which are then immediately executed (see Appendix A.2 for details).

One distinct advantage of this imperative semantics, as opposed to an alternative formulation in which the

Table 3.1: The grammar for SHAPEASSEMBLY, our low-level domain-specific "assembly language" for shape structure. A program consists of `Cuboid` statements which instantiate new geometry and `attach` statements which connect these geometries together at specified points on their surfaces. Macro functions (`reflect`, `translate`, `squeeze`) form complex spatial relationships by expanding into multiple `Cuboid` and `attach` statements.

Start $\rightarrow$ BBlock; CBlock; ABlock; SBlock;
BBlock $\rightarrow$ bbox $=$ `Cuboid`$(l, h, w,$ `True`$)$
CBlock $\rightarrow c_n =$ `Cuboid`(l, w, h, a) ; CBlock — None
ABlock $\rightarrow$ Attach ; ABlock — Squeeze ; ABlock — None
SBlock $\rightarrow$ Reflect ; SBlock — Translate ; SBlock — None
Attach $\rightarrow$ `attach`$(c_{n_1}, c_{n_2}, x_1, y_1, z_1, x_2, y_2, z_2)$
Squeeze $\rightarrow$ `squeeze`$(c_{n_1}, c_{n_2}, c_{n_3}, f, u, v)$
Reflect $\rightarrow$ `reflect`$(c_n,$ axis$)$
Translate $\rightarrow$ `translate`$(c_n,$ axis$, m, d)$
$f \rightarrow$ right — left — top — bot — front — back
axis $\rightarrow$ X — Y — Z
$l, h, w \in \mathbb{R}^+$
$x, y, z, u, v, d \in [0, 1]^2$
$a \in$ [`True`, `False`]
$n, m \in \mathbb{Z}^+$

program specifies constraints which are jointly optimized, is that the entire process of executing a program is end-to-end differentiable. That is, it is possible to compute the gradient of the program's output geometry with respect to the continuous parameters in the text of the program (e.g., cuboid dimensions, attachment point locations). We make use of this feature in results shown later in this chapter.

Handling hierarchy Thus far, we have described a language that can generate flat assemblies of parts, but not hierarchical ones. The extension to hierarchical shapes is straightforward: we represent hierarchical shapes by treating select non-leaf cuboids as the bounding box of another program (e.g., the contents of its "BBlock"). Figure 3.3 shows an example of a program in which cuboids expand into sub-programs.

3.3 Turning Shapes into Training Programs

SHAPEASSEMBLY allows us to write programs that generate new shapes. However, we are interested in using the language to represent existing shapes in a dataset, so that we can learn to generate novel instances from the same underlying shape distribution. In this section, we describe how we accomplish this goal. Given an input shape, represented as a hierarchical part graph, the process divides into three steps: extracting program information, creating candidate programs, and checking program validity.

3.3.1 Extracting Program Information

To convert hierarchical part graphs into SHAPEASSEMBLY programs, we perform a series of data regularizations, record cuboid parameters, locate cuboid-to-cuboid attachments, and identify symmetry groups (Figure 3.5). We provide a high-level overview of the steps involved here, and a detailed description in Appendix A.3.

Regularization Before we parse program attributes, we attempt to create more regularized part graphs through a series of data-cleaning steps. For instance, in the flattening phase, we restructure the part graph hierarchy so that leaf parts with spatial relationships are more often siblings. In the shortening phase, we decrease the dimensions of leaf cuboids that interpenetrate other leaf cuboids (to create more surface-to-surface part connections).

Cuboids Ground truth cuboid dimensions are provided in the input part graphs. A cuboid is marked as aligned if its orientation matches its parent cuboid (with an allowable error of 5-degrees).

Attachment To locate cuboid-to-cuboid attachments, we sample a uniform, dense point cloud on each cuboid in the scene. For each pair of cuboids, we compute the intersection of the point clouds. If the intersection set is non-zero, we record an attachment point within the volume formed by the intersection, with preference for locations on the centers of faces. For every cuboid, we then check if any of its parsed attachments could be represented as a squeeze relationship, and replace any that can.

Symmetry To find symmetry groups, we identify collections of cuboids that share a reflectional or translational relationship about either the X, Y, or Z axis of their parent cuboid. For each collection, if all of the member cuboids have the same connectivity relationships, we form them into a symmetry group. Each symmetry group is represented by a transform applied to a single cuboid, and all other members are removed from the graph.

3.3.2 Creating Candidate Programs

Given the extracted program information, we know the content of the program, but not how the lines should be ordered. To make the task of learning a generative model of programs easier, we aim to extract only a single, "canonical" program for each shape. As the ordering of cuboid and symmetry lines doesn't change the executed geometry, this consistency is enforced by ordering these lines according to the semantic label of each part involved in the line. Ties in this ordering between same part-type cuboids are broken by sorting on centroid position.

Deciding on a single ordering of the attach and squeeze statements is more challenging. Since SHAPEASSEMBLY has an imperative execution semantics, the order in which these commands are executed is significant: different orderings can potentially create different output geometries. To reduce the space of possible orderings, we only consider programs which follow a *grounded* attachment order, which we define as follows:

- Initially, only the shape bounding box is grounded.
- The only valid attachments to perform are those which connect a cuboid to a grounded cuboid.
- After executing an attachment, the newly-attached cuboid becomes grounded.

If there are multiple valid grounding orders, we first discard any orderings that produce worse geometric fits to the target shape. If ambiguities in the attachment ordering still remain, we break ties using (1) the semantic ordering of the cuboids involved in the attachment (2) preferring attachments from non-aligned to aligned cuboids and finally (3) preferring attachments from cuboid face-centers.

3.3.3 Validating Programs

Once we extract a canonical SHAPEASSEMBLY program, we perform a series of checks to verify the results of our procedure. Programs must pass the following validation steps in order to be added to our training data:

Reconstruction Executed programs should recreate the geometry of their respective ground truth part graph. To verify this, we sample point clouds from the surfaces of the ground truth shape and the geometry generated by executing the canonical program. These point clouds are compared using the F-score [108] metric; a program is filtered out if it produces an F-score lower than 75.

Semantics Programs must respect the semantics of SHAPEASSEMBLY. For instance, within each program, the connectivity graph of all parts should have only one component. Likewise, executed programs should not create geometry that extends beyond the bounding volumes they define.

Complexity Programs that are overly complex (more than 12 leaf cuboid instantiations) are discarded. Note that, when executed, programs can still produce more than 12 leaf cuboids through expanding symmetry macros.

3.4 Learning to Generate Programs

Given the programs extracted from our dataset, we now have the data we need to train a neural network to write novel hierarchical SHAPEASSEMBLY programs for us. In this section, we describe the generative model architecture we use, our learning procedure, and how we sample new shapes from the learned model.

3.4.1 Model Architecture

Figure 3.6 shows our generative model architecture. It is a hierarchical sequence VAE. The encoder branch embeds a hierarchical SHAPEASSEMBLY program into a latent space. The decoder branch converts a point in this latent space into a hierarchical SHAPEASSEMBLY program. The bottleneck of our network is parameterized by separate μ and σ vectors in the standard variational autoencoder (VAE) setup.

The dark grey callout in Figure 3.6 illustrates the operation of our decoder within a single node of the program hierarchy. The decoder receives as input the latent code z_{par} of its parent node (or the root latent code from the encoder, if it is the root node of the hierarchy). This latent code is used to initialize the hidden

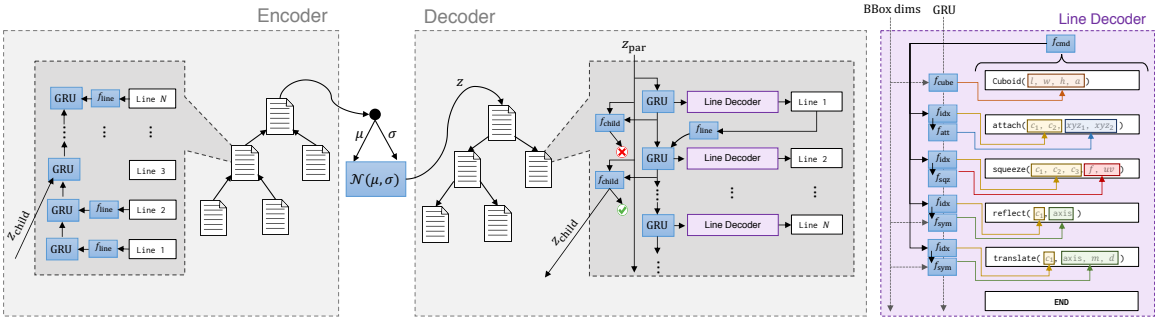

Figure 3.6: Architecture of our hierarchical sequence VAE for SHAPEASSEMBLY programs. Given a SHAPEASSEMBLY program, the encoder ascends the hierarchy from the leaves to the root, encoding each sub-program into a latent z vector. Given a latent code, the decoder recursively decodes a hierarchical SHAPEASSEMBLY program. Within each hierarchy node, a recurrent neural network decodes each line of the program.

state of a Gated Recurrent Unit (GRU), a recurrent language model which is responsible for constructing a representation of the program state. The output of the GRU cell is sent to the line decoder sub-routine, which predicts a line in the SHAPEASSEMBLY grammar, that is then passed as input back to the GRU cell at the next time step.

The purple callout in Figure 3.6 gives a detailed depiction of the line decoder sub-routine. The line decoder receives the hidden state of the GRU cell, along with conditioning information about the size of the current bounding volume, and uses a collection of multilayer perceptrons (MLPs) to predict a 63-dimensional vector representing a single line in SHAPEASSEMBLY . The sub-networks it uses are:

- $f_{\textbf{cmd}}$**: (7):** Predicts the type of command to execute. This is a one-hot vector whose seven entries correspond to `<start>` (the special program start token), `<stop>` (the special program stop token), `Cuboid`, `attach`, `squeeze`, `translate` and `reflect`.
- $f_{\textbf{cube}}$**: (4):** Predicts the length, width, height, and aligned flag for `cuboid` lines, conditioned on the bounding volume dimensions.
- $f_{\textbf{idx}}$**: (11 $\times$ 3):** Predicts the indices of the cuboids involved in the line represented as 3 one-hot vectors, conditioned on the predicted command. We limit each node in the hierarchy to contain at most 10 children parts, so there are 11 choices (10 cuboids and the bounding volume).
- $f_{\textbf{att}}$**: (3 $\times$ 2):** Predicts the (x, y, z) coordinates involved in an `attach` line, conditioned on the cuboids involved in the attach.
- $f_{\textbf{sqz}}$**: (8):** Predicts the the face involved in a `squeeze` line as a one-hot vector in the first 6 indices. The last 2 indices predict the (u, v) coordinates. Both predictions are conditioned on the cuboids involved

in the squeeze operation.

- f_{sym}**: (5):** Predicts the axis involved in a symmetry line as a one-hot vector in the first 3 indices. For `translate` lines, the 4th index is the number of cuboids involved in the symmetry group, and the 5th index is the scale of the symmetry. All predictions are conditioned on the cuboid involved in the symmetry and the bounding volume dimensions.

Hierarchical decoding To generate a hierarchical program, our decoder also includes a submodule f_{child} which is executed after every predicted `Cuboid` command to determine whether that cuboid should be recursively expanded. This is another MLP which takes as input both the current hidden state of the GRU as well as z_{par}, the overall latent code for this hierarchy node. f_{child} produces two outputs: a Boolean flag for whether the current cuboid should be expanded into a child program, and a new latent code z_{child} which is used to initialize the decoder for this child program.

3.4.2 Learning Procedure

We implement our models in PyTorch[158]. All training is done with the Adam optimizer [106], with a learning rate of 0.0001 without batching. All multilayer perceptrons have 3 layers and use leaky ReLU [129] with $\alpha = 0.2$.

We train our model in a seq2seq fashion, where the ground truth input sequence is teacher forced to the model, and our model is tasked with predicting each subsequent line. During training, we use a program reconstruction loss that only considers entries of the predicted 63 dimensional vector that are relevant to the target line. For instance, when predicting a `Cuboid` line, no part of the reconstruction loss comes from the indices in the tensor associated with symmetry. The program reconstruction loss is comprised of a cross-entropy component for each one-hot prediction (with weight 1) and an l1 loss for each continuous component (with weight 50). Additionally we use a KL loss in the standard VAE setup with weight 0.1 [107].

Enforcing semantically-valid output As our model generates shape *programs*, rather than raw shape geometry, we can use the semantics of the SHAPEASSEMBLY language to detect outputs that would be invalid, and prevent them from happening. For instance, attaches must be made in a grounded order. If a predicted `attach` line violates such a constraint, we use a backtracking procedure to find new 'valid' parameter values whenever possible. During unconditional generation, if we cannot fix the line through backtracking, we reject the sample. During interpolation, if we cannot fix the line through backtracking we don't add the predicted

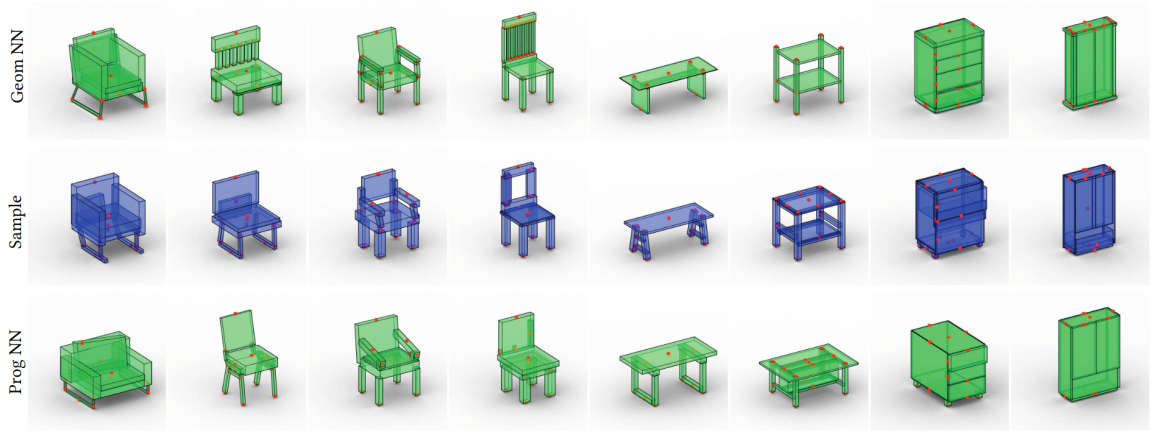

Figure 3.7: In the middle row, we show samples from our generative model of SHAPEASSEMBLY programs. In the top row, we show the nearest neighbor shape in the training set by Chamfer distance. In the bottom row, we show the nearest neighbor shape in the training set by program edit distance. Our method synthesizes interesting and high-quality structures that go beyond direct structural or geometric memorization. We quantitatively examine SHAPEASSEMBLY's generalization in Table 3.4. Refer to the supplemental material for the corresponding program text.

line to the program. Appendix A.4 describes the complete semantic validity procedure we enforce. We also note that this approach to forbidding the generation of invalid outputs is similar to that of the Grammar Variational Autoencoder [114]. However, that model only uses grammar *syntax* to determine whether an output is valid, whereas as we use program *semantics*.

3.5 Results and Evaluation

In this section, we demonstrate our learned generative model's ability to synthesize high-quality hierarchical SHAPEASSEMBLY programs, and we compare it to alternative generative models of 3D shape structure. All of the experiments described were run on a GeForce RTX 2080 Ti GPU with an Intel i9-9900K CPU, and consumed 3GB of GPU memory.

We use objects from the PartNet dataset [141] as our training data. It contains 3D shapes in multiple categories, each with a hierarchical part segmentation and labeling. For the experiments in this chapter, we use the *Chairs*, *Tables*, and *Storage* categories. After running the program extraction procedure described in Section 3.3, we obtain 3835 ground truth programs from *Chairs*, 6536 ground truth programs from *Tables*, and 1551 ground truth programs from *Storage*.

3.5.1 Novel Shape Synthesis

In this section, we present both qualitative and quantitative evaluations of our method's ability to produce novel shape structures. Figure 3.7 includes some unconditionally generated samples from our learned generative model for each of the three shape categories. Above each sample we show its nearest neighbor in the training data based on Chamfer distance. Additionally, below each sample we visualize its nearest neighbor in the training data based on program distance, the string edit distance of a tokenized version of our hierarchical programs. As shown, our method is able to generate complex and interesting structural variation without copying either the geometry or program structure of its training data.

As our model directly generates programs, its outputs can be easily edited to produce variants. In Figure 3.8 we demonstrate how by changing just the continuous parameters of programs generated by our model, we are able to create a wide variety of output geometry, all the while maintaining part-to-part attachment relationships.

We compare the generated results of our method against two baselines:

- **StructureNet** is a variational autoencoder that generates hierarchical part graphs with cuboids at each node [140].
- **3D-PRNN** is a recurrent neural network that generates a sequence of cuboids [247]. It enforces global bilateral symmetry by only generating cuboids with some part of their geometry on the negative side of the $x = 0$ plane, and then reflecting generated cuboids which fall entirely on that side of the plane.

We compare against the StructureNet models released by the authors. These were trained on the subset of PartNet that they were able to represent within the constraints of their problem formulation. This is a heavily overlapping set, but not identical, with the shapes we were able to find valid SHAPEASSEMBLY programs for. In direct comparisons with StructureNet for reconstruction tasks, we only consider shapes that appear in the validation splits of both methods. We compare against a version of 3D-PRNN that was re-trained on the data we use for our generative model. Figure 3.9 shows a qualitative comparison of unconditionally generated samples from each method. Our method is capable of generating diverse, structurally complex, 3D shape structures across multiple categories. Attachment as a primary operation provides a strong inductive bias for generating physically plausible shapes that maintain realistic part-to-part relationships. In contrast, both comparison methods that directly predict part placements in 3D space are prone to producing floating cuboids or jumbled collections of spatially colocated parts.

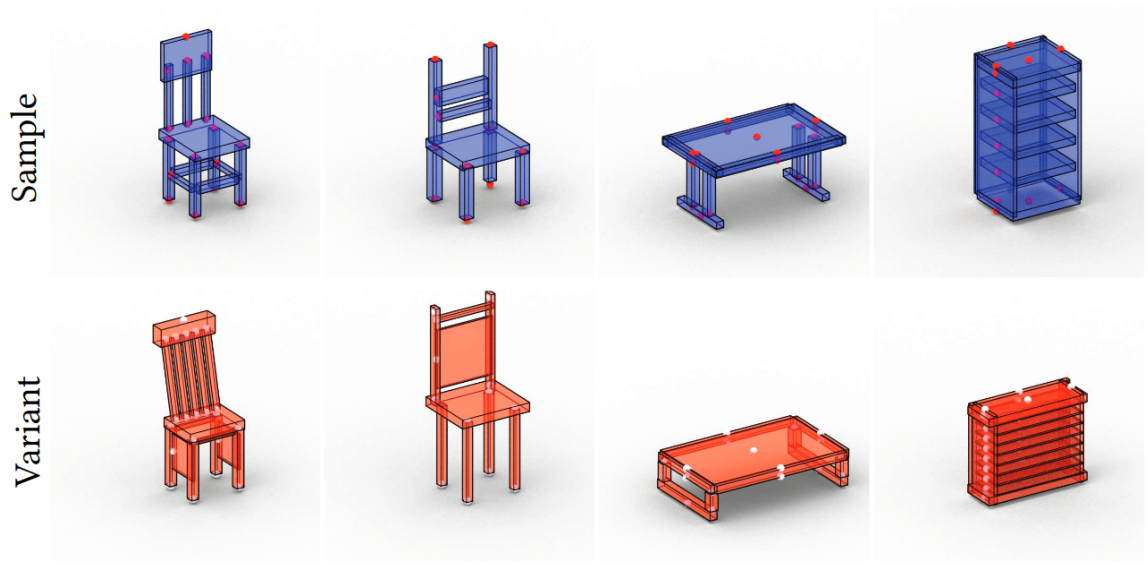

Figure 3.8: Programs, by way of representational form, allow for easy semantic editing of generated output. Each column shows a sample from our model in the top row. In the bottom row we create a variant with the same structure, but different geometry, by editing only the continuous parameters of the program. Program text can be found in the supplemental material.

Analysis of Shape Quality

We also quantitatively compare the quality of the shape structures generated by different methods. Our desiderata for generated shape structures is that they should be physically plausible and come from the same distribution that the model was trained on. In order to asses the quality of generated output, we use the following metrics:

- **Rootedness ⇑ (% rooted):** The percentage of shapes for which a connected path exists between the ground and all leaf parts.
- **Stability ⇑ (% stable):** The percentage of shapes which remain upright under gravity and small forces in a physical simulation.
- **Realism ⇑ (% fool):** The percentage of test set shapes classified as "generated" by a PointNet classifier trained to distinguish between generated shapes and shapes from the training dataset.
- **Frechet Distance ⇓ (FD):** Measurement of distributional similarity between generated shapes and the training dataset using the feature space of a pre-trained PointNet model [73]

Further details about these metrics are provided in Appendix A.5.

We show results for these metrics on 1000 unconditional generated shapes in Table 3.2. Our method

largely outperforms 3D-PRNN and StructureNet across these metrics for three categories of shapes. While StructureNet achieves good rootedness scores, especially for the Storage category, our method performs better in the other three metrics along all categories. The samples from 3D-PRNN, achieve similar FD and % fool scores with StructureNet, but perform markedly worse on the rootedness and stability metrics.

Additionally in this experiment we compare our model with a series of ablated versions:

- **Flat:** Training on programs with no hierarchies, only leaf parts.
- **No Order:** Training on programs without canonical ordering as described in Section 3.3.
- **No Align:** Training on programs without an aligned flag for cuboids.
- **No Macros:** Training on programs without `squeeze`, `translate`, or `reflect` commands.
- **No Reject:** At generation time, discard unfixable, invalid program line predictions instead of rejecting the entire sample.

Training without hierarchy (Flat) slightly improves rootedness, but drastically lowers the quality of output as seen in the % fool and FD columns. Training on programs without a canonical ordering (No Order) performs worse on every metric. Removing the alignment flag (No Align) actually improves performance on the Chair category for % rooted and % fool, but drastically worsens the physical validity of generations for Tables and Storage, categories where parts are much more often aligned with their parent cuboid. Training without macros (No Macros) once again decreases the performance of all metrics, but not by a substantial margin. Finally, we see that while the rejection sampling step does improve the quality of our generated samples, without it we still outperform 3D-PRNN and StructureNet by a wide margin.

Analysis of Editability

In this section, we quantitatively analyze our previous claim that directly predicting programs improves editability. We claim that a program is more editable if it is both compact and compromised of higher level functions. That is, a shorter program that uses higher-level constructs will be easier to understand and make changes to.

As a strong baseline, we evaluate the editability of our programs against the generated outputs of 3D-PRNN and StructureNet. As 3D-PRNN and StructureNet do not directly produce SHAPEASSEMBLY programs, we use our extraction procedure described in Section 3.3 in order to convert their generations into programs. As StructureNet predicts part graph hierarchies, the representational form our extraction procedure takes as input, we use our procedure without any of the data cleaning steps. As 3D-PRNN has no notion

Table 3.2: Comparing the quality of generated samples. Our method outperforms other generative methods for 3D shape structure in terms of realism and physical validity. Through a series of ablation baselines, we validate various design decisions of our method.

Category	Method	% rooted ⇑	% stable ⇑	% fool ⇑	FD ⇓
Chair	3D-PRNN	73.1	50.9	12.60	39.30
	StructureNet	89.7	74.9	4.04	64.79
	Ours (Flat)	**95.0**	60.0	11.58	77.45
	Ours (No Order)	82.4	58.4	12.36	64.17
	Ours (No Align)	94.6	84.6	**28.68**	29.32
	Ours (No Macros)	92.0	77.9	19.56	36.78
	Ours (No Reject)	92.9	79.7	23.36	**20.63**
	Ours	94.5	**84.7**	25.06	22.34
	Ground Truth	100	88.0	—	—
Table	3D-PRNN	71.2	29.4	2.12	140.07
	StructureNet	94.4	76.8	3.94	173.35
	Ours (Flat)	87.0	66.0	29.84	148.63
	Ours (No Order)	84.5	56.0	27.38	114.10
	Ours (No Align)	92.2	61.5	23.64	**46.64**
	Ours (No Macros)	95.9	85.0	33.16	53.21
	Ours (No Reject)	94.1	76.4	29.20	52.78
	Ours	**96.2**	**85.9**	**33.21**	49.07
	Ground Truth	100	93.1	—	—
Storage	3D-PRNN	44.8	20.8	4.62	94.08
	StructureNet	**96.2**	75.0	5.04	92.85
	Ours (Flat)	95.9	74.0	7.44	81.17
	Ours (No Order)	87.9	63.4	8.70	107.42
	Ours (No Align)	89.7	49.3	11.04	**30.15**
	Ours (No Macros)	87.5	69.9	5.92	72.80
	Ours (No Reject)	94.3	80.9	11.66	31.69
	Ours	95.3	**83.7**	**13.50**	31.72
	Ground Truth	100	87	—	—

of hierarchy, we create single node part graphs out of their output samples, which are then run through our program extraction logic.

Table 3.3 shows results from an experiment where we compare the SHAPEASSEMBLY programs of each method's generations (directly predicted by our method, parsed programs from comparisons). The metrics we use are the number of lines in each program (as a coarse measure of compactness) and the percentage of lines which are macros (split by macro type).

Compared with programs parsed from StructureNet, the programs generated by our model are much more compact and have higher rates of macro usage across all categories of shapes. While our method also has higher macro rate usage compared with 3D-PRNN, 3D-PRNN programs are more compact in the Chair and Table categories. Based on 3D-PRNN's poor performance within our shape quality experiments (Table 3.2), and its significant deviation from the number of lines found in the ground truth programs (the cleanest set of

Table 3.3: Markers of program editability for SHAPEASSEMBLY programs predicted by our generative model compared with SHAPEASSEMBLY programs parsed from outputs of other generative methods. Training our model in the space of programs allows us to represent geometry more compactly. We find higher rates of macro functions per program line in our method's generations compared with extracting programs from other generative models' predictions.

			Macros Per Line			
Category	**Method**	**Lines ⇓**	**Refl ⇑**	**Trans ⇑**	**Squeeze ⇑**	**Total ⇑**
Chair	3D-PRNN	**15.7**	**0.1100**	0.0020	0.0240	0.1430
	StructureNet	27.1	0.0600	0.0004	0.0700	0.1330
	Ours	20.4	0.0880	**0.0054**	**0.0920**	**0.1860**
	Ground Truth	24.4	0.0800	0.0090	0.1130	0.2070
Table	3D-PRNN	**13.1**	**0.1300**	**0.0010**	0.0680	0.1990
	StructureNet	24.8	0.0270	0.0006	0.0620	0.0900
	Ours	19.0	0.0990	0.0002	**0.1440**	**0.2440**
	Ground Truth	20.0	0.0950	0.0050	0.1450	0.2460
Storage	3D-PRNN	22.6	0.0170	0.0060	0.0530	0.0770
	StructureNet	30.7	0.0390	0.0040	0.0770	0.1200
	Ours	**19.8**	**0.0820**	**0.0080**	**0.1440**	**0.2340**
	Ground Truth	24.7	0.0650	0.0147	0.1510	0.2320

SHAPEASSEMBLY programs we have access to), there is reason to believe that the compactness of its parsed programs more likely reflects shape simplicity rather than useful editability.

Analysis of Variability

Beyond quality and editability, we also consider the variability of outputs of each method. Specifically, for generated shapes, we care about their novelty with respect to the training data, their complexity, and their variety. We present results of an experiment using Chamfer distance to quantify performance across these areas in Table 3.4.

The *Generalization* metric measures the average distance of each generated sample to its nearest neighbor in the training set. As all methods have higher generalization scores than the validation set, we can conclude that none of the methods appear to be overfitting. For our method specifically, this re-enforces the qualitative nearest neighbor results presented in Figure 3.7.

The *Coverage* metric measures the average distance of each validation shape to its nearest neighbor in the set of generated shapes. Across all categories our method achieves the best results, and by a wide-margin for tables, which indicates that our generations have enough complexity to match the distribution of the validation shapes.

The *Variety* metric measures the average distance of each generated shape to its nearest neighbor in the

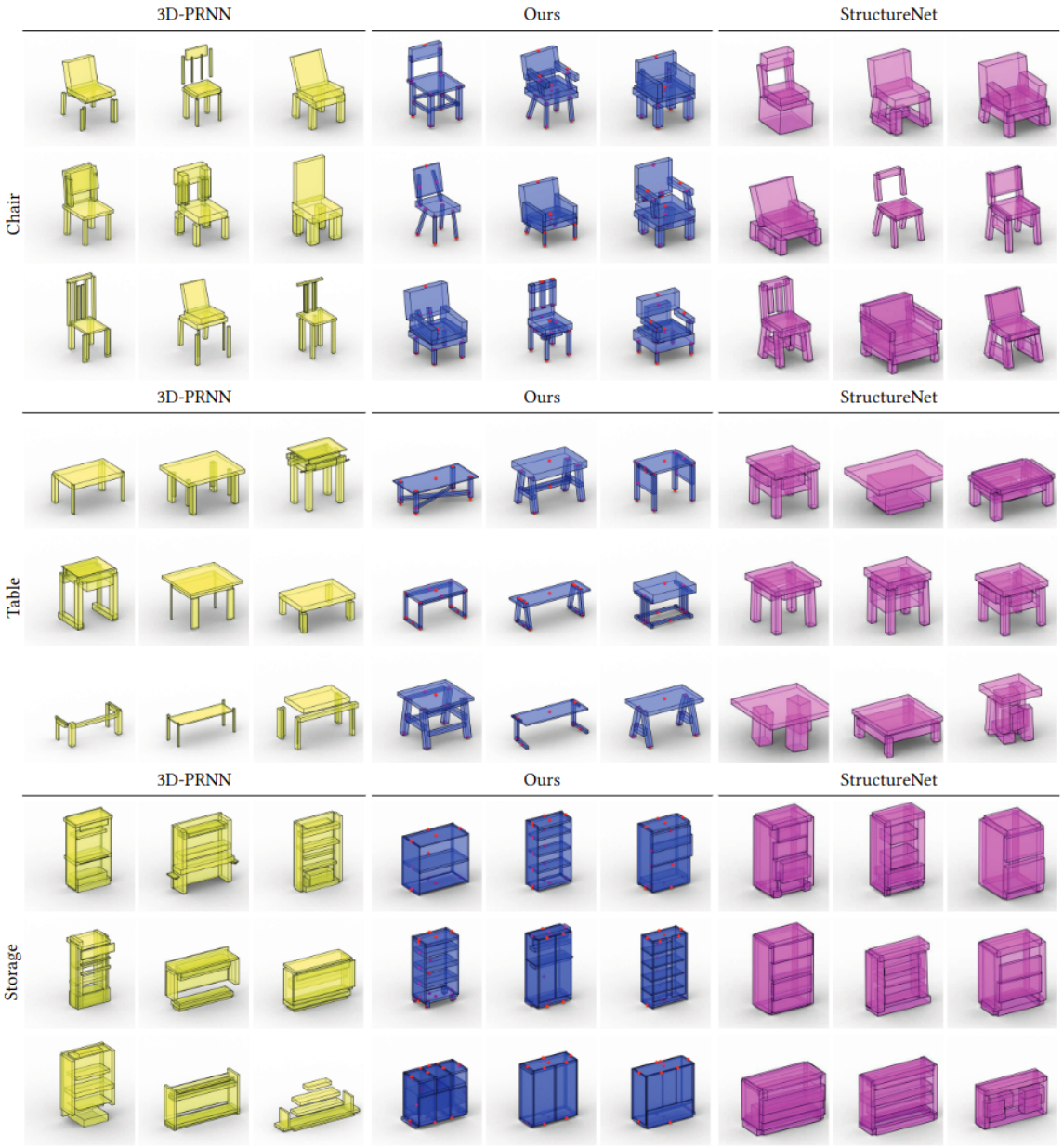

Figure 3.9: Qualitative comparison between generated samples from our method, StructureNet, and 3D-PRNN. Across different categories, our method creates novel SHAPEASSEMBLY programs that, when executed, produce shape structures that maintain realistic and physically valid part-to-part relationships. Comparison methods that directly predict 3D shape geometry exhibit failure cases where parts become disconnected or intersect in an implausible manner.

set of generated shapes besides itself. Once again, across all categories our method achieves top, or tied for top performance.

Additionally, we look at average number of leaf parts as a coarse proxy for the complexity of a shape's structure, which is shown in Table 3.5. While our method has a similar number of leaf parts to the comparison methods for the Chair and Table categories, we do have fewer leaf parts on average for Storage. Qualitatively, these additional parts in the comparison methods often manifest as collections of spatially colocated cuboids, and not necessarily more complex shape structures.

In terms of the variability of programs generated by our method, we note that 65% of Chair programs, 85% of Table programs, and 53% of Storage programs contained SHAPEASSEMBLY program structures not present in the training data. Thus our method not only exhibits novelty in the geometric domain, but also in the structural domain.

Program Clustering

Our approach is predicated on the assumption that a single program can represent a parametric family of multiple shapes, allowing for this shape space to be explored via manipulation of interpretable program parameters. To verify whether this is true, we cluster shapes that are represented by *structurally-equivalent* programs (i.e. programs that are the same up to continuous parameter variations). Figure 3.10 shows program clustering results for the ground truth programs we parse from PartNet. These results demonstrate how the structure of a single SHAPEASSEMBLY program is able to represent related shapes through different parameterizations. The marked improvement in clustering when splitting by intermediate part programs compared with clustering on entire shape programs, provides additional support for our hierarchical approach; shape programs are more likely to share structure within a node of the hierarchy than they are to match entire hierarchies exactly.

Synthesizing Surface Geometry

While collections of part proxies are a useful modeling representation for 3D shape structures, they do not directly attempt to capture the wide range of intra-part variability present in man-made objects. We demonstrate how SHAPEASSEMBLY programs can additionally be used to model parts at finer levels of detail by turning SHAPEASSEMBLY programs into dense point clouds. As a proof of concept, we augment our generative model with a point cloud encoder that consumes dense point cloud samples of ground truth leaf parts, and a point cloud decoder that generates dense point clouds for every leaf part within its predicted bounding

Table 3.4: We compare the geometric variability of generated shapes from different methods. In the first column, we measure generalization as the average nearest neighbor distance (NND) from generated samples to shapes in the training set. In the second column we measure coverage as the average NND from shapes in the validation set to generated samples. In the last column, we measure variety as the average NND from shapes in the generated samples to other generated shapes in the same set. Across three categories of shapes, our method performs the best on the coverage and variety metrics, while outperforming validation on generalization (demonstrating we are not overfitting).

		Generalization	*Coverage*	*Variety*
		NND to Train ⇑	NND from Val ⇓	NND to Self ⇑
Category	**Method**	**CD**	**CD**	**CD**
Chair	3D-PRNN	**0.111**	0.123	**0.104**
	StructureNet	0.104	0.119	0.087
	Ours	0.108	**0.118**	**0.104**
	Validation	0.105	—	0.114
Table	3D-PRNN	0.095	0.130	0.086
	StructureNet	**0.129**	0.141	0.0925
	Ours	0.101	**0.108**	**0.102**
	Validation	0.09	—	0.099
Storage	3D-PRNN	**0.134**	0.132	**0.119**
	StructureNet	0.129	0.135	0.107
	Ours	0.125	**0.129**	**0.119**
	Validation	0.11	—	0.125

Table 3.5: We compare the average number of leaf parts in generated shapes, as a coarse proxy for complexity of shape structure. Our method generates similar numbers of leaf parts compared with other methods for Chairs and Tables, but fewer leaf parts for Storage. Qualitatively, the additional leaf parts measured in comparison methods often manifests as spurious overlapping cuboids, rather than more complex structural variety.

Category	**Method**	**Avg # Leaf Parts**
Chair	3D-PRNN	8.6
	StructureNet	8.7
	Ours	7.9
	Ground Truth	9.7
Table	3D-PRNN	7.07
	StructureNet	8.16
	Ours	7.84
	Ground Truth	8.4
Storage	3D-PRNN	10.6
	StructureNet	12.3
	Ours	8.4
	Ground Truth	10.8

volume. Figure 3.11 shows some qualitative results of our method, trained on point clouds sampled from the dense geometry of Chairs found in PartNet. These generated surfaces provide additional detail over the geometry specified by their cuboid part proxies, as evidenced by both the rounding in the legs and back slats, and also in the curvature of the chair back surfaces.

Table 3.6: We measure smoothness along random high-frequency interpolation sequences in each method's latent space. The Geo column measures smoothness with Chamfer distance, while the Prog column measures smoothness with program edit distance. Note that 3D-PRNN is missing because it is not a latent variable model and thus does not support interpolation.

		Avg. Step Size ⇓	
Category	**Method**	**Geo**	**Prog**
Chair	StructureNet	**0.0384**	3.90
	Ours	**0.0384**	**1.33**
Table	StructureNet	0.0474	4.75
	Ours	**0.0389**	**2.48**
Storage	StructureNet	0.0512	4.29
	Ours	**0.0482**	**2.6**

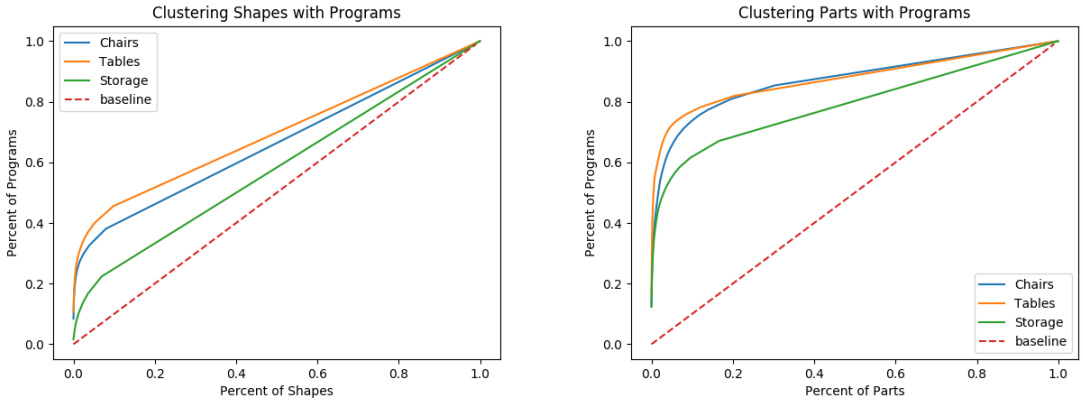

Figure 3.10: Clustering results that demonstrate how the structure of a single SHAPEASSEMBLY program is capable of capturing a family of related shapes. Using ground truth programs found with our program extraction procedure, in the left graph we plot the percentage of shapes captured as we consider more program structures extracted from the data. In the right graph we show the same plot but with parts (nodes) instead of shapes (full hierarchy).

3.5.2 Latent Space Interpolation

Beyond novel shape generation, we evaluate the ability of our method to interpolate between two points in our latent space. The presence of smooth, semantic transitions between end-points indicates a well-formed latent space. In Figure 3.12 we qualitatively compare our method with StructureNet on the task of interpolating between shapes in the validation sets of both models. Our interpolations demonstrate both geometrically smooth and semantically consistent transitions. For instance, in the top interpolation sequence, the surface of the chair back in the source shape gradually shrinks vertically until in the target shape it is just a horizontal bar. At the same time, the number of vertical slats in the chair back gradually increases from 2, to 4, to 5.

Figure 3.11: Converting generated SHAPEASSEMBLY programs into dense point clouds. We use a point cloud decoder to predict the surface geometry of each leaf part proxy in our 3D shape structure. In this process, geometric details begin to take form, at the cost of some artifacts. We discuss a method for improving this procedure in section 3.6.

In Table 3.6, we attempt to quantify the smoothness along random interpolation sequences within the latent space of each generative model. In this experiment, 100 interpolation sequences were computed from sources to targets that were randomly sampled in each model's latent space, with 100 interpolation steps per sequence. Each method's geometric smoothness is computed by taking the average Chamfer distance (normalized by shape scale) between each interpolation step. The lower geometric smoothness of our method, compared to StructureNet in the Table and Storage categories, demonstrates the quality of the latent space learned by our method. Moreover, using our procedure to turn StructureNet outputs into SHAPEASSEMBLY programs, we can measure the program smoothness along these interpolation paths. Each method's program smoothness is computed by taking the average tokenized program edit distance between each interpolation step. As a measure for structural change throughout the transitions of an interpolation sequence, our lower program smoothness metric again shows how our method benefits by operating within the space of 3D shape programs.

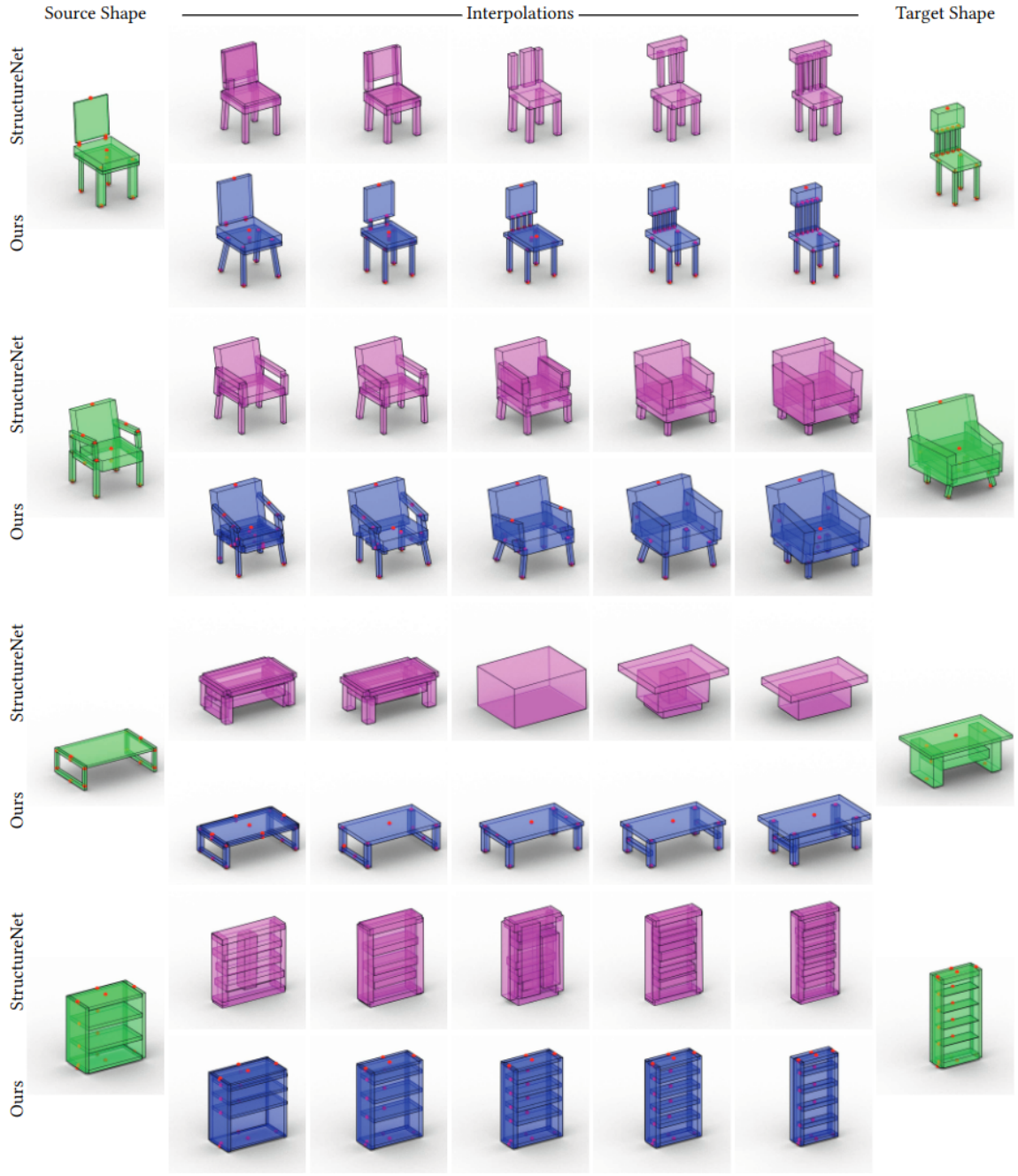

Figure 3.12: A qualitative comparison of latent space interpolation between our method and StructureNet on shapes from the validation set. Our method's interpolations within program space produce sequences that combine smooth continuous variation with discrete structural transitions.

3.5.3 Synthesis from Unstructured Geometry

Another way to inspect the structure of a generative model's latent space is through performing "synthesis from X", by projecting X into the latent space of the generative model. As an application for 3D reconstruction, we are able perform such a projection with point clouds, demonstrating how our generative model's latent space can synthesize SHAPEASSEMBLY programs from unstructured geometry.

Specifically, we train a PointNet++ encoder [163] to map point clouds sampled on dense mesh geometry to the latent space learned by our generative model. These latent codes are then converted into programs by our trained decoder.

In Table 3.7, we show an experiment comparing our method against StructureNet on the task of reconstructing point cloud samplings of dense geometry on the intersection of each method's validation set for Chairs in Partnet (463 shapes total). We evaluate reconstruction accuracy with F-score [108], and the physical validity of reconstructions with the rootedness and stability metrics. When projecting point clouds into the latent space of each method (top two rows), our method outperforms StructureNet on both reconstruction accuracy and maintaining physical validity. This demonstrates, once again, the well-structured nature of our method's latent space.

Moreover, as the SHAPEASSEMBLY interpreter is differentiable, we can further refine the continuous parameters of a program by minimizing the Chamfer distance between executed geometry and a target point cloud with a gradient-based optimizer. We compare this procedure **(Ours + Opt Program)** against the following conditions:

- **SN + Opt Cuboids**: Starting with StructureNet's reconstruction, then directly optimizing predicted cuboids to minimize Chamfer distance to the target point cloud.
- **SN + Opt Program**: Parsing StructureNet's reconstruction into a SHAPEASSEMBLY program, then optimizing the program to minimize Chamfer distance to the target point cloud.
- **Ours + Opt Cuboids**: Starting with our reconstruction, directly optimizing predicted cuboids to minimize Chamfer distance to the target point cloud.

We show results for this experiment in the last four rows of Table 3.7. All of the optimization procedures improve reconstruction accuracy at the cost of physical validity. However, Ours + Opt Program is the only condition that achieves a desirable trade-off in this exchange, gaining much more reconstruction accuracy improvement than it loses in physical validity.

We show some qualitative results of this experiment in Figure 3.13. Through latent space projection,

Table 3.7: Results from our point cloud reconstruction experiment. Our model's well-formed latent space allows for more accurate and physically valid reconstructions without further optimization. With additional optimization, using the reconstructed program from our method and our differentiable interpreter finds the best trade-off between reconstruction accuracy and maintaining physical validity.

Method	**F1** ⇑	**% rooted** ⇑	**% stable** ⇑
StructureNet	24.3	95.1	78.4
Ours	**31.1**	**95.5**	**84.4**
SN + Opt Cuboids	**80.0**	92.9	72.7
SN + Opt Program	77.4	90.0	71.9
Ours + Opt Cuboids	77.6	93.1	72.9
Ours + Opt Program	75.8	**95.3**	**80.2**

our model is able to output the rough 3D shape structure (column 1) of an input unstructured point cloud (column 0). Through our differentiable interpreter, we are able to find continuous parameters for the predicted program structure that ultimately lead to better reconstruction fits (column 3). Shape programs place a strong structural regularization prior over unstructured 3D data, and thus our presented method is less prone to "losing" semantic parts, such as small legs, in comparison to the other conditions.

3.6 Discussion

In this chapter, we took a first step toward marrying the complementary strengths of neural and procedural 3D shape generative models by introducing a hybrid neural-procedural approach for synthesizing novel 3D shape structures. We introduced SHAPEASSEMBLY, a low-level "assembly language" for shape structures, in which shapes are constructed by declaring cuboidal parts and attaching them to one another. We also introduced a differentiable interpreter for SHAPEASSEMBLY, allowing the optimization of program parameters to produce desired output geometry. After describing how to extract consistent programs from existing shape structures in the PartNet dataset, we then defined a deep generative model for SHAPEASSEMBLY programs, effectively training a neural network to write novel shape programs for us. We evaluated the quality of the generative model along several axes, showing that it produces more plausible and physically-valid shapes, and that its latent space is better-structured than that of other generative models of shape structure. We also found that directly generating shape programs leads to more compact, editable programs than extracting programs from shapes generated by methods that directly output 3D geometry.

Limitations As mention in Section 3.3, we do not successfully extract training programs from every shape in our dataset. For instance, our program extraction procedure assumes that the orientation of all parts can

Figure 3.13: Qualitative comparison of synthesis from point clouds of our method against StructureNet (SN). Our method is able to infer good program structures that match well with the unstructured geometry. The continuous parameters of this program structure can be further refined through an optimization procedure in order to better fit the target point cloud without creating artifacts.

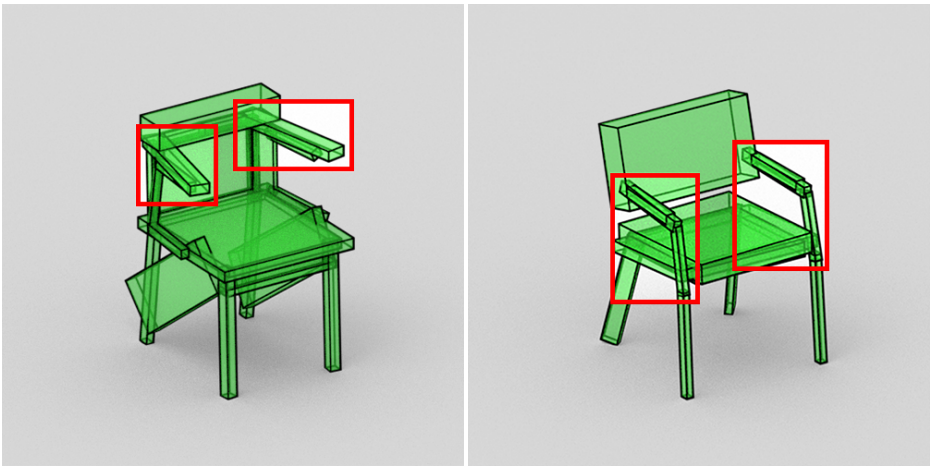

Figure 3.14: Examples of PartNet shapes that contain parts whose orientations cannot be inferred from part-to-part attachments alone. While these shapes can be represented with SHAPEASSEMBLY programs that attach parts to "floating" points within the bounding volume, such programs are not added to our training data during our program extraction phase. As a result, our generative model never learns to produce shapes that require this type of attachment pattern.

be specified through solely part-to-part attachments, yet as demonstrated in Figure 3.14, this does not hold for all shapes. While it is possible to reconstruct these shapes with SHAPEASSEMBLY programs (through attaching parts to "floating" points in space via the bounding volume) such programs will never be added to our training data, and thus our generative model won't learn to produce such constructs. Our design decision

to discard training programs with more than 12 total `Cuboid` declarations has a similar effect: it limits our generative model from synthesizing the most complex of shape structures that exist in our dataset. We impose such strict criteria in order to make our training programs exhibit more regularity, simplifying the learning task for our neural network at the expense of its potential expressivity.

This highlights a central tradeoff: higher variability in the training programs may result in lower quality shapes synthesized by a generative model. This phenomenon is not unique to our setting: it is well-known that e.g., image generative models perform better on very-regularly-structured domains, such as human faces. The question, looking forward, is how to capture more data variability while keeping a high-degree of regularity in the input data representation? We believe that using programs as a data representation is the best avenue of attack, here. As we have shown in our work, a single program can capture a wide range of parametrically related shapes. One program, many shapes; strong regularity, but also high variability.

While SHAPEASSEMBLY has a strong inductive bias for generating physically-connected shapes, it is not guaranteed to do so. Hierarchical part structures which are locally connected everywhere may occasionally still exhibit disconnected leaf cuboids. This is more likely to happen with very non-axis-aligned structures that result in loose bounding cuboids at the intermediate levels of the hierarchy.

Chapter 4

Learning to Infer Shape Programs with Pseudo-Labels and Approximate Distributions

Having access to a procedure which generates a visual datum reveals its underlying structure, facilitating high-level manipulation and editing by a person or autonomous agent. In $\mathbb{R}^2$, inferring shape programs has applications in the design of diagrams, icons, and other 2D graphics. In $\mathbb{R}^3$, it has applications in reverse engineering of CAD models, procedural modeling for 3D games, and 3D structure understanding for autonomous agents.

We formally define shape program inference as obtaining a latent program $\mathbf{z}$ which generates a given observed shape $\mathbf{x}$. We model $p(\mathbf{z}|\mathbf{x})$ with deep neural networks that train over a distribution of real shapes in order to amortize the cost of shape program inference on unseen shapes (e.g. a test set). This is a challenging problem: it is a structured prediction problem whose output is high-dimensional and can feature both discrete and continuous components (i.e. program control flow vs. program parameters). Nevertheless, learning $p(\mathbf{z}|\mathbf{x})$ becomes tractable provided that one has access to paired $(\mathbf{X}, \mathbf{Z})$ data (i.e. a dataset of shapes and the programs which generate them) [217].

In this chapter, we study a collection of methods that create (shape, program) data pairs used to train $p(\mathbf{z}|\mathbf{x})$ models with maximum likelihood estimation (MLE) updates while treating the program executor as a black-box. As discussed, ground-truth (shape, program) pairs are often unavailable, so these techniques must

make compromises in how they formulate paired data. In wake-sleep, a generative model $p(\mathbf{z})$ is trained to convergence on alternating cycles with respect to $p(\mathbf{z}|\mathbf{x})$. When training $p(\mathbf{z}|\mathbf{x})$, paired data can be created by sampling from $p(\mathbf{z})$. Each program label $\mathbf{z}$ is valid with respect to its associated $\mathbf{x}$ shape, but there is often a distributional mismatch between the generated set of shapes, $\mathbf{X}$, and shapes from the target distribution, S^*. In self-training, one uses $p(\mathbf{z}|\mathbf{x})$ to infer latent $\mathbf{z}$'s for unlabeled input $\mathbf{x}$'s; these $\mathbf{z}$'s then become "pseudo-labels" which are treated as ground truth for another round of supervised training. In this paradigm, there is no distributional shift between $\mathbf{X}$ and S^*, but each $\mathbf{z}$ is only an approximately correctly label with respect to its paired $\mathbf{x}$.

We observe that shape program inference has a unique property that makes it especially well-suited for self-training: the distribution $p(\mathbf{x}|\mathbf{z})$ is known *a priori*—this is a delta distribution defined by the program executor. When using a model $p(\mathbf{z}|\mathbf{x})$ to infer a program $\mathbf{z}$ from some shape $\mathbf{x}^*$ of interest, one can use this executor to produce a shape $\mathbf{x}$ that is consistent with the program $\mathbf{z}$: in the terminology of self-training, $\mathbf{z}$ is guaranteed to be the "correct label" for $\mathbf{x}$. However, similar to wake-sleep, formulating $\mathbf{X}$ as shape executions produced by model inferred programs can cause a distributional shift between $\mathbf{X}$ and S^*. Since this variant of self-training involves executing the inferred latent program $\mathbf{z}$, we call this procedure latent execution self-training (LEST).

As all of the aforementioned fine-tuning regimes use either Pseudo-Labels or Approximate Distributions to formulate (shape, program) pairs, we group them under a single conceptual framework: PLAD. We evaluate PLAD methods experimentally, using them to fine-tune shape program inference models in multiple shape domains: 2D and 3D constructive solid geometry (CSG), and assembly-based modeling with ShapeAssembly, a domain-specific language for structures of manufactured 3D objects (Chapter 3). We find that PLAD training regimes offer substantial advantages over the de-facto approach of policy gradient reinforcement learning, achieving better shape reconstruction performance while requiring significantly less computation time. Further, we explore combining training updates from a mixture of PLAD methods, and find that this approach leads to better performance compared with any individual method.

We provide code for our method at https://github.com/rkjones4/PLAD .

4.1 Approaches for fine-tuning visual program induction models

A common practice of methods that train networks to infer shape program is to start with a model that has been pretrained on synthetically generated (shape, program) pairs with supervised learning, and then

Method	**Models**	**Black-Box $p(\mathbf{x}\|\mathbf{z})$?**	$X = S^*$	**Low variance, unbiased gradients**
Policy gradient RL	$p(\mathbf{z}\|\mathbf{x})$	✓	✓	X
Differentiable executor	$p(\mathbf{z}\|\mathbf{x})$	X	✓	✓
Variational Bayes	$p(\mathbf{z}\|\mathbf{x})$, $p(\mathbf{z})$	X	✓	✓
Wake-sleep, EM	$p(\mathbf{z}\|\mathbf{x})$, $p(\mathbf{z})$	✓	X	✓
Self-training	$p(\mathbf{z}\|\mathbf{x})$	✓	✓	X
LEST	$p(\mathbf{z}\|\mathbf{x})$	✓	X	✓

Table 4.1: Comparison of different methods for fine-tuning $p(\mathbf{z}|\mathbf{x})$, in terms of the models that must be trained, if they treat the program executor as black-box, if their distribution of training shapes matches the distribution real shapes ($X = S^*$), and if their loss gradients are unbiased with low-variance. The last three rows describe methods that fall under the PLAD framework.

perform fine-tuning towards a distribution of interest. However, as there is typically significant distributional mismatch between these synthetic shapes and "real" shapes from the distribution of interest, S^*, various techniques must be employed to fine-tune $p(\mathbf{z}|\mathbf{x})$ models towards S^*. In this section, we discuss prior work for fine-tuning such program inference models, organized by methodology used to learn $p(\mathbf{z}|\mathbf{x})$; see Table 4.1 for an overview.

Policy Gradient Reinforcement Learning The most general method for fine-tuning a pretrained $p(\mathbf{z}|\mathbf{x})$ is reinforcement learning: treating $p(\mathbf{z}|\mathbf{x})$ as a policy network and using policy gradient methods [216]. The geometric similarity of the inferred program's output to its input is the reward function; the program executor $p(\mathbf{x}|\mathbf{z})$ can be treated as a (non-differentiable) black-box. CSG-Net uses RL for fine-tuning [187, 188], as does other recent work on inferring CSG programs from input geometry [41]. While CSG-Net has been improved to allow it to converge without supervised pretraining [243], not starting from the supervised model results in worse performance. The main problem with policy gradient RL is its instability due to high variance gradients, leading to slow convergence. Like RL, PLAD methods treat the program executor as a black-box, but as we show experimentally, they converge faster and achieve better reconstruction performance.

Differentiable Executor If the functional form of the program executor $p(\mathbf{x}|\mathbf{z})$ is known and differentiable, then the gradient of the reward with respect to the parameters of $p(\mathbf{z}|\mathbf{x})$ can be computed, making policy gradient unnecessary. Shape programs are typically not fully differentiable, as they often involve discrete choices (e.g. which type of primitives to create). UCSGNet uses a differentiable relaxation to circumvent this issue [101]. Other work trains a differentiable network to approximate the behavior of the program executor [206], which introduces errors. PLAD regimes do not require the program executor to be differentiable,

yet they perform better than other approaches (e.g. policy gradient RL) that share this desirable property.

Generative Model Learning Shape program inference has also been explored in the context of learning a generative model $p(\mathbf{x}, \mathbf{z})$ of programs and the shapes they produce. The most popular approach for training such models is variational Bayes, in particular the variational autoencoder [107]. This method simultaneously trains a generative model $p(\mathbf{x}, \mathbf{z})$ and a recognition model $p(\mathbf{z}|\mathbf{x})$ by optimizing a lower bound on the marginal likelihood $p(\mathbf{x})$. When the $\mathbf{z}$'s are shape programs, the program executor is $p(\mathbf{x}|\mathbf{z})$, so learning the generative model reduces to learning a prior over programs $p(\mathbf{z})$. Training such models with gradient descent requires that the executor $p(\mathbf{x}|\mathbf{z})$ be differentiable. When this is not possible, the wake-sleep algorithm is a viable alternative [76]. This approach alternates training steps of the generative and recognition models, training one on samples produced by the other. Recent work has used wake-sleep for visual program induction [74, 42]. If one trains the generative model and the inference model to convergence before switching to training the other, this is equivalent to expectation maximization (viewed as alternating maximization [147]).

Self-Training Traditionally, self-training has been employed in weakly-supervised learning paradigms to increase the predictive accuracy of simple classification models [183, 232, 135]. Recently, renewed interest in self-training-inspired data augmentation approaches have demonstrated empirical performance improvements for neural models in domains such as large-scale image classification, machine translation, and speech recognition [246, 70, 100]. But while self-training has been shown to yield practical gains for some domains, for others it can actually lead to worse performance, as training on too many incorrect pseudo-labels can cause learning to degrade [17, 194]. For self-training within the PLAD framework, the assigned pseudo-label for each example changes during fine-tuning whenever the inference model discovers a program that better explains the input shape; similar techniques have been proposed for learning programs that perform semantic parsing under the view of iterative maximum likelihood [124]. To our knowledge, self-training has not been applied for fine-tuning visual program inference models, likely because it is somewhat unintuitive to view a program as a "label" for a visual datum.

4.2 Method

In this section, we describe the PLAD framework: a conceptual grouping of fine-tuning methods for shape program inference models. Our formulation assumes three inputs: a training dataset of shapes from the distribution of interest, S^*, a program inference model, $p(\mathbf{z}|\mathbf{x})$, and a program executor that converts programs

Input: $(S^*, p(\mathbf{z}|\mathbf{x}), E)$
Output: $p(\mathbf{z}|\mathbf{x})$ fine-tuned on S^*
  $P^{\text{BEST}} \leftarrow \{\}$
  for Number of Rounds **do**
    // Update Best Programs
    $P^{\text{BEST}} \leftarrow$
    inferProgs$(p(\mathbf{z}|\mathbf{x}), S^*, P^{\text{BEST}})$
    // Create Training Data
    if Self-Train **then**
      $\mathbf{Z} \leftarrow P^{\text{BEST}}$
      $\mathbf{X} \leftarrow S^*$
    else if LEST **then**
      $\mathbf{Z} \leftarrow P^{\text{BEST}}$
      $\mathbf{X} \leftarrow \{E(\mathbf{z}) \mid \mathbf{z} \in \mathbf{Z}\}$
    else if Wake-Sleep **then**
      $p(\mathbf{z}) \leftarrow$ trainGenerative(P^{BEST})
      $\mathbf{Z} \leftarrow$ sample$(p(\mathbf{z}), |S^*|)$
      $\mathbf{X} \leftarrow \{E(\mathbf{z}) \mid \mathbf{z} \in \mathbf{Z}\}$
    end if
    // Train inference model
    $p(\mathbf{z}|\mathbf{x}) \leftarrow$ trainMLE$(\mathbf{X}, \mathbf{Z})$
  end for

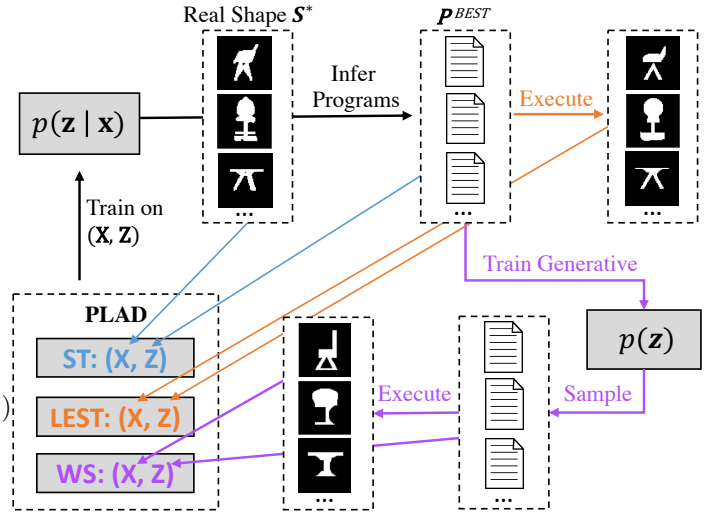

Figure 4.1: *(Left)* Pseudocode for fine-tuning shape program inference models, $p(\mathbf{z}|\mathbf{x})$, towards a shape distribution of interest, S^*, with Pseudo-Labels and Approximate Distributions (PLAD). PLAD methods iterate through three steps: infer programs for S^* with $p(\mathbf{z}|\mathbf{x})$, create a dataset of $(\mathbf{X}, \mathbf{Z})$ shape-program pairs, and train $p(\mathbf{z}|\mathbf{x})$ on batches from $(\mathbf{X}, \mathbf{Z})$. Self-training, latent execution self-training, and wake-sleep differ in how $(\mathbf{X}, \mathbf{Z})$ is constructed. *(Right)* A visual illustration of the algorithm's dataflow.

into shapes, E. Throughout this chapter, we assume that the $p(\mathbf{z}|\mathbf{x})$ passed as input has undergone supervised pretraining on a distribution of synthetically generated shapes. Methods within the PLAD framework return a fine-tuned $p(\mathbf{z}|\mathbf{x})$ specialized to the distribution of interest from which S^* was sampled.

We depict the PLAD procedure both algorithimcally and pictorially in Figure 4.1. To fine-tune $p(\mathbf{z}|\mathbf{x})$ towards S^*, PLAD methods iterate through the following steps: (1) use $p(\mathbf{z}|\mathbf{x})$ to find visually similar programs to S^*, (2) construct a dataset of shape and program pairs $(\mathbf{X}, \mathbf{Z})$ using the inferred programs, and (3) fine-tune $p(\mathbf{z}|\mathbf{x})$ with maximum likelihood estimation updates on batches from $(\mathbf{X}, \mathbf{Z})$. Through successive iterations, these steps bootstrap one another, forming a virtuous cycle: improvements to $p(\mathbf{z}|\mathbf{x})$ create $(\mathbf{X}, \mathbf{Z})$ pairs that more closely match the statistics of S^*, and training on better $(\mathbf{X}, \mathbf{Z})$ pairs specializes $p(\mathbf{z}|\mathbf{x})$ to S^*.

Methods that fall within the PLAD framework differ in how the paired $(\mathbf{X}, \mathbf{Z})$ data is created within each round. We detail this process for wake-sleep (Section 4.2.1) , self-training (Section 4.2.2), and latent

execution self-training (Section 4.2.3). In Section 4.2.4 we explain our program inference procedure (inferProgs, Fig 4.1). Finally, in Section 4.2.5 we discuss how a single $p(\mathbf{z}|\mathbf{x})$ can be fine-tuned by multiple PLAD methods.

4.2.1 Wake-Sleep (X, Z) Construction

Wake-sleep uses a generative model, $p(\mathbf{z})$ to construct $(\mathbf{X}, \mathbf{Z})$. In our implementation, we choose to model $p(\mathbf{z})$ as a variational auto-encoder (VAE) [107], where the encoder consumes visual data and the decoder outputs a program. To create data for $p(\mathbf{z})$, we take the current best programs discovered for S^*, P^{BEST}, and execute each program to form a set of shapes $\mathbf{X}^G$. $p(\mathbf{z})$ is then trained on pairs from $(\mathbf{X}^G, P^{\text{BEST}})$ in the typical VAE framework. Note that the design space for $p(\mathbf{z})$ is quite flexible, for instance, $p(\mathbf{z})$ can trained without access to $\mathbf{X}^G$ if implemented with a program encoder.

Once $p(\mathbf{z})$ has converged, we use it to sample $|S^*|$ programs by decoding normally distributed random vectors. This set of programs becomes $\mathbf{Z}$, and $\mathbf{X}$ is formed by executing each program in $\mathbf{Z}$. In this set of $(\mathbf{X}, \mathbf{Z})$ programs, $\mathbf{Z}$ is always the correct label for $\mathbf{X}$, so the gradient estimates during $p(\mathbf{z}|\mathbf{x})$ training will be low-variance and unbiased. However, $\mathbf{X}$ is not guaranteed to be close to S^*, it is only an approximate distribution. Note though, that as P^{BEST} better approximates S^*, the distributional mismatch should become smaller, as long as the generative model has enough capacity to properly model $p(\mathbf{z})$.

4.2.2 Self-Training (X, Z) Construction

Self-training constructs $(\mathbf{X}, \mathbf{Z})$ by assigning labels from the current best program set, P^{BEST}, to shape instances from S^*. Formally, $\mathbf{X} \leftarrow S^*$ and $\mathbf{Z} \leftarrow P^{\text{BEST}}$. This framing maintains the nice property that $\mathbf{X} = S^*$, so there will never be distributional mismatch between these two sets. The downside is that unless programs from P^{BEST} exactly recreate their paired shapes from S^* when executed, we know that the pseudo-labels from P^{BEST} are 'incorrect'. From this perspective, we can consider gradient estimates that come from such $(\mathbf{X}, \mathbf{Z})$ pairs to be biased. However, as we will show experimentally, when X forms a good approximation to S^*, sourcing gradient estimates from these pseudo-labels leads to strong reconstruction performance.

4.2.3 LEST (X, Z) Construction

A unique property of shape program inference is that the distribution $p(\mathbf{x}|\mathbf{z})$ is readily available in the form of the program executor E. We leverage this property to propose LEST, a variant of self-training that does

not create mismatch between pseudo-labels and their associated visual data. Similar to the self-training paradigm, LEST first constructs $\mathbf{Z}$ as the current best program set, P^{BEST}. Then, differing from self-training, LEST constructs $\mathbf{X}$ as the executed version of each program in $\mathbf{Z}$. By construction, the labels in $\mathbf{Z}$ will now be correct for their paired shapes in $\mathbf{X}$. The downside is that, like wake-sleep, LEST may introduce a distributional mismatch between $\mathbf{X}$ and S^*. But once again, as P^{BEST} better approximates S^*, the mismatch between the two distributions will decrease.

4.2.4 Inferring Programs with $p(z|x)$

During each round of fine-tuning, PLAD methods rely on $p(\mathbf{z}|\mathbf{x})$ to infer programs that approximate S^*. We propose to train PLAD methods such that the best matching inferred programs for S^* are maintained across rounds. Specifically, we construct a data structure P^{BEST} that maintains a program for each training shape in S^*. In this way, as more iterations are run, P^{BEST} always forms a closer approximation to S^*. There is an alternative framing, where P^{BEST} is reset each epoch, but we show experimental results in the supplemental material that this can lead to worse generalization.

To update P^{BEST} each round, we employ an inner-loop search procedure. For each shape in S^*, $p(\mathbf{z}|\mathbf{x})$ suggests high-likelihood programs, and the P^{BEST} entry is updated to keep the program whose execution obtains the highest similarity to the input shape; the specific similarity metric varies by domain. While there are many ways to structure this inner-loop search, we choose beam-search, as we find it offers a good trade-off between speed and performance. Experimentally, we demonstrate that PLAD methods are capable of performing well even as the time spent on inner-loop search is varied (Section 4.3.4).

4.2.5 Training $p(z|x)$ with multiple PLAD methods

As detailed in the preceding sections, the main difference between PLAD approaches is in how they construct the $(\mathbf{X}, \mathbf{Z})$ dataset used for fine-tuning $p(\mathbf{z}|\mathbf{x})$. However, there is no strict requirement that these different $(\mathbf{X}, \mathbf{Z})$ distributions be kept separate. We explore how $p(\mathbf{z}|\mathbf{x})$ behaves under fine-tuning from multiple PLAD methods, such as combining LEST and self-training. We implement these mixtures on a per-batch basis. Before $p(\mathbf{z}|\mathbf{x})$ training, we construct distinct $(\mathbf{X}, \mathbf{Z})$ distributions for each method in the combination. Then, during training, each batch is randomly sampled from one of the $(\mathbf{X}, \mathbf{Z})$ distributions. We experimentally validate the effectiveness of this approach in the next section.

Method	2D CSG CD ⇓	3D CSG IoU ⇑	ShapeAssembly IoU ⇑
Supervised Pretraining (SP)	1.580	41.0	37.6
REINFORCE (RL)	1.097	53.4	50.8
Wake-Sleep (WS)	1.118	67.4	57.2
Self-Training (ST)	0.841	67.3	61.3
LEST	0.976	69.8	56.5
LEST+ST	0.829	70.8	66.0
LEST+ST+WS	**0.811**	**74.3**	**66.4**

Table 4.2: Test-set reconstruction performance across multiple shape program inference domains. The top row contains results for the pretrained $p(\mathbf{z}|\mathbf{x})$ model fine-tuned by the other methods. For 2D CSG the metric is Chamfer distance (CD, lower is better). For 3D CSG and ShapeAssembly the metric is intersection over union (IoU, higher is better). Individual PLAD methods outperform RL, and combining PLAD methods achieves the best performance across all domains (LEST+ST+WS).

4.3 Results

We evaluate a series of methods on their ability to fine-tune shape program inference models across multiple domains. We describe the different domains in Section 4.3.1 and details of our experimental design in Section 4.3.2. In Section 4.3.3, we compare the reconstruction accuracy of each method, and study how they are affected by varying the time spent on inner-loop search (Section 4.3.4) and the size of the training set (Section 4.3.5). Finally, we explore the convergence speed of each method in Section 4.3.6.

4.3.1 Shape Program Domains

We run experiments across three shape program domains: 2D Constructive Solid Geometry (CSG), 3D CSG, and ShapeAssembly. Details can be found in the supplemental.

In CSG, shapes are created by declaring parametric primitives (e.g. circles, boxes) and combining them with Boolean operations (union, intersection, difference). CSG inference is non-trivial: as CSG uses non-additive operations (intersection, difference), inferring a CSG program does not simply reduce to primitive detection. For 2D CSG, we follow the grammar defined by CSGNet [187], using 400 shape tokens that correspond to randomly placed circles, triangles and rectangles on a 64 x 64 grid. For 3D CSG, we use a grammar that has individual tokens for defining primitives (ellipsoids and cuboids), setting primitive attributes (position and scales), and the three Boolean operators. Attributes are discretized into 32 bins.

ShapeAssembly is designed for specifying the part structure of manufactured 3D objects. It creates objects by declaring cuboid part geometries and assembling those parts together via attachment and symmetry

operators. Our grammar contains tokens for each command type and parameter value; to handle continuous values, we discretize them into 32 bins.

4.3.2 Experimental Design

Fine-Tuning Methods We compare the ability of the following training schemes to fine-tune a model on a specific domain of interest:

- **SP:** $p(\mathbf{z}|\mathbf{x})$ with supervised pretraining.
- **RL:** Fine-tuning with REINFORCE.
- **WS:** Fine-tuning with wake-sleep.
- **ST:** Fine-tuning with self-training.
- **LEST:** Fine-tuning with latent execution self-training.
- **LEST+ST:** combining LEST and ST.
- **LEST+ST+WS:** combining LEST, ST and WS.

Shape Datasets Fine-tuning methods learn to specialize $p(\mathbf{z}|\mathbf{x})$ against a distribution of real shapes S^*. For each domain, we construct a dataset of shapes S^*, and split it into train, validation, and test sets. We perform early-stopping with respect to the validation set. For 2DCSG, we use the CAD dataset from CSGNet [187], which consists of front and side views of chairs, desks, and lamps from the Trimble 3D warehouse. We split the dataset into $10K$ shapes for training, $3K$ shapes for validation, and $3K$ shapes for testing. For 3D CSG and ShapeAssembly, we use CAD shapes from the chair, table, couches, and benches categories of ShapeNet; voxelizations are provided by [25]. We split the dataset into $10K$ shapes for training, $1K$ shapes for validation, and $1K$ shapes for testing.

Model Architectures For all experiments, we model $p(\mathbf{z}|\mathbf{x})$ in an encoder-decoder framework, although the particular architectures vary by domain. In all cases, the encoder is a CNN that converts visual data into a latent variable, and the decoder is an auto-regressive model that decodes the latent variables into a sequence of tokens. For 2D CSG, we use the same $p(\mathbf{z}|\mathbf{x})$ architecture as CSGNet. A CNN consumes 64×64 binary mask shape images to produce a latent code that initializes a GRU-based recurrent decoder. For 3D CSG and ShapeAssembly, we use a 3D CNN that consumes $32 \times 32 \times 32$ occupancy voxels. This CNN outputs a latent code that is attended to by a Transformer decoder network [209]; this network also attends over token sequences in a typical auto-regressive fashion.

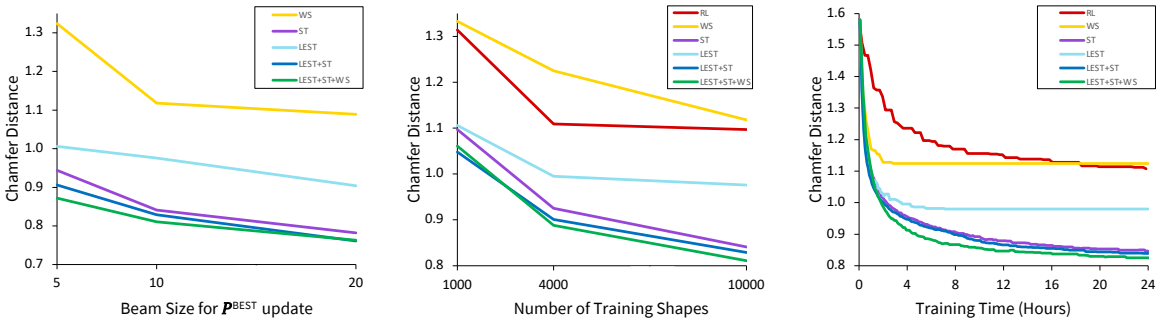

Figure 4.2: Experiments exploring properties of PLAD methods on 2D CSG. On the X-axis we plot the beam size used during the P^{BEST} update *(Left)*, the number of training shapes *(Middle)*, and the training time *(Right)*. The Y-axis of each plot measures reconstruction accuracy on test-set shapes.

Supervised Pretraining Before fine-tuning, $p(\mathbf{z}|\mathbf{x})$ undergoes supervised pretraining on synthetically generated programs until it has converged on that set. For 2D CSG, we follow CSGNet's approach. For 3D CSG, we construct valid programs by (i) sampling a set of primitives within the allotted grid (ii) identifying potential overlaps (iii) constructing a binary tree of boolean operations using these overlaps. For ShapeAssembly, we propose programs by sampling random grammar expansions according to the language's typing system. We then employ a validation step where a program is rejected if any of its part are not the sole occupying part of at least 8 voxels (to discourage excessive part overlaps). For 3D CSG and ShapeAssembly we sample 2 million synthetic programs and train until convergence on a validation set of 1000 programs. Full details provided in the supplemental.

4.3.3 Reconstruction Accuracy

We evaluate the performance of each fine-tuning method according to reconstruction accuracy: how closely the output of a shape program matches the input shape from which it was inferred, on a held out set of test shapes. The specific metric varies by domain. For 2D CSG, we follow CSGNet and use Chamfer Distance (CD), where lower distances indicate more similar shapes. For 3D CSG and ShapeAssembly, we use volumetric intersection over union (IoU).

For each domain, we run each fine-tuning method to convergence, starting with the same $p(\mathbf{z}|\mathbf{x})$ model that has undergone supervised pretraining. The reward for RL models follows the similarity metric in each domain: CD for 2D CSG; IoU for 3D CSG and ShapeAssembly. For PLAD fine-tuning methods, the similiarity metric in each domain determines which program is kept during updates to P^{BEST}. At inference time, when evaluating the reconstruction performance of each $p(\mathbf{z}|\mathbf{x})$, we employ a beam search procedure,

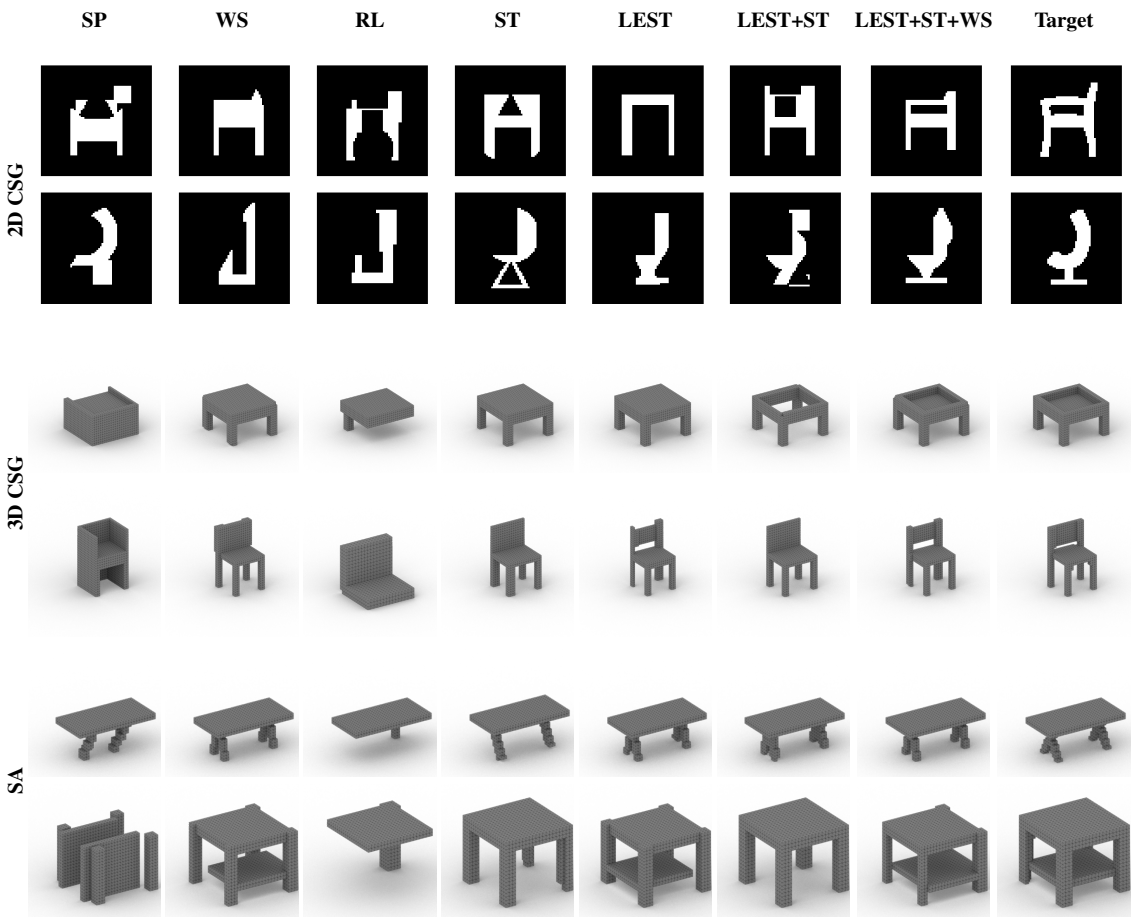

Figure 4.3: Qualitative comparisons of shape programs inferred for test-set shapes made by different fine-tuning methods for 2D CSG *(Top)*, 3D CSG *(Middle)*, and ShapeAssembly *(Bottom)*. We provide additional qualitative results in the supplemental.

decoding multiple programs in parallel, and choosing the program that achieves the highest similarity to the target shape. We use a beam size of 10, unless otherwise stated.

We present quantitative results in Table 4.2. Looking at the middle four rows, the two self-training variants (ST and LEST) outperform RL as a fine-tuning method in all the domains we studied. The wake-sleep variant (WS) also outperforms RL for both 3D CSG and ShapeAssembly. These are more challenging domains with larger token spaces, posing difficulties for policy gradient fine-tuning. As demonstrated by the last two rows, further improvement can be had by combining multiple methods: for each domain, the best performance is achieved by LEST+ST+WS. In fact, for 2DCSG, the test-set reconstruction accuracy achieved by LEST+ST+WS (0.811) outperforms previous state-of-the-art results, CSGNet (1.14) [187] and CSGNetStack (1.02) [188], for paradigms where the executor is treated as a black-box.

Mixing updates from multiple PLAD methods is beneficial because, in this joint paradigm, each method can cover the other's weaknesses. For instance, employing ST ensures that some samples of $\mathbf{X}$ are sourced from S^*, and employing LEST ensures that some samples of $\mathbf{X}$ have paired $\mathbf{Z}$ programs which are exact labels. We present qualitative results in Figure 4.3. The reconstructions from the PLAD combination methods better reflect the input shapes, reinforcing the quantitative trends.

4.3.4 Inner-loop Search Time

PLAD methods make use of P^{BEST} to generate $(\mathbf{X}, \mathbf{Z})$ datasets that train $p(\mathbf{z}|\mathbf{x})$. To study how time spent on inner-loop search affects each technique, we ran an experiment using different beam sizes to update P^{BEST}. We present results in Figure 4.2, left. On the X-axis we plot beam size; on the Y-axis we plot test-set reconstruction Chamfer distance. Unsurprisingly, spending more time on the inner-loop search leads to better performance; finding better programs for training shapes improves test time generalization. That said, across all beam sizes, we find that it is always best to train under a combination of PLAD methods; the LEST+ST and the LEST+ST+WS variants always outperform any individual fine-tuning scheme. Note, RL is not included in this experiment because REINFORCE, as defined, has no inner-loop search mechanism. In this way, PLAD provides an additional control lever, where time spent on inner-loop search modulates a trade-off between convergence speed and test-set reconstruction performance.

4.3.5 Number of Training Shapes from S^*

All the fine-tuning methods make use of a training distribution of shapes that are sampled from S^*. For some domains, the size of available samples from S^* may be limited. We run an experiment on 2D CSG to see how different fine-tuning methods are affected by training data size. We present the results of this experiment in Figure 4.2, middle. We plot the number of training shapes on the X-axis and test set reconstruction accuracy on the Y-axis. All fine-tuning methods improve as the training size of S^* increases, but once again, combining multiple PLAD methods leads to the best performance in all regimes. This study also demonstrates the sample efficiency of PLAD combinations: LEST+ST and LEST+ST+WS trained on 1,000 shapes achieve better test set generalization than RL trained on 10,000 shapes.

4.3.6 Convergence Speed

Beyond reconstruction accuracy, we are also interested in the convergence properties of a fine-tuning method. Policy gradient RL is notoriously unstable and slow to converge, which is undesirable. For 2D CSG, we record the convergence speed of each method and present these results in Figure 4.2, right. We plot reconstruction accuracy (Y-axis) as a function of training wall-clock time (X-axis); all timing information was collected on a machine with a GeForce RTX 2080 Ti GPU and an Intel i9-9900K CPU. All PLAD techniques converge faster than policy gradient RL. For instance, RL took 36 hours to reach its converged test-set CD of 1.097, while LEST matched this performance at 1.1 hours (32x faster) and LEST+ST matched this performance at 0.85 hours (42x faster).

4.4 Discussion

We presented the PLAD framework to group a family of techniques for fine-tuning shape program inference models with Pseudo-Labels and Approximate Distributions. Within this framework, we proposed LEST: a self-training variant that creates a shape distribution $\mathbf{X}$ approximating the real distribution S^* by executing inferred latent programs. Experiments on 2D CSG, 3D CSG, and ShapeAssembly demonstrate that PLAD methods achieve better reconstruction accuracy and converge faster than policy gradient RL, the current standard approach for black-box fine-tuning. Finally, we found that combining updates from multiple PLAD methods outperforms any individual technique.

Limitations While fine-tuning $p(\mathbf{z}|\mathbf{x})$, PLAD methods construct $(\mathbf{X}, \mathbf{Z})$ sets approximating the statistics of S^*, specializing $p(\mathbf{z}|\mathbf{x})$ towards S^*. As a consequence, $p(\mathbf{z}|\mathbf{x})$ may not generalize as well to shapes outside of S^*; we explore this phenomenon in B.5. Training a general-purpose inference model for all shapes expressible under the grammar is an interesting line of future work. While our work focuses on reconstruction quality, producing programs with 'good' structure matters just as much, if the program is to be used for editing tasks. Currently, the synthetic pretraining data is the only place where knowledge about what constitutes "good program structure" can be injected. Such knowledge must be expressed in procedural form, which may be harder to elicit from domain experts than declarative knowledge (i.e. "a good program has these properties" vs. "this is how you write a good program").

Chapter 5

Learning to Edit Visual Programs with Self-Supervision

People seldom write code with a linear workflow. The process of authoring code often involves substantial trial-and-error: possibly correct programs are evaluated through execution to see if they raise exceptions or break input-output assumptions. When an error is identified, an edit is made, and this process is repeated. It is difficult to imagine writing any moderately complex program in a *one-shot* paradigm, without being able to debug intermediate program versions.

In this chapter, we present a model that learns how to edit visual programs in a goal-directed manner for the task of visual program induction (VPI). Our network consumes a complete input program, this program's executed state, and a visual target. It then proposes a local edit operation that modifies the input program to better match the target. In contrast with *one-shot* approaches, this framing allows our network to explicitly reason over a complete program and its execution, in order to decide how this program should be modified.

We train our network without access to any ground-truth program annotations. To accomplish this, we propose an integration of our edit network with the *one-shot* VPI models produced by PLAD (Chapter 4). During iterative finetuning rounds, we source paired training data for our edit network by first constructing pairs of start and end programs, and then using a domain-aware algorithm to find a set of edit operations that would bring about this transformation. This process jointly finetunes both our edit network and a *one-shot* network, and we propose an integrated inference algorithm that leverages the strengths of both of these paradigms: the *one-shot* model produces rough estimates that are refined with the edit network. We find

that this joint self-supervised learning set-up forms a virtuous cycle: the *one-shot* model provides a good initialization state for the edit network, and the edit network improves inner-loop inference, creating better bootstrapped training data for the *one-shot* model.

We experimentally compare the effectiveness of integrating our edit network into this joint paradigm against using *one-shot* models alone. Controlling for equal inference time, over multiple visual programming domains, we find that using the edit network improves reconstruction performance. Moreover, we find that the reconstruction gap between these two paradigms widens as more time is spent on test-time program search. Further, we demonstrate our method performs remarkably well even with very limited data, as learning how to edit is an inherently more local task compared with learning how to author a complete program. Finally, we run an ablation study to understand and justify our system design.

We release code for our experiments at: https://github.com/rkjones4/VPI-Edit

5.1 Method

In this section, we present our approach for learning how to edit visual programs. First we formalize our task of unsupervised visual program induction. For a particular domain, we are given a domain-specific language (DSL) L and an executor E that converts programs z from L into visual outputs x. Given visual inputs from a target visual dataset that lacks program annotations, $x^* \in X^*$, our goal is to find find $z^* \in L$, such that $E(z^*) \sim x^*$. This measure of similarity is usually checked under a domain specific reconstruction metric M.

A general approach employed by prior visual program induction works is to use an autoregressive model (e.g. a Transformer) that is conditioned on a visual encoding to predict a well-reconstructing program: $p(z|x)$. These *one-shot* models iteratively predict the next program token until the program is complete. We present a framework that employs a similar autoregressive model, but instead of predicting a complete program from scratch, we instead predict a local edit that modifies an input program. In the rest of this section, we first present how we design our edit network (Sec. 5.1.1). Then we discuss our unsupervised training procedure where we jointly finetune an edit network along with a *one-shot* network (Sec. 5.1.2. Finally, we describe how we combine these networks to search for visual programs (Sec. 5.1.3).

5.1.1 Edit Network Design

Our edit network $p(e|z, x)$ learns how to predict a local edit operation that improves an input program towards a visual target (see Figure 5.1). We provide our network with a triplet input state: the tokens of an input

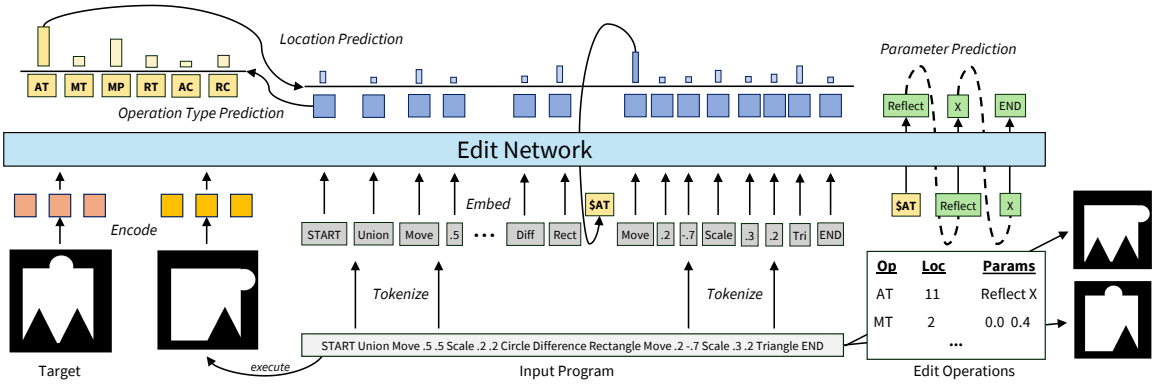

Figure 5.1: We design a network that learns how to locally edit an input program towards a target. It first predicts what type of edit operation should be applied, then it predicts where that edit operation should be applied, and finally it autoregressively samples any parameters the edit operation requires.

program z, this program's executed output $E(z)$, and a visual target x. From this state, our network is tasked with predicting an edit operation e that could be applied to the input program.

Edit Operations. There are many ways to parameterize the space of possible program edits. We choose to constrain the possible edit operations our network can produce by forcing it to select from a set of local editing operations designed for visual programs. For instance, for functional visual programming DSLs with transformation and combinator functions, we allow for seven different edit operations: modifying a transform's parameters (*MP*), modifying a transform (*MT*), adding a transform (*AT*), removing a transform (*RT*), modifying a combinator (*MC*), removing a combinator (*RC*), or adding a combinator (*AC*). We provide more details in Appendix C.4. Some of these edit operations do not take in parameters (removing a transform) while others require new parameters (e.g. to modify the parameters of a transform we need to know the new parameters). Each of these edit operations can be applied to a program at a specific token location, and results in a local change. Subsequently, we task our edit network with predicting three items: an edit operation type, a location for that edit operation, and any extra parameters that operation requires.

We design our system with this somewhat constrained edit operation set as it has a number of advantages. First, the application and effect of each edit operation is local; this simplifies the learning task and allows us flexibility at inference time. Moreover, ensuring that edit operations are tied to the semantics of the underlying DSL helps to promote program edits that result in syntactically valid modified programs. We compare our edit operation design against alternative formulations in our experimental results (Sec. 5.2.5).

Architecture. We implement our edit network as a Transformer decoder. This network has full attention over the conditioning information: each visual input (the executed output of the input program and the target)

is encoded into a sequence of visual tokens (e.g. with a CNN) and each token of the input program is lifted with an embedding layer.

To predict the edit operation type, we take the output Transformer embedding from the first index of input program sequence. This embedding is sent through a linear layer which predicts a distribution over the possible edit operation types (yellow boxes, Fig. 5.1).

To predict the edit operation location, we consider the embeddings that the Transformer produces over the tokens of the input program. Each of these location codes is sent through a linear layer, which predicts a value for each operation type. For a chosen operation type, we then normalize these values into a probability distribution across the length of the input program sequence (dark-blue boxes, Fig. 5.1). This distribution models the likelihood of where a specific edit operation type should be applied.

Finally, we use our network to autoregressively sample any extra parameters that a chosen edit operation might require. To accomplish this, we first slightly reformat the input program by inserting a special 'sentinel token' [166] associated with the chosen edit operation in two places: (1) at the specified edit operation location and (2) at the end location of the current program (*$AT*, Fig. 5.1). This 'sentinel' tokens allows the network to know what operation is being applied to which position. Then, starting from the location of the second sentinel token, we can use the network to iteratively generate a sequence of parameter predictions with causal attention-masking, until an 'END' token is chosen (green boxes, Fig. 5.1).

Training. Given an input program, how do we know which edit operations are helpful? If we have access to not only a visual target, but also its corresponding program, we can find a set of edit operations that would transform the input program into this target. We follow this logic to source training data for our edit network: given a start program and an end program, we analytically identify a set of edit operations that would bring about this transformation with a *findEdits* function. We can then convert this set of edit operations into a large set of (input, output) pairs that our network can train on. We provide further details on this algorithm in Appendix C.4. Once we have sourced paired data, through teacher-forcing we can train our network in a supervised fashion with a cross-entropy loss on the predicted operation type, location, and each parameter token. Though we lack known programs for the target domain of interest, we next discuss a bootstrapped finetuning procedure that provides a work-around for this issue.

5.1.2 Learning Paradigm

As we operate in a paradigm where we don't have access to ground-truth programs for our target set X^*, we take inspiration from recent self-supervised approaches that employ bootstrapped finetuning for visual

Algorithm 1: Network Training

1: $p(z|x) \leftarrow$ pretrain(L)
2: $p(e|z,x) \leftarrow$ pretrain(L, $p(z|x)$)
3: $P^{\text{BEST}} \leftarrow \{\}$
4: **for** num_rounds **do**
5: $P^{\text{BEST}} \leftarrow$ Infer(X^*, $p(z|x)$, $p(e|z,x)$)
6: $p(z) \leftarrow$ trainGen(P^{BEST})
7: $P^G \leftarrow$ sample($p(z)$, $\{\}$)
8: $P^S \leftarrow$ sample($p(z|x)$, $E(P^G)$)
9: $ES \leftarrow$ findEdits(P^S, P^G)
10: $p(e|z,x) \leftarrow$ trainEdit(ES)
11: $p(z|x) \leftarrow$ trainPLAD(P^{BEST}, P^G)
12: **end for**

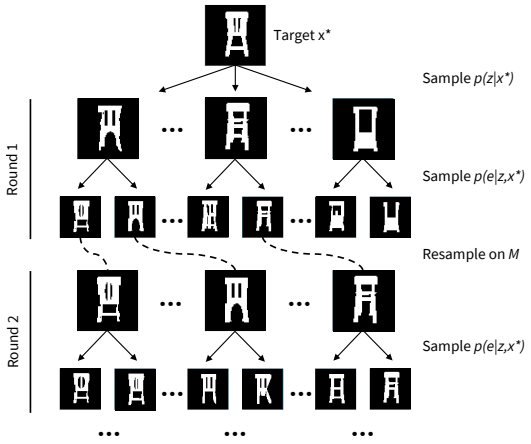

Figure 5.2: *Left:* our bootstrapping algorithm that finetunes an edit network and a *one-shot* model towards a target dataset. *Right:* our inference algorithm that initializes a population with a *one-shot* model and then mutates it towards a visual target through iterative rounds of edits and resampling.

program induction [98, 53]. Specifically, we develop an algorithm (Alg. 1) that integrates edit network training into the PLAD finetuning framework.

PLAD Finetuning. We begin with an overview of the PLAD method, which is depicted with the black text in Alg. 1 (see Chapter 4 for details). At the start of each round, the program inference network $p(z|x)$ is run over the target dataset X^*; the results of this inference procedure populate the entries of a best programs data-structure P^{BEST} according to M. Then an unconditional generative model $p(z)$ is trained over the entries of P^{BEST}, and a set of 'dreamed' programs, P^G, are sampled from this network. The weights of $p(z|x)$ are then finetuned using paired data sourced from P^{BEST} and P^G. These steps are repeated for a set number of rounds, or until convergence.

Edit Model Finetuning. The blue-colored lines in Alg. 1 indicate the modifications we make to the PLAD algorithm to incorporate our edit network. Lines 8-10 explain the training logic. First we use $p(z|x)$ to sample a set of programs P^S conditioned on the executed outputs of the generated programs P^G. Treating P^S as the starting points and P^G as the end points, we can then use our *findEdits* operation to find sets of edit operations ES that would realize these transformations. This provides us with paired data that we can use to finetune the weights of the edit network through teacher forcing, as explained in the prior section.

Synthetic Pretraining. PLAD finetuning is typically initialized with a synthetic pretraining phase (Alg. 1, line 1). During pretraining, random programs are sampled from L, and $p(z|x)$ can be trained on the paired data produced by executing these samples. Similarly, as we discuss in the results section, we find it useful to 'pretrain' the edit network on synthetic data (Alg. 1, line 2). While multiple formulations are possible here,

Table 5.1: Across multiple visual programming domains we evaluate test-set reconstruction accuracy. In all cases, we find that our joint paradigm that integrates an edit network with *one-shot* models outperforms the alternative of using only *one-shot* models.

	Layout *cIoU* ⇑	2D CSG *CD* ⇓	3D CSG *IoU* ⇑
OS Only	0.94	0.156	83.3
OS + Edit (Ours)	**0.98**	**0.111**	**85.3**

we re-use the same logic shown on lines 8-10, except we replace the set of target programs P^G with random programs sampled from L.

5.1.3 Inference Algorithm

With the above procedure we can train our edit network, but how can we use this network to find improved visual programs? This question is not only relevant at test-time, but also impacts bootstrapped training, as we run an inner-loop search to populate the entries of P^{BEST}(Alg. 1, line 5). As depicted on the right side of Figure 5.2, we design a search procedure that combines the strengths of the *one-shot* and editing paradigms. This search procedure maintains a population of programs, which are evolved over a number of rounds. The initial population is produced by sampling $p(z|x)$. Then for each round, we use the edit network to sample sets of edits for every program in the current population. We apply each of these sampled edits, and then re-sample the population for the next round according to a ranking based on M.

This formulation has a number of advantages. Instead of starting from a blank canvas, or with random samples, we allow $p(z|x)$ to produce initial rough program estimates. These guesses are then refined through mutations over a series of editing rounds that are all directed at improving similarity towards the visual target. In Section 5.2.5 we compare this algorithm against alternative formulations. Critically, by applying this joint inference procedure during finetuning we form a virtuous cycle: improving the inference strategy leads to better P^{BEST} entries, which results in better training data for $p(z|x)$ and $p(e|z, x)$, which in turn allows us to find to better P^{BEST} entries in subsequent finetuning rounds. Finally, we note that this formulation maintains a nice symmetry between $p(z|x)$ and $p(e|z, x)$: in out joint finetuning algorithm $p(e|z, x)$ trains on sequences sourced from sampling $p(z|x)$, and in this way its training distribution of edit operations well matches the population used to initialize the inference algorithm.

5.2 Results

We evaluate our edit network with experiments over multiple domains. First we describe our experimental design (Sec. 5.2.1). Then we compare the ability of different methods to accurately infer visual programs in terms of reconstruction performance (Sec. 5.2.2). We analyze how this performance changes as a function of time spent on inference (Sec. 5.2.3) or the size of the training target dataset (Sec. 5.2.4). Finally, we discuss results of an ablation study on our method in Section 5.2.5.

5.2.1 Experimental Design

We provide a high-level overview of our experimental design. See Appendix C.3 for details.

Methods. We compare our approach (*OS+Edit*) against the alternative of using only a *one-shot* model (*OS Only*). As described in Section 5.1, our approach jointly finetunes an edit network along with a *one-shot* network, and uses both of these networks to infer visual programs (Fig. 5.2). To control for the added time cost incurred by our inference procedure, we adapt a sampling-based inference loop for the *OS Only* variant, which we find results in a surprisingly strong baseline.

Domains. We consider three VPI domains (see Appendix C.2): Layout, 2D CSG, and 3D CSG. In the Layout domain, scenes are created by placing colored 2D primitives on a canvas, and optionally modifying them by changing their size, location, or forming a symmetry group. In constructive solid geometry (CSG), complex shapes are formed by combining simple shapes with boolean set operations (union, intersection, difference). Our 2D CSG and 3D CSG domains differ in terms of their primitive types (e.g. squares vs cuboids) and the parameterizations of transformation functions: generalizing notions of scaling, translating, rotating, and symmetry grouping from $\mathbb{R}^2$ to $\mathbb{R}^3$.

Network Details. For each domain, we implement $p(z|x)$ as a decoder-only Transformer [209] that conditions on a set of visual tokens and predicts up to a maximum sequence length *SL*. Similarly, we implement $p(e|z, x)$ as a Transformer with the same architecture, except that it conditions on (i) two sets of visual tokens and (ii) an input program of length *SL*, and it is only allowed to predict edit parameters up to a length of *EL*. Our visual encoders are all standard CNNs. For Layout we use a 2D CNN that takes in an RGB 64x64 image, for 2D CSG we use a 2D CNN that takes in a binary 64x64 image, and for 3D CSG we use a 3D CNN that takes in a 32^3 voxel grid.

Reconstruction Metric. The reconstruction metric M guides the inference algorithm and also performs

early stopping with respect to a validation set. For Layout we use *cIoU*, an intersection over union metric which only counts intersections on color matches (see Chapter 6). For 2D CSG we use an edge-based Chamfer distance (*CD*) [187]. For 3D CSG we use intersection over union (*IoU*).

Target Data. Like prior bootstrapping methods, our finetuning algorithm specializes our networks towards a target dataset of interest, X^*, that lacks known programs. For 2D CSG we use shapes from the dataset introduced by CSGNet [187], originally sourced from Trimble 3D warehouse. For 3D CSG we use shapes from the dataset introduced by PLAD, originally sourced from ShapeNet [16]. While we use the same test-sets as prior work (3000 / 1000 for 2D CSG / 3D CSG), we find that our method is able to offer good performance with much less training data. In our base experiments, we use 1000/100 train/val shapes for 2D CSG (from 10000 / 3000 available) and and 1000/100 train/val shapes for 3D CSG (from 10000 / 1000 available). For the Layout domain, we use the manually designed scenes sourced from the method described in Chapter 6 (1000 train / 100 val / 144 test).

5.2.2 Reconstruction Accuracy

We compare our *OS+Edit* approach against *OS Only* on each method's ability to infer visual programs that accurately reconstruct test-set inputs in Table 5.1. As demonstrated, our joint finetuning paradigm that combines an edit network with a *one-shot* network consistently improves reconstruction performance. In these experiments, we ensure that each method gets to spend the same amount of time on inference by setting search parameters so that the average inference time per shape was equal: ~ 5, ~ 10, ~ 60 seconds per shape for Layout, 2D CSG, and 3D CSG respectively. For *OS Only*, we use a sampling-based inference search where the model samples a population of complete programs for a set number of rounds. Though this approach provides a strong baseline, it was not as effective as combining our edit networks with *one-shot* initializations. In fact, for the 2D CSG domain, our formulation achieves reconstruction scores that surpass the performance of related methods that assume access to executor-gradients. On the 2D CSG test-set, we achieve a Chamfer distance (CD) of 0.111 (lower is better), whereas UCSG-Net [101] gets a CD of 0.320, SIRI [53] gets a CD of 0.260, and ROAP [201] gets a CD of 0.210 . Note that as the DSL, architecture, objective, and inference procedures differ across these various works, it's hard to make any absolute claims from this direct comparison. Nevertheless we would like to emphasize that our method's reconstruction performance on this task is very strong in the context of the related literature. We visualize reconstructions from this experiment in Figure 5.3, and find that qualitative evidence supports the quantitative trends.

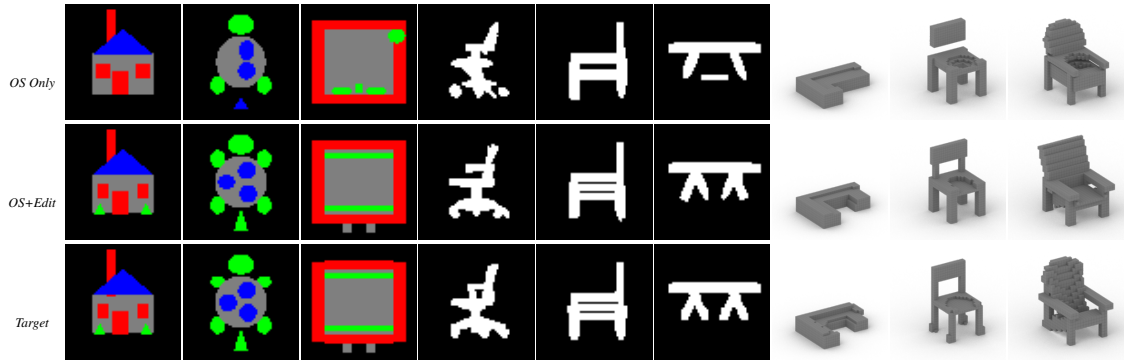

Figure 5.3: Comparing reconstructions of *one-shot* models (*top*) against our joint approach (*middle*).

5.2.3 Search Time

While *one-shot* models must author new programs from scratch without execution-feedback, our edit network has the capacity to reason over an input program, compare its execution versus the visual target, and decide how this program should be modified. As such, we hypothesize that integrating our edit network into our inference procedure will be increasingly advantageous over the *OS Only* approach as more time is spent on test-time search. To validate this hypothesis, we explore how the reconstruction gap between these paradigms changes as a function of time spent on search (Figure 5.4, *left*). For 2D CSG we take a subset of the test-set (300 shapes) and run more rounds of our inference algorithm. As demonstrated, as more time is spent on test-time search (i.e. as the number of rounds increases) the reconstruction gap between *OS Only* and *OS+Edit* grows wider. Moreover, we note that even on the first round there is a gap between the methods, as the *one-shot* network trained in the *OS+Edit* paradigm had access to better P^{BEST} entries throughout the finetuning process (i.e. the aforementioned virtuous cycle). We present qualitative results that show how the edit network evolves the population of programs towards the visual target in Figure 5.5.

5.2.4 Training with limited data

While both *OS+Edit* and *OS Only* are unsupervised in the sense that they don't have access to any ground-truth program annotations, they do require an input set of visual data to form a target training set. We hypothesize that our edit network will be especially useful for domains with limited data (even limited unannotated data) as the program editing task is inherently more local than trying to author a complete program. Consider for instance that during finetuning, in a *one-shot* paradigm each visual datum can only contribute a single training example, while in our paradigm an entire distribution of edit operations can be sourced by considering the many possible edit paths one could take to transform a start program into an end program.

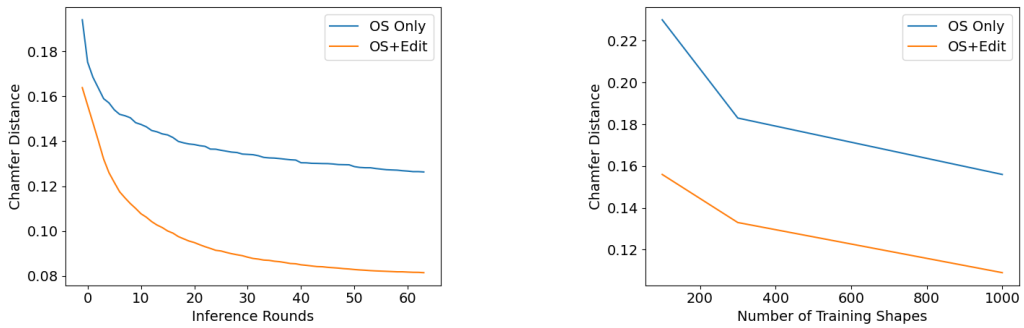

Figure 5.4: For 2D CSG, we compare reconstruction accuracy (Chamfer distance, lower is better, Y-axis) between using an edit network and using only a *one-shot* network while varying time spent on inference (*left*) and training set size (*right*).

We validate this hypothesize with an experiment where we train versions of these systems while varying the size of the target training set (Fig. 5.4, *right*). Our joint paradigm offers very strong performance even while finetuning towards an input set of just 100 training shapes, matching the performance of *OS Only* when it has 10x more data.

5.2.5 Method Ablations

We run an ablation experiment to evaluate the design of our system on the Layout domain. We present results of this experiment in Table 5.2. In the rest of this section we detail all of the alternative formulations we compare against.

Edit Operations. Our default edit networks learn how to predict local edit operations from a limited set of options. We compare this paradigm with two alternatives. In the *next program* mode, we task the edit network with predicting all of the tokens of the program that would be created by applying the target edit operation to the input program. In the *final program* mode, we task the edit network with predicting the tokens of the final program associated with the visual target. This formulation was inspired by the success of denoising diffusion models for visual synthesis tasks [77], though in our setting this variant is basically an alternative *one-shot* model with extra conditioning information but with the same target sequences. As demonstrated, neither of these approaches is as performant as our formulation where edits are predicted as local operations. Moreover, predicting an entire program is much slower compared with predicting an edit, so fewer rounds of our inference algorithm can be run with the same search time budget.

Program Corruption. We source paired training data for our edit network by constructing (start, end) program pairs and then analytically finding a set of edit operations that would complete this transformation.

For an alternative, we can look towards discrete diffusion methods [202, 242, 215, 172]. In our *corruption* variant we take inspiration from these works and design a program corruption algorithm for the Layout domain. This corruption algorithm takes an end program as input, and then samples corruption operations (i.e. inverse edit operations) that can be used as paired data for our edit network (Appendix C.5). As seen, this alternative formulation was not as performant as our default approach. One reason for this is that it hard to design a corruption process that converts end programs (e.g. P^G) into the distribution of programs that we have access to at inference time (e.g. P^S). Conversely, by applying our *findEdits* operation on P^G and P^S pairs, we can source paired data for our edit network that *does* match this distribution.

Pretraining and Finetuning. In our default version there are three training phases. First, $p(z|x)$ undergoes pretraining on synthetic data. Second, $p(e|z, x)$ undergoes pretraining on synthetic data using samples from $p(z|x)$. Then both of these networks are jointly finetuned with respect to X^*. In the *No FT* variant, we don't finetune either network, in *no one-shot FT* we don't finetune $p(z|x)$, in *no edit FT* we don't finetune $p(e|z, x)$, and in *no edit PT* we don't pretrain $p(e|z, x)$. While the performance of our system remains remarkable strong even under these ablations, we get the best results by using all three training phases. Interestingly, for settings where $p(z|x)$ is not specialized for X^*, the reconstruction accuracy gap dramatically increases between the best sample in the starting population and the best sample in the final population of our inference procedure. For instance, for the *no one-shot FT* variant, the first round cIoU score is 0.88 which gets increased to 0.972 (0.092 improvement) through the mutations proposed by the edit model, while in our default variant the first round cIoU is 0.925 (an improvement of .055).

Inference Algorithm. We compare our inference algorithm with two alternative versions. In *Naive OS* we initialize the first population with $p(z|x)$, and make edits to each population member with $p(e|z, x)$, but we skip the population resampling step according to M , and instead apply the highest likelihood edit from $p(e|z, x)$. While the edit network is still helpful in this paradigm (0.022 improvement from the first to the last round), it performs worse compared with our default implementation. In *Rand+Edit*, we remove $p(z|x)$ and instead fill the initial population with random program sampled from L. This provides a much worse initialization (0.302 cIoU in the first round), and though our edit network successfully mutates these samples towards the target, better reconstruction performance is gained by combining our edit network with initial guesses from a *one-shot* model.

Table 5.2: Ablation study comparing our method against alternative formulations.

Method	Final cIoU ⇑
Ours	**0.980**
Next program	0.941
Final program	0.920
Corruption	0.964
No FT	0.955
No one-shot FT	0.972
No edit FT	0.976
No edit PT	0.953
Naive OS	0.947
Rand+Edit	0.906

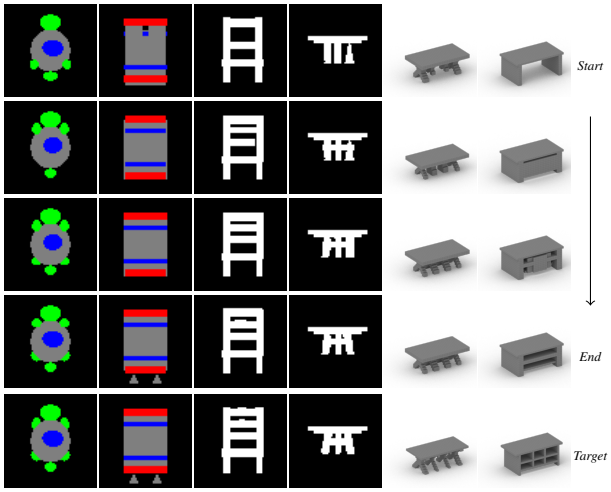

Figure 5.5: Our inference procedure edits samples from an initial population (*top*) towards a target (*bottom*).

5.3 Discussion

In this chapter, we've introduce VPI-Edit: a system that learns how to edit visual programs in a goal-directed fashion. We develop a self-supervised bootstrapping approach that allows us to train an edit network for domains that lack ground-truth program annotations. We compare our proposed paradigm, that jointly finetunes a *one-shot* model and an edit network, against the alternative of using only a *one-shot* model, and find that our approach infers more accurate program reconstructions. Further, we find this performance gap is more pronounced when more time is spent on program search or when less training data is available. Finally, we justified the design of our method with an ablation experiment.

Limitations Compared with prior work, we need to train another network, this impacts the time required for both pretraining and finetuning stages. Moreover, the full benefit of using an edit network is best realized with a more complex program search, and as such we advocate for search-time budgets that are slightly more costly compared with prior work. Though our formulation would offer improved performance for work-flows that can afford to spent more time on program search, it would be useful to consider potential speed-ups of our system [29]. Finally we note that our current formulation requires access to a domain-aware *findEdits* operation that can analytically find a set of edits that realizes a transformation from a start program to an end program.

5.3.1 Relation with SIRI

SIRI [53] is another method for visual program induction that builds off of, and improves upon, the PLAD framework introduced in Chapter 4. Like the VPI-Edit method, it treats program as structured objects than can be improved through rewriters. Instead of training a network that learns how to edit programs, it integrates domain-specific analytical code rewriting operations into the bootstrapped finetuning loop of a *one-shot* network. It considers three types of rewriting operations: parameter optimization, code pruning, and code grafting. These sub-modules are interleaved together to find rewritten program versions that improve an objective function (some combination of reconstruction and program length). While these types of rewrites are very helpful for correcting certain types of errors (e.g. better alignment by finding improved continuous valued parameters) they also can lead to rewritten programs that are 'out-of-distribution' for the *one-shot* network. As a result, SIRI only sparsely integrates these rewritten programs during network finetuning, whereas in our VPI-Edit framework the *one-shot* network can finetune on the entire set of rewritten programs without convergence issues. As the ideas of these two methods are largely complimentary, and it would be interesting to consider ways to integrate these paradigms into a single system.

5.3.2 Relation with Tree Diffusion

Tree Diffusion [102] is a contemporary approach with many similarities to the method we present in this chapter. This paper takes inspiration from execution-guided visual program synthesis works [41], and like VPI-Edit trains networks that try to edit 'complete programs' in a goal-directed fashion. This editing process is framed as a sort of discrete diffusion occurring over syntax trees of programs from context-free grammars. To get training data, the high-level idea is to sample a node in this syntax tree, and then place the node with a

same-typed newly sampled expression from the grammar. Much like our corruption ablation condition, this noising process can then be converted into supervised paired training data for an edit network. In fact, their edit network does not learn to undo the corruption directly, but instead treats the corrupted program as a *start* program, and then computes a minimal set of edits that would convert it into the *end* program. Though their version of *findEdits* is considerably more simple (as their DSLs and edit operations are more limited), this is otherwise a remarkably similar paradigm as our pretraining phase, where the only difference is in how *start* and *end* program pairs are sourced (Alg 1, L:2).

Beyond implementation/experimental details (network/domain/language design) there are two other major philosophical differences between these works. First of all, our VPI-Edit system not only pretrains an edit network with respect to some DSL, it also finetunes this network towards a target shape distribution (jointly with a *one-shot* network). On the other hand, Tree Diffusion only focuses on synthetic pretraining, and then tries to adapt to a target distribution at inference time. Secondly, different mechanisms are used to guide test-time search. VPI-Edit initializes a population of programs with the *one-shot* network, proposes mutations with an edit network, and then re-samples the next generation according to a reconstruction metric. Tree diffusion initializes a population of programs by randomly sampling a grammar, proposes mutations with an edit network, and then re-samples the next generation according to a value network's prediction. This value network predicts the edit distance (e.g. how many edit operations would be required) between two programs, and is trained on the same distribution of data as the edit network. Though well-calibrating this value network, especially to out-of-distribution samples, might prove challenging, it does offer an intriguing alternative to the more local signals provided by reconstruction based rewards, and this paradigm is certainly worthy of further investigation; see Section 10.1 for further discussion.

Chapter 6

Learning to Infer Generative Template Programs for Visual Concepts

Humans understand the visual world through concepts [143]. Concept-level reasoning allows us to perform a multitude of tasks over a range of situations, even after seeing a new concept only a few times [205]. In this Chapter, we propose an inverse procedural modeling scheme that aims to endow machines with similar abilities to learn flexible, general purpose visual concepts. For instance, to support creative applications, we would like to be able to feed it a small set of visual exemplars and have it synthesize novel generations that match the input concept. Or to support analysis tasks, our system should be able to parse the input exemplars into corresponding parts in a consistent fashion. We desire a system capable of achieving these goals across different visual domains.

Past work in the field of *concept learning* has explored systems capable of meeting some of these desiderata [116] Some attempts have proposed purely 'neural' approaches that learn to perform well on a single concept-related tasks, like classification [211, 190] or generation [48, 173, 57]. A smaller number of 'neural' approaches have investigated how to learn concept representations that support multiple tasks [40, 75]. While these methods often achieve domain-flexibility by learning from visual data directly, they are data-hungry and don't always generalize well to out-of-distribution concepts.

More relevant to the subject matter of this dissertation are concept-learners that construct structured task-general representations. To the best of our knowledge, such methods have only been successfully developed for stroke-based drawing domains. BPL [115] fits a structured hierarchical model of handwritten character

production to human stroke data under a Bayesian framing, achieving human-level performance across generative and discriminative tasks. GNS [46] extends this framework with a neurosymbolic method, where the distribution and correlation of strokes are modeled with learned networks. While these approaches demonstrate impressive performance, their design is specialized for datasets such as Omniglot; we are unaware of any successful attempts to generalize these approaches to other domains.

Working towards domain and task general concept learning, we introduce the *Template Program* framework. Our *neurosymbolic* system *learns* how to infer *programs* that capture visual concepts. This framework extends the 'single input' visual program induction method described in Chapter 4, introducing networks that learn to find procedural models that explain a collection of visual inputs. Beyond simply parsing concepts, our Template Programs can also be sampled to synthesize new generations from a particular concept.

Template Programs are structured symbolic objects from a domain-specific language that capture structural and parametric attributes common to a particular concept. They admit instantiated programs that accord with these constraints, and convert these programs into visual outputs with a domain-specific executor. We train networks that learn how to infer Template Programs with a training regime that works across visual domains. This paradigm requires only a domain-specific language (DSL) and a visual dataset (e.g. images) with concept groupings (e.g. class annotations). Our two-step learning approach first pretrains a series of inference networks on synthetic data sampled from the DSL, and then specializes these networks towards the target dataset with a bootstrapped fine-tuning procedure.

We experimentally validate that our method is capable of inferring Template Programs across multiple visual domains: 2D layouts, Omniglot characters, and 3D shapes. We demonstrate that Template Programs natively support a number of downstream applications, including few-shot generation and co-segmentation. We are unaware of any other method that is able to perform these tasks in a domain-general fashion, so we compare against either task-specific or domain-specific alternatives. With respect to task-specific approaches, we find that our neurosymbolic method achieves superior performance. For the one domain, Omniglot [115], where task-general methods have been proposed, we compare our domain-general method against domain-specific approaches and find that we are able to achieve competitive performance.

We release code for our experiments at: https://github.com/rkjones4/TemplatePrograms

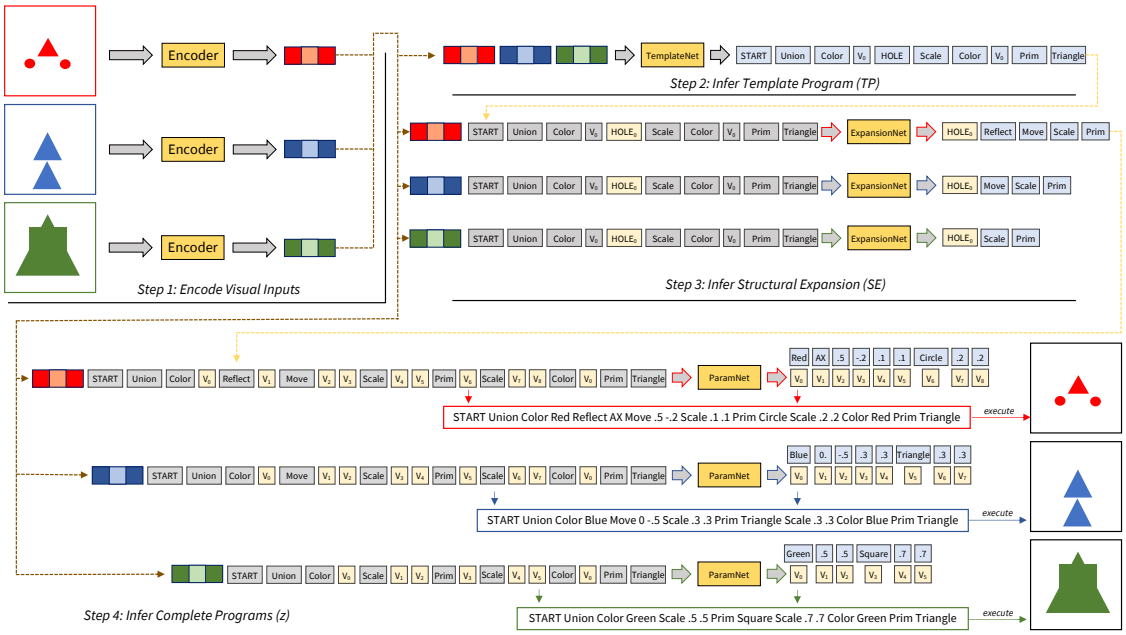

Figure 6.1: Our inference process. First, a group of visual inputs are encoded (Step 1). Next, our TemplateNet uses these encodings to infer a Template Program (TP, Step 2). The TP and each encoding are then sent to the ExpansionNet to produce a Structural Expansion (SE) for each input (Step 3), which are finally passed to the ParamNet to produce a set of complete programs that explain the inputs (Step 4).

6.1 Method

Our framework learns how to infer Template Programs (Section 6.1.1) that capture visual concepts. We describe our inference networks in Section 6.1.2 and our learning paradigm in Section 6.1.3.

6.1.1 Template Programs

Given a collection of related visual inputs, our goal is to find a symbolic structure capable of representing this group as a concept. This structure must be able to account for both (i) the shared attributes across the group and (ii) the allowable divergences that differentiate various group members.

Towards this goal, we introduce *Template Programs* to represent visual concepts. A Template Program (TP) is a partial program specification from a domain-specific language (DSL). We assume this DSL is a functional language, where each function takes other functions or parameter arguments as input. Template Programs admit fully instantiated *programs* (z). These programs can be run through a domain-specific executor (E) to produce visual outputs.

Template Programs are composed of a hierarchy of function calls (i.e an expression tree) and are optionally allowed to define relationships between parameter arguments (e.g. variable reuse). All instantiations from a Template Program must invoke the specified functions and use the described relations. To allow instantiations to vary structurally (i.e. use different functions), we introduce a special *HOLE* construct. Each *HOLE* in the Template Program can be filled in with an arbitrary expression tree. This process creates a *Structural Expansion* (SE), which completely specifies the function call sequence of an instantiation. Any function parameters that lack a specified relation in the SE are allowed to differ freely in the output programs.

6.1.2 Inference Networks

We use a learning-based approach to infer Template Programs and their instantiations. Given a group of visual inputs X^G from some concept $\widetilde{X}$, our goal is to infer a Template Program TP, such that for each x in X^G there is a program instantiation z from TP so that $E(z) = x$.

We solve this difficult inverse structured prediction problem with a series of inference networks p_{inf} that we depict in Figure 6.1. To start, each x is converted into a latent code with a domain-specific visual encoder (e.g. a 2D CNN for image inputs). These latent codes are then passed through a series of auto-regressive networks, explained below.

The TemplateNet, $p(TP|X^G)$, is responsible for inferring Template Programs. Attending over all of the latent codes from X^G as conditioning information, it autoregressively predicts a series of tokens that form the Template Program. We linearize this composition of functions with prefix notation. Using Figure 6.1 as reference, these tokens are either (i) functions from the DSL (`SCALE`), (ii) *HOLE* tokens, or (iii) parametric relations, such as static variable assignment (`Triangle`) or variable reuse (V_0).

Given the inferred TP, we use the ExpansionNet and ParamNet to instantiate a complete program z. The ExpansionNet, $p(SE|TP,x)$, conditions on TP along with a single visual input x, and autoregressively produces a SE by filling in *HOLE* tokens with a series of functions. This SE is then reformatted to expose any free parameters and their relations. The ParamNet, $p(z|SE,x)$, conditions on this representation and the same visual input x in order to autoregressively predict the value of each parameter which instantiates a complete program z.

6.1.3 Learning Paradigm

How can we train our inference networks? With ground-truth program annotations, we could employ supervised learning, but datasets with this level of annotation do not exist. As our goal is to design a domain-general framework, our problem formulation assumes the following as input: a target dataset of interest X^* and a relevant DSL. We assume that we can sample groups of visual concepts from this dataset (e.g. by using class annotations), but otherwise assume the visual data is unstructured. Under these assumptions, we employ a two-step process: we first initialize our networks by pretraining on synthetic data sampled from the DSL, and then we specialize p_{inf} towards X^* with bootstrapped finetuning.

Synthetic Pretraining We implement each autoregressive network within p_{inf} as a Transformer decoder with causal masking (where the conditioning information varies across networks). With paired (input, output) data, each of these networks can be trained with maximum likelihood updates (i.e. cross-entropy loss). We can produce (input, output) pairs for all of our networks if we have an associated (X^G, TP^G, Z^G) group, where targets for the ExpansionNet and ParamNet can be derived by comparing the TP^G to each $z \in Z^G$(further details in Appendix D.4.1).

One way to produce paired data is to generate it synthetically. Following previous VPI approaches [206, 187], we sample synthetic data from our DSL and use it to pretrain our inference networks in a supervised setting. At a high level, this sampling procedure invokes the following steps: (1) sample a full program from the DSL (e.g. by stochastically expanding the grammar), (2) convert the full program into a TP^G (e.g. by collapsing random expression trees into *HOLE* tokens and randomly assigning parameter relations), (3) sampling a group of programs Z^G from the TP^G (e.g. through random expansion) and recording their executions, $X^G = (E(z) \; \forall \; z \in Z^G)$.

Bootstrapped Finetuning While synthetic pretraining attunes p_{inf} to the DSL, it produces overly general networks that make inaccurate predictions when run over concepts from X^*. To specialize p_{inf} towards X^*, we develop an unsupervised bootstrapped finetuning approach that generalizes the PLAD framework designed for single-input, deterministic programs (Chapter 4).

Our algorithm oscillates between inference and training steps. In each inference step, we run p_{inf} over groups of visual inputs X^G drawn from concepts in the target dataset $\widetilde{X} \in X^*$. We run a beam-search to find the Template Program whose instantiations best match X^G under an objective O (Eq. 6.1). For each X^G, we record the best inferred (TP^G, Z^G) pair for use in the training step.

The training step uses this paired data to finetune p_{inf}. Specifically, we convert (X^G, TP^G, Z^G) inferred

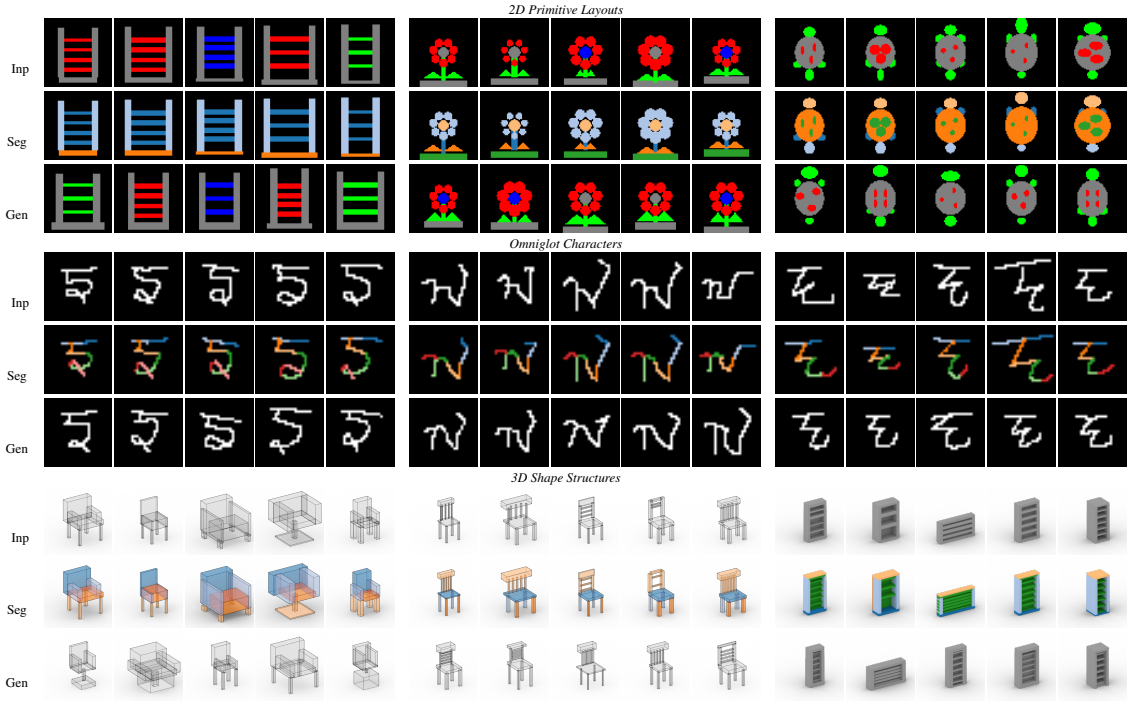

Figure 6.2: We learn to infer Template Programs that capture input concepts (*Inp*). Template Programs produce consistent concept parses (*Seg*) and synthesize new generations (*Gen*). Our framework flexibly extends across different visual domains and input representations.

groups into paired training data for p_{inf} under different self-supervised learning formulations. In the self-training (*ST*) formulation, we leave the group as is. In the latent execution self-training formulation (*LEST*), we replace X^G by executing each program in Z^G. Our wake-sleep formulation (*WS*) first trains a generative model p_{gen}(Appendix D.3.3). This model is a modified variant of p_{inf}, where the visual latent codes are masked out, so that visual information does not affect the conditioning. We train p_{gen} to model the inferred (TP^G, Z^G) data, and then we sample a collection of synthesized (TP^G, Z^G) pairings from the network. Finally, we produce an associated X^G for each generation by employing our program executor, following the same procedure as in LEST.

From these three self-supervised approaches (ST, LEST, WS), we get three distinct datasets of (X^G, TP^G, Z^G) groups. We use these datasets to finetune p_{inf}, using the same maximum likelihood updates as in our synthetic pretraining phase. We randomly sample batches from each of these datasets in a training loop until we reach convergence with respect to concepts from the validation set of X^*.

Objective Our inference procedure takes in a visual group X^G and tries to find a Template Program TP^G whose instantiations Z^G best explain the group. We formalize this notion of *best* with an objective composed of two

terms (i) reconstruction error (under a domain-specific metric M) and (ii) the description length difference between each z and the TP it originated from. Specifically, we try to minimize:

$$O = \lambda_1 * \sum_{(x,z)\in(X^G,Z^G)} M(x, E(z)) + \lambda_2 * \sum_{z\in Z^G} |z| - |TP^G| \tag{6.1}$$

In short, we search for Template Programs that encode as much commonality as possible while still producing instantiations that capture the visual input.

6.2 Results

We validate the benefits of our method through comparisons with alternative approaches across three visual domains. We describe the domains in Section 6.2.1 and our experimental design in Section 6.2.2. Next, we evaluate performance on downstream tasks: few-shot generative modeling (Section 6.2.3, Figure 6.2 *Gen* rows) and parsing-based cosegmentation (Section 6.2.4, Figure 6.2 *Seg* rows). Finally, we discuss out-of-distribution generalization, method ablations, and additional capabilities of Template Programs in Section 6.2.5.

6.2.1 Visual Domains

We experiment over three visual domains that differ in input modality and concept groupings. We provide an overview of each domain here, and further information in Appendix D.2.

2D Primitive Layouts We design a procedurally generated domain where concepts are represented with a layout of simple 2D colored primitives. In addition to functions that move, scale, and color primitives, our DSL also contains simple symmetry functions (e.g. `REFLECT`, Fig. 6.1). We hand-design 20 high-level meta-procedures that correspond with manufactured or organic concepts (e.g. cats or clocks). Each meta-procedure creates a distribution of concepts by expressing different combinations of four attributes, allowing us to produce 384 distinct concepts. We divide these into 216 training-validation concepts and 168 testing concepts, where this split is designed to investigate out-of-distribution generalization performance (Section 6.2.5).

Omniglot Characters [115] introduced the Omniglot dataset which contains handwritten characters from 50 languages. These characters are split between a background set (964 characters) and a generalization

Table 6.1: Across multiple visual domains we quantitatively evaluate few-shot generation and co-segmentation performance. Our method outperforms domain-general but task-specific alternatives, and is competitive against approaches that specialize for Omniglot.

Domain		**Omniglot**					**2D Layouts**					**3D Shapes**			
Task		*Few-shot gen*				*Co-seg*	*Few-shot gen*				*Co-seg*	*– Few-shot gen –*			*Co-seg*
	Method	**FD⇓**	**Conf⇑**	**MMD⇓**	**Cov⇑**	**mIoU⇑**	**FD⇓**	**Conf⇑**	**MMD⇓**	**Cov⇑**	**mIoU⇑**	**FD⇓**	**MMD⇓**	**Cov⇑**	**mIoU⇑**
Domain	BPL	130	57.9	9.58	**61.1**	**79.9**	-	-	-	-	-	-	-	-	-
Specific	GNS	123	55.0	9.47	58.1	73.8	-	-	-	-	-	-	-	-	-
	FSDM	196	5.17	12.6	48.6	-	-	-	-	-	-	-	-	-	-
Task	VHE	139	2.46	10.4	52.0	-	81.9	59.0	8.06	22.4	-	-	-	-	-
Specific	arVHE	137	12.3	10.2	55.8	-	45.3	77.0	6.34	45.1	-	128	8.57	53.6	-
	BAE	-	-	-	-	34.3	-	-	-	-	34.5	-	-	-	53.2
	Ours	**115**	**59.9**	**9.40**	50.7	78.7	**30.7**	**90.9**	**5.49**	**50.6**	**82.5**	**84.5**	**6.49**	**53.9**	**68.6**

set (659 characters), where each concept comes with 20 examples. We use the background characters for training and validation, and test on the generalization characters. Our DSL for drawing characters produces strokes by moving a virtual pen. The pen moves at an angle, for varying distances, optionally bowing inwards or outwards. It can be lifted up or put down and has the option to back-track to previous positions. As we are more interested in modeling stroke structure than physical handwriting dynamics, we adopt a simplified ink model compared with previous work: any pixel the pen passes through is filled completely.

3D Shape Structures Beyond 2D domains, we also run experiments on a dataset of 3D shapes. Following past work, we use a structured part-based representation, where 3D shapes are modeled as a combination of primitives (i.e. cuboids) [18, 80]. For our DSL, we use the ShapeAssembly modeling language (Chapter 3), which creates complex 3D shapes by instantiating cuboids and assembling them together through attachment and symmetry operators. We source 10,000 3D shape structures from the chair, table, and storage categories of PartNet [141], holding out 1000 of these for our test set. We use the associated structural annotations in PartNet to identify groupings of these shapes that correspond to concepts that are more fine-grained than object category. While we use annotations to partition the dataset into groups, our networks receive only a visual representation of each shape during training: either an unordered collection of primitives or a 3D voxel grid.

6.2.2 Experimental Design

Networks We implement each autoregressive component of p_{inf} with Transformer decoder models that have 8 layers, 16 heads, and a hidden dimension of 256. We use causal attention masks with a prefix that contains conditioning information (see Section 6.1.2, Appendix D.3). For the 2D layout and Omniglot domains we

model our visual encoders with 2D CNNs that respectively take in RGB images of size 64x64 and binary images of size 28x28. We train two different versions of p_{inf} for 3D shapes. When shapes are represented as an unordered collection of primitives (*primitive soup*), we use a Transformer encoder with order-invariant positional encodings (Fig. 6.2, left & middle). We additionally explore using a 3D CNN that takes in a 64^3 occupancy grid of voxels (Fig. 6.2, right). For each domain, we train p_{inf} with the procedure described in Sec. 6.1.3 until we reach convergence on the validation set (additional training details in Appendix D.4).

Inference logic We infer Template Programs and their instantiations with a beam search. This algorithm has two parameters: BM_{TP} controls the size of the beam used to find Template Programs under $p(TP|X^G)$, while BM_z controls the size of the beam used to find instantiated programs under $p(SE|TP,x)$ and $p(z|SE,x)$. This search concludes by evaluating each candidate under O, which requires a domain-specific reconstruction metric. We use a color-based IoU for 2D layouts, an edge-based Chamfer distance for Omniglot, and either a primitive-matching score or IoU for 3D shapes depending on the input format (details in Appendix D.2). During fine-tuning, we set BM_{TP} and BM_z to 5 ($\sim$1 second for inference per input group). For evaluation tasks, we set BM_{TP} to 40 and BM_z to 10 ($\sim$20 seconds for inference per input group).

Comparison Conditions We compare how our method performs on concept-related tasks against alternative approaches. For the Omniglot domain, we compare against the task-general but domain-specific *BPL* [115] and *GNS* [46] methods. Though they are designed to operate under one-shot paradigms, we adapt them for our task settings. We also compare against alternatives that are domain-general but task-specific. For few-shot generation, we compare against *FSDM* [57] and *VHE* [75]. These approaches both train deep generative networks that condition on a group of input images but use different generative models: VHE uses a VAE [107], while FSDM uses diffusion [77]). During our experiments, we found VAE training to be highly unstable, so we also introduced an autoregressive VHE variant: *arVHE*. Our arVHE model first tokenizes visual data (e.g. through vector-quantization [208]) then learns an autoregressive model over this tokenization that is conditioned on groups of visual inputs. For co-segmentation tasks, we compare against *BAE-NET* [25]. BAE-NET forms consistent parses by training a parameter-constrained implicit network to solve an occupancy reconstruction task. Though this method is designed primarily for 2D and 3D shapes, we adapt it to create segmentations across all of our domains. We provide additional details for all of our comparisons conditions in Appendix D.6.

6.2.3 Concept Few-shot generation

For few-shot generation, a method is given a set of examples from a concept as input and is tasked with producing new instances that demonstrate variety while maintaining concept membership. Our method accomplishes this with a two step process: first we infer a Template Program that explains the input group, then we sample new instantiations from the Template Program. To sample these instantiations, we use variants of our $p(SE|TP,x)$ and $p(z|SE,x)$ that condition on a mean-pooled visual encoding of the input group (Appendix D.3.3). Across our three domains, we show examples of our method's few-shot generative capabilities in Figure 6.2, *Gen* rows. Our method is able to capture input concepts and synthesize new outputs that demonstrate interesting variations while preserving concept identity.

We present quantitative few-shot generation results in Table 6.1 (details in App. D.5). For each domain and test-set concept, we provide every method with a group of 5 visual inputs and ask it to synthesize 5 generations. Comparing these generations to a reference set of held-out examples from the same concept, we compute the following metrics using the latent space of a domain-specific auto-encoder: Frechet Distance (*FD*), Minimum Matching Distance (*MMD*), and Coverage (*Cov*). For the Omniglot and 2D layout domains, we also report class confidence (*Conf*), the average predicted probability of each generation being a member of the target class under a classifier trained on all domain concepts.

As demonstrated, our method vastly outperforms task-specific alternatives (FSDM, VHE, arVHE) for few-shot generation. Over all domains, we find that our method scores much better along metrics that measure output concept consistency (*Conf*) and fidelity to the reference set (*FD*, *MMD*), while maintaining reasonable output variability (*Cov*). Moreover, our domain-general method is able to largely match, and even somewhat outperform, domain-specialized approaches (BPL, GNS) along measurements of concept consistency and fidelity to the reference set.

We visualize few-shot generation results for Omniglot characters in Figure 6.3. While we again offer much improved performance over the task-specific alternatives, we note that the methods that specialize for Omniglot typically demonstrate a wider range of output variability, which confirms the trend we observe with the *Cov* metric. We hypothesize this difference is due to BPL and GNS learning priors over human stroke patterns (learning how people typically produce characters). In contrast, our method finds a Template Program attuned to the visual data present in the input group without regard for structured priors beyond the input DSL.

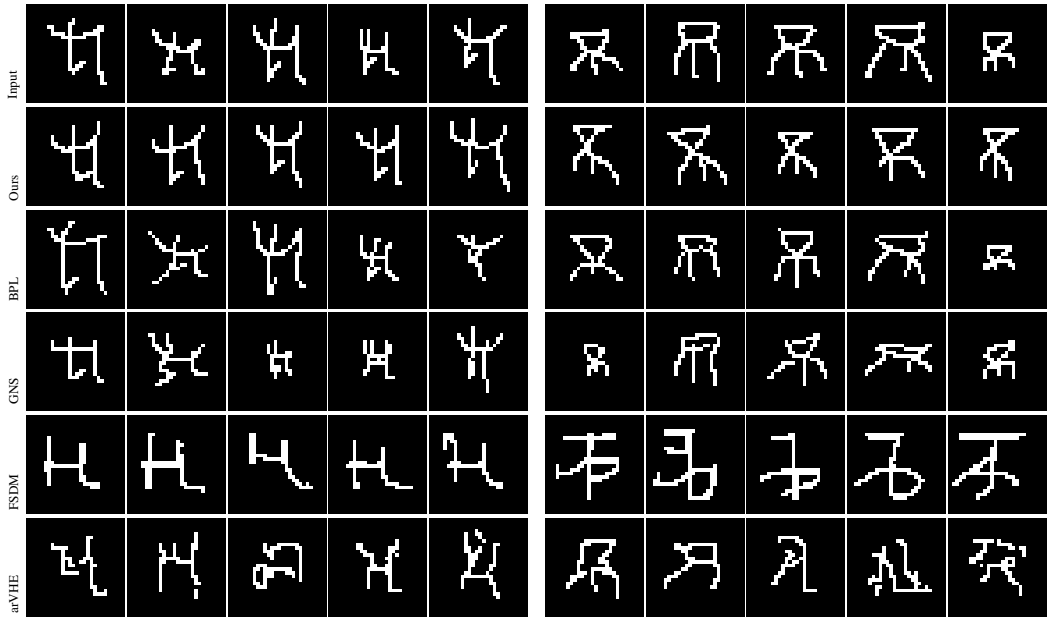

Figure 6.3: Comparing few-shot generations of Omniglot characters.

Perceptual Study To further investigate few-shot generative performance, we designed a two-alternative forced-choice perceptual study (Appendix D.5.2). We recruited 20 participants, and presented a series of questions that compared generations from competing methods to the input group. We report results for this study in Table 6.2. For the Omniglot domain, we compared our method against our best performing task-general method (arVHE) and the domain-specific GNS method. We additionally compared our method against arVHE for the shape domain. We observed that there was an overwhelming preference for our method compared with task-specific alternatives (our generations were preferred at rates of 94% and 84% against those produced by arVHE). Even when our method was compared with GNS, we found participants had a slight preference for the few-shot generations our system produced, with 64% preference rate. We point to this result as another strong indication of the impressive performance that our domain-general method is capable of achieving.

6.2.4 Concept Co-segmentation

Our method also natively supports co-analysis tasks. When we infer a Template Program and instantiations that explain an input visual group, we can use the shared structure of the Template Program to parse the group members in a consistent fashion. We visualize this capability in the *Seg* rows of Figure 6.2. This consistent parsing allows us to perform a co-segmentation task: given an input visual group, where exactly one member

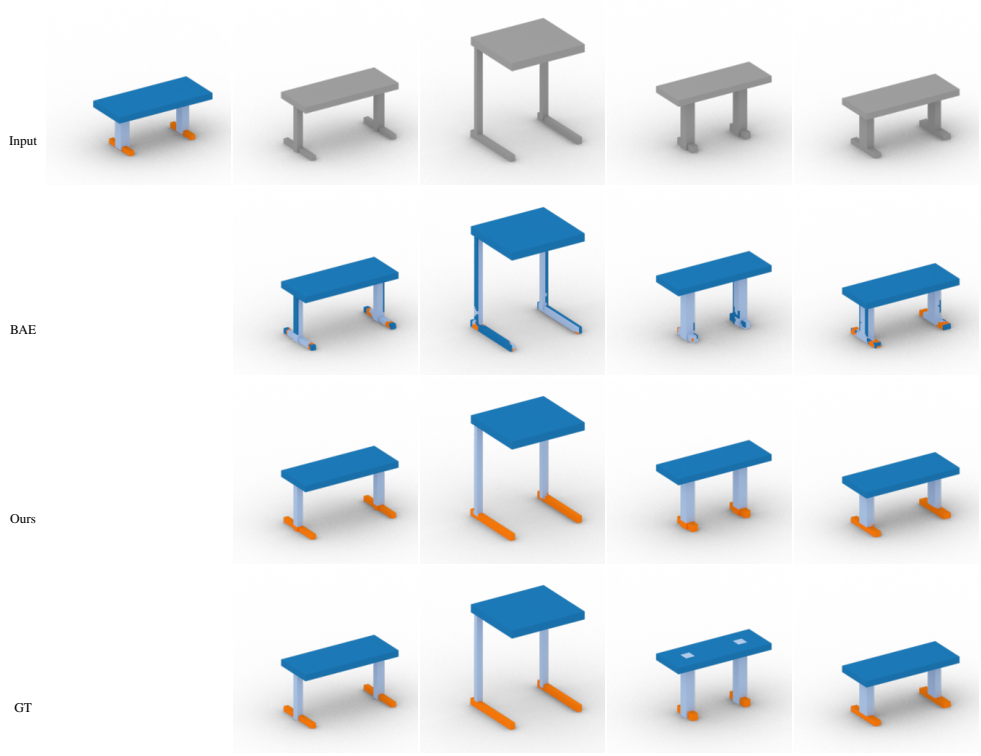

Figure 6.4: We compare co-segmentations produced from voxelized shapes (*Input*) to ground-truth annotations (*GT*)

of the group has a labeled segmentation, our goal is to propagate this labeling to the other group members. We provide further details in Appendix D.5.3.

We compare how our method does on co-segmentation tasks across domains. Our main comparison is against BAE-NET [25], which is designed specifically for this task. For Omniglot, BPL and GNS can also perform this task by parsing visual inputs to ordered strokes. We report results of our experiments in Table 6.1. We evaluate performance with a mean intersection over union metric (*mIoU*) that measures how closely the output segmentation predictions match the ground-truth labelings. Despite the fact that our method never trains on human stroke data, we achieve a better mIoU on this co-segmentation task compared with GNS, and nearly match the metric value achieved by BPL. Though our output co-segmentations are less structured compared with the ordered stroke parses BPL and GNS can produce, we are encouraged by our method's performance in this task. For our 3D shapes domain, as BAE-NET was originally designed to operate over voxels, our comparisons against it use a variant of our method that also takes in voxel inputs. We visualize an example co-segmentation of each method in Figure 6.4. Across domains and input modalities, we find that we outperform BAE-NET for this task.

6.2.5 Discussion

Out-of-distribution generalization Different domains require different levels of generalization. For instance, in the Omniglot dataset there is no alphabet overlap between train and test characters, so strong generalization capabilities are required for each test concept. As we procedurally generated the 2D layout domain, we are able to control and evaluate the level of out-of-distribution generalization required for each test-set concept. We consider three settings. *Easy* concepts have a new combination of attributes, but each attribute has been seen before (e.g. chair back, top-left of Fig. 6.2). *Medium* concepts have a new attribute not seen during training (e.g. double-sided leaves, top-middle of Fig. 6.2). *Hard* concepts are from a meta-procedure that was not used at all during training (e.g. turtles, top-right of Fig. 6.2). We find that while our method does become worse when evaluated on more difficult concepts, its performance remains more consistent compared with alternative approaches. We explore this phenomenon further in Appendix D.1.1.

Ablations We consider the effect of different design decisions on our method with an ablation study. We provide the details of this study and quantitative results in Appendix D.1.2. We find that our bootstrapped fine-tuning process is critical to adapting networks pretrained on synthetic data towards a target dataset of interest. We validate that our scheme of allowing the Template Program to capture parametric relationships improves performance on downstream tasks. Finally we compare our three step inference approach ($TP \rightarrow SE \rightarrow z$) against a two step alternative where each z is predicted directly from the TP. In this comparison, we find that our formulation, which allows the ParamNet to attend over the complete expression tree, outperforms this alternative formulation.

Unconditional Concept Generation Though we mainly evaluate our method on few-shot generation and co-segmentation, these are not the only concept-related tasks our framework can support. For Omniglot, we explore how our approach can be used for unconditional concept generation. In fact, this is a task we naturally solve as part of our fine-tuning procedure: the wake-sleep component of each training loop uses an unconditional generative model to sample *Template Programs* that represent new concepts. We visualize some of these generations in Figure 6.5.

Table 6.2: Perceptual study results evaluating few-shot generation performance. Our method is greatly preferred over task-specific alternatives and slightly preferred over domain-specific alternatives.

Domain	——*Omniglot*——		*3D Shapes*
	arVHE	**GNS**	**arVHE**
Ours vs.	94%	64%	84%

6.3 Discussion

We presented the *Template Programs* framework: a neurosymbolic method that learns to capture visual concepts with structured symbolic objects. We demonstrated that our method flexibly learns to infer Template Programs across multiple visual domains: 2D primitive layouts, Omniglot characters, and 3D shape structures. Our approach supports multiple downstream tasks of interest, such as few-shot generation and cosegmentation. On these tasks, we achieve superior performance over other domain-general, *task-specific* alternatives, and find that we match, and in some cases slightly outperform, *domain-specific*, task-general alternatives for the limited areas where they exist.

Figure 6.5: Qualitative examples of unconditional concept generations on the Omniglot domain. We show 30 concepts synthesized by our method where each concept is associated with two rows of five images. The bottom five images depict five samples from each concept, and the top five images show the nearest neighbor in the training set by Chamfer distance to each sample.

Chapter 7

Macro Operation Discovery for Shape Programs

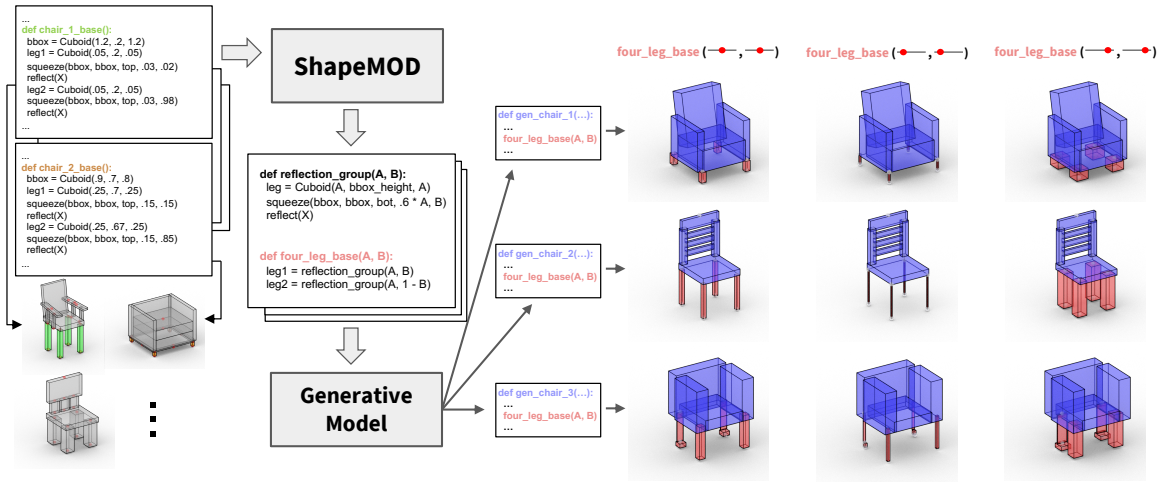

Figure 7.1: We propose ShapeMOD, an algorithm which takes as input a collection of 3D shape programs and makes them more compact by automatically discovering common *macros* which can be re-used across the collection. We apply ShapeMOD to datasets of ShapeAssembly programs and find that generative models which train on refactored programs containing these macros produce more plausible output shapes than those trained on the original programs. The discovered macros also facilitate shape editing by exposing only a small number of meaningful parameters for manipulating shape attributes. For example, the *four_leg_base* macro exposes two parameters (visualized as sliders with red handles); one parameter controls leg size, while the other controls leg spacing.

To maximally realize the benefits of its representation, a good shape program should be compact and expressed at a high level while still exposing important degrees of freedom for editing. One way to create

such programs is to introduce higher-level functions, or *macros*, into the shape DSL. We define a macro to be a function that, when executed, expands into a series of commands from the base DSL.

In this chapter, we present ShapeMOD, an algorithm for automatically discovering such macros from a collection of shape programs. ShapeMOD operates on any imperative, statement-based language whose commands are parameterized by discrete and continuous parameters. It is designed around the principle of discovering macros that make programs more *compact*, where compactness is measured by the number of function calls and number of free parameters required to represent the input shape collection.

In pursuit of compactness, one must consider the cost incurred by adding more functions (i.e., macros) to the DSL. At one extreme, one could use no macros, which results in the maximum number of free parameters (i.e., minimal compactness). At the other extreme, one could define a macro for each shape program in the input collection—this is maximally compact, but makes applications such as shape manipulation or learning to generate novel shape programs impossible. Our insight is that the trade-off space between these extremes can be navigated via optimization to find a middle-ground where a small set of macros explain a high percentage of variations across the input collection of shape programs. Critically, these frequently-used macros expose sufficient degrees of freedom to allow for shape manipulation and exploration across a shape collection.

To demonstrate the benefits of ShapeMOD, in this chapter we apply it to collections of ShapeAssembly programs (Chapter 3), where, in order to discover more useful macros, we modify the grammar slightly, as described in Appendix E.1. In its original form, ShapeAssembly contains two base functions and three macro functions. The base functions are `Cuboid`, which creates an cuboid part proxy, and `attach`, which moves a Cuboid to satisfy the described spatial relationship. The `squeeze`, `reflect` and `translate` commands are expert-defined macros that abstract common patterns of structural and geometric variation. Each of these macros expands into a sequence of lower-level `Cuboid` and `attach` commands. As we observed that these macros improved downstream task performance, in this chapter we explore how such similarly beneficial macro operations could be automatically identified. ShapeMOD is the first approach that discovers macro operations capturing parametric relationships between continuous parameters across a collection of imperative programs. While specialized data structures such as version spaces and E-graphs can efficiently reason about rewrites of functional programs [42, 31], they cannot efficiently reason over semantic line re-orderings of imperative programs (i.e. maintaining correct execution behavior) and thus are not applicable to languages such as ShapeAssembly.

We run ShapeMOD on multiple collections of shape programs expressed in the ShapeAssembly DSL to

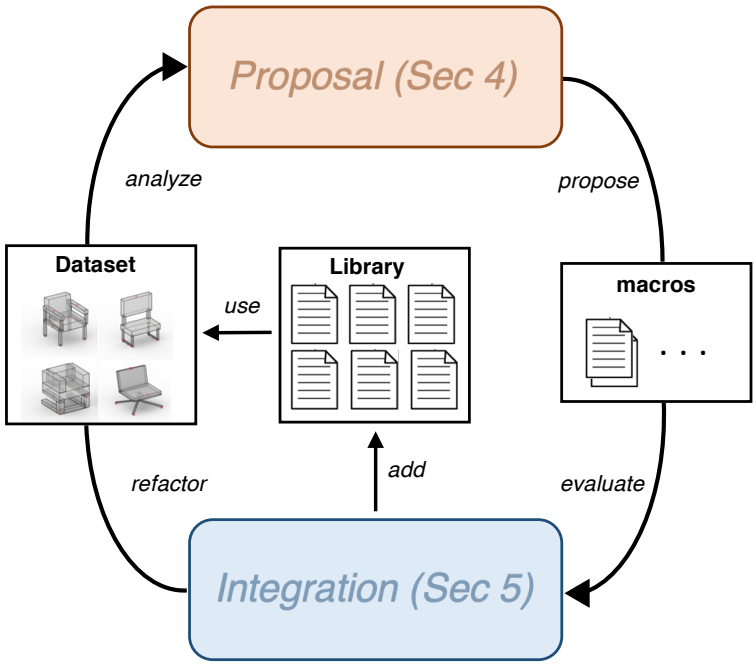

Figure 7.2: ShapeMOD consists of two alternating phases: proposing new candidate macros (top) and refactoring programs to use some of the proposed macros (bottom).

discover new libraries of macros. For example, in Figure 7.1, starting from a set of chair shape programs, ShapeMOD discovers a reusable macro for four leg chair bases which exposes a compact set of associated control sliders. We demonstrate the benefits of working with these discovered macros, by evaluating how adding the discovered macros into the language affects performance on downstream tasks: learning a generative model for shape programs, learning to infer shape programs from unstructured geometry, and goal-directed editing of shapes via their programs. In all cases, task performance is improved by using automatically discovered macros. Finally, we show that ShapeMOD can find useful macros even when trained on a set of ShapeAssembly programs from multiple categories.

We provide code for our method at https://github.com/rkjones4/ShapeMOD .

7.1 Macro Operator Discovery

ShapeMOD's goal is to take a dataset of programs $\mathcal{D}$ and the library of DSL functions used to express them $\mathcal{L}$, and return a new library (with additional macros) which is able to express the programs in $\mathcal{D}$ with fewer free parameters. The motivation here is that macros should remove free parameters that correspond to *extraneous* degrees of freedom, i.e. degrees of freedom that can create implausible output shapes, such as

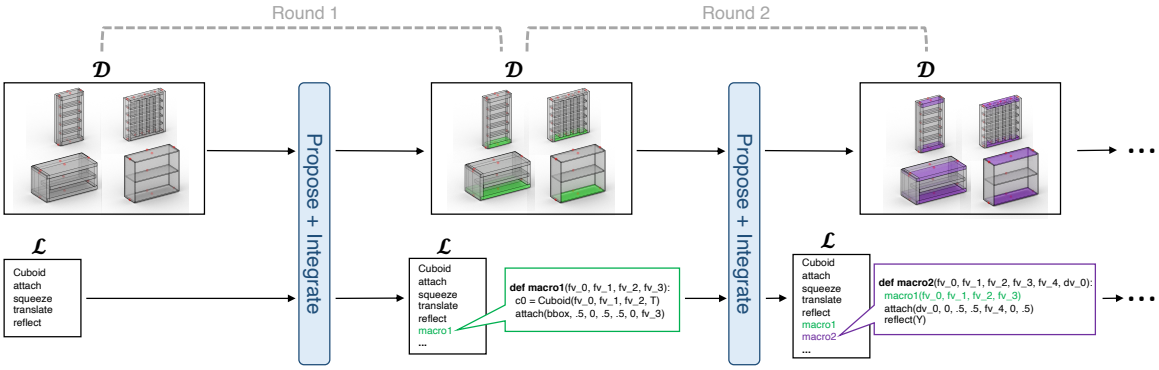

Figure 7.3: Running ShapeMOD for multiple rounds allows for discovery of increasingly complex macros. Here, a macro discovered in Round 2 uses a macro previously found in Round 1 as part of its function body.

independently changing the length of each leg of a table. At the same time, we want to keep the number of functions in our library relatively small, so as not to remove *necessary* degrees of freedom that can create meaningful shape manipulations. We formalize this trade-off in an objective function f which the algorithm attempts to minimize.

7.1.1 Overview

The ShapeMOD algorithm has two phases. First, a *proposal phase* (Section 7.2) finds clusters of similar programs and uses these clusters to propose a set of candidate macros. Then, an *integration phase* (Section 7.3) greedily iterates through a ranked list of these candidate macros and adds them to the library $\mathcal{L}$ whenever it would improve the objective function f. These phases can be alternated one after the other for multiple rounds, with the output of one phase treated as the input for the next (Fig. 7.2). By iterating this procedure for multiple rounds, increasingly complex macros can be found; as a macro discovered in round t can use a previously-discovered macro from round $t-1$ in its definition (Fig. 7.3).

Working with imperative programs that contain real-valued parameters presents unique challenges. For instance, it is difficult to reason about valid line re-orderings of imperative programs when discovering macros and deciding when they can be applied. ShapeMOD uses a sampling-based approach to discover macros by creating clusters of shapes with shared program structure (Section 7.2.1) and a beam search procedure to decide how to apply discovered macros to existing programs (Section 7.1.4). Moreover, when dealing with real-valued parameters, it is challenging to find meaningful (non-spurious) parametric relationships, especially within a single program. To achieve generality, ShapeMOD finds abstracted expressions that simultaneously describe multiple programs from a cluster of related shapes (Section 7.2.2).

Algorithm 2: ShapeMOD

Input: Library of functions $\mathcal{L}$, Program dataset $\mathcal{D}$, Objective f
Output: Updated $\mathcal{L}$ with macros, best programs $\mathcal{P}^*(\mathcal{D}, \mathcal{L})$

for *num_rounds* **do**
  candidate_macros $\leftarrow$ Set() {Proposal Phase}
  for *num_proposal_steps* **do**
    $z, o \leftarrow$ sampleProgAndOrder($\mathcal{D}$) {Sec 7.2.1}
    $\mathcal{P}_{\textbf{matches}} \leftarrow$ findMatchingProgs($\mathcal{D}$, z, o)
    $\mathcal{P}_{\textbf{cluster}} \leftarrow$ sampleByParamSim($\mathcal{P}_{\textbf{matches}}$)
    $z_{\textbf{abs}} \leftarrow$ findAbstractProg($\mathcal{P}_{\textbf{cluster}}, \mathcal{L}$) {Sec 7.2.2}
    $\mathcal{M} \leftarrow$ proposeMacrosForProg($z_{\textbf{abs}}$) {Sec 7.2.3}
    $\mathcal{M} \leftarrow$ generalize($\mathcal{M}$) {Sec 7.2.4}
    candidate_macros $+= \mathcal{M}$
  end for
  $\tilde{\mathcal{D}} \leftarrow$ subsample($\mathcal{D}$) {Integration Phase}
  for *num_integration_steps* **do**
    $M \leftarrow$ getTopRankedMacro(*candidate_macros*) {Sec 7.3.1}
    $\mathcal{L}' \leftarrow$ optimize(f, $\mathcal{L}$, $\mathcal{L} + \{M\}$, $\tilde{\mathcal{D}}$) {Sec 7.3.2}
    if $\mathcal{L}' \neq \mathcal{L}$ **then**
      $\mathcal{L} \leftarrow \mathcal{L}'$; **continue**
    end if
    $\mathcal{M}_{\text{infreq}} \leftarrow$ findInfrequentMacros($\tilde{\mathcal{D}}$, $\mathcal{L}$, $\mathcal{L} + \{M\}$)
    if $\mathcal{M}_{\text{infreq}} = \varnothing$ **then**
      continue
    end if
    $\mathcal{L}' \leftarrow$ optimize(f, $\mathcal{L}$, $\mathcal{L} + \{M\} - \mathcal{M}_{\text{infreq}}$, $\tilde{\mathcal{D}}$)
    if $\mathcal{L}' \neq \mathcal{L}$ **then**
      $\mathcal{L} \leftarrow \mathcal{L}'$
      for $M \in \mathcal{M}_{\text{infreq}}$ **do**
        $\mathcal{L} \leftarrow$ optimize(f, $\mathcal{L}$, $\mathcal{L} + \{M\}$, $\tilde{\mathcal{D}}$)
      end for
    end if
  end for
  for $M \in \mathcal{L}$ **do**
    $\mathcal{L} \leftarrow$ optimize(f, $\mathcal{L}$, $\mathcal{L} - \{M\}$, $\tilde{\mathcal{D}}$)
  end for
  $\mathcal{D} \leftarrow$ filterBadOrders(f, $\mathcal{D}$, $\mathcal{L}$) {Sec 7.3.3}
end for
return $\mathcal{L}$, $\mathcal{P}^*(\mathcal{D}, \mathcal{L})$ {Sec 7.1.4}

Complete pseudocode for ShapeMOD is shown in Algorithm 2; Sections 7.2 and 7.3 explain this procedure in more detail. As input, it takes in a starting library of functions $\mathcal{L}$, a dataset of imperative programs $\mathcal{D}$ and an objective function f to be minimized. Each element of $\mathcal{D}$ is a tuple $(z, \mathcal{O}_z)$ containing program lines z and the set of valid orderings for those lines $\mathcal{O}_z$ (i.e. re-orderings of the lines which produce the correct output when executed).

7.1.2 Initialization

In our experiments, the library $\mathcal{L}$ is initialized with the 5 manually designed functions from the ShapeAssembly grammar. Then, starting with a collection of hierarchically-organized 3D cuboid structures from PartNet [141], we use ShapeAssembly's data parsing algorithm to find program lines z which recreate each shape. We then developed a procedure to determine the set of valid orderings $\mathcal{O}_z$ for that program (i.e. all orderings which produce the correct output geometry) to form our input dataset $\mathcal{D}$. Further details about the data parsing and valid ordering procedures can be found in the supplemental (Section A.3).

7.1.3 Objective Function

Our goal is to represent an entire dataset of programs compactly (removing free parameters) while also keeping the number of functions in the library small. Specifically, our objective is to minimize a weighted sum of the number of functions in $\mathcal{L}$ and the number of free parameters needed to represent programs in the dataset $\mathcal{D}$. For ShapeAssembly, free parameters can have multiple types T: Choice of function per line (**fn**), cuboid ID (**cid**), float/continuous (**f**), discrete (**d**), Boolean (**b**). One may care about compressing these types differently, we allow each parameter type to be weighed differently in the objective defined as,

$$f = \lambda_{\mathbf{n}}|\mathcal{L}| + \frac{1}{|\mathcal{D}|}\sum_{\tau\in\mathrm{T}} \lambda_\tau |\tau(\mathcal{P}^*(\mathcal{D},\mathcal{L}))| + \lambda_\epsilon \epsilon(\tau, \mathcal{D}, \mathcal{P}^*(\mathcal{D},\mathcal{L}))$$

where $\mathcal{P}^*(\mathcal{D},\mathcal{L})$ returns the best programs for $\mathcal{D}$ using the functions in $\mathcal{L}$ (Section 7.1.4), $\tau(\mathcal{P})$ returns the set of all τ-typed free parameters in the programs $\mathcal{P}$, and $\epsilon(\tau, \mathcal{D}, \mathcal{P})$ returns the sum of errors in τ-typed parameters incurred by using $\mathcal{P}^*(\mathcal{D},\mathcal{L})$ in place of the original programs in $\mathcal{D}$. The weights $\lambda_{\mathbf{n}}$, $\{\lambda_\tau|\tau\in\mathrm{T}\}$ and λ_ϵ can be adjusted to express preferences for the types of macros the algorithm aims to find. In our experiments, we use $\lambda_{\mathbf{n}} = 1$, $\lambda_{\mathbf{fn}} = 8$, $\lambda_{\mathbf{cid}} = 8$, $\lambda_{\mathbf{f}} = 1$, $\lambda_{\mathbf{d}} = 0.5$, $\lambda_{\mathbf{b}} = 0.25$, and $\lambda_\epsilon = 10$.

7.1.4 Finding the Best Program for a Given Library

Calculating the value of f over a dataset of shapes requires finding the program under $\mathcal{L}$ that minimizes the objective function for each program $(z, \mathcal{O}_z) \in \mathcal{D}$. As $\mathcal{O}_z$ is a collection of valid orderings of the program lines z, we solve this problem by finding the best scoring program under $\mathcal{L}$ for every $o \in \mathcal{O}_z$. Combining an ordering o with program lines z produces a program expressed in terms of base library functions z_o. We then want to find the best program, z^*, that uses the functions in $\mathcal{L}$ (including macros, if $\mathcal{L}$ contains them)

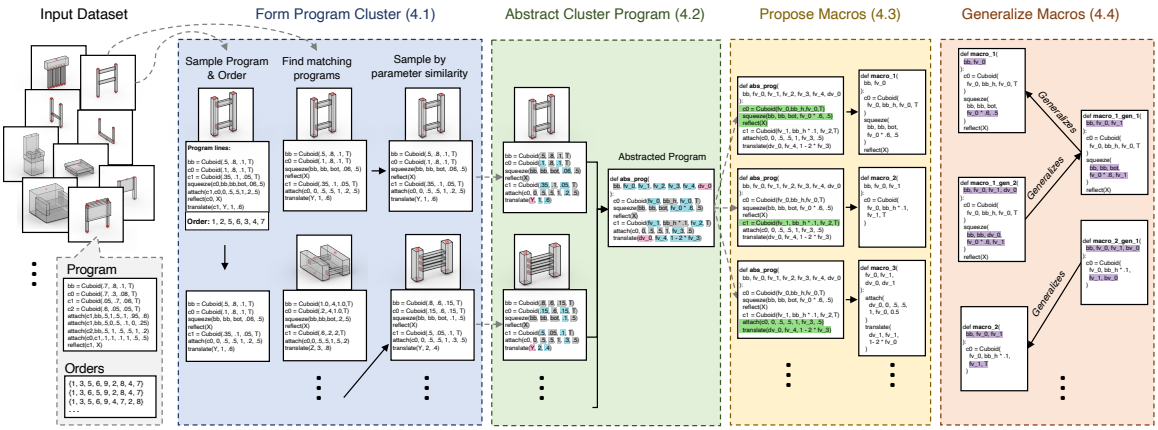

Figure 7.4: ShapeMOD's proposal phase, which proposes candidate macros to be added into $\mathcal{L}$. Each round of this phase begins by identifying a cluster of structurally-identical programs with similar parameter values within the input dataset (Section 7.2.1). It then finds a single abstracted program which subsumes most or all of the programs in this cluster (Section 7.2.2); here, gray parameter values are abstracted as constants, blue ones as continuous free variables, and pink ones as discrete free variables. Subsequences of lines in this abstracted program (shown in green) are isolated to form potential macros which could be used to re-write the program (Section 7.2.3). Finally, this set of candidate macros is expanded by including generalizations of the initial set (Section 7.2.4); purple lines show lines that are generalized. Best viewed on a high-resolution screen.

to recreate z_o while minimizing f. We implement this procedure with a beam search that iteratively builds partial programs in the beam by adding calls to functions from $\mathcal{L}$ whose expansions cover lines in z_o. For a function expansion to *cover* a sequence of program lines, the expansion must match those lines on command type, the values of the discrete / Boolean parameters must match exactly, and the continuous parameters must differ by an amount no greater than ϵ. We set $\epsilon = 0.05$, finding that larger values lead to abstracted programs with degenerate geometry. We rank partial programs in the beam by their objective value, normalized by the number of lines in z_o it is covering. This search runs until all programs in the beam have no more lines in z_o to cover; the program with lowest objective value is returned as the best program z^*. In the case of ties, we choose the program with the most canonical ordering, as explained in the supplemental material. In our implementation, we use a beam width of 10. Other search strategies could be applied here; we chose beam search as it was relatively fast and found good solutions.

7.2 Proposal Phase

The goal of ShapeMOD's proposal phase is to construct a set of candidate macros which might be useful for compressing the dataset of shape programs $\mathcal{D}$. A schematic overview of the proposal phase is shown in

Figure 7.4. In each proposal round, the algorithm first forms a cluster of similar programs sampled from $\mathcal{D}$ (Section 7.2.1). Then, using the functions of $\mathcal{L}$, it finds an abstracted program that explains the majority of examples in the cluster while trying to remove free parameters whenever possible (Section 7.2.2). It converts this abstracted program into a set of candidate macros (Section 7.2.3) and finds potential generalizations of these macros (Section 7.2.4). This process is repeated for *num_proposal_steps* (we use 10000) to build up a large collection of candidate macros.

7.2.1 Form a Program Cluster

The goal of the cluster formation step is to find a set of programs from $\mathcal{D}$ that can be represented by a single abstracted program, i.e. a program with free variables. The blue box in Fig. 7.4 illustrates the procedure. The algorithm first randomly samples a program z from $\mathcal{D}$ and then randomly samples an order o from the possible valid orderings in $\mathcal{O}_z$ (Algorithm 2, line 4). It then finds the set of programs $\mathcal{P}_{\textbf{matches}}$ from $\mathcal{D}$ that *structurally match* z and also have o as one of their valid orderings in (Algorithm 2, line 5). In ShapeAssembly, two programs structurally match if they use the same set of commands which refer to the same cuboid IDs (though their other parameters may vary).

For each program in $\mathcal{P}_{\textbf{matches}}$, we record the norm n of the difference of its continuous parameters compared with those in z. We then form a probability distribution over $\mathcal{P}_{\textbf{matches}}$, where each program is given a weight proportional to $1 - \frac{n}{n*}$, where $n*$ was the maximum observed n. Taking the parameter distance between programs into account results in clusters that are more semantically consistent, which increases the likelihood the abstracted program we produce can discover meaningful parametric relationships. Finally, we sample k programs from $\mathcal{P}_{\textbf{matches}}$ using this probability distribution in order to form $\mathcal{P}_{\textbf{cluster}}$ (line 6). We use $k = 20$ in our implementation.

7.2.2 Find Abstracted Program for Cluster

Given the cluster of programs $\mathcal{P}_{\textbf{cluster}}$ identified in the previous section, the next step is to use the library of functions $\mathcal{L}$ to find the most compact program (fewest free parameters) that can represent the majority of programs in $\mathcal{P}_{\textbf{cluster}}$ (Algorithm 2, line 7). By construction, the sequence of functions and cuboid IDs is the same across all programs in $\mathcal{P}_{\textbf{cluster}}$. To build up the abstracted program $z_{\textbf{abs}}$, the algorithm uses a similar procedure to the best-program-finding routine in Section 7.1.4: covering each line in the cluster by choosing functions from $\mathcal{L}$. However, instead of using a beam search to find the sequence of functions, here

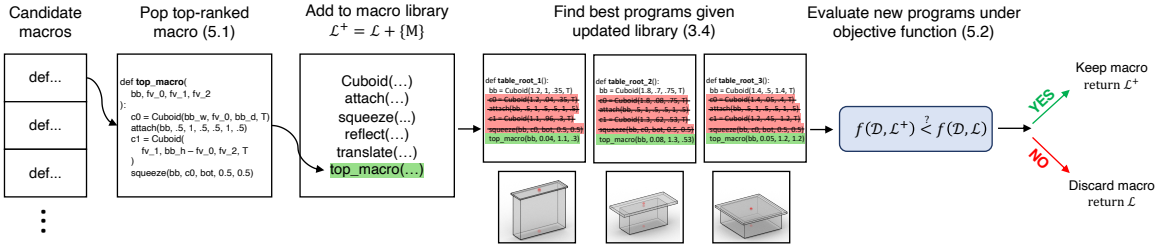

Figure 7.5: ShapeMOD's integration phase, which chooses which candidate macros to add to the DSL library $\mathcal{L}$. On each round of this phase, the algorithm heuristically ranks candidate macros based on which are likely to improve program compression, adds the top-ranked macro to the library, then finds the best refactored program for each program in the input dataset $\mathcal{D}$ under this new library. If this refactoring lowers the objective value $f(\mathcal{D}, \mathcal{L})$, then the macro is kept in the library; otherwise, it is discarded.

we employ a greedy strategy. We create a preference ordering over the functions of $\mathcal{L}$ based on how many free parameters each function constrains (weighted by their respective λ_τ weights). Then, whenever we need to pick a function, we step through this ordering, until we find a function that is able to match the parameters of at least $p = 70\%$ of the next lines from $\mathcal{P}_{\textbf{cluster}}$.

For each function added to the abstracted program, we iterate through its parameter slots to see if we can remove more degrees of freedom. For discrete parameters, a constant can be used, a previously defined parameter can be used, or a new free parameter can be declared. For continuous parameters, a constant can be used, an expression over previously defined parameters can be used, or a new free parameter can be declared. The details of this logic can be found in the supplemental material (Section B.1). In all cases, the value chosen for each parameter must still be valid for at least p percent of programs in $\mathcal{P}_{\textbf{cluster}}$. This process iterates until there are no remaining uncovered lines in the programs of $\mathcal{P}_{\textbf{cluster}}$. At this point, $z_{\textbf{abs}}$ is complete. The green box in Fig. 7.4 shows an example of finding a single abstracted program for two base programs.

7.2.3 Proposing Candidate Macros

The abstracted program $z_{\textbf{abs}}$ found in the previous step represents multiple shape programs from our dataset (via leaving some of its parameters as free parameters). Thus, its function body likely contains re-usable shape programming patterns—in other words, it is a good source of potential macros $\mathcal{M}$ (Algorithm 2, line 8). In this next step, the algorithm iterates through the lines of $z_{\textbf{abs}}$ and finds all line sequences that could be turned into a valid macro (yellow box in Fig. 7.4). A valid macro M is a sequence of program lines that simplifies the program, i.e. it must remove some degree of freedom from the program lines it aims to cover. For both computational efficiency, and to encourage the creation of more meaningful macros, we impose some additional restrictions on the definition of a valid macro; see the supplemental material (Section

B.2). For each created candidate macro, we record what cluster it was found in and the lines of the cluster it covered, in order to calculate frequency statistics used later in the integration phase (Section 7.3).

7.2.4 Generalizing Macros

As $z_{\textbf{abs}}$ is designed to maximally condense all of the programs in $\mathcal{P}_{\textbf{cluster}}$, the generated candidate macro operators $\mathcal{M}$ may be somewhat overly-specific to the subset of programs in $\mathcal{P}_{\textbf{cluster}}$. Furthermore, $\mathcal{M}$ may also contain some very similar macros that are treated as distinct. To get around these issues, the proposal phase concludes with a generalization step, where for each discovered candidate macro, we also find all generalizing macros that are within n program edits (Algorithm 2, line 9). We set $n = 2$ due to running time constraints; in principle, higher values of n will lead to better solutions. For a given macro M, another macro M' is defined to be generalizing if for every parameterization of M, M'could be parameterized to produce the same output. From this generalization procedure we form a graph where each node is a macro and edges between two nodes indicates a generalizing relationship (orange box in Fig. 7.4). This graph is used to update frequency statistics (in that generalizing macros also cover all lines covered by macros they generalize) which influences the candidate macro ranking logic used by the integration phase (Section 7.3).

7.3 Integration phase

Given candidate macros from the proposal phase, the integration phase chooses which macros to add to the library $\mathcal{L}$ in order to minimize its objective function f. Figure 7.5 shows an overview. Solving such a subset selection problem optimally is intractable, so this phase instead employees a greedy approximation. It iterates through the candidate macro operators, on each iteration taking the highest ranked macro based on expected improvement to f (Section 7.3.1). It then decides whether to add the macro into the library $\mathcal{L}$ by evaluating its effect on the objective function (Section 7.3.2).

7.3.1 Ranking Candidate Macros

The proposal phase can generate tens of thousands of candidate macros; it is computationally intractable to consider all of them. To prioritize which candidate macros to consider within a finite time budget, the algorithm employs a heuristic ranking scheme (Algorithm 2, line 13). The rank of a candidate macro M is based on an estimate of how much using M would improve the score of the objective function. The ranking scheme first calculates the *gain* of the macro over the functions already in $\mathcal{L}$. The gain g of a macro M is

the weighted sum of the number of free parameters (weighted by their respective λ_τ weights) that would be removed each time M were used in a program instead of the lowest-cost sequence of functions currently in $\mathcal{L}$ that is equivalent to or generalizes M. Then our ranking scheme calculates the percentage of shapes p that produced M as a candidate macro during the proposal phase. The ranking score of M is then simply $p \cdot g$. This score is a simple estimate of the effect on the actual objective value $f(\mathcal{D}, \mathcal{L} + \{M\})$ that does not require the expensive step of finding the best programs for the whole dataset.

7.3.2 Evaluating & Selecting Candidate Macros

Given a candidate macro operator M, the next step is to see if adding it to $\mathcal{L}$ would actually improve the value of the objective function f. For this, we define a function `optimize` which takes in f, the current library $\mathcal{L}$, a modified version of the library $\mathcal{L}^+$, and a subset of programs from the dataset $\tilde{\mathcal{D}} \subset \mathcal{D}$. It returns whichever version of the library has the lower objective value, i.e. argmin$(f(\tilde{\mathcal{D}}, \mathcal{L}^+) < f(\tilde{\mathcal{D}}, \mathcal{L}))$. Using a subsample $\tilde{\mathcal{D}}$ of the full dataset reduces computation time, i.e. we are using an unbiased estimator of the true objective value for the dataset.

The algorithm first calls `optimize` with a modified library where M is added to $\mathcal{L}$ (line 14). If this leads to a library change, then it continues to the next candidate macro operator (lines 15-16). If $\mathcal{L}$ remains unchanged, it checks if any of the functions currently in $\mathcal{L}$ are used significantly less in finding the best programs over $\tilde{\mathcal{D}}$ when the modified library version is used (line 17). If the set of functions in $\mathcal{L}$ whose frequency decreased significantly, $M_{\textbf{infreq}}$, is not empty, then it runs `optimize` once again with a modified version of the library that includes M but removes all elements of $M_{\textbf{infreq}}$ (lines 18-20). This step allows the algorithm to avoid a local minima where M would not be added to $\mathcal{L}$, even if it could ultimately improve f, because similar macros to M had been added to $\mathcal{L}$ earlier. If this step changes the library, then $\mathcal{L}$ has been updated to include M, but it does not include any of the functions in $M_{\textbf{infreq}}$. Thus, the algorithm attempts to add each $M \in M_{\textbf{infreq}}$ back into $\mathcal{L}$, by once again calling `optimize` and keeping the library version with the better score (lines 23-24). Finally, after evaluating *num_integration_steps*=20 macros, the algorithm checks if f can be improved by removing any of the functions in $\mathcal{L}$ (lines 25-26). This can be beneficial, for instance, when a macro discovered in an early round becomes a sub-routine of a macro discovered in a later round, and therefore appears less frequently (or not at all) in $\mathcal{P}^*(\mathcal{D}, \mathcal{L})$.

7.3.3 Removing Bad Program Orders

When $\mathcal{L}$ is composed of only original library functions, any valid ordering in $\mathcal{O}_z$ for z will lead to a program that produces the same score under f. As macros are added into $\mathcal{L}$, using different line orderings in $\mathcal{O}_z$ may result in different scores under f (as some line orders will prohibit certain macros from being applied). As such, after each integration round, the algorithm removes any orders from $\mathcal{O}_z$ that lead to objective function scores that are significantly worse (using a threshold of $\tau_o = 1$) then the score produced by the order, o^*; the order that leads to the best objective function score for z (Algorithm 2, line 27). The following proposal rounds will then only be able to use orderings that have not been filtered out of $\mathcal{D}$. Keeping the orderings that perform best during the preceding integration phase produces more accurate heuristic rankings of macros from the proposal phase (Section 7.3.1). We found this encouraged the discovery of complex macros, e.g. without this step, the 'four leg base' macro was not discovered.

7.4 Results and Evaluation

We experimentally evaluate ShapeMOD's effectiveness at compressing shape programs and at supporting downstream tasks. Our experiments use three categories of manufactured shapes (Chairs, Tables, Storage) from CAD models in PartNet. We use the same data parsing procedure as described in Chapter 3 to produce 3836 Chair programs, 6536 Table programs, and 1551 Storage programs. In Section 7.4.1, we examine the properties of ShapeMOD's discovered macros on dataset compression. In Section 7.4.2, we show that using these macros improves the performance of generative models of 3D shape structures. In Section 7.4.3, we demonstrate that macros aid in visual program induction tasks. And finally, in Section 7.4.4, we report the results of a user study comparing performance on goal-directed shape editing tasks with and without discovered macros.

7.4.1 Discovered Macros

For each shape category, we run ShapeMOD until $f$stops decreasing (5 rounds in all cases) to discover a small set of macro operators. Instead of applying ShapeMOD directly on hierarchical programs, we form $\mathcal{D}$ by decomposing each ShapeAssembly program into a collection of non-hierarchical sub-programs (e.g., a single Chair might contribute one program for its back sub-part and one program for its base sub-part). We implement ShapeMOD in Python and run the algorithm on a computer with an Intel i9-9900K CPU, which

Table 7.1: We measure how well different libraries can compress a dataset of shape programs (metric details in Section 7.4.1). For all compression metrics, lower values are better, as our goal is to find a small collection of functions that remove many degrees of freedom from the underlying shape programs. ShapeMOD operates by attempting to minimize f, and we show that it does in fact improve f compared to the No Macros version.

Category	Method	f	$\lvert\mathcal{L}\rvert$	$\mathbf{fn}(\mathcal{P}^*)$	$\mathbf{d}(\mathcal{P}^*)$	$\mathbf{f}(\mathcal{P}^*)$	$\mathbf{b}(\mathcal{P}^*)$
Chair	No Macros	411	**5**	29.8	17.8	84.4	11.3
	Baseline Macros	312	36	21.7	7.0	80.2	**4.2**
	ShapeMOD	**260**	17	**21.0**	**6.4**	**58.1**	8.6
Table	No Macros	356	**5**	25.6	16.3	70.7	9.6
	Baseline Macros	263	36	18.0	6.4	65.8	**3.2**
	ShapeMOD	**214**	15	**17.4**	**5.1**	**48.7**	5.6
Storage	No Macros	453	**5**	30.4	21.6	92.2	11.7
	Baseline Macros	314	48	**18.4**	**7.6**	88.45	**2.65**
	ShapeMOD	**283**	17	21.1	**7.6**	**68.9**	4.0

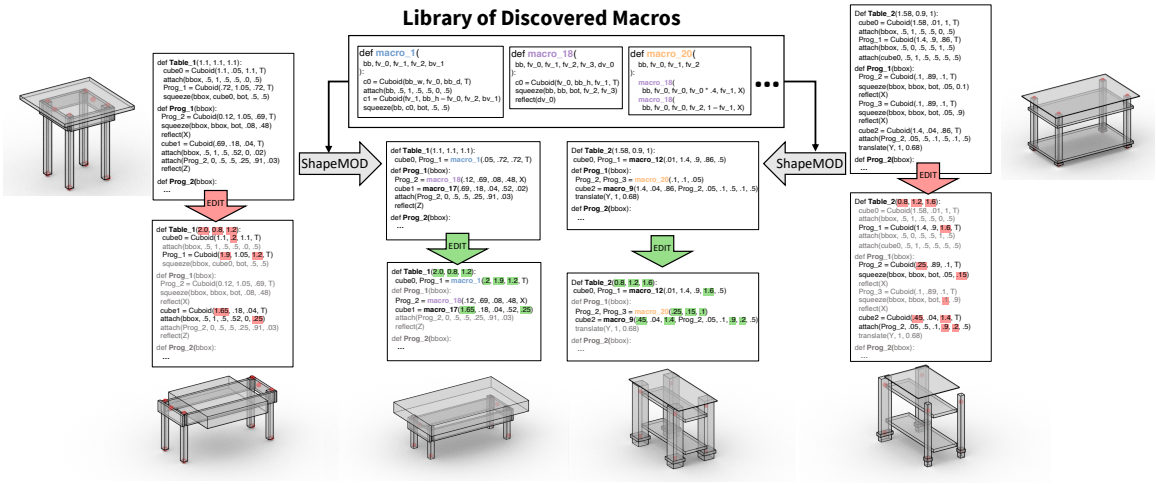

Figure 7.6: We show some macros (top-middle) that ShapeMOD discovered when run on the Table dataset, and program refactors that use these macros to significantly compress the number of exposed free parameters (ShapeMOD arrows from outside to inside). We show program edits (down arrows) of corresponding parameters in both programs with macros (green) and without macros (red). The discovered macros capture parametric relationships that better preserve shape plausibility under manipulation; for example, all chair legs remain the same size in the third column (macros), while the shape in the fourth column (no macros) becomes disconnected and physically implausible .

takes 5 hours for Chairs, 12 hours for Storage, and 19 hours for Tables.

Fig. 7.6 shows examples of some of the macros discovered for Tables; see the supplemental material for complete discovered libraries for all shape categories (Section F). These macros are used by multiple shape programs in our dataset, explaining common patterns and shortening programs that use them. They also better facilitate editing: making edits to a few parameters in macro-refactored programs tends to produce more plausible shape variations than edits to the corresponding parameters of the macro-free program. For instance, discovered macro_1 introduces a relationship that the heights of the table base and the table top

should sum to the height of the table bounding box. Without this macro, edits to base ShapeAssembly functions can easily cause the table top to overlap and intersect parts of the table base in an implausible manner (left side of figure).

We compare the library of functions generated by our ShapeMOD procedure to two baselines:

1. **No Macros**: The base library of functions from ShapeAssembly that is used to initialize our ShapeMOD procedure.
2. **Baseline Macros**: A naive single-pass approach for macro discovery that creates macros out of the most common structural sequences present in the dataset and replaces parameters with constants whenever a high percentage of its parameterizations share similar values. See Appendix E.2 for details.

Table 7.1 compares these baselines to ShapeMOD's discovered language on the task of compressing a dataset of 3D Shape programs. We consider the following metrics:

- Value of ShapeMOD's objective function (f)
- Number of functions in library ($|\mathcal{L}|$)
- Average number of lines in the best programs ($\mathbf{fn}(\mathcal{P}^*)$)
- Average number of discrete parameters in the best programs ($\mathbf{d}(\mathcal{P}^*)$
- Average number of continuous parameters in the best programs ($\mathbf{f}(\mathcal{P}^*)$)
- Average number of Boolean parameters in the best programs ($\mathbf{b}(\mathcal{P}^*)$)

By adding only a handful of macros to the language, ShapeMOD significantly compresses programs in terms of number of lines and number of free parameters. For instance, the 12 Chair macros discovered remove 30% of program lines, 64% of the discrete parameters, and 30% of the continuous parameters needed to represent the same dataset without macros. In total, these macro functions are able to decrease the value of the objective function we aim to minimize by 37%. Moreover, ShapeMOD is able to compress programs to a greater degree than the baseline approach, especially for continuous parameters, while using half as many (or fewer) new macros.

The examples shown in Fig. 7.6 suggest that programs refactored using ShapeMOD macros produce more plausible shapes under variations of their free parameters. We ran an experiment to quantify this behavior. Given a set of ground-truth Chair programs, we run ShapeMOD and our baseline macros procedure on them to create a set of macro-refactored programs. We then perturb the free parameters of both macro refactored and no macro programs by increasingly large perturbations, and we check how distributionally similar the outputs of the perturbed programs are to a held-out validation set of Chair shapes using Frechet Distance [73]

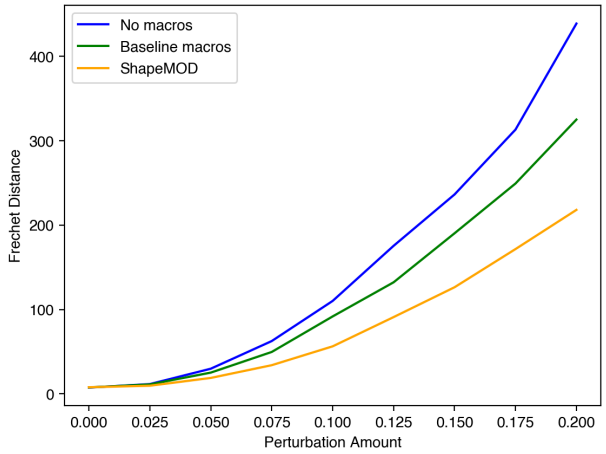

Figure 7.7: We measure distributional similarity (Frechet Distance [73]) between a set of reference chairs and a set of chair programs subjected to perturbations. We simulate perturbations by adding noise from a normal distribution (x-axis is standard deviation) to continuous parameters in the programs. Programs with ShapeMOD macros retain more similarity under larger perturbations, suggesting the macros remove degrees of freedom that permit shapes to move outside of their original distribution.

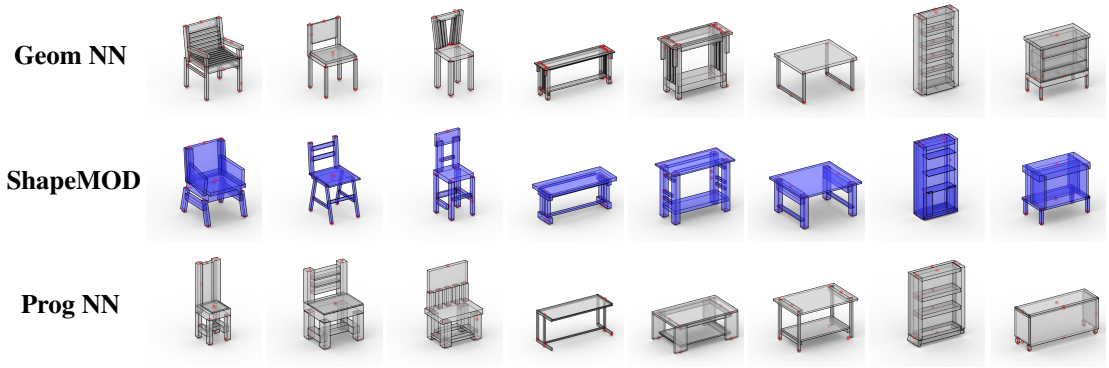

Figure 7.8: Some example outputs of generative models trained to produce ShapeAssembly programs expressed with macros discovered by ShapeMOD, along with their training set nearest neighbors (NN) by geometric and program similarity. Each cuboid represents a part proxy bounding volume. Structures are formed through attaching parts to one another (red dots). The generative models produce a variety of plausible structures without memorizing their training data. All corresponding programs can be found in supplemental material.

in the feature space of a PointNet classifier pre-trained on ShapeNet [162, 16]. Figure 7.7 plots this distance against the magnitude of parameter perturbation. Frechet Distance increases more slowly for programs that use macros, and increases the slowest for macros found using ShapeMOD. This indicates that the modes of variation in programs expressed with our method's macros are better at producing plausible output shapes that stay within the distribution that the collection of input programs originally came from. In Section 7.4.4,

we conduct a shape-editing user study to further validate this behavior.

7.4.2 Generating 3D Shapes

We are interested in how well ShapeMOD's discovered macros support the downstream task of generative shape modeling. Our hypothesis is that using macros will restrict the output space of a program-generating model, making it harder to output 'garbage' shapes. To test this hypothesis, we train generative models on programs with and without ShapeMOD macros.

For our generative model, we use the variational autoencoder architecture from Chapter 3, modified slightly to support programs that use an arbitrary number of functions as opposed to a fixed, predefined set (see Appendix E.3 for details). We train each model for 5000 epochs with a learning rate of $2e^{-4}$ and a batch size of 64. At the end of training, we choose the model from whichever training epoch produced the lowest Frechet Distance [73] to the training set; we report all other metrics on a held out set. Training was done on a computer with a GeForce RTX 2080 Ti GPU with an Intel i9-9900K CPU, consumed 2GB of GPU memory, and takes approximately 14 hours for Chairs, 22 hours for Tables, and 8 hours for Storage.

Fig. 7.8 shows some examples of novel shapes synthesized by these generative models, as well as their nearest neighbor from the training set according to both program similarity and geometric similarity. The generative models are capable of producing valid, plausible output shapes, and they do not simply memorize their training data.

We quantitatively assess the quality of the generative models' output shapes using the following metrics (additional details in supplemental Section C):

- **Rootedness ⇑ (% rooted):** percentage of shapes whose leaf parts all have a path to the ground.
- **Stability ⇑ (% stable):** percentage of shapes which remain upright when subjected to a small vertical drop.
- **Realism ⇑ (% fool):** percentage of test set shapes classified as "generated" by a PointNet [162] trained to distinguish between generated shapes and training set shapes.
- **Frechet Distance ⇓ (FD):** distributional similarity between generated shapes and training set shapes in the feature space of a pre-trained PointNet [73].

Table 7.2 shows the results of this experiment. Metrics related to realism/plausibility (% fool, FD) are always best for programs that use ShapeMOD macros as opposed to other language variants. Complexity (# Parts) and validity (% rooted, % stable) metrics also generally improve. The simple baseline macros are

Table 7.2: Comparing the quality of programs sampled from a learned generative model. Generative models trained on programs with ShapeMOD macros tend to produce more visually plausible, physically valid, and complex shapes than those trained on programs expressed with other libraries.

Category	Method	% fool ⇑	FD ⇓	# Parts ⇑	% rooted ⇑	% stable ⇑
Chair	No Macros	21.2	17.8	7.6	**93.9**	**82.3**
	Baseline Macros	16.9	24.1	8.5	89.8	74.2
	ShapeMOD	**25.6**	**16.7**	**8.6**	92.7	79.5
Table	No Macros	27.7	26.0	**8.0**	88.8	76.1
	Baseline Macros	11.5	38.1	7.0	90.2	79.6
	ShapeMOD	**29.2**	**23.2**	7.8	**93.2**	**84.3**
Storage	No Macros	4.9	70.0	6.0	92.4	85.5
	Baseline Macros	5.5	78.9	7.6	86.2	78.3
	ShapeMOD	**11.1**	**38.1**	**7.7**	**95.1**	**90.5**

Figure 7.9: Example visual program induction results from our point cloud → program inference experiment. ShapeMOD macros are especially helpful for the heterogeneous Storage category. All corresponding programs can be found in the supplemental material.

considerably worse; worse, in fact, than using no macros at all. We provide some qualitative comparisons of generated outputs from ShapeMOD vs No Macros in Appendix E.4.

7.4.3 Inferring 3D Shape Structures

Another downstream task is visual program induction: inferring a shape program from unstructured input geometry. Here, we consider inferring ShapeAssembly programs from a point cloud. As with generative modeling, our hypothesis is that macros will regularize this problem, making it harder to output invalid shapes.

We train the program inference networks end-to-end in an encoder-decoder paradigm. The encoder uses

Table 7.3: Quantitative results from our visual program induction experiment, where we train encoder-decoder models that learn to infer ShapeAssembly programs from point clouds. ShapeMOD macros regularize the output program space, leading to significant and consistent improvement in both reconstruction accuracy and physical validity. Note: Chamfer Distance (CD) values are multiplied by 1000 for clarity and we use a F-Score threshold of 0.03 [108].

Category	Method	CD ⇓	F-Score ⇑	% rooted ⇑	% stable ⇑
Chair	No Macros	44.2	54.8	93.7	83.6
	ShapeMOD	**41.7**	**56.1**	**96.9**	**88.0**
Table	No Macros	41.1	64.0	92.8	78.2
	ShapeMOD	**36.7**	**68.7**	**95.2**	**88.5**
Storage	No Macros	56.5	41.1	95.0	87.7
	ShapeMOD	**47.0**	**53.0**	**97.6**	**92.6**

a PointNet++ architecture to embed a point cloud sampled from dense surface geometry into a latent space. The decoder is identical the one used for generative modeling, it converts a point in this latent space into a hierarchical shape program. We create a 80/10/10 training/validation/test set split for all categories. Each network is trained for 2000 epochs with a learning rate of 2e-4 and a batch size of 32. We report metrics on test set shapes, and choose the model that reported the best Chamfer distance on the validation set.

Table 7.3 shows the results of this experiment. As with generative modeling, using ShapeMOD macros results in significantly better performance. Using ShapeMOD macros leads to better reconstruction accuracy, in terms of Chamfer distance and F-score, for all categories (average relative improvement for both is 11%). Moreover, the programs that are inferred with macros also always result in shapes that are more physically valid in terms of stability and rootedness. Fig. 7.9 shows some example input point clouds and the shapes produced by their inferred programs. Macros help considerably, especially for Storage, which is the most structurally- and geometrically-heterogeneous category and thus most likely to cause structured prediction models to output garbage.

7.4.4 Interactive Shape Editing

Our final downstream task is interactive shape editing. We hypothesize that programs with macros will support easier, more efficient shape editing. To test this hypothesis, we built an interactive ShapeAssembly editor and conducted a user study with it.

Editing interface We designed an interactive editing interface tailored to the goal-directed editing task of modifying a ShapeAssembly program such that its output shape matches a target output shape as closely

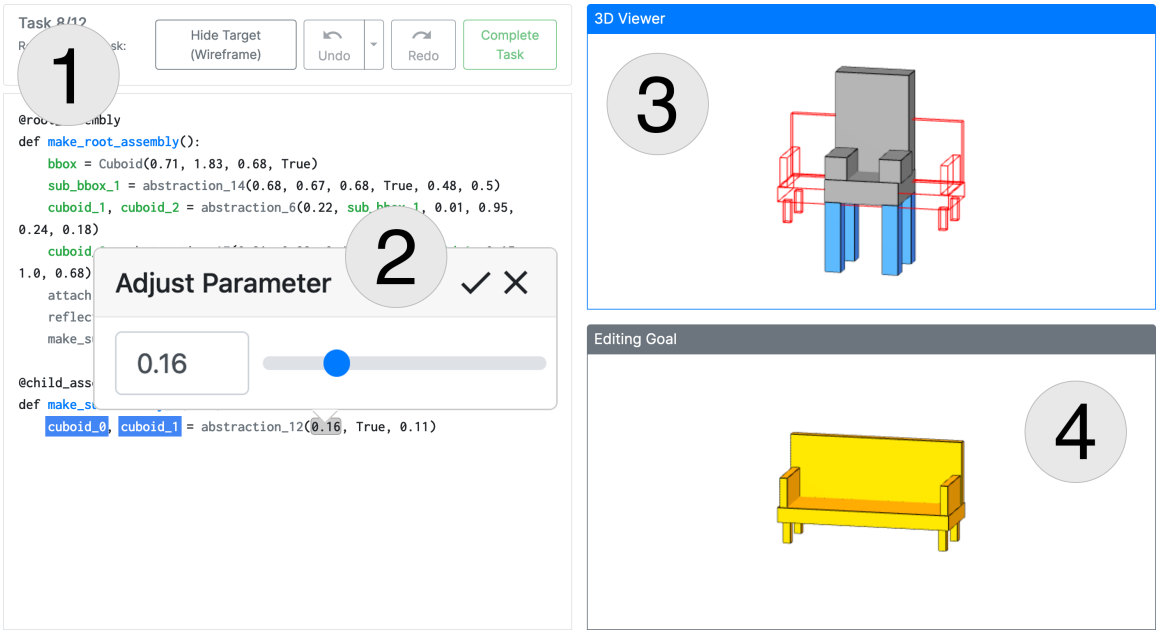

Figure 7.10: A screenshot of our editing interface. The key elements are: (1) A view of the ShapeAssembly program's text. (2) Contextual sliders (enlarged in the figure) that allow the user to edit program parameters. (3) A view of the current program's output. Note the optional wireframe of the target shape and the ability to highlight correspondences between cuboids in the text and the 3D viewer (blue highlights shown). (4) The target shape.

as possible. Fig. 7.10 shows our interactive editing interface. The left panel shows the text of the current ShapeAssembly program. The top-right panel shows the current output shape produced by this program; the bottom-right panel shows the target shape. The cameras of the two shape view panels are synchronized, such that if a user moves the viewpoint of one, the other one follows. The user also has the option of toggling a wireframe display of the target shape overlaid on the current output shape, which can assist with making fine-tuning edits. Finally, in this interface, the text of the program is frozen: users are only allowed to manipulate the continuous programs parameters via contextual slider widgets that appear when a parameter is clicked. See the supplemental video for a demonstration of the interface.

Experiment design Our study asked participants to perform a series of goal-directed editing tasks. To ensure that it was possible to complete these tasks, we selected each target shape by finding a program in our dataset that was identical to the input program up to continuous parameters. We recruited 38 participants, all of whom were university students with some programming background. Participants were randomly divided into one of two conditions: editing programs with ShapeMOD macros or programs without them. Participants were not told the meaning of their assigned condition. First, each participant was shown a short tutorial which

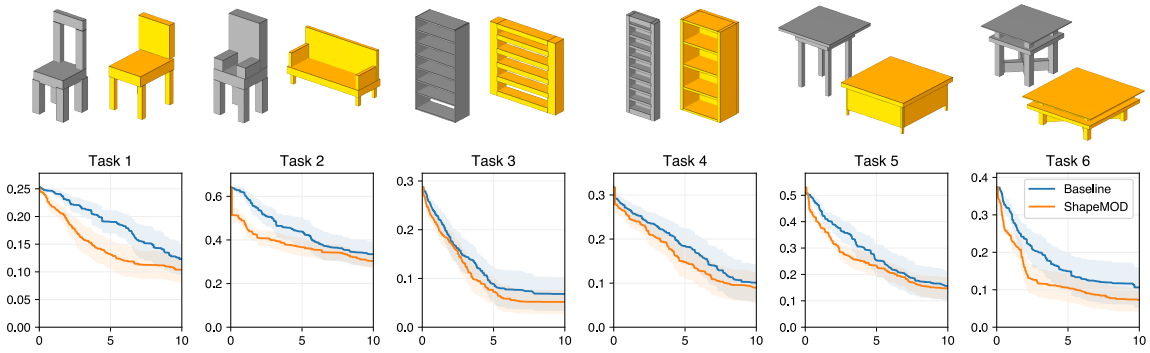

Figure 7.11: Top row: the initial program output shape (gray) and target shape (yellow) for each task in our goal-directed editing study. Bottom row: plots of how quickly participants were able to edit a program's parameters to match the target shape, with 95% confidence intervals shown. The x axis is time elapsed in minutes, while the y axis is the mean of the running minimum of each participant's corner distance to the target shape. In general, participants using ShapeMOD macros more quickly converged to the target shape and achieved a closer fit. To allow users to take breaks between tasks, time starts when the user makes their first edit for each task .

explained the features of ShapeAssembly and allowed them to become familiar with the editing interface. Then, participants completed six editing tasks (two for each of Chair, Table, and Storage). Participants were given 10 minutes to complete each task. After completing these tasks, participants completed an exit survey which asked them to rate the ease of each task (1-5, with 5 being easiest) as well as to provide qualitative feedback about their experience.

Results We first ask the question: how long did it take participants to edit the program to produce a close match to the target shape? Fig. 7.11 plots the running lowest corner distance of the program output to the target shape as a function of task time elapsed, for each of the six study tasks, averaged across participants in each condition. For all tasks, participants using ShapeMOD macros more quickly converged to the target shape.

We also examined the participants' responses to survey questions. Fig. 7.12 shows the ease rating given to each task, averaged across participants in each condition. For most tasks, participants using ShapeMOD macros rated the task as slightly easier to complete.

7.4.5 Cross-category Macro Discovery

We also wondered: can one discover useful macros from a dataset consisting of multiple categories of shapes? To answer this question, we ran the ShapeMOD algorithm on the union of our Chair, Table, and Storage

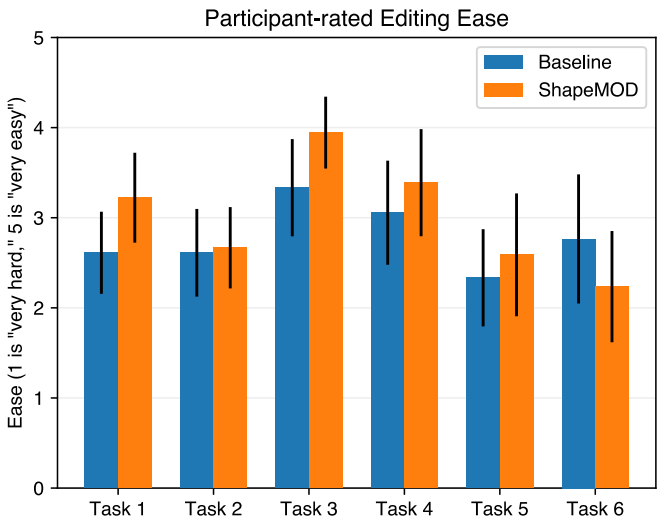

Figure 7.12: Participants in our user study rated the ease of completing each task; here, we plot each task's average difficult rating for each condition (5 = very easy, 1 = very difficult) with 95% confidence intervals shown. Participants using ShapeMOD macros generally rated tasks as easier to complete.

datasets, and report full quantitative results in the supplemental material (Section E). Interestingly, the library of functions discovered across multiple categories led to better program compression statistics, but slightly degraded performance on novel shape generation and program inference tasks, compared with libraries discovered by category specific ShapeMOD runs. These experiments show that for downstream tasks it is slightly better to run ShapeMOD on a per-category basis, although the marginal performance gap provides evidence that the discovered macros can generalize.

7.5 Discussion

We presented ShapeMOD, an algorithm for discovering useful macros across a dataset of shape programs. To our knowledge, ShapeMOD is the first method that discovers common abstractions from a set of imperative programs with relationships between continuous variables. The macros ShapeMOD finds significantly compress the input programs, and these compressed programs lead to better results when used to train models for generating shape structures and inferring shape programs from point clouds. We also conducted a user study which showed that compressed programs allow for more efficient shape program editing.

Limitations The abstractions that ShapeMOD currently considers when proposing macros are relatively simple refactorings of free parameters (e.g. into constants or expressions of other variables).

As mentioned in Section 7.3, ShapeMOD's integration step is intractable to solve optimally. But even the greedy approximation we use can be slow for large collections of shape programs. The major computational bottleneck is the cost of finding optimal programs $z^*(\mathcal{D}, \mathcal{L})$.

While ShapeMOD finds macros that are useful across shape program collections, it does not give them semantic names. In fact, some users in our editing study found the base ShapeAssembly functions easier to work with than the macros for this reason (even though they edited more efficiently with the macros).

Chapter 8

Discovering Abstractions for Visual Programs from Unstructured Primitives

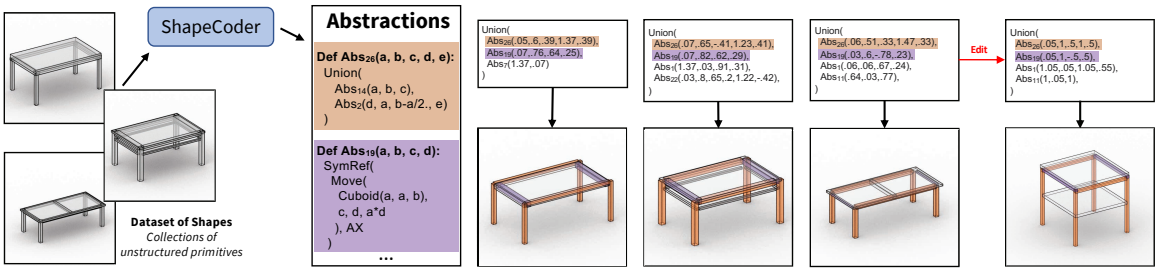

Figure 8.1: ShapeCoder automatically discovers abstraction functions, and infers visual programs that use these abstractions, to compactly explain an input dataset of shapes represented with unstructured primitives. For example, the orange abstraction uses only five parameters to encode a distribution of 4-legged table bases with adjoining horizontal support bars.

In Chapter 7, we demonstrated that not all visual programs are equally useful. Well-structured programs that capture and constrain properties of the visual data they represent typically benefit downstream applications (e.g. editing, generation, analysis). On the other hand, badly written programs lose this advantage. For instance, given an input visual scene composed of a collection of primitives, a visual program that simply unions instantiated primitives together might achieve a perfect reconstruction, but would lose all of the aforementioned benefits of the underlying representation. The functions a DSL contains influences the types of programs it can represent, and access to a 'good' collection of functions is often a prerequisite for finding

well-structured programs. Abstraction functions that extract out common patterns of structural and parametric use for a particular domain, can significantly improve visual program quality, but these types of programs (and their abstractions) are hard to obtain without expert manual design.

In this chapter, we present ShapeCoder, a method that is able to discover useful abstractions for visual data under relaxed assumptions. ShapeCoder consumes a base DSL and a dataset of shapes represented as collections of primitives without any additional annotations. It discovers a collection of abstraction functions (a library) over the base DSL that is tailored to the input distribution. It uses the discovered library to find programs with abstractions that explain the shapes from the dataset (Figure 8.1).

Our approach is inspired by, and improves upon, other abstraction discovery approaches, especially DreamCoder [42] and ShapeMOD (Chapter 7). ShapeMOD can discover abstractions that extract out meaningful relationships in terms of both parametric expressions and program structure. Yet, it does not solve the problem completely. ShapeMOD is able to find these abstractions under fairly stringent input assumptions: it requires a collection of imperative programs as input, as its integration stage relies on enumerative search over a limited, curated subset of possible program line-orderings. ShapeCoder shares the same goals as ShapeMOD, but aims to discover useful abstractions while making much weaker assumptions: it does not assume access to ground-truth programs, canonical line-orderings, or hierarchy decompositions. Instead ShapeCoder takes in a dataset where each shape is expressed as an unordered set of primitives. Discovering abstractions under these assumptions requires both developing logic to infer programs that explain the input shapes, along with extending the abstraction phase so that it is able to reason over arbitrary reorderings of the inferred programs. We solve the latter problem through the use of e-graphs and a conditional rewrite scheme. We provide a more detailed discussion comparing ShapeCoder and DreamCoder in Section 8.6.1

We run ShapeCoder over multiple visual domains, and demonstrate that across all domains ShapeCoder finds abstractions that dramatically simplify the input datasets by discovering meaningful parametric and structural relationships. With respect to an objective function that tracks how well the input dataset has been abstracted, we find that ShapeCoder significantly outperforms ShapeMOD (even when given access to our wake phase) and DreamCoder (which fails to converge without a curriculum of tasks). In a series of ablation experiments, we justify the design decisions of our method, and demonstrate the importance of our conditional rewrite scheme and bottom-up recognition network. Finally, we investigate combining our approach with methods that automatically convert 3D shapes into primitives in an unsupervised fashion, allowing us to discover programs and abstraction functions directly from ‘in the wild’ 3D meshes [16]. In this setting, we observe ShapeCoder still discovers interesting, high-level abstractions, even over noisy,

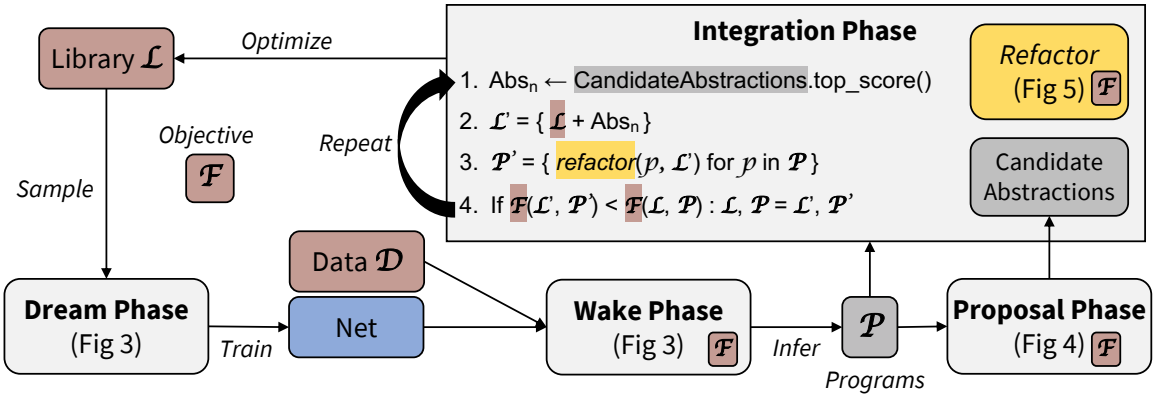

Figure 8.2: **Overview.** ShapeCoder consumes an initial library $\mathcal{L}$, an objective $\mathcal{F}$, and a dataset of shapes $\mathcal{D}$ (brown boxes). Each round of the algorithm iterates through a series of phases that progressively add abstractions into $\mathcal{L}$ to improve $\mathcal{F}$. A **dream** phase trains a recognition network by sampling from $\mathcal{L}$. A **wake** phase infers programs for shapes in $\mathcal{D}$. A **proposal** phase produces candidate abstractions. An **integration** phase uses a *refactor* operation to decide when these abstractions should be added into $\mathcal{L}$.

inconsistent primitive decompositions.

We provide code for our method at https://github.com/rkjones4/ShapeCoder .

8.1 Overview

ShapeCoder automatically discovers a library of abstraction functions tailored for an input dataset of shapes. It takes the following as input: a library $\mathcal{L}$ describing a functional domain-specific language, a dataset of shapes $\mathcal{D}$, and an objective function $\mathcal{F}$. Each $d \in \mathcal{D}$ is represented as a collection of unstructured primitives, and we assume that there exists some program expansion of $\mathcal{L}$, z, such that executing $z$would recreate d.

ShapeCoder's goal is to minimize $\mathcal{F}$ (Section 8.1.1), which expresses a trade-off between how well-suited $\mathcal{L}$ is for $\mathcal{D}$ (program complexity) and how many abstractions functions have been added to $\mathcal{L}$ (library complexity). We break this task into multiple steps that each tackle a tractable sub-problem. We depict the distinct phases of ShapeCoder in Figure 8.2. The *dream* phase (Section 8.2.2) samples scenes from $\mathcal{L}$ to train a program recognition network. The *wake* phase (Section 8.2.3) uses this network to infer programs $\mathcal{P}$ that recreate shapes in $\mathcal{D}$. The *proposal* phase (Section 8.3.1) consumes $\mathcal{P}$ as input, and generates candidate abstraction functions. Finally, the *integration* phase (Section 8.3.2) considers proposed candidate abstractions and finds modified versions of $\mathcal{L}$ to improve $\mathcal{F}$, which can be passed in to a subsequent *dream* phase. Of note, the integration phase uses a *refactor* function (Section 8.4) to find minimal cost equivalent programs under different libraries in a tractable manner through use of e-graphs and a novel conditional rewriting scheme.

In the following sections, we walk-through these various stages, where examples in the text and figures use programs from a toy 2D grammar for rectilinear shapes (Appendix F.1). Further implementation details are provided in Appendix F.2.

8.1.1 Optimization Objective $\mathcal{F}$

ShapeCoder's objective function $\mathcal{F}$ takes in two arguments: a library $\mathcal{L}$ and a collection of programs from $\mathcal{L}$ that correspond with a shape dataset $\mathcal{D}$. $\mathcal{F}$ measures the trade-off between two competing terms: the complexity of $\mathcal{L}$ and $\mathcal{P}$.

The complexity of each $z \in \mathcal{P}$ is computed according to Occam's razor: all else equal, shorter programs are better. We compute program length with a weighted sum of program tokens: if $\mathcal{L}$ has token types T (e.g. booleans, floats, etc.), we allow users to specify a weight λ for each $\tau \in$ T. Further, ShapeCoder employs a geometric error function, *err*, that compares the executed geometry of each $z \in \mathcal{P}$ against its corresponding shape, $d \in \mathcal{D}$. If *err*(z, d) returns a value above a user-defined threshold, $\mathcal{F}$ returns ∞. Otherwise, the error is added into $\mathcal{F}$ with weight λ_e.

Library complexity can be measured by tracking the number of functions that $\mathcal{L}$ contains. ShapeCoder allows users to specify a function weighting scheme, ω. ω consumes a function f from $\mathcal{L}$ and returns a value in the range (0, ∞). Lower ω values make it easier to add f into $\mathcal{L}$. As an example, we find it useful to increase the ω of f according to the number of input parameters f consumes, as this often indicates an overly general pattern.

With this machinery, where τ (z) expresses the number of tokens in $z$that have type τ, we can express ShapeCoder's objective as:

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

8.2 Inferring Visual Programs

While ShapeCoder consumes a shape dataset $\mathcal{D}$ as input, it doesn't know what programs $\mathcal{P}$ from a given library version $\mathcal{L}$ can best represent $d \in \mathcal{D}$. To solve this problem, ShapeCoder uses a program recognition network (Section 8.2.1), trained on randomly sampled programs from $\mathcal{L}$ (dream phase, Section 8.2.2), to infer $\mathcal{P}$ that minimize $\mathcal{F}$ (wake phase, Section 8.2.3).

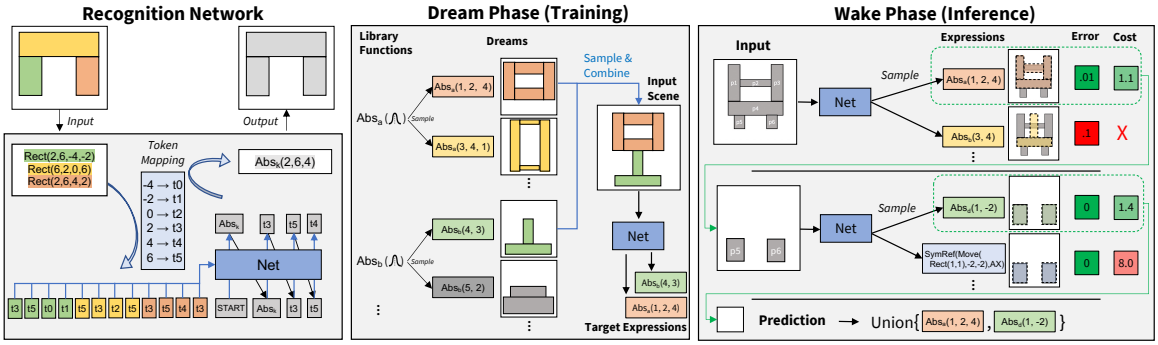

Figure 8.3: **Dream and Wake Phases.** *(Left)* ShapeCoder's recognition network is a Transformer decoder that attends over tokenized input primitives and autoregressively predicts functions and parameterizations. *(Middle)* The dream phase trains the recognition network by sampling expressions from library functions, which are randomly combined together to form (input, target) training pairs. *(Right)* The wake phase uses the recognition network to find programs that explain input shapes. In a series of iterative steps, it samples expressions, chooses the expression that achieves the best cost, and removes covered primitives from the input canvas, until the canvas is empty.

To simplify this search, our recognition network learns to infer partial solutions: expressions from $\mathcal{L}$ that recreate a subset of input primitives. Found expressions are then combined together to form a complete program that explains an input scene. This framing requires that $\mathcal{L}$ contains a combinator operation (e.g. `Union`). To ensure that our search procedure never fails to find *some* solution, we assume access to an analytical procedure for finding expressions in $\mathcal{L}$ that can recreate any primitive in d (e.g. any cuboid can be represented with a scale, rotation, and translation sequence).

8.2.1 Recognition Network

We depict ShapeCoder's recognition network on the left side of Figure 8.3. The recognition network consumes a scene of geometric primitives as input, and aims to output an expression from $\mathcal{L}$ that corresponds with a subset of the input primitives. We implement this network as a Transformer [209] decoder that autoregressively predicts a sequence of tokens from $\mathcal{L}$. The network is conditioned (through causal-masking) on an encoding of the input primitives: if M primitives are each represented with K parameters, the network attends over $K \times M$ conditioning tokens ($M = 3$ and $K = 4$ in the figure example). To convert expressions into token sequences, discrete elements of $\mathcal{L}$ are given a unique index. To tokenize real-valued parameters, we employ a simple mapping procedure: for a given input scene, we take all real values in the primitive parameterizations, bin them through rounding (to 2 decimal places), and sort them to produce a token mapping (light-blue box). This mapping is used to form the conditioning tokens, and converts network predictions back into real values.

8.2.2 Dream Phase

The dream phase trains the recognition network by randomly sampling example scenes from $\mathcal{L}$. We show this process in the middle box of Figure 8.3. To begin the dream phase, for each function $f \in \mathcal{L}$, ShapeCoder creates N_D number of dreams for f. Dreams are generated by sampling random instantiations of each parameter slot of f. Rejection sampling is employed to avoid dreams that create bad geometry by checking easy to enforce properties (geometry outside scene bounds, primitives with negative dimensions, primitives wholly contained by other primitive, etc.).

However, as shapes in $\mathcal{D}$ often contain scenes best explained by more than one function, its not enough to train on function-specific dreams directly. We solve this issue with composite scenes formed by sampling function-specific dreams and combining their output primitives together (blue arrow). If a composite scene was formed by combining K sampled dreams, then we can derive K paired training examples for the recognition network: the input to the network will be the composite scene, and each of the K sampled dreams would be a target output. For instance, given the input scene with orange and green primitives in Figure 8.3, we would train the network to predict both the green and orange expression sequences (i.e. there is a one-to-many mapping). Once this paired data has been assembled, by ensuring that each $f \in \mathcal{L}$ appears in at least N_D target sequences, the recognition network can be trained in a supervised fashion with maximum likelihood updates.

8.2.3 Wake Phase

The wake phase takes an input shape d and aims to infer a program $z$that minimizes $\mathcal{F}$ using the recognition network. We depict this process on the right side of Figure 8.3.

To begin, the scene is initialized to contain the primitives of d. Then the wake phase performs the following steps in an iterative fashion. First the input scene is used to condition the recognition network, which samples a large set of expressions from $\mathcal{L}$ according to its output probabilities, up to a timeout (1 second). For every sampled expression, e, we record its cost: the program complexity of e under $\mathcal{F}$, normalized by the number primitives it explains. Note that if e does not recreate a subset of primitives in the input scene, it will have a high geometric error, and $\mathcal{F}$ will return ∞ (red X in figure). The wake phase chooses the lowest cost e^* (dotted green lines), and removes all primitives it covers from the input scene, which is then fed back into the recognition network. These steps are repeated until the canvas is empty. Once this condition is met, the final program $z$explaining d is formed by applying the combinator operation in $\mathcal{L}$ over each e^* (e.g. the

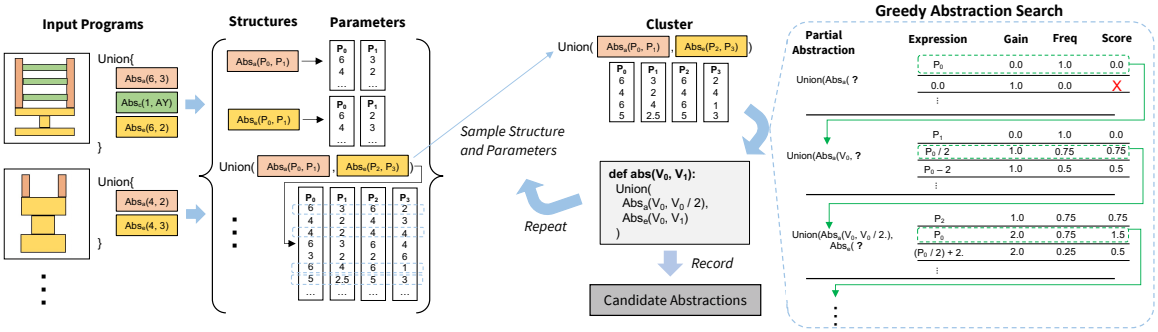

Figure 8.4: **Proposal Phase.** The proposal phase consumes a collection of programs and outputs a set of candidate abstractions. First, possible structures and their parameterizations are recorded from the input programs. Then clusters are formed by sampling a structure and a subset of parameterizations. For each cluster, a greedy abstraction search generates a possible abstraction, which is recorded.

`Union` of the orange and green expressions in the bottom-row). For every input scene, the 'naive' expression for a single primitive under $\mathcal{L}$ is added to the sampled set of expressions, so that a valid solution is guaranteed to be found.

During each ShapeCoder round, the wake phase uses the recognition network to infer a set of programs that explain $\mathcal{D}$. But should we treat these predictions independently? One option is to clear all program entries in $\mathcal{P}$ before every wake phase. However, this would cause ShapeCoder to 'forget' good solutions discovered in previous rounds. Instead, we use the following approach: for round r, $r > 0$, if $\mathcal{P}$ contains previously discovered programs, and $\mathcal{P}_r$ contain programs discovered in round r's wake phase, then we set each entry of $\mathcal{P}$ to be the result of *combine*(z, z_r), where *combine* performs a greedy replacement search to optimize $\mathcal{F}$.

8.3 Proposing and Integrating Abstractions

Together, the dream and wake phases train and use a recognition network to infer a set of programs $\mathcal{P}$ that explain the shapes of the input dataset $\mathcal{D}$. The proposal phase (Section 8.3.1) reasons over $\mathcal{P}$ to suggest candidate abstractions functions, used by the integration phase (Section 8.3.2) to find library variants that improve $\mathcal{F}$.

8.3.1 Proposal Phase

The goal of the proposal phase is to search over $\mathcal{P}$ for abstraction functions that would improve $\mathcal{F}$ if added into $\mathcal{L}$. As this search is computationally intractable to solve globally, ShapeCoder's proposal phase instead

solves more tractable sub-problems (subsets of $\mathcal{P}$), and aggregates local solutions. Figure 8.4 outlines this process.

Identifying Structures and Parameters. As $\mathcal{L}$ is a functional language, generating an abstraction *a* requires two steps: deciding the structure of *a* (what are its sub-functions) and deciding how *a* is parameterized (what input does *a* take, and how are those mapped to its sub-functions). What structures should we consider for possible abstractions? Each program $z \in \mathcal{P}$ is found in the wake phase by combining expressions that solve sub-tasks, so $z$will have no consistent or canonical ordering. Therefore, we would like to factor out expression ordering by considering structural variants over any possible function reordering of each $z \in \mathcal{P}$. However, as the general solution is intractable, we instead consider a limited set of potential abstraction structures: singleton and paired combinations of sub-expressions found in $\mathcal{P}$. We record all such observed structures as keys and how those structures were parameterized as values (see bracketed data structure in figure). We additionally find it useful to apply a simple filtering step that removes infrequently observed structures in $\mathcal{P}$ from this mapping (seen in less than 5% of $\mathcal{P}$).

Cluster Sampling and Search. Once potential structures and their observed parameterizations have been recorded, the proposal phase begins an iterative process. To convert the global problem into a local one, a random structure and a subset of its parameterizations are sampled to form a cluster. Then a greedy search is run over this cluster to find an abstraction *a* that would optimize $\mathcal{F}$. The generated function is recorded into a candidate abstraction data structure that keeps track of a coverage set of $z \in \mathcal{P}$ that could be simplified through applications of *a*. This procedure is repeated many times, and coverage sets are expanded whenever the candidate abstraction data structure receives a previously observed abstraction.

Greedy Abstraction Search We employ a greedy search to find an abstraction *a* for a given cluster (right side Figure 8.4) This search is guided by a *score* function that provides a heuristic estimate of how *a* would improve $\mathcal{F}$ if it were added into $\mathcal{L}$. The *score* of *a* is a product of two terms: the *frequency* and the *gain*. The *frequency* (Freq column in figure) is the percentage of instances in the cluster that *a* could recreate (with the correct parameterization). The *gain* tracks the number of parameters removed from a program z, whenever $z$could be rewritten with *a*, denoted as z_a. For instance, the proposed abstraction in Figure 8.4 would remove two float-typed parameters whenever it could be applied, corresponding with slots P_1 and P_3 in the cluster.

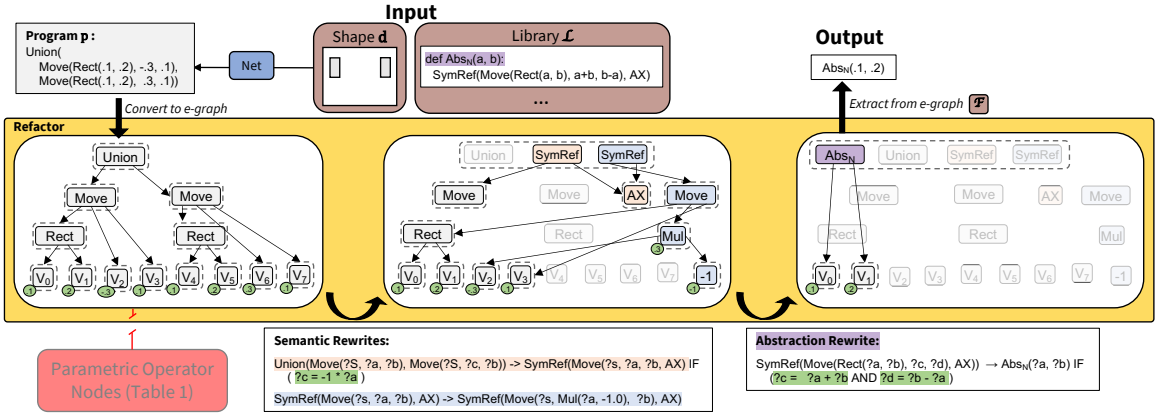

Figure 8.5: **Refactor.** The refactor operation uses e-graphs to identify when abstractions can be applied. Input programs are converted into e-graphs, which are expanded with semantic and library-specific rewrites to uncover lower-cost equivalent expressions that can be extracted. We develop a conditional rewrite scheme that reasons over parametric relationships (green highlights) without adding excessive e-nodes for parametric operators (red box).

Using the weighting from $\mathcal{F}$ (Section 8.1.1), we have:

$$gain(a) = \sum_{\tau \in T} \lambda_\tau * (\tau(z) - \tau(z_a)) \quad .$$

The function sequence in the proposed abstraction is determined by the structure of the sampled cluster, but how should we fill in the parameter slots? For each slot, we consider a set of possible expressions, calculate the *score* of each option, and add the expression with the highest score into the partial abstraction. If the *frequency* is ever zero, then the *score* is voided. For float-typed parameter slots, ShapeCoder produces expressions by iterating over a preference ordering of possible parametric relationships. For discrete-typed parameter slots, a previously instantiated parameter can be reused, or a static value can be assigned. This search always includes defining a new free parameter (e.g. using the parameterization in the sampled cluster) as an option (depicted as the top-row of each step).

8.3.2 Integration Phase

The integration phase takes in a library $\mathcal{L}$, a set of programs $\mathcal{P}$, and candidate abstractions from the proposal phase. It searches for modified version of $\mathcal{L}$ that can be used to refactor $\mathcal{P}$ to improve $\mathcal{F}$. The *refactor* operation (Section 8.4) uses e-graphs to efficiently search for minimal cost equivalent programs under different $\mathcal{L}$ variants.

The integration phase begins by first recording the starting objective value: $\mathcal{F}$ ($\mathcal{L}$, $\mathcal{P}$). It then iterates

through a series of steps in an attempt to greedily improve this value. First, a new library variant $\mathcal{L}$' is formed by sampling a candidate abstraction and adding it into $\mathcal{L}$. The abstraction with the top *score* value is chosen, where the notion of *frequency* is generalized from clusters to all of $\mathcal{P}$. Then a new program set, $\mathcal{P}$', is formed by applying the *refactor* operation over each $z \in \mathcal{P}$ under $\mathcal{L}$'. Finally, if $\mathcal{F}$ ($\mathcal{L}$', $\mathcal{P}$') is better than $\mathcal{F}$ ($\mathcal{L}$, $\mathcal{P}$), both $\mathcal{L}$ and $\mathcal{P}$ are replaced with their modified versions.

Evaluating a modified library $\mathcal{L}$' is expensive, as it requires running the *refactor* operation for every $z \in \mathcal{P}$, so we usually consider a small number, N_A, of top-ranked candidate abstractions during each integration phase. To keep the *score* heuristic as accurate as possible, whenever $\mathcal{L}$', that added a to $\mathcal{L}$, improves $\mathcal{F}$, we check which $z \in \mathcal{P}$ contributed to the *frequency* of a and discount the *frequency* of other abstractions that overlapped on the covered set.

Beyond this greedy search, two other forms of library variants are also considered during the integration phase. Whenever adding a to $\mathcal{L}$ does not improve $\mathcal{F}$, we compute the set of functions whose frequency between $\mathcal{P}$ and $\mathcal{P}$' decreased significantly; call this set f_{dec}. We then consider $\mathcal{L}_{dec}$ = $\{$ $\mathcal{L}$ + a - f_{dec} $\}$ as a library variant. This procedure allows the greedy integration search to avoid a local minima where a would not be added to $\mathcal{L}$ because similar (but worse) functions already exist in $\mathcal{L}$. In addition, to finish the integration phase, we consider library variants where each $f \in \mathcal{L}$ is removed one at time. In all comparisons, the library variant becomes the new default if it improves the objective function. At the end of the integration phase, the $\mathcal{L}$ that achieved the best $\mathcal{F}$ score is then passed into the subsequent dream phase to begin a new ShapeCoder round.

8.4 Refactoring Programs with E-Graphs

ShapeCoder's integration phase evaluates how library variants can be used to compactly represent $\mathcal{P}$ but how does it know when abstractions can be applied? For this task, we use the *refactor* operation: it takes as input a program, z, and aims to find p^*, an equivalent program to $z$that minimizes $\mathcal{F}$. This is a hard search problem, which we make tractable through the use of e-graphs [204] and a conditional rewriting scheme. In the rest of this section, we provide a quick background on e-graphs, and walk-through their role in *refactor* with a running example, depicted in Figure 8.5.

Background on e-graphs. E-graphs are a specialized data structure capable of efficiently representing a large set of equivalent programs. We show an example e-graph in the left call-out of the figure. E-graphs

are made up of e-nodes (solid boxes) and e-classes (dotted boxes) Each e-node is associated with a term from $\mathcal{L}$ and has a pointer (arrows) to e-class children, if that term is a function. Each e-class contains a set of equivalent e-nodes. The root of the e-graph is the e-class that contains the e-node associated with the outermost operator in the input expression (`Union` in the figure).

This representation becomes useful when it is combined with *rewrite* rules. Rewrite rules are domain-specific, pattern matching program transformations that maintain semantic equivalence. For instance, for any ?a and ?b: `Union`(?a, ?b) is equivalent to `Union` (?b, ?a). E-graphs are expanded by iteratively applying rewrite rules to create new e-classes and new e-nodes. These newly created constructs reference existing e-class and e-nodes, allowing the e-graph to represent a large set of equivalent programs in a space-efficient manner. Importantly, e-graphs also provide support for quickly finding minimal cost rewritten versions of a starting expression, by running a greedy recursive algorithm starting at the root e-class.

Refactor Operation. The refactor operation consumes an input program $z$from the wake phase. First, it converts $z$into an e-graph, as depicted in the left call-out of Figure 8.5. In this step, each float-typed token is replaced with an independent variable (V_0 to V_7).

The operation also consumes a library $\mathcal{L}$ as input. It uses $\mathcal{L}$ to source two types of rewrite operations. *Semantic* rewrites express domain-knowledge over base DSL functions and are provided as part of the language definition. For instance, the blue rewrite expresses the following logic: a sub-expression ?s moved to *xy* position (?a, ?b) and reflected over the X axis is equivalent to moving ?s to *xy* position (-1 $\times$?a, ?b) and reflecting it over the X axis. *Abstraction* rewrites correspond with the abstractions in $\mathcal{L}$, where rewrites express the conditions that need to be met in order for the abstraction to be applied. For instance, Abs_N (top-middle) in the input library creates the purple highlighted abstraction rewrite (lower-right).

ShapeCoder expands the e-graph by iteratively applying these rewrite operators. In the middle-frame, the orange rewrite first introduces a new `AX` e-node into a new e-class and a new `SymRef` e-node into the root e-class. Following this, the blue rewrite can be applied, matching on the orange e-nodes, to add the blue highlighted e-nodes. At this point, the purple abstraction rewrite can be applied, and a new Abs_N e-node is added into the root e-class. The refactor operation will continue expanding the e-graph until it is saturated (nothing can be added) or a timeout is reached.

Once the rewrites have expanded the e-graph, we can run an extraction procedure on the root e-class to find the minimum cost expression p^* in the e-graph equivalent to the starting program z. In this example, p^* will be equal to $Abs_N(V_0, V_1)$, which we can rewrite to $Abs_N(.1, .2)$ using the reverse of the parameter

mapping we used to convert the initial program into an e-graph.

Conditional Rewrite Scheme. The above explanation is complete up to one critical step: how do know when rewrites can be applied? E-graphs typically search for structural pattern-based matches, and some semantic rewrites can be included in this framework (e.g. the blue rewrite). However, other rewrites, such as the purple abstraction rewrite, require both structural and parametric matches. For instance, the structural matching requirement to apply Abs_N would be finding some sub-graph of e-classes that matches the pattern of: `SymRef(Move(Rect(?a,?b),?c,?d),AX)`, where ?a through ?d can be filled in with any e-class. Beyond this, applications of Abs_N also require parametric matching with logic expressed in green highlights: the ?c spot must be equal to the sum of the ?a and ?b slots, and the ?d spot must be equal to the ?b slot minus the ?a slot.

How we can support this type of parametric matching? A naive solution would convert parametric constraints into structural ones:
`SymRef(Move(Rect(?a,?b),Add(?a,?b),Sub(?b,?a)),AX)`. The issue with this approach is that it requires adding e-nodes for parametric operations (e.g. `Add` or `Sub`) into the e-graph, before it is known whether or not that e-node will be useful. When there are many input parameters (V_i's) this naive solution will blow up the size of the e-graph, making the refactor operation ineffective. We visualize our choice to avoid this blowup with the disconnected red box in the figure.

ShapeCoder addresses this issue of exploding e-graph size by leveraging a conditional rewrite scheme. Conditional rewrites are rewrite operations that first find structural matches but only make a rewrite application if additional checks pass. In this way, each parametric relationship (green highlights on rewrites) is only evaluated lazily, after a structural match has been identified.

Concretely, in the working example applying the purple rewrite will find the following matches: ?a with V_0, ?b with V_1, ?c with `Mul` (V_2,-1), and ?d with V_3. To check that the parametric relationships hold, we need to know the real value associated with each matched e-class. Then to check a relationship such as ?d = ?b - ?a, we can simply compare the difference in values between V_3 and V_1 - V_0. This check does not enforce exact matches, but rather allows the user to specify a maximum error threshold, allowing us to apply *approximately*-equivalent rewrites, which is typically a limitation of e-graphs.

For some e-classes, finding their associated real-values is trivial: for each e-class associated with a float-typed parameter e-node (V_0 to V_7) we record a mapping between e-class ids and values. This procedure is complicated by the fact that some rewrites create new float-typed nodes (e.g. the blue `Mul` e-class). We handle

Table 8.1: Comparing our conditional rewriting scheme against the naive alternative. The conditional scheme is able to quickly saturate the e-graph (time reported in seconds), even for complex input expressions with many parameters. The naive approach times out when the complexity is too high.

Rewrite Scheme	8 params	16 params	32 params
Naive	.22	2.6	X
Conditional	.01	0.04	2.1

this case by dynamically updating the e-class-to-real-value mapping during all rewrite steps (represented with green-highlights on e-classes), which is a constant time operation. Our conditional rewrite step is just as fast as a non-conditional rewrite step and critically avoids unnecessarily expanding the e-graph with unneeded parametric operator e-nodes. In sum, conditional rewrites provide a dramatic speedup over the naive approach for the kinds of refactoring problems that ShapeCoder typically reasons over (see Table 8.1).

8.5 Results and Evaluation

We run ShapeCoder over distributions of visual shapes represented as collections of unstructured primitives. We describe these domains in Section 8.5.1. In Section 8.5.2, we compare how well the abstractions discovered by ShapeCoder improve the objective function compared to alternative approaches. Our main comparison is against ShapeMOD (Chapter 7). In the main text, we do not include comparisons against DreamCoder [42], as we found it performed poorly on a toy grammar with parametric relationships (see supplemental). In Section 8.5.3, we analyze properties of the discovered abstractions and investigate their generality with a post hoc inference procedure. In Section 8.5.4, we run an ablation experiment to investigate the importance of various algorithm components. In Section 8.5.5 we show another application of our method: inferring visual programs, that contain abstractions, given only a dataset of 3D meshes as input, where we leverage noisy primitives sourced from a pretrained unsupervised cuboid decomposition approach [230]. Finally, in Section 8.5.6 we explore how ShapeCoder's discovered abstractions benefit downstream tasks.

8.5.1 Experimental Domains

For the main result section, we consider domains of 3D shapes. We provide experimental results over a toy dataset of 2D shapes in the supplemental. Our experiments use manufactured objects sourced from PartNet [141], where manual annotations are used to convert each 3D object into an unstructured collection of cuboids, that represent part bounding boxes. We follow past-work in the 3D shape abstraction discovery

Table 8.2: Abstraction discovery performance, measured with objective function $\mathcal{F}$, for libraries of abstractions discovered by different methods.

Category	Method	$\mathcal{F} \Downarrow$	$\lvert\mathcal{L}\rvert$	Num Struct	Num Param
Chair	Input Prims	146.0	6	29	61
	ShapeMOD+Input	109.0	21	16	46
	ShapeMOD+Wake	83.0	21	12	36
	ShapeCoder	**63.6**	33	10	27
Table	Input Prims	125.0	6	25	51
	ShapeMOD+Input	84.2	25	11	34
	ShapeMOD+Wake	69.1	17	10	30
	ShapeCoder	**40.9**	37	8	18
Storage	Input Prims	154.0	6	30	62
	ShapeMOD+Input	119.0	16	20	48
	ShapeMOD+Wake	103.0	10	19	45
	ShapeCoder	**71.3**	31	11	33

space, and run experiments on shapes from the *Chair*, *Table*, and *Storage* categories of PartNet. We perform the cuboid simplification steps outlined in ShapeAssembly (Chapter 3), so that our starting primitive set is the same as that used by ShapeMOD, except we remove all hierarchy and canonical ordering information.

The DSL (Appendix F.1) we use for our experiments has 4 low-level operations: (i) instantiating a primitive (`Cuboid`); (ii) moving a shape (`Move`); (iii) rotating a shape (`Rotate`); and (iv) unioning two shapes together (`Union`). We also provide two mid-level symmetry operations in the base DSL, that correspond with (v) reflectional and (vi) translational symmetry (`SymRef` and `SymTrans`).

8.5.2 Discovering Abstractions

For each PartNet category, we run ShapeCoder for four rounds over 400 shapes from that category. ShapeCoder is implemented in Python and Rust, using PyTorch and Egg, an e-graph library [218]. We run ShapeCoder on a machine with a GeForce RTX 3090 Ti GPU and an Intel i7-11700K CPU, and find that it takes less than 24 hours to finish discovering abstractions for a single category (taking at most 4GB of GPU memory).

Discovering abstractions that improve our objective We report how the abstractions discovered from ShapeCoder impact the objective function we optimize over, in Table 8.2. From left to right, the columns express the objective function score ($\mathcal{F}$, where lower is better), the number of functions that the library contains ($|\mathcal{L}|$), and the average number of operations (*Num Struct*) and parameters (*Num Param*) that are needed to represent the input dataset of shapes using programs that make use of the discovered abstractions.

The top *Input Prims* row for each category conveys the starting objective function value for ShapeCoder. This row reports the cost of using 'naive' programs to cover the primitives of the input shapes, where each

primitive is rotated, moved, and instantiated, whenever that command would have an effect (e.g. moving zero distance would be ignored). The final objective function score found by ShapeCoder, in the bottom rows, is dramatically better than this starting point. For *Chairs*, *Tables*, and *Storage*, the starting objective function value drops by 56%, 67%, and 53%, respectively. This improvement is achieved by adding abstraction functions (2nd column) that remove degrees of freedom needed to represent the shapes of the input set (3rd and 4th columns).

We also compare how ShapeCoder performs against ShapeMOD in this setting. The ShapeMOD algorithm requires a dataset of imperative programs as input, along with the possible ways that the lines of the programs can be ordered. As we lack ground-truth programs for our problem setting, we compare against two versions of ShapeMOD, that attempt to optimize the same objective function as ShapeCoder:

- *ShapeMOD+Input:* We take the 'naive' programs that can be directly parsed from the input collection of primitives, and provide this as input to ShapeMOD.
- *ShapeMOD+Wake:* We take the output from ShapeCoder's first wake phase as the input to ShapeMOD. Note that the only 'non-trivial' functions in the library for the first wake phase are the symmetry operations, roughly equivalent to running symmetry detection on the 'naive' programs.

For both program datasets, we have no way of knowing how the various expressions (e.g. sub-shapes combined through `Union`) should be ordered, so we pass a random subset of all possible valid orderings to ShapeMOD, as without limiting the set of orders ShapeMOD takes prohibitively long to run (see supplemental).

Comparing ShapeMOD variants and ShapeCoder in Table 8.2, it is clear that ShapeCoder finds abstractions that significantly improve the objective function over those found by ShapeMOD. While ShapeCoder's wake phase provides a better starting point than the 'naive' programs, in either case, the complexity of the input programs is too high for ShapeMOD to handle-well when canonical orderings and hierarchy annotations are absent.

We also compare ShapeCoder against approaches that operate over single programs, like Szalinski [145]. Szalinski also uses e-graphs in the context of visual programs, and while its fixed rewrite rules are well-suited for simplifying a single heuristically-inferred CAD program of a mechanical object, we found that these rules did not significantly compress shape programs in our domain: Szalinksi's rewrites improved our objective function from 146 to 131, for chairs, whereas ShapeCoder reached 63.

Table 8.3: We measure the generality of the abstractions that ShapeCoder discovers by comparing how well it can compress shapes (objective function $\mathcal{F}$) from a held-out set (Val) with post hoc inference (PHI) compared with the programs it discovers during normal operation (top-row).

Shape Set	Inference Method	$\mathcal{F}$	Abs Count
Train	ShapeCoder	63.6	4.31
Train	PHI	67.5	4.67
Val	PHI	70.6	4.77

8.5.3 Analysis of Discovered Abstractions

We visualize a subset of abstractions discovered by ShapeCoder when run over PartNet shapes in Figure 8.6. The recognition network learns how to use these abstractions to explain shapes in the input dataset (first three columns). Programs rewritten with these abstractions can be edited to create new shapes, as we show in the fourth column. The discovered abstractions contain many desirable properties: they capture diverse geometric expressions and constrain many extraneous degrees of freedom by introducing parametric relationships. Abstractions in later rounds of ShapeCoder can reference previously discovered abstractions in sub-function calls, forming a nesting hierarchy of abstractions. In extreme cases, ShapeCoder can even discover single abstractions that explain entire input shapes, e.g., in the first and third columns of the top-row, a single abstraction function, that consumes five input parameters can output an entire chair when executed. Access to these types of abstractions can even be helpful for structural analysis of 3D shapes. For instance, the shown abstraction for tables (2nd row) is consistently mapped to the same semantic part (regular table legs), even though the part has a wide range of possible output geometries. For each abstraction, we also visualize a subset of random parameterizations (i.e. dreams), to give a sense of the possible output space described by each function.

Post hoc inference During the course of abstraction discovery, ShapeCoder finds programs that use abstractions to explain the shapes in its input dataset. We investigate if these abstractions can generalize to shapes from the same distribution that were not included in its optimization procedure. We leverage ShapeCoder's recognition network to find programs that explain shapes that were not included in the 'training' phase of abstraction discovery. We run the wake phase over these shapes, to find programs that explain the input set of primitives. These programs are then passed through the *refactor* operation, to see if any of the library rewrites can further improve the program.

We present the results of this *post hoc inference* (PHI) procedure in Table 8.3, for shapes from the *Chair*

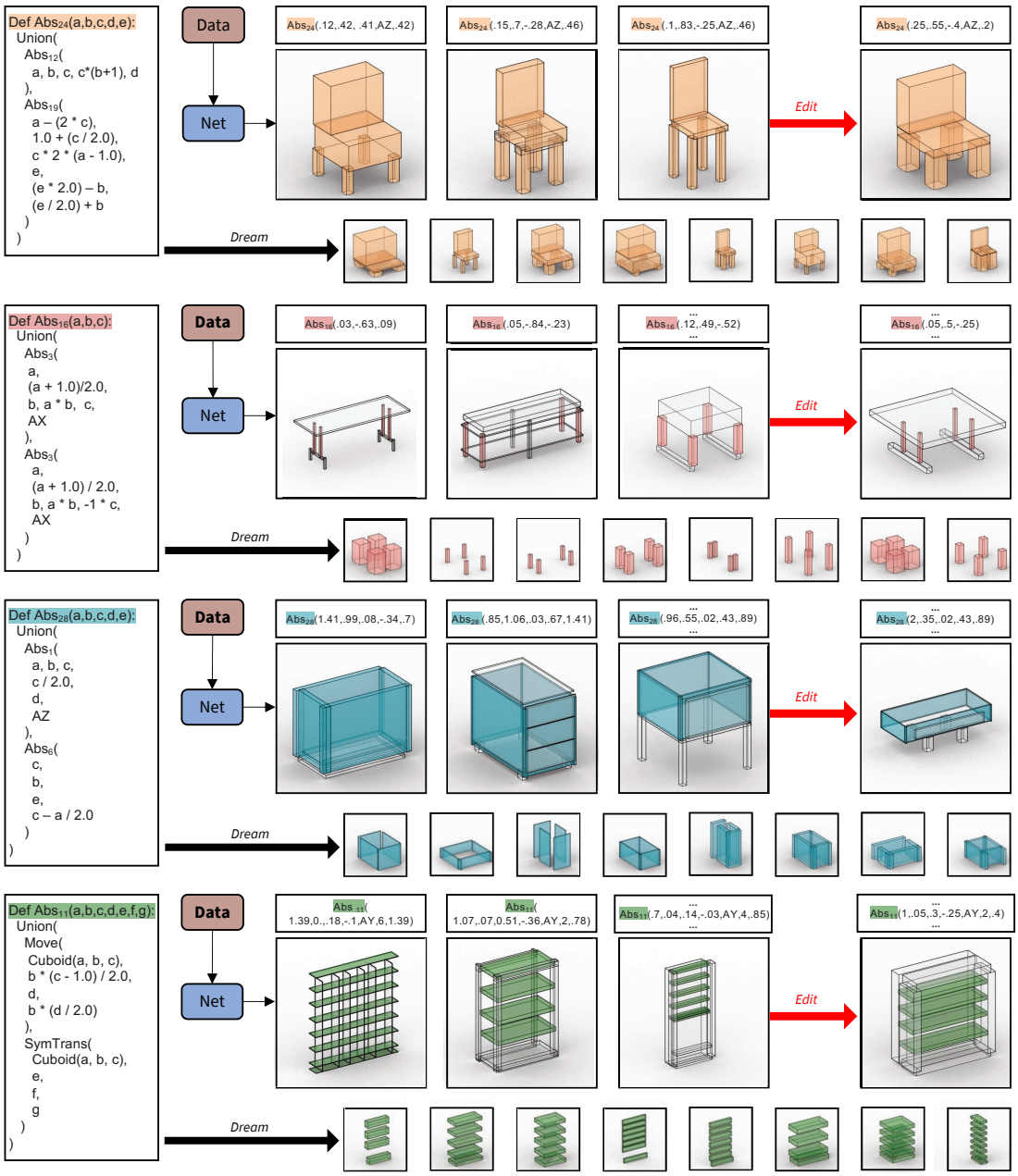

Figure 8.6: Qualitative examples of discovered abstractions. We show one abstraction each for *Chair* and *Table*, and two abstractions for *Storage* furniture. The abstraction code is shown on the left, followed by three different usages of the abstraction in our shape dataset discovered by ShapeCoder. In the right-most column, we manually edit the discovered program to create a new shape. Along the bottom, we visualize randomly sampled dreams.

Table 8.4: *(Left)* Ablating design decisions of ShapeCoder by tracking objective function improvement (see condition details in Section 8.5.4). Our default configuration (bottom) performs best. *(Right)* Measuring output execution validity (with Frechet Distance) under increasing perturbations (Noise Level) for programs with, or without, abstractions. Abstractions help keep shapes 'in distribution' under parameter edits.

Condition	$\mathcal{F} \Downarrow$
No Abstraction	104.9
Single Iter	81.6
No Dream+Wake	99.0
No Semantic Rws	75.2
No Conditional Rws	100.0
No Abs Preferences	70.7
ShapeCoder	**63.6**

Noise Level	No Abs	With Abs
0.1	**8**	**8**
0.2	18	**13**
0.3	40	**27**
0.4	88	**48**
0.5	157	**84**

category of PartNet. The top row of this table shows the objective function values, and the average number of abstraction-uses, for the programs that were iteratively built up during ShapeCoder 'training' (e.g., abstraction discovery). In the middle row, we take this same set of shapes, 'forget' the programs discovered during abstraction discovery, and run the PHI procedure, which aims to infer programs from scratch. In the last row, we run PHI on validation shapes, never before seen by ShapeCoder. While doing inference post hoc is slightly worse than iteratively discovering programs over multiple rounds, the difference between running PHI over the 'training' shapes and 'validation' shapes, is relatively small. This fact, along with the consistently high-values in the abstraction usage column, indicates that many of the abstractions that ShapeCoder discovers can generalize beyond the dataset of shapes it optimizes over.

8.5.4 ShapeCoder Ablations

To evaluate the design decisions behind ShapeCoder, we run an ablation experiment, by tracking how the removal of different components of our method impacts the types of abstractions we discover, and how those abstractions impact the optimization of the objective function. We consider the following ablation conditions:

- *No Abstraction:* We report the results of running just the wake phase, once, without an abstraction phase.
- *Single Iter:* We only run ShapeCoder for a single round.
- *No Dream+Wake:* We run multiple rounds of ShapeCoder without access to a recognition network. Instead 'naive' programs are used to initialize the algorithm.
- *No Semantic Rws:* We remove all of the semantic rewrites associated with our base DSL in the refactor operation.

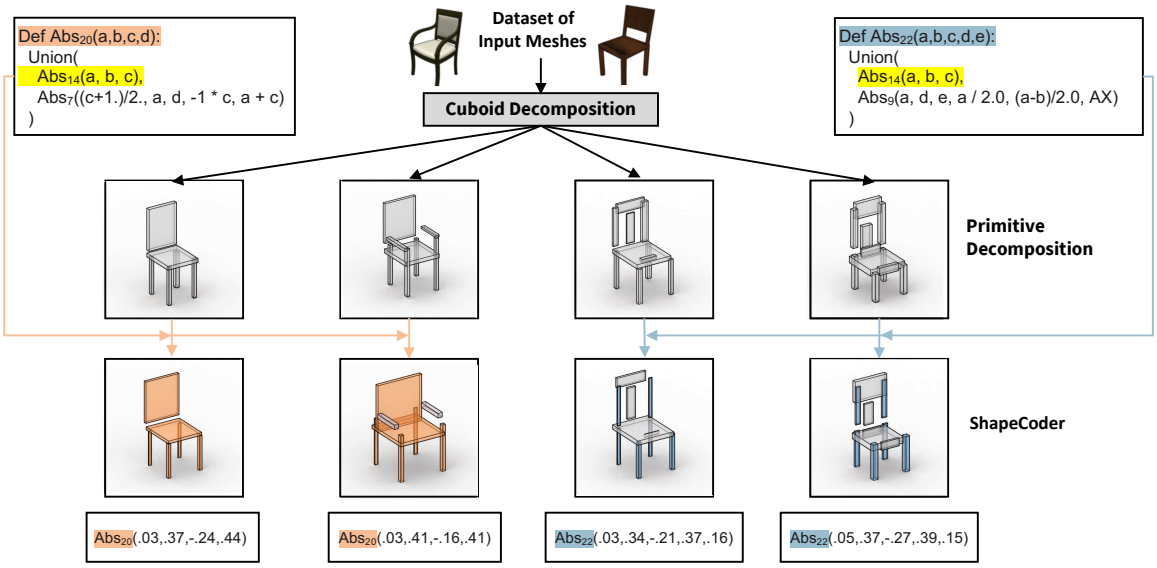

Figure 8.7: We leverage an unsupervised primitive decomposition approach [230] to run ShapeCoder over datasets of 3D meshes. Even on these noisy primitive decompositions, our method still finds high-level, useful abstractions that capture meaningful degrees of shape variation. Interestingly, the two top-level abstractions we show, in orange and blue, both make use of the same abstraction sub-function (highlighted in yellow) to create a four-leg base.

- *No Conditional Rws:* We replace our conditional rewriting scheme with the 'naive' approach described in Section 8.4.
- *No Abs Preferences:* We remove the preference weighting ω, described in Section 8.1.1.

We report how these different variants perform in Table 8.4, left, using shapes from the *Chair* category of PartNet. All ablation conditions lead to worse optimization behavior than our default configuration (bottom row). Without an abstraction phase, the programs returned from wake can't leverage higher-order functions. With just a single iteration of ShapeCoder, hierarchical abstractions can't be discovered, and the wake phase can't learn to apply the discovered abstractions more broadly. When the abstraction phase is run without a dream or wake phase, the method runs into a similar problem, where the abstractions can be underutilized, and won't be integrated into all of the shapes that they could be used to represent. The semantic rewrites allow e-graphs to represent a large set of equivalent programs that we efficiently search over during refactoring; when we don't consider this large set of equivalent programs, we, once again, under-apply proposed abstractions. The importance of our conditional rewrite scheme is made evident by the no conditional rewrite ablation: within the computational budget allotted for this ablation experiment (3 days) the version of ShapeCoder that used the 'naive' rewrite scheme failed to finish a complete abstraction phase. As such, we report its objective

function value at this 3-day cut-off. Finally, our preference weighting scheme helps ShapeCoder avoid local minima: mostly by down-weighting obviously bad (e.g. too constrained or too general) candidate abstraction functions.

8.5.5 Discovering Abstractions from Unstructured Shapes

As an illustrative application of ShapeCoder, we investigate its ability to jointly discover a library of abstraction functions and programs that use those abstractions, when run over a dataset of 3D meshes. To source this kind of input data, we use a method that performs unsupervised cuboid decomposition of 3D shapes [230]. Specifically, we employ this approach to convert sets of ShapeNet meshes into arrangements of unstructured, noisy primitives – a data format that ShapeCoder can reason over. We provide details of this data preprocessing in Appendix F.2.8

Similar to the experiments in Section 8.5.2, we construct a dataset of 400 shapes, with primitives produced by this unsupervised algorithm. We run ShapeCoder over a dataset of chairs sourced from ShapeNet [16] for three rounds and show results of some of the discovered abstractions in Figure 8.7. Even though the primitive decompositions that ShapeCoder receives are noisy and irregular, it still manages to discover a collection of meaningful abstraction functions that expose higher-order properties and can be applied across instances of the input distribution. For instance, the discovered Abs_{20}, captures the same fundamental chair structure found by ShapeCoder when run over PartNet annotations (Abs_{24}, Figure 8.6). In fact, over the course of 3 rounds, ShapeCoder improves the objective function score by 61% ($140 \rightarrow 53.9$), which is similar to the quantitative improvement observed when ShapeCoder operates over clean, manually annotated parts. These results are promising, and indicate that systems like ShapeCoder can be used to discover useful high-level programmatic representations of complex visual phenomena, without reliance on manual annotations.

8.5.6 Downstream Benefits of Abstractions

In this section, we investigate how ShapeCoder's discovered abstractions can benefit downstream applications with two experiments: maintaining validity under perturbations and novel shape synthesis.

Maintaining validity under perturbation As we aim to discover abstractions that remove extraneous degrees of freedom, we can evaluate success by perturbing degrees of freedom in shape programs, and checking whether they 'stay in distribution'. We take two shape program datasets, where programs are written with or without abstractions, and perturb their parameters under different noise levels. Specifically, the noise level

modulates the standard deviation of Gaussian noise distributions fit to each parameter slot of each DSL function. For each perturbed set of programs, we measure how similar their output executions are to a validation set with Frechet Distance (FD) in the feature space of a pretrained model. We report results of this experiment in Table 8.4, right. We find that rewriting programs with abstractions discovered by ShapeCoder helps to keep shapes 'in distribution' under parameters perturbations, which is an important property for goal-directed editing tasks.

Novel Shape Synthesis We evaluate if generative models that learn to write novel shape-programs benefit from training over programs that have been rewritten with discovered abstractions. For this experiment, we use the PHI procedure (Section 8.5.3) to construct a dataset of 3600 chair-programs written with ShapeCoder discovered abstractions. We use this dataset to train an auto-regressive network, a Transformer decoder, that learns to generate sub-programs conditioned on a canvas that tracks the execution output of previously predicted program parts (Appendix F.2.9). To synthesize novel shapes, the network starts with a blank canvas, and then gradually builds up a complex program by iteratively sampling expressions, and adding their outputs to the canvas, until a `STOP` token is predicted.

We visualize outputs of this model in Figure 8.8. Qualitatively, we find that this model can create new shapes not observed from the training set, that clearly stay within the training-distribution. Quantitatively, we compare the outputs of this model against an ablated version that trains over programs without abstractions, and find that learning over programs written with abstractions improves Frechet Distance (against a validation set) from 17.1 to 13.8, a 19% improvement. Moreover, generative models of visual programs that learn over abstractions are particularly attractive, because the programs they output have less extraneous degrees of freedom, and will be better suited for downstream tasks.

8.6 Discussion

We have presented ShapeCoder, a system capable of discovering visual program abstractions in a collection of shapes represented as unstructured primitives. Our method does not require any additional supervision such as ground truth programs, any specific ordering of program operations, or any program curriculum. We have shown that ShapeCoder discovers high-level abstractions, that result in significant compression, on domains that other state-of-the-art methods cannot handle. ShapeCoder can find programs that use these abstractions to explain shapes not observed during optimization, compactly. Finally, we demonstrated the

Figure 8.8: Sampled programs (top) from a generative model that writes programs containing abstractions, along with nearest neighbors (bottom).

flexibility of ShapeCoder by showing that it can discover useful abstractions, that capture meaningful degrees of freedom when run over noisy primitive decompositions produced by an unsupervised method.

8.6.1 Relation with DreamCoder

DreamCoder proposes a system that jointly discovers abstractions and performs program induction over arbitrary functional programming languages [42]. At its core DreamCoder uses three phases to perform this hard task. A dream phase samples random programs from a library (optionally augmented with abstractions). A wake phase trains a recognition network to infer programs based on the dream samples. An abstraction phase looks over a corpus of returned programs from the wake phase, and proposes and integrates abstractions that improve an objective function. The objective function trade-offs program likelihood under the library with the complexity of the library.

Similar to this framing, ShapeCoder employs an iterative procedure with interleaved phases (dream, wake, proposal, and integration). These phases are run repeatedly, gradually discovering a library of abstraction functions that minimize a compression-based objective function. The dream phase trains a recognition network, which is used by the wake phase to infer visual programs that explain input shapes. Critically, we design our recognition network in a way that allows it to find partial solutions for difficult input scenes. This allows ShapeCoder to still work on input datasets that lack a curriculum of examples (some inputs are easy to solve under the base DSL).

While DreamCoder's generality allows it to effectively scale across a wide-variety of program inference tasks, its abstractions are purely structural, treating real-valued program components as discretizations. This means that it is not well-suited for shapes (or other visual domains) where ideally abstractions would capture both complex parametric and structural relationships. Another challenge of applying DreamCoder to shape programs is that its iterative procedure is reliant on a curriculum to solve tasks: all of its stages (dreaming, waking, abstraction) rely on the assumption that solutions to at least some of the input tasks have a high probability under the current library functions. When the input tasks form a curriculum (e.g. some tasks are very easy to solve under the base DSL), then this procedure works very nicely, gradually discovering more and more abstractions that allow it to solve increasingly complex VPI tasks. Unfortunately, when this curriculum assumption is broken, DreamCoder can fail to discover any programs or abstractions for a given domain. Based on these properties, we ran investigations of how DreamCoder fairs on a simple grammar with parametric relationships, and found that it wasn't able to discover the kinds of abstractions that ShapeCoder is able to find. We provide details in the Appendix F.4.

8.6.2 ShapeCoder Limitations

While ShapeCoder is the first method to discover non-trivial program abstractions directly from unstructured primitives, it does have some important limitations:

(i) Redundant abstractions. We find multiple abstractions that explain the same concept. While these can be seen as structural variations for the same semantic concept (e.g. pedestal chair bases and four-leg chair bases), the abstracted programs can feel redundant for downstream tasks. This is hard to avoid as, at present, we do not 'execute' the programs to compare their geometric output.

(ii) Unsaturated e-graphs. For complicated input expressions, it can be computationally infeasible to fully saturate e-graphs, as they lack the ability to efficiently represent associativity-commutativity constraints. While ShapeCoder doesn't offer a direct solution to this issue, our use of conditional rewrites avoids inserting extraneous parametric operation nodes. This helps to alleviate exponential blowup, and allows ShapeCoder to explore a much richer range of possible program structures than prior work. Despite this, we cannot always saturate our e-graphs within the allotted computational budget. This implies that some possibly useful rewrites go unexplored and never get appended to the abstraction library.

(iii) Bottom-up wake network. ShapeCoder's recognition network (used in the wake phase) solves subproblems that are stitched together through combinator operations. A downside of this design decision is that the recognition network must be retrained whenever the library version changes. Further, as the network does not predict an entire program in one-shot, inference can be expensive to run, and there is less consistency in how programs will be inferred across a dataset.

Chapter 9

Designing a Library of Procedural Shape Abstractions with LLMs

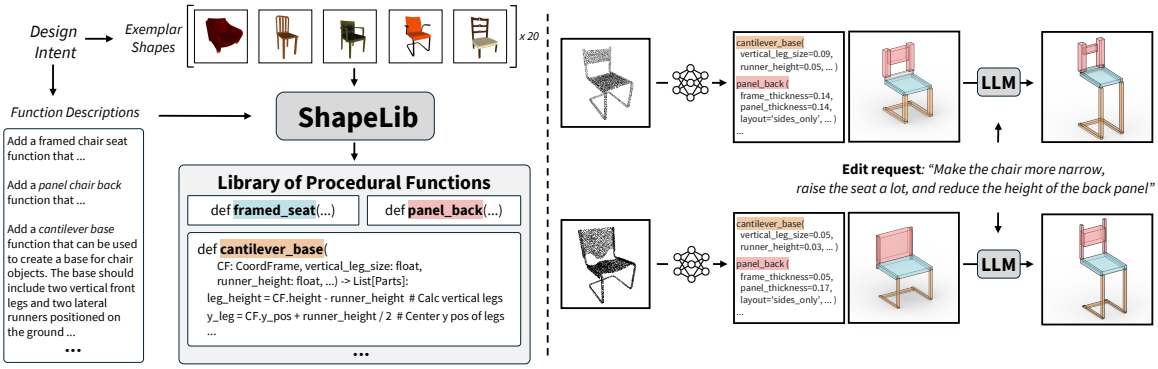

Figure 9.1: ShapeLib guides an LLM to design a library of procedural shape functions from a given set of (20) seed shapes and textual descriptions. Using an LLM prior makes the functions semantically interpretable and easy to edit, while aligning them with the seed shapes specializes the functions to a given domain and reduces LLM hallucinations. The library can be used to train a network for visual program induction that generalizes well beyond the seed shapes.

Methods like ShapeMOD (Chapter 7) and ShapeCoder (Chapter 8) aim to automatically discover good libraries of procedural shape functions. These methods use data-driven approaches to optimize a compression-based objective. They operate in a 'bottom-up' fashion, starting from a base modeling language with elementary functions, and gradually grow their library, in a greedy manner, by defining new and more domain-specific abstraction functions based on how much they help to compress shapes from a large dataset. While these approaches can successfully optimize their compression objective, they base their library development

solely on compressing out common geometric patterns over a large shape dataset, without any semantic 'top-down' guidance. As a result, the functions they produce can only align to shape semantics by chance, making them difficult to interpret and meaningfully manipulate.

As an alternative, we investigate how Large Language Models (LLMs) can help with this procedural language design problem. LLMs have demonstrated remarkable success over a surprisingly diverse range of tasks, from 3D layout synthesis [81] to general code generation [88]. There are reasons to believe they might be useful in helping to design procedural models. They have top-down world knowledge about the semantic relationships of parts within shapes and they are proficient at writing code. Despite these properties, LLMs also have limitations that temper their procedural modeling capabilities. As we demonstrate experimentally, latest frontier LLMs still struggle to understand complex geometric layouts and often misinterpret or misattribute constraints and relations between parametric controls. Their mistakes manifest as hallucinations, leading to implausible geometry or structures that cannot represent assets in existing 3D datasets.

In this Chapter, we introduce ShapeLib, a hybrid system that guides an LLM through the creation of a library of procedural abstraction functions from a specified design intent. An expert user provides this design intent to our system with two modalities: (i) function descriptions in natural language, and (ii) a seed set of exemplar shapes. The two modalities are complementary: the first mode allows the user to specify the kinds of functions they would like to interface with; while the second mode provides geometric references that guide and constrain library development.

ShapeLib breaks the complex library design process into a series of sub-problems. First, we use an LLM to design the library interface with a prompting workflow conditioned on the function descriptions. Next, we task an LLM with proposing applications of these functions to explain shapes from the seed set (from the interface only, without any actual implementations). We then use these proposed applications to automatically formulate input/output examples that guide the LLM to propose implementations of each function. We finalize the library with a validation step that performs a geometric analysis over the proposed function implementations and applications. To apply these functions to represent shapes beyond the seed set, we additionally train a recognition network that learns to map input shapes to output programs written with the library functions. To train this network, we create a synthetic data generator by prompting an LLM with the finalized library implementation and asking it to produce a function that randomly generates an input shape using the abstraction functions. In this way, even starting from only a small seed set, ShapeLib can find programs that use these abstraction functions to explain a much larger collection of shapes (see 9.1).

We evaluate ShapeLib by using it to design libraries of procedural functions over multiple shape categories (`chair`, `table`, `storage`, `lamp`, `faucet`). We find that our method generates functions that (i) adhere to the top-down semantics provided by the natural language descriptions, and (ii) produce geometric outputs that reflect structures observed from the exemplar shapes. Beyond this, we experimentally validate that our discovered library helps us to realize the benefits of representing shapes procedurally along a number of axes. *Generalization (a)*: they are useful for modeling shapes outside of the seed set; *Interpretability (b)*: they are aligned with semantics and expose a small number of parameters that produce predictable edits; *Plausibility (c)*: they constrain outputs to maintain shape semantics under manipulation. We compare against alternative problem framings, and find that our dual modality design intent is crucial for our success. When semantic information from (i) is missing, systems like ShapeCoder find abstractions that improve compression, but lack interpretability and do not maintain plausibility. When reference geometry from (ii) is missing, LLMs design sensible library interfaces, but produce function implementations that can not generalize across shape distributions.

9.1 Overview

ShapeLib guides an LLM through the process of developing a library of procedural functions that matches an input design intent. In our problem framing, we assume that a user has a procedural modeling domain in mind (e.g., a particular category of shapes). The user will communicate their design intent to our system, which is then tasked with producing a fully realized library of abstraction functions that meet our desiderata: (a) they should generalize, (b) they should be interpretable, and (c) they should produce plausible outputs.

Our system receives a number of benefits from the prior knowledge encoded in LLMs. Since LLMs have been trained extensively on human-written code, they are able to author functions with meaningful names and parameters. This exposes an interface that a person can easily work with and understand. However, we also find that LLMs are prone to hallucinate, generating mismatches from 'real' distributions of shapes (e.g., collections of 3D assets).

To overcome this issue, we guide and ground the LLM outputs under the supervision of the user provided design intent, consisting of a textual description and a set of seed shapes. Textual descriptions of desired function properties help constrain the interface design, prompting the semantic prior of the LLM to attune towards a particular modeling task. Each seed set we consider is composed of twenty 3D shapes with part-level semantic segmentations and textured renders. Our system validates the plausibility of its productions by

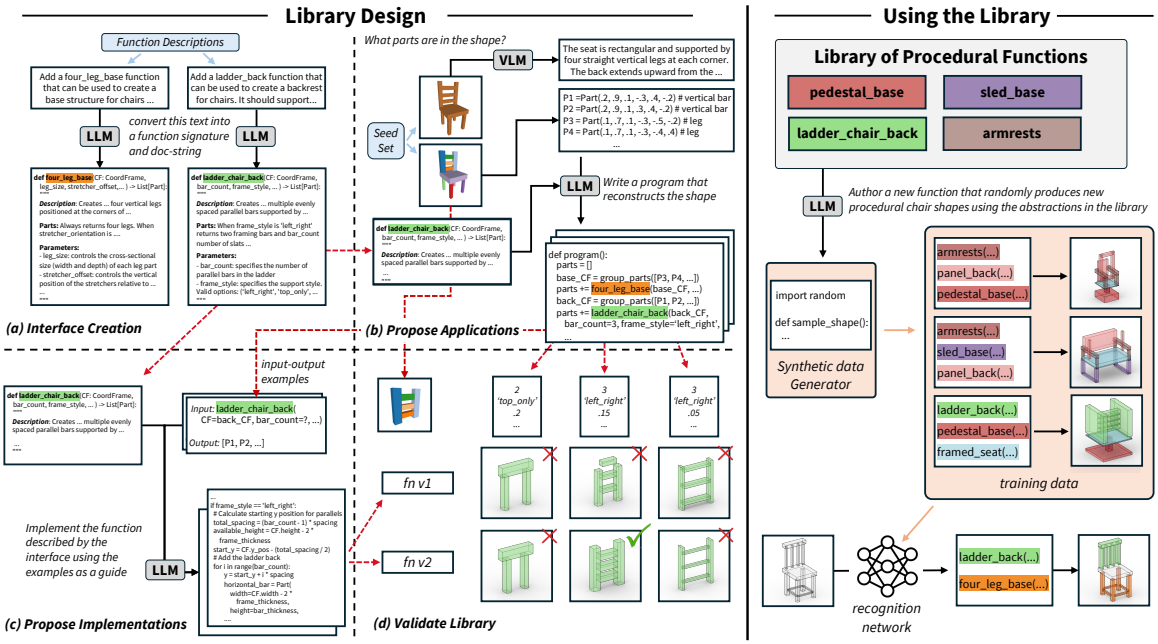

Figure 9.2: Method overview. We design a function library in four steps, starting from a user intent (light blue) that consists of function descriptions and a set of seed shapes. First, (a) we prompt an LLM to create function interfaces that define parameters and annotate the function's purpose. Then, (b) the LLM is prompted to propose multiple applications of the functions that reconstruct the seed shapes. Next, (c) we use this information to guide the LLM to propose multiple function implementations. The library is finalized with a validation step (d) that searches for pairs of applications and implementations that best reconstruct the seed shapes. We can use the library to extend beyond the seed shapes by guiding the LLM to author a synthetic data generator with the library functions, and using the resulting paired data to train a recognition network for visual program induction.

searching for function implementations and applications that can explain sub-structures in these exemplars.

In the following, we explain how ShapeLib solves this problem. In Section 9.2, we describe how we convert design intent into a fully realized library of abstraction functions. In Section 9.3, we describe how we can expand the usage of this library beyond the seed set by training a recognition network on synthetic data.

9.2 Library Design

ShapeLib converts design intent into a library of functions through a series of steps, which we depict in Figure 9.2. The interface creation step converts function descriptions into a library interface (Section 9.2.1). The application proposal step identifies which library functions should model which seed set shapes (Section 9.2.2). The implementation proposal step generates candidate function implementations (Section 9.2.3).

The library is then finalized with a validation step that checks combinations of proposed function applications and implementations against seed set examples (Section 9.2.4).

9.2.1 Interface Creation

ShapeLib first converts user function descriptions into a library interface (Fig. 9.2, a). We prompt an LLM to produce a structured interface, where for each function it produces a typed signature and an accompanying doc-string.

We provide the LLM with two default classes: a 'Part' class that creates primitives that abstract detailed geometry and a 'CoordFrame' class that defines a local bounding volume. Our prompt contains task instructions and in-context expert demonstrations sourced from different categories. By default, we use axis-aligned cuboid primitives, though this design decision could be generalized by modifying prompt instructions and examples.

The LLM produces function signatures that expose parametric handles, e.g. the numbers of bars in a ladder back or the height of base runner. Each function is instructed to take in a special first parameter, *CF*, a 'CoordFrame' that specifies the expected extents of the functions outputs. Functions are typed so that they return a List of 'Part' objects.

Through our in-context examples and instructions, we prompt the doc-string to have a particular structure. First, it defines a *description* field to explain the high-level goals of the function. Then, it defines a *parts* field, that specifies what parts should be produced depending on the input parameters. Finally, it defines a *parameter* field, that explains how they should affect the output structure. This interface is then used to guide the library development.

9.2.2 Proposing Function Applications

As LLMs are prone to hallucinate, we do not directly implement each function following the prior step. Instead, we would like to ground each function implementation by referencing structures from the seed set. To find such references, we propose programs that apply library functions that explain exemplar shapes (Fig. 9.2, b).

This step begins by sampling a shape from the seed set. We ask a VLM to describe the parts that is sees from a render of the shape. We also convert the 3D semantic part annotations into a list of labeled 'Part' objects. We combine these inputs together, and task an LLM with deciding what parts should be explained by

which library functions (even though these functions lack implementations). The LLM outputs this decision by authoring a 'program()' function that proposes library function applications (along with parameters). We ask the LLM to use a special 'group_parts' function when constructing this program, that consumes a list of input 'Part' objects and returns a bounding 'CoordFrame' object. In this way, the 'program' provides information about which parts of the input shape should be explained by which library functions.

As we later demonstrate empirically, the accuracy of individual LLM calls has a high variance which makes them hard to trust. Therefore, instead of finding a single program for each shape, we run this procedure K times for each shape in the seed set (K=5).

9.2.3 Propose Function Implementations

ShapeLib now has the information from the prior steps it needs to author good function implementations: typed signatures, doc-string guidance, and input-output examples. These input-output example pairs can be automatically found from the proposed function applications. From this input, we ask the LLM to complete the implementation of each function so that it matches the signature type, meets the doc-string specification, and respects the observed patterns present in the usage examples (Fig. 9.2, c).

Of note, we find that the LLM predictions in the previous application proposal step do a good job of identifying which functions should explain which parts, but do a much worse job at predicting parameter values. With this in mind, we mask out parameter values with a special token '?' in all input-output examples. We do this for every parameter value, except for the first *CF* 'CoordFrame', as the correct value for this parameter can be found automatically with the 'group_parts' function.

Similar to previous step, we find that some implementations produced by the LLM produce better or worse matches against the input specification. So for each function in our library, we propose K different ways that it could be implemented (K=4).

9.2.4 Library Validation

At this point we are close to having a fully realized library. From the prior steps we have (a) function doc-strings and signatures, (b) proposals of how the functions should be applied to explain groups of parts in seed-set shapes, and (c) proposals of how the function should be implemented. This validation step is responsible for deciding which of these proposals are 'good', and not just LLM hallucinations (Fig 9.2, d).

To make this decision, we search over pairs of proposed implementations and parameterizations, and

record those that geometrically match structures present in the seed set shapes. For each proposed function implementation from (c) we check which of proposed part groups from (b) this implementation can explain. Specifically, we try executing the function with the proposed parameterizations sourced from (b), calculate the observed error between the target parts and function output, and record the parameterization that achieves the best error. Our error metric compares corner-to-corner distances between sets of geometric primitives, and mark function applications as invalid if the paired structures are not similar enough (see Appendix G.1.1 for details).

At this point, for each group of parts from (b) we know which implementation from (c) best matches the observed part structure. We keep the implementation that achieves the *best* error across the *most* part groups, and remove all others proposals. If this *best* implementation found valid applications across multiple seed set shapes, we update the library interface entry with its implementation logic. Otherwise, we remove the function entry from the interface.

9.3 Using the Library for Program Synthesis

In Section 9.2, we constructed a library of functions that have meaningful signatures and structured doc-strings. Each function has an implementation that is capable of producing structures that capture patterns observed in the seed set, but a question remains: how can we use these functions to represent new shapes?

In this section, we describe our strategy for expanding library function usage beyond the seed set (Fig. 9.2, *right*). To begin, we once again make use of the strong prior of LLMs by providing it with our library interface and asking it to design a procedure that uses the abstraction functions to randomly synthesize synthetic shapes. Once we've developed this synthetic data sampler, we can use it to produce paired training data for a recognition network that learns how to solve an inverse task: given an input shape structure, write a program using the library functions that explain its parts.

Generating a synthetic shape sampler In this step, we design a prompt that describes the library we've developed, including the interface of each function and examples of how to use it (sourced from the validation stage). We give this prompt to an LLM and ask it to write a 'sample_shape' function that randomly produces new shapes using the provided abstractions. Interestingly, we find that frontier LLMs are able to provide useful implementations of such a 'sample_shape' function. A shown in Figure 9.2, some of these random outputs produce good shape abstractions, while other random samples violate class semantics. With this in

mind, instead of attempting to get the LLM to perfect its implementation, we treat its output as a synthetic data generator for a recognition network. To broaden the coverage and variety of structures that these 'sample_shape' functions produce, we employ an iterative refinement loop that provides automatic feedback to the LLM. This refinement procedure ensures that all functions and parameters in the library get used, and instructs the 'sample_shape' function to produce outputs spanning the observed structures from validation step (see Appendix G.1.3).

Training a recognition network Once we've improved the 'sample_shape' function through rounds of iterative refinement, we can use it to produce training data for a recognition network. This network takes as input a shape represented as a set of unordered primitives (e.g., Cuboid dimensions and positions). It outputs a program that uses library functions to reconstruct this input shape. We implement this network as an autoregressive Transformer decoder [209] with a causal prefix mask over the input shape representation. We train this network from scratch, streaming random samples from the synthetic data generator: each program we sample becomes a target output and we execute the program to find the corresponding input. Once trained, we can use this network to find library function applications that explain shapes from outside of the starting seed set (Fig. 9.2, *right-bottom*). Our inference procedure prompts the network with an input set of unordered primitives and samples a large number of programs according the network's predicted distribution. We try executing each program, and we record its complexity (the number of tokens it uses) and its geometric error against the input set. We choose the program that minimizes an objective that is a simple weighted combination of these two values.

9.4 Results and Evaluation

We run experiments over multiple categories of 3D shapes (`chair`, `table`, `storage`, `lamp`, `faucet`). For each category, an expert user provides design intent as (a) natural language descriptions of functions that would be useful for this category and (b) a set of 20 seed shapes sourced from PartNet [141], which has per-part annotations. We obtain corresponding renders of each shape from ShapeNet [16]. This input is provided to ShapeLib, which then produces libraries of abstraction functions for each category. Unless otherwise noted, we use OpenAI's o1-mini as the LLM.

We find that ShapeLib discovers libraries that match the design intent, with validated implementations for almost all of the functions specified in natural language (`chair` 8/8, `table` 5/6, `storage` 6/6, `faucet`

Table 9.1: We compare how well ShapeLib's library of abstraction functions can generalize from the seed set to held-out validation shapes. We report the objective score achieved by our method compared with alternatives. Obj is a weighted average of the program DoF and the geometric error.

Set	Method	Obj ↓	Prog DoF ↓	Error ↓	# Lib Fns ↓	Dev Time ↓
Seed	Prims	73.0	73.0	0.000	0	0 h
	LLM-Direct	64.0	61.6	0.242	5.6	0.25 h
	ShapeCoder	43.8	39.9	0.389	19.2	20.26 h
	ShapeLib	**43.5**	39.6	0.393	5.6	0.85 h
	Method	**Obj ↓**	**Prog DoF ↓**	**Error ↓**	**# Shape Fns ↓**	**Inf Time ↓**
Val	Prims	71.5	71.5	0.000	17.133	5.137 s
	LLM-Direct	65.3	63.5	0.184	14.482	4.792 s
	ShapeCoder	52.1	48.6	0.354	13.485	7.361 s
	ShapeLib	**51.5**	47.8	0.369	9.592	5.137 s

5/5, `lamp` 4/4). Figure 9.3 shows examples of these implementations and applications.

We verify that our method is able to help realize the benefits of representing shapes in a procedural fashion with experiments that match our stated desiderata (see Figure 9.4.). To evaluate generalization, we compare recognition networks that infer programs from structured inputs (Section 9.4.1) and from unstructured geometry (Section 9.4.2). We then evaluate how well function applications are aligned with class semantics (Section 9.4.3). Finally, we show that our interface is interpretable and maintains plausibility under manipulations with a perceptual study that evaluates how well an LLM can edit our shape programs compared to a baseline (Section 9.4.4)

9.4.1 Library Function Generalization

We measure how well our library generalizes beyond the patterns in the seed shapes. We compare against three alternatives: *Prims*, *LLM-Direct*, and *ShapeCoder*. *Prims* refers to our representation of input shapes as collection of unordered primitives – it is used as lower performance bound; *LLM-Direct* is an ablated version of our method that only reasons over the natural language descriptions to discover a library of abstraction and does not use seed shapes; while *ShapeCoder* only uses seed shapes. In our evaluations we show that ShapeLib, which uses both forms of design intent, offers clear advantages over these alternatives.

We evaluate the ability of different methods to compress programs in Table 9.1. We report this over two different shape sets: the seed set (20 shapes per category) and a held-out validation set (400-1000 shapes per category). For ShapeLib and LLM-Direct, program applications are found for validation shapes using the recognition network that takes as input a shape represented as a collection of unordered primitives (Section 9.3). ShapeCoder develops and learns such a recognition network during its 'library learning' stage.

Table 9.2: We train networks that learn to map unstructured geometry (point clouds or voxels) to shape programs. Learning with ShapeLib functions improves reconstruction Chamfer distance and voxel IoU.

Method	**CD** ↓ *(Point Clouds)*	**IoU** ↑ *(Voxels)*
ShapeCoder	0.0490	0.5708
ShapeLib	**0.0467**	**0.6404**

For both the seed set and the validation set, we report the total compression objective value (Obj). This is a weighted sum of the degrees of freedom the program exposes (Prog DoF, weight 1), and the geometric error of the reconstructions (Error, weight 10). We also report the number of functions used in each library (# Lib fns), the time it took to discover each library (Dev time), the number of functions used per shape (# Shape fns) and the average time it takes to infer a program for a validation shape (Inf Time).

From the results, we note LLM-Direct performs poorly, and its function implementations can't find applications that match well to real geometry (resulting in its limited objective improvement over Prims). ShapeCoder is designed solely to perform well at program compression, but despite this, ShapeLib is able to match or slightly outperform ShapeCoder with respect to the objective. Moreover, we achieve this result much faster, using a smaller collection of library functions, and require less function calls to reconstruct shapes during inference. We find library implementations in under an hour, whereas ShapeCoder's bottom up procedure takes around a day to converge (though our LLM API calls cost $5-10 per category).

9.4.2 Shape Programs from Unstructured Geometry

So far, we demonstrated that our recognition network from Section 9.3 can successfully convert semi-structure geometric inputs into programs, but what about completely unstructured geometry such as point clouds or voxels? To support this application, we train new recognition networks that take either point clouds or voxels as input. We source training data using the original 'structured' recognition network to annotate shapes in PartNet with corresponding programs. Per category, we use 400-4000 shapes for training and re-use the same 400-1000 shapes as described previously for validation. We sample both point clouds and voxels for each of the shapes.

Table 9.2 compares the reconstruction performance of a recognition network trained with function from ShapeLib to a recognition network trained with functions from ShapeCoder. For point clouds, we track Chamfer distance [45] between input point cloud (sampled from mesh geometry) and point cloud sampled from abstracted cuboid outputs. For voxels, we track IoU between input voxelizations and voxelizations of program outputs. We find that functions from ShapeLib enable more accurate reconstructions compared to

Table 9.3: We measure the 'semantic entropy' of library function applications by analyzing the distribution of functions used to reconstruct parts in validation shapes. Lower values indicate more semantically aligned usage.

Method	Chair	Table	Storage	Lamp	Faucet
ShapeCoder	1.67	1.578	2.077	1.732	2.103
ShapeLib	**0.484**	**1.095**	**0.745**	**0.684**	**1.243**

Table 9.4: Fine-grained semantic segmentation performance found by applying functions over validation shapes, and assigning labels with a voting scheme decided by seed-set usage patterns.

Method	Precision ↑	Recall ↑	F1 Score ↑
ShapeCoder	0.25	**0.30**	0.27
LLM-Direct	0.34	0.12	0.18
ShapeLib	**0.50**	**0.30**	**0.36**

functions from ShapeCoder. We visualize some qualitative results for some validation shapes in Figure 9.4. In addition to leading to better reconstructions, we also see that the application of our functions are more strongly correlated with class semantics.

9.4.3 Sematic Consistency of Function Usages

Beyond reconstruction, the *way* in which functions are used also impacts the usefulness of the resulting model. We design an experiment to evaluate the *semantic consistency* of function usages. We track how each function is applied when reconstructing validation shapes, and record the semantic labels of the parts that it matches against. Then, for each semantic label, we analyze the distribution of functions that were used to construct parts of this type. If functions are well-aligned with semantics, i.e. have a consistent usage pattern, then this distribution should have low entropy. We report results of this experiment in Table 9.3. Compared with ShapeCoder, ShapeLib has a much lower semantic entropy, indicating that its assignment of functions to part structures is more semantically aligned.

Semantic Segmentation Alternatively, we judge the semantic alignment of these libraries by using them to perform semantic segmentation. We design an experiment to test these capabilities. For each function, we look at validated applications made over the seed set, and record the semantic labels of parts that each function explains. We then aggregate this information by counting the most commonly covered part labels to produce a simple voting function to assign semantic labels when the function is applied. We evaluate the semantic segmentation performance on fine-grained part labels from PartNet over validation shapes, and report results of this experiment in Table 9.4. ShapeCoder and ShapeLib achieve a similar recall, but ShapeLib is twice as

Table 9.5: Results of our perceptual study evaluating edits made by an LLM to programs that use shape abstraction libraries. We report judgments along two axes: shape plausibility and match to edit intent.

	More Plausible(%)	**Better Matches Intent (%)**
vs. ShapeCoder	75%	73%

precise in its semantic predictions. LLM-Direct is more precise then ShapeCoder, however without access to seed set exemplars it cannot find many successful function application, resulting in poor recall.

9.4.4 Editing Shape Programs with LLMs

In this section, we investigate two critical questions concerning our library: is it interpretable and does it help constrain shape plausibility. We consider these questions under the framing of a shape editing study. First, we use the application network from Section 9.3 to find programs that explain validation shapes, using either functions from ShapeCoder or ShapeLib. We then design a series of shape edit requests, and ask an LLM to edit the text of the shape program to meet the request (i.e. change function parameters and how functions are used, as depicted in Figure 9.1, for example).

To evaluate performance, we designed a two alternative forced choice perceptual study. We choose 5 shapes from the validation set of each category, and consider 4 edits per shape, giving us a cross-product of 100 total comparison conditions. We provide *o1mini* with the fully implemented function library for both ShapeLib and ShapeCoder conditions. For the ShapeCoder condition, we observed that *o1mini* produced a program that failed on execution for 11/100 editing tasks, so we omit those from the study. *o1mini* never produced a program that failed on execution for the ShapeLib condition. We recruited 13 participants who made 50 perceptual judgments each. For each comparison, we show the original shape in the middle, and arrange edits made using ShapeCoder/ShapeLib programs on either side, randomizing the left/right order. We then ask each participant to make two judgments: (i) which manipulated shape was more plausible, and (ii) which edit better matched the input edit request.

We report preference rates of ShapeLib over ShapeCoder along these two axes in Table 9.5. These results support our claim that our library of shape abstraction functions provides an interface that is easy to interpret and maintains strong plausibility under parameter variations. We show qualitative demonstrations of these edits in Figure 9.4, and observe higher semantic alignment of LLM edits, when these edits are made over ShapeLib programs.

9.5 Discussion

We have presented ShapeLib as the first method that combines general semantic priors from LLMs with domain-specific information in the form of small seed set of shapes to produce a function library that *generalizes* to a full category of shapes and exposes *interpretable* parameters that produce *plausible* results under manipulation. This addresses the long-standing problem in visual program induction to create programs that are not only compact, but also semantically well-aligned and thus easy to work with for both humans and LLMs.

9.5.1 Relation with LILO

LILO is a related contemporary approach that proposes making use of an LLM prior for general (i.e., non-shape-specific) library learning [58]. Interestingly, the LLM in this method does not guide a top-down search for new abstractions, but rather tries to add semantic information to functions proposed in a bottom-up fashion. The method is a spiritual successor to DreamCoder [42], with a few critical differences. It does not train a recognition network, so there is no dream phase. Instead the wake phase uses both enumerative search and a LLM as a program synthesizer. The interface of the abstraction phase is unchanged, but the STITCH [11] algorithm replaces the version space reasoning from DreamCoder. Once abstractions have been proposed, LILO uses the LLM to *automatically document* each function: given its definition and example usages the LLM gives the function a readable name and a doc-string explanation. This reformatting is what allows the LLM to act as successful program synthesizer in the wake phase.

Like DreamCoder, LILO is dependent on the wake phase for finding complete solutions for at least some tasks before any abstraction discovery can take place. While LLM program synthesis priors can help alleviate this issue for many types of general domains, zero-shot visual program induction for complex shapes is outside of the capabilities of current models. Further, when abstraction functions are proposed in a purely bottom-up fashion, interpretability issues will remain, even with LLM integration, as there can be no guarantee that the function logic, or exposed interface, will have relevant semantic mappings.

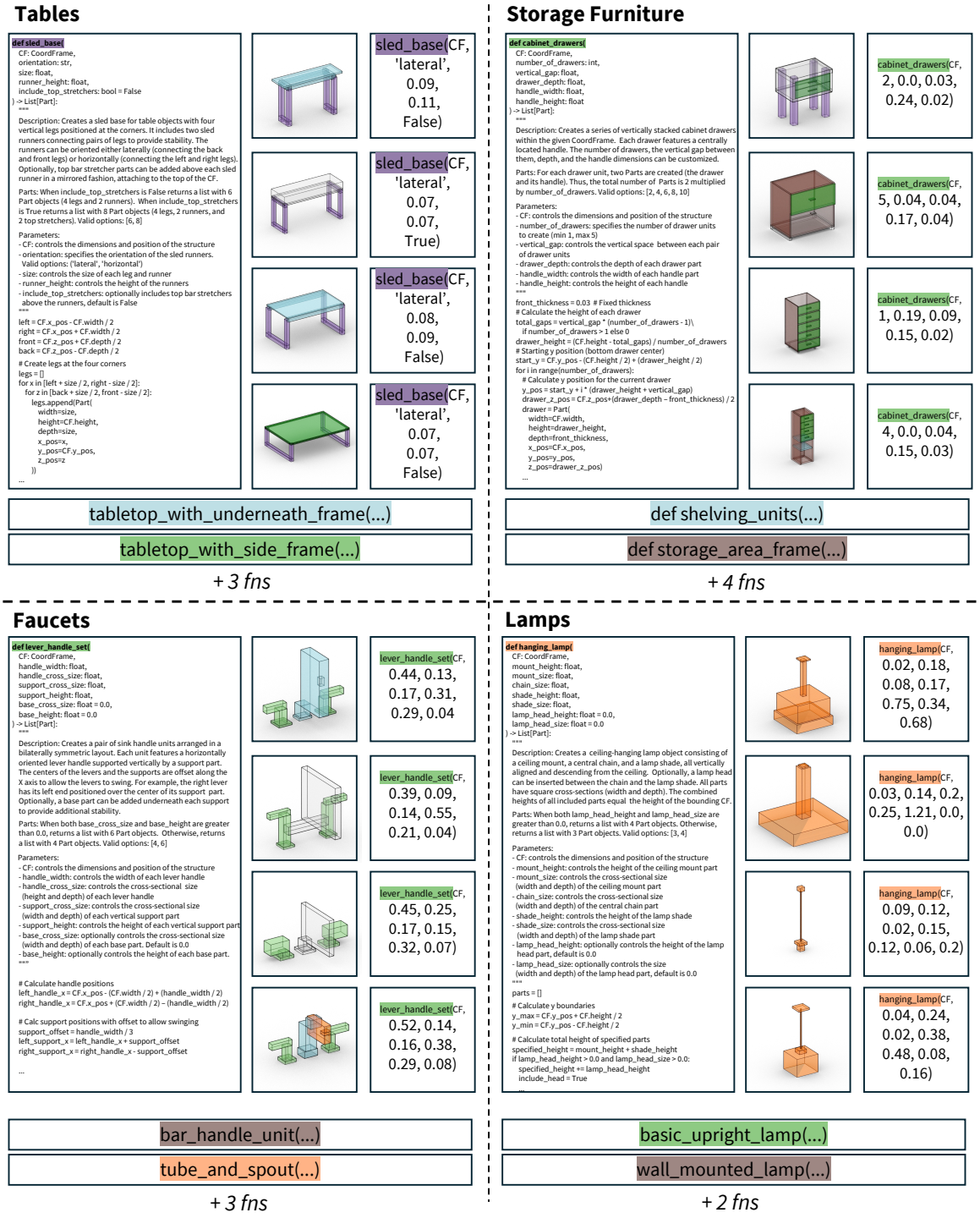

Figure 9.3: Examples of functions from the shape libraries discovered by ShapeLib. For each category, we show a function implementation, and a few example applications of the function. For each application, we show the full output shape, with parts corresponding to the function marked in the same color as the function name, and the function parameters. We can see that function applications are well-aligned with part semantics and that each function typically requires only a small set of parameters to represent a rich variety of part shapes.

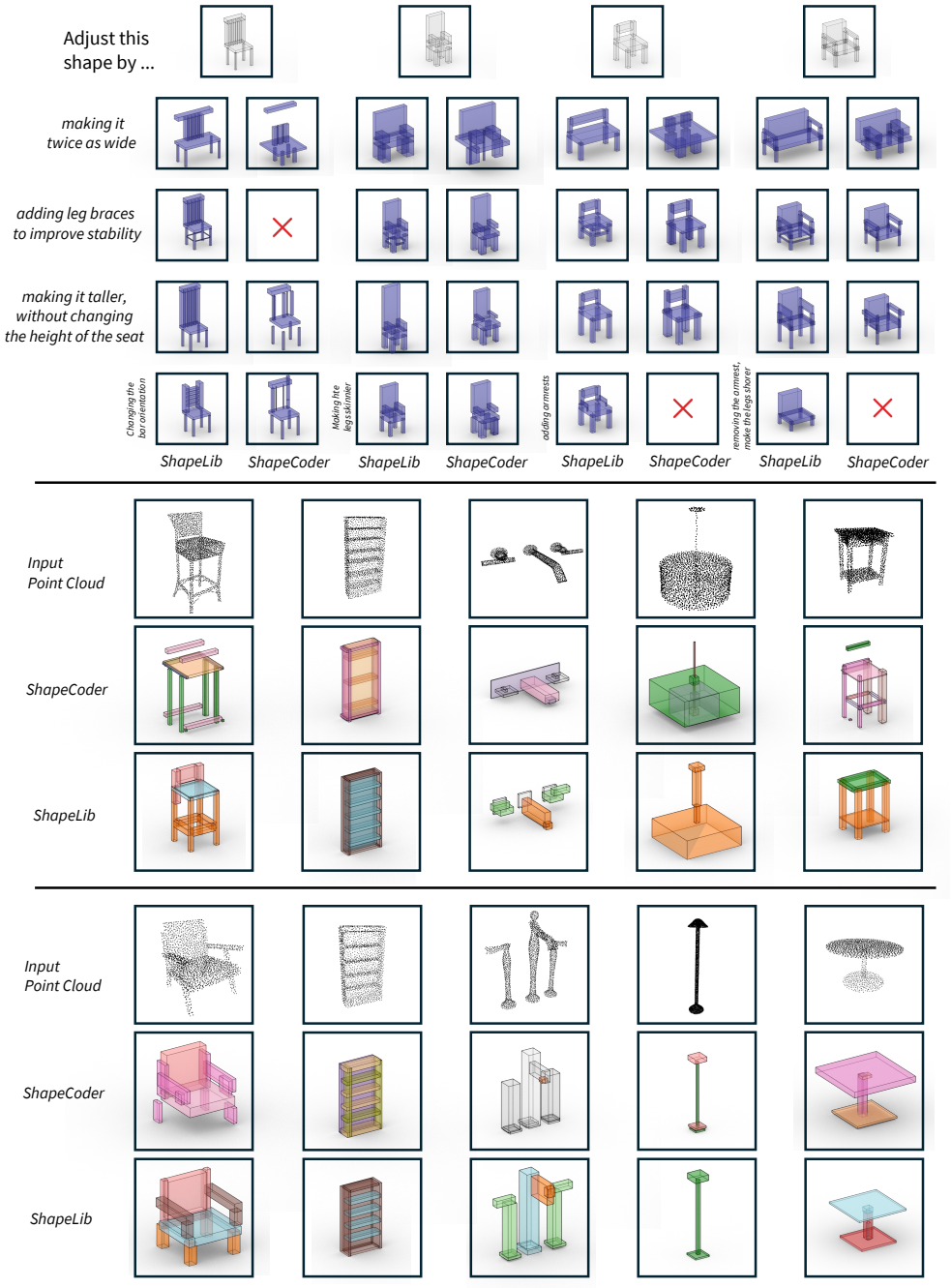

Figure 9.4: ShapeLib's abstraction functions provide a semantically aligned and interpretable interface that support downstream applications: text-based LLM editing and visual program induction from unstructured geometry.

Chapter 10

Conclusion and Future Directions

This thesis has introduced a series of neurosymbolic methods that aid in shape analysis and generation. These works demonstrate how the traditional limitations of procedural shape representations can be mitigated through the thoughtful integration of learning-based components and sub-modules. Our discussion focused on three lines of investigation. *Generating Shape Programs (a):* We proposed one of the first systems that realized the complementary strengths of neural and procedural generative models by introducing a hybrid neural-procedural approach for synthesizing novel shape structures. *Visual Program Induction (b):* We developed PLAD, a flexible and general framework that trains VPI networks without program annotations, treating the executor as a black-box, and offering better convergence properties compared with policy gradient alternatives. We explored extensions of PLAD, finding that we could improve VPI performance by training networks that learn how to edit programs and that this self-supervised framework could be used to infer partial programs that could capture a collection of visual inputs, e.g. a concept. *Abstraction Discovery (c):* We investigated methods that automatically produce DSLs tailored for a particular modeling task (e.g. a category of objects). In ShapeMOD, we introduced the first work that successfully scaled library learning techniques to complex 3D shape structures, starting from a input dataset of imperative programs. ShapeCoder relaxed this input assumption, discovering abstraction libraries from a dataset of shapes represented as collections of unstructured primitives. As these bottom-up approaches lack semantic guidance, we developed an alternative top-down solution in ShapeLib, that guides a LLM through the process of designing procedural abstraction functions.

While these narrative delineations might imply that these lines of investigation are isolated, the reality is that they are actually quite complementary. Training generative neurosymbolic models (a) often requires

access to a dataset of shape programs, these could be sourced by methods from (b). Conversely, the training schemes we've developed within (b) often make use of generative models that author shape programs (a), e.g. in 'wake-sleep' phases. Access to better domain-specific libraries produced by abstraction discovery methods (c) simplifies learning tasks (a, b), while identifying when procedural libraries are better or worse often requires the ability to know how candidate abstractions could be used to represent shapes from some collection (b). Together, these neurosymbolic approaches form a cohesive toolkit, promising a virtuous cycle of improvement, and help to realize the strengths of procedural shape representations in a flexible, adaptable fashion.

10.1 Future Work

While we propose a catalog of neurosymbolic techniques that help to alleviate the limitations of procedural shape representations, there are many avenues of future work left to explore. Our contributions advance the state-of-the-art performance of neurosymbolic methods for shape analysis and generation tasks, yet these capabilities are still largely tied to relatively simple domains and problem framings when compared with 'production-level' procedural assets. While there is some hope that existing methodology could narrow this gap through resource scaling (e.g. data, annotations, computation), closing this gap completely (or, at the very least, in a cost-effective manner) will likely require further research and innovation. We conclude this dissertation by discussing future directions of particular note.

10.1.1 Controllable Dense Geometry

In this dissertation, we have mostly discussed *structured* shape representations. Instead of trying to capture the geometric surface details (i.e. dense geometry) directly, these methods might use primitives as coarse part proxies to represent a shape at a slightly abstracted level. Some complementary analysis and manipulation works have made use of such structured representations for tasks such as segmentation [51] or deformation [192, 234]. In terms of shape generation, end-users typically want high-fidelity assets that could be placed directly in artificial worlds and applications; this demand is driving the rising popularity of easy-to-use text to 3D models [65, 125, 79]. To create such detailed assets in a structured, controllable fashion, some methods have been proposed for shape stylization, where (decoupled) coarse geometry can be 'up-sampled' through a style encoding [20, 23]. These aforementioned approaches treat structure generation and detalization as separate steps, but is this the only way neurosymbolic methods can be used for shape modeling?

One alternative is to 'bottom-out' symbolically: e.g., attempt to find a program that can explain the detailed version of a shape. This is a very sensible approach for certain domains, like CAD modeling of mechanical parts, as having access to a 'complete' program can be important for downstream analysis and many such shapes were designed through CAD programmatic workflows. On the other hand, for some domains the benefits of this framing is less clear: scans of real-world 3D objects might not have a good procedural equivalent, or even if one can find a good procedural equivalent it may be too complex to offer any real benefit. For these domains, we may want to keep an explicit separation between 'structure' and 'style', but this doesn't mean that these modes need to be ignorant of one another. One option here is to guide the outputs of an unstructured generative shape model through structured conditioning [185, 239]. Alternatively, the DSL could contain 'neural' operators that produce controllable sub-shapes that can be manipulated by other programmatic functions [34]. Beyond a successful decoupling of shape structure and style, these methods will realize their full potential when this style decoupling matches our design criteria for the shape structure: exposing a controllable, interpretable interface that allows for analysis and creative manipulations.

10.1.2 Visual Program Induction beyond Shapes

We've introduced a number of methods that advance the field of visual program induction. Most of this analysis has been performed on manufactured 2D and 3D shapes, but there exist many exciting opportunities to scale these insights to related domains. Closely related to manufactured shapes are organic shapes like humans, animals and plants: great effort has already been expended to develop structured proxies for some members of these classes [117, 128, 179, 226]. When procedural analogs are available, this inverse task becomes more constrained, though often there is still room for these representations to become more 'programmatic'. Building and facade modeling is another interesting problem area with higher visual complexity, but more within-sample regularity [142, 186]. For graphics content more broadly, one could imagine finding programmatic representations for movement, e.g. agent behavior [127], repetitive exercise [112], sports analytics [238]; environments, e.g. indoor rooms [212], floor plans [214], city scapes [225]; or even 'natural' images or videos. Further afield, some of our insights might even be relevant to more general program synthesis problems that are non-visual, including music synthesis [83] or molecule generation [196].

In terms of methodology, there are few unanswered questions that would be interesting to explore. In Chapter 5, we found that training program editing networks on data sourced from edit difference scripts outperformed data sourced from corruption processes. Both approaches require a hand-crafted machinery (how to corrupt, how to convert one program into another), so it would be interesting to try to automate these

in a fully domain-agnostic fashion. One could also investigate if edit scripts and program corruption are complimentary with one another, or what types of domains favor either approach. Beyond that, it would be interesting to formulate a system that integrates such 'learning' based program rewriters with 'non-learning' based rewriters like those used in SIRI [53]. When the executor is not just a black-box, some types of program manipulations can be better done automatically (e.g. using parameter optimization to tweak continuous values), while other types of program manipulations might be better left for the edit network (e.g. structural manipulations). Relatedly, extending these types of VPI methods to more natural image/video domains will likely require some form of 'neural' concept integration [240], and one could imagine concept embedding manipulation as another form of 'program-rewriting'.

A subtle, though powerful, benefit of performing visual program induction over shape domains lies in the reconstruction-based reward formulation. Shape program 'goodness' is usually evaluated with geometric error with respect to a target shape: this error metric provides a *relatively* non-sparse signal that rewards partially correct predictions. Like other unsupervised program synthesis approaches, the PLAD framework (Chapter 4), and its extensions, require signal to judge the fidelity of predicted programs. Finding an error metric that is compatible with PLAD training may be more challenging for other domains, and some alterations to the method may need to take place to explicitly reward well-matching local structures produced by self-consistent sub-program components (i.e. a chunk of code that reconstructs one building out of many in a city). An attractive alternative might be to instead use a policy network to guide self-supervised training, for instance a network that models how 'far-away' a program is from a target state [102]. Though learning such a network is challenging, this framing could be very powerful under the right conditions.

Typically, networks for visual program induction have been trained 'from scratch' (i.e. without a pretraining phase on some other task / dataset). As mentioned in Chapter 4, these networks are often initialized by training on *synthetic data*: programs (and their executions) sample from some random procedure. The rise of LLMs challenges this paradigm, as many visual program induction tasks can be framed as program synthesis tasks in more general languages (e.g. a python program that imports a visualization/graphics library). Though this approach is clearly superior for general languages, its less clear what advantages LLMs (or VLMs) offer for program synthesis tasks under more constrained DSLs. Their pretraining phase gives them a strong 'coding prior', but when synthetic data generators can be produce infinite in-domain data, how much of this prior is needed? Investigations that offer robust analysis into this tradeoff would likely be well received. A related idea has explored how LLMs can act as an 'easy-to-implement' data generator. The aforementioned random program sampling procedures typically require some domain-specific logic, which

is sometimes carefully crafted to match expected test-time distributions [206]. With LLMs, this process can be dramatically simplified, if one is willing to pay a compute/API cost: simply append a few *seed* example programs to a prompt, and ask the LLM to produce similar programs [123, 122]. One unexplored idea in this space is to take a hybrid framing: ask a LLM to gradually improve the design of a programmatic sampling procedure.

10.1.3 Procedural Abstraction Discovery

This dissertation has introduced three works that discover libraries of procedural abstractions functions: ShapeMOD (Chapter 7), ShapeCoder (Chapter 8), and ShapeLib (Chapter 9). The formulation presented in ShapeLib reflects our current best guess concerning the future of library learning for complex 3D shapes: using LLMs, under minimal human guidance, to search for abstractions that meet an input specification. In the following paragraphs we reflect on the future avenues along this research direction.

ShapeLib extensions One limitation of the current framing of ShapeLib is that we require users to specify up-front all of the abstraction concepts they would like the system to discover. This could be improved by recasting this process in an iterative loop: the user specifies some initial concepts, ShapeLib tries to find implementations, and then reports back to the user. The user could then update concept descriptions to better align them with LLM priors, or may be inspired to suggest new concept descriptions by looking at shapes that are poorly covered by the proposed abstractions. Alongside this, it would be interesting to investigate how well an LLM prior could be used to automatically propose new abstraction concepts based on the initial design intent (descriptions or seed set).

ShapeLib's functions produce cuboid primitives that represent part bounding boxes. Instead of trying to completely reproduce surface geometry, we capture a structured shape representation useful for downstream tasks. While this representation can already be directly useful for analysis and manipulation tasks, more machinery must be developed to convert these structured representations into production quality assets. Ideally, this could be done in a way that decouples local geometry (so each individual function can still be edited) while still exposing relevant tunable parameters for the style (dense geometry / texture / materials).

While ShapeLib is able to find abstractions that meet a user's design intent, currently this process starts from scratch with each invocation. With the proper infrastructure, one could imagine ShapeLib blossoming into an organic ecosystem, where successful abstractions are maintained and curated by a community of procedural modelers. This centralized knowledge-base would at once simplify ShapeLib's directive, allowing

it to reuse or be inspired by its previous solutions, while at the same time reforming task-specific abstraction functions into a category-general procedural modeling library.

Merging bottom-up and top-down methods While ShapeLib's top-down framing offers many benefits, this perspective cannot make full use of the bottom-up abstraction discovery machinery developed within ShapeMOD and ShapeCoder. One might then ask: is there a hybrid solution that meets somewhere in the middle? This may be possible, consider for instance a method that first partitions the broad library learning objective into digestible sub-tasks, that are then satisfied through bottom-up candidate proposal, and finally validated with top-down semantics. The LILO system [58] takes a hybrid stance along this line, using top-down semantic knowledge to document abstractions found from a bottom-up procedure, but there may be further opportunities to integrate top-down reasoning into such library learning systems.

Leveraging existing procedural assets The library learning works we've proposed often try to reduce the amount of structured system inputs as much as possible. Though this framing is general, and intellectually interesting, it neglects to make use of an available resource: existing procedural models! Why might these be useful? A well-structured procedural model could become relevant context as part of LLM prompts, e.g. as a guide for what 'well-designed' means. One could also imagine a gradual evolution of a procedural model, where a starting version is improved through iterative edits to capture a more diverse output distribution or specialize its productions with respect to certain criteria. Beyond making use of single procedural models, there are also opportunities for works in this space to learn from or condition on datasets of procedural representations and functions [165, 168, 167].

10.1.4 Programmatic Shape Analysis

Up to this point, we've discussed methods that aim to represent shapes in a neurosymbolic fashion to support generation and analysis tasks. A subtly different line of investigation might try to analyze shapes in a neurosymbolic manner. For instance, there has been some recent works that explore how programs can be used in visual analysis tasks, especially within the field of visual question answering, or VQA. In VQA, a visual input (usually an image) and a question (in the form of natural language) are presented to a system, which must output an answer. Prior works have explored converting such questions into 'query programs', which can then operate over a processed version of the image (e.g. object-centric representations). While a number of these methods have shown proficiency on artificial domains [89, 90, 131, 233, 69], only very

recently have similarly inspired approaches seen success in scaling to 'real-world' use-cases by leveraging LLMs [67, 199, 195].

Though these advances for neurosymbolic image analysis have not yet proved useful for 3D shapes, this is an area ripe for investigation. Semantic segmentation is a foundational visual computing problem that might especially benefit from this sort of neurosymbolic framing. Many applications and methods, including some of the ones proposed in this dissertation, require fine-grained hierarchical part decompositions of 3D shapes. A large body of research has investigated learning-based approaches for this task, but a key issue is that labeled data is often limited, especially for fine-grained label sets. Though this issue can be mitigated somewhat through modular approaches that factorize this task into more manageable sub-problems ([96], [97]), these approaches are still quite data-hungry and significantly underperform against expert annotators.

For inspiration on how these systems could be further improved, one can look to how people go about decomposing shapes into parts. PartNet [141], is one of the only existing large-scale datasets of 3D shapes that has fine-grained part annotations. Interestingly, in the interface that PartNet labelers used to make the annotations, the instructions for how to label each semantic part were given in the form of both rules & examples. For instance, when labeling a back-frame part, an annotator would be shown an example of a back-frame part in the context of a chair, and then also given a definition that a back-frame typically 'outlines the backbone of a chair back'. Could learning-based approaches for semantic segmentation benefit from the same kinds of symbolic information? Exploring how to convert discrete part-based relationships, like one part outlining another part, into programmatic rule-expressions promises an intriguing future direction.

Appendix A

Additional Details for ShapeAssembly

In Appendix A, we supply additional details for the ShapeAssembly method introduced in Chapter 3.

A.1 Semantics of the `attach` Command

In designing the SHAPEASSEMBLY interpreter, our goal is to ensure that its internal operations stay limited to simple fixed-function, differentiable operations. Thus, implementing the `attach` command, we opt not to use any constrained optimization routines which could resolve a globally-optimal configuration of cuboids given the attachment constraints. Instead, the interpreter immediately executes each attachment as it is declared, i.e. it greedily solves for attachments. To make the behavior of this procedure as predictable as possible, the greedy attachment procedure should induce the fewest changes possible to the current cuboid shapes.

With these desiderata in mind, we designed the following procedure for attaching cuboid c_1 to cuboid c_2 (see Figure A.1). The logic that executes depends upon how many prior attachments c_1 has and the aligned flag of c_1:

No prior attachments In this case, cuboid c_1 can connect to cuboid c_2 by simply translating until the attach points are colocated.

One prior attachment Here, the interpreter scales cuboid c_1 along one of its axes and then rotates it such that the attachment is satisfied. To choose the axis along which to scale c_1, the interpreter checks how quickly scaling each of its three dimensions would reduce the ratio n/k, where n is the distance between c_1's existing attachment point and the new target attachment point, and k is the distance between c_1's existing attachment

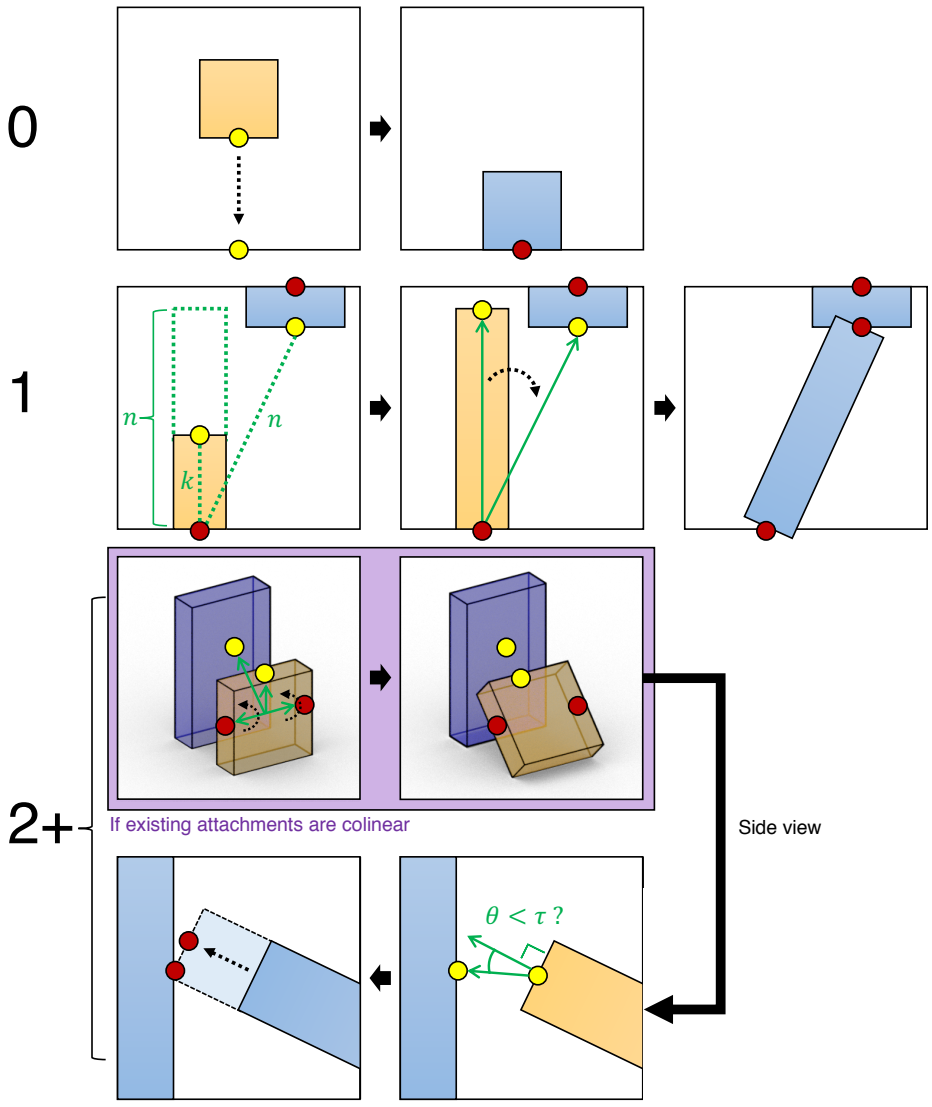

Figure A.1: Illustrating how the `attach` command executes, depending on the number of existing attachments (left column) to the cuboid in question. Cuboids with no existing attachments can simply be translated into place (top). Cuboids with one existing attachment can be scaled along one axis and then rotated (middle). Cuboids with two or more existing attachments are more complicated, and the attachment may not always be satisfiable. Our interpreter attempts to rotate and scale the cuboid to get as close as possible to valid solution.

point and the new source attachment point. The interpreter then scales c_1 by n/k along this dimension, which gives it the correct length. Finally, c_1 is rotated such that the source and target attachment points are colinear (and thus colocated).

Two or more prior attachments In this case, it is not always possible to satisfy the attachment, as three point constraints on a cube may be overconstrained. If a solution exists, however, our interpreter will find it. And in the case where no solution exists, it attempts to approximately satisfy the attachment (which we decided to be more user-friendly behavior than throwing an error).

First, the interpreter checks if c_1's existing attachment points are all colinear. If they are, then it rotates c_1 about this axis of colinearity to make the source attachment point face the target attachment point. The final step is to scale c_1 along the normal of the face containing the source attachment point. If the existing attachment points were not colinear, and this face was not rotated to point toward the target attachment point, then this may not be a useful operation (i.e. it may introduce undesirable change to the cuboid shape while doing little to bring the source point closer to the target point). Thus, the interpreter only executes this scale if the angle between the source face normal and the vector to the target point is smaller than a threshold τ (25 degrees in our implementation).

Aligned Cuboids Cuboids that are marked as aligned in SHAPEASSEMBLY programs cannot have their orientations changed through attachment. In fact, with correct cuboid dimension parameterization, a single attachment is enough to properly position and orient an aligned cuboid. However, in order to ensure that aligned Cuboids remain connected through edits and predictions of our generative model, we minimally grow aligned cuboid dimensions to satisfy the part-to-part connectivity specified through attachments. That is, for aligned cuboids we do not guarantee attachment point colocation after the first attachment, as this is often impossible to exactly fulfill without changing a cuboid's orientation. Rather, we guarantee that aligned cuboids will fulfill attachment relationships with cuboids they are attached to at *some* attachment point.

A.2 Semantics of SHAPEASSEMBLY Macro Functions

We provide an account of the logic for macro function expansion in SHAPEASSEMBLY :

Squeeze. The `squeeze` macro is parameterized by three cuboids (c_{n1}, c_{n2}, c_{n3}) a face f and a (u, v) position on f's 2D coordinate system. A `squeeze` command expands into two `attach` functions. The first attach function attaches the center of c_{n1}'s f face to the (u, v) position on the opposite face of f on c_{n2}. The second attach function attaches the center of c_{n1}'s opposite face of f to the (u, v) position on the face of f

on c_{n3}. For example, the line `squeeze` (c_{n1}, c_{n2}, c_{n3}, left, .1, .4). It expands into `attach`(c_{n1}, c_{n2}, 0.0, .5, .5, 1.0, .1, .4) and `attach`(c_{n1}, c_{n3}, 1.0, .5, .5, 0.0, .1, .4).

Reflect. The `reflect` macro is parameterized by a cuboid c_n and an axis a. A `reflect` command first expands into one `Cuboid` function, that creates a new cuboid $c_{n'}$ with the same parameters as c_n. Then for every previous attachment line pair that had moved c_n, of the form `attach`$(c_n, c_m, x_1, y_1, z_1, x_2, y_2, z_2)$, the reflect command creates a new attachment line: `attach`$(c_{n'}, c_m, x_1, y_1, z_1, R(x_1, y_1, z_1, c_n, c_m, a))$. R is a function that applies a reflection of the global point specified by (x_1, y_1, z_1) in the local coordinate frame of c_n about the axis a, and then returns the local coordinates of that point within c_m.

Translate. The `translate` macro is parameterized by a cuboid c_n, an axis a, a number of members m, and a distance d. A `translate` command first expands into m `Cuboid` functions, that each creates a new cuboid c_{n_i} with the same parameters as c_n. Then for every previous attachment line pair that had moved c_n, of the form `attach`$(c_n, c_m, x_1, y_1, z_1, x_2, y_2, z_2)$, the translate command creates a new attachment line `attach`$(c_{n_i}, c_m, x_1, y_1, z_1, T(x_1, y_1, z_1, c_n, c_m, a, d))$. T is a function that applies a translation of the global point specified by (x_1, y_1, z_1) in the local coordinate frame of c_n along the axis a (of the bounding volume) for for a distance of d (where d is normalized by the size of the bounding volume), and then returns the local coordinates of that point within c_m.

A.3 Program Extraction Procedure

Here, we provide an account of our program extraction procedure in greater detail:

Part Shortening Before any hierarchical processing, we first attempt to regularize any artifacts in the input data. Specifically, for each leaf cuboid part proxy, we check if any of its faces are completely contained within any other leaf cuboid. If we find that we can shorten a leaf cuboid without changing the visible, non-intersecting, geometry of the part graph, we do so.

Semantic Hierarchy Arrangement During our data preprocessing stage when converting PartNet part graphs into SHAPEASSEMBLY programs, we locally flatten part graph hierarchies based on semantic rules as depicted in Figure 3.5. For chairs we flatten the following nodes: back, arm, base, seat, footrest and head. For tables we flatten the following nodes: top and base. For storage we flatten the following nodes: cabinet frame, cabinet base. For storage, we move the following nodes into the cabinet frame sub-program: countertop, shelf, drawer, cabinet door and mirror. We also perform a semantic collapsing step where the intermediate

nodes containing detailed geometry are converted into leaf nodes and their children are discarded. For chairs we collapse the following nodes: caster and mechanical control. For tables we collapse the following nodes: caster, cabinet door, drawer, keyboard tray. For storage we collapse the following nodes: drawer, cabinet door, mirror and caster. Empirically we observed that this method of hierarchy re-arrangements produces cleaner and more regularized training data for our generative model.

Attachment Point Detection In order to identify which cuboids connect, and where they connect, we use a point cloud intersection procedure. We sample a uniform 20x20x20 point cloud within the volume defined by each cuboid. To check if two cuboids are attached, we find the set of points in the pairwise point cloud comparison that have a minimum distance to any point in the other point cloud within a distance threshold determined by the scale of the larger cuboid. For cuboids that attach (i.e. this intersection set is non-zero) we sample a denser 50x50x50 point cloud within the bounds of the detected intersection volume, forming a set of candidate attachment points. From this set we first filter all attachment points that are outside of either cuboid. If any remaining attachment points form face-to-face connections between cuboids we choose them, otherwise we define the attachment as taking place at the mean of the remaining attachment points. With the same procedure, we also record if cuboids connect to the top or bottom of the bounding volume. Sampled points with bounding volume local y-coordinates in the ranges of $[0, 0.05]$ and $[.95, 1.0]$ are assigned to the bottom and top respectively.

Symmetry Detection We enforce that all members of a symmetry group share the same connectivity structure in the input part graph. Cuboids are grouped together by symmetry if they: (i) connect to the same cuboids, (ii) share a reflectional or translational symmetry about the X, Y or Z axis of their parent bounding volume, and (iii) each attachment point involved in their outgoing connections also shares this same symmetrical relationship. Two cuboids, or two attachment points, are considered to share a symmetrical relationship if applying the symmetry transformation matrix to one member produces a parameterization close to that of the other member.

Notice that this procedure can disqualify symmetry formation about groups of interconnected cuboids that share a symmetrical relationship. As such, before forming symmetry groups about individual cuboids, we attempt to form symmetry groups about connected components of multiple cuboids. Whenever such a component is found, we locally abstract its structure with a bounding volume, and create a symmetry group

sub-program. In this manner, we capture additional spatial symmetries while continuing to enforce the relationship between symmetry and part connectivity. The ”H-leg” program (Program3) in Figure 3.2 shows an example of where such a symmetry sub-program was formed.

In total, our parsing procedure finds valid SHAPEASSEMBLY programs for 46% of Chairs, 65% of Tables and 58% of Storage shapes in PartNet.

A.4 Decoder Semantic Validity Checks

During the process of decoding a latent code, our generative network enforces the following semantic validity conditions on its outputs:

- XYZ attachment coordinates are clamped between 0 and 1.0. Additionally, attachments to the bounding box can only be at the top or bottom faces with an allowable error of .05.
- Cuboid dimensions are clamped between 0.01 and the corresponding bounding box dimension
- Bounding box cuboids can have no sub-programs
- Cuboids only attach at a single location. As an exception, cuboids are allowed to attach to both the top and bottom faces of the bounding volume.
- The bounding box cannot be moved by an attach command
- Attachment orderings must be grounded. Upon terminating, any ungrounded cuboids instantiations are discarded.
- Symmetries can only operate on grounded cuboids
- The ordering of Cuboid, attach, squeeze, reflect, and translate lines must be consistent with the SHAPEASSEMBLY grammar.
- Commands must keep cuboids within the bounds of the defined bounding volume with an allowable error of 10%.

During generation, if our model predicts a non-semantic program line, we attempt to back-track until we are able to find a semantically valid solution. For instance, if we predict a new line to be a `reflect`

command, but no cuboids have been grounded, we pick a new command type for the line by zeroing out the logits for the reflect command index.

In some cases, a combination of bad continuous parameters and program structure predictions produce a violating line that cannot be easily fixed. During unconditional generation, we reject the sample if we encounter this behavior (this happens for 10% - 20% of our random samples across the categories we consider). We run an ablation on this rejection sampling in Table 3.2. During interpolation, we never reject a sample. Instead, we simply do not add lines to the predicted program for which we could not find a fix.

A.5 Shape Quality Metrics

We provide additional details about the metrics used in Table 3.2:

- **Rootedness :** We check if a connected path exists between the ground and all parts in the shape. We judge two parts to be connected if they are separated by a distance no larger than 2% of the overall shape's bounding box diagonal length.
- **Stability :** We convert generated 3D shape structures into rigid bodies and place them in a physical simulation with gravity. A vertical force is applied to each shape proportional to its mass, along with some other small random forces and torques. If the resting height of any connected component of the shape changes by more than 10% after these perturbations we declare it unstable. Note that this is by definition less than or equal to the percentage of rooted shapes, as a shape must be rooted in order to be stable.
- **Realism:** The percentage of test set shapes classified as "generated" by a binary PointNet classifier trained to distinguish between generated shapes and shapes from the training dataset. The classifier is trained on an equal amount of positive and negative examples for 300 epochs. We hold out a portion of shapes from the test set, and measure the percentage of them incorrectly classified as "fake". To reduce fluctuation, the percentage is averaged over the last 50 epochs.

Appendix B

Additional Details and Results for PLAD

In Appendix B, we supply additional details for the PLAD method introduced in Chapter 4.

B.1 Details of Domain Grammars

2D CSG We follow the grammar from CSGNet [187]. This grammar contains 3 Boolean operations (intersect, union, subtract), 3 primitive types (square, circle, triangle), and parameters to initialize each primitive (L and R tuples). Please refer to the CSGNet paper for details.

$$
\begin{aligned}
&S \rightarrow E; \\
&E \rightarrow EET \mid P(L, R); \\
&T \rightarrow intersect \mid union \mid subtract; \\
&P \rightarrow square \mid circle \mid triangle; \\
&L \rightarrow \big[8:8:56\big]^2; \;\; R \rightarrow \big[8:4:32\big].
\end{aligned}
$$

3D CSG We design our own grammar for 3D CSG similar in spirit to the grammar of CSGNet. While CSGNet does contain a 3D CSG grammar, we find that it overly discretizes the possible spacing and positioning of primitives. Therefore in our grammar, we allow each primitive to be parameterized at the same granularity as the voxel grid (32 bins). In this way, each primitive takes in 6 parameters (instead of 2 parameter tuples), where the 6 parameters control the position and scaling of the primitive.

$$
\begin{aligned}
&S \rightarrow E; \\
&E \rightarrow EET \mid P(F, F, F, F, F, F); \\
&T \rightarrow intersect \mid union \mid subtract; \\
&P \rightarrow cuboid \mid ellipsoid; \\
&F \rightarrow \big[1 : 32\big]
\end{aligned}
$$

ShapeAssembly ShapeAssembly is a domain-specific language for creating structures of 3D Shapes (Chapter 3). It creates structures by instantiating parts (*Cuboid* command), and then attaching parts to one another (*attach* command). It further includes macro operators that capture higher-order spatial patterns (*squeeze*, *reflect*, *translate* commands). To remain consistent with our CSG experiments, we further modify the grammar such that all continuous parameters are discretized.

$S \rightarrow BBoxBlock; ShapeBlock;$

$BBoxBlock \rightarrow$ bbox $= \texttt{Cuboid}(1.0, x, 1.0)$

$ShapeBlock \rightarrow PBlock; ShapeBlock \mid None$

$PBlock \rightarrow c_n = \texttt{Cuboid}(x, x, x); ABlock; SBlock$

$ABlock \rightarrow Attach \mid Attach; Attach \mid Squeeze$

$SBlock \rightarrow Reflect \mid Translate \mid None$

$Attach \rightarrow \texttt{attach}(cube_n, f, uv, uv)$

$Squeeze \rightarrow \texttt{squeeze}(cube_n, cube_n, face, uv)$

$Reflect \rightarrow \texttt{reflect}$(axis)

$Translate \rightarrow \texttt{translate}$(axis$, m, x)$

$f \rightarrow right \mid left \mid top \mid bot \mid front \mid back$

axis $\rightarrow X \mid Y \mid Z$

$x \in [1, 32]/32.$

$uv \in [1, 10]^2/10.$

$n \in [0, 10]$

$m \in [1, 4]$

B.2 Details of Synthetic Pretraining

2DCSG We follow the synthetic pretraining steps from CSGNet and directly use their released pretrained model weights. Please refer to their paper and code for further details.

3DCSG We generate synthetic programs for 3D CSG with the following procedure. First, we sample K primitives, where K is randomly chosen between 2 and 12. To sample a primitive, we sample a center position within the voxel space, and then we sample a scale, such that the scale is constrained so that the primitive will not extend past the borders of the voxel grid. We then find if the bounding boxes of any two primitives overlap in space (using the position and scale of each primitive). We then construct a binary tree of Boolean operations by randomly merging the K primitives together, until only one group remains. Each Boolean operation merges two primitive groups into a single primitive group. The type of semantically valid Boolean operation depends on the overlaps between primitives of the two groups. When a group of primitives A and a group of primitives B is merging: union is always a valid operation, difference is a valid operation if each primitive in group B shares an overlap with some primitive in group A, and intersection is a valid operation if each primitive in group A shares an overlap with some primitive in group B and each primitive in group B shares an overlap with some primitive in group A. We can then unroll this binary tree of boolean operations into a sequence of tokens from the CSG grammar, forming a synthetic program. We sample 2,000,000 synthetic programs according to this procedure, that are used during supervised pretraining, and we sample another 1000 synthetic programs that we use a validation set. We pretrain our model for 40 epochs, where each epoch takes around 1.5 hours to complete. At this check-point, the model had converged to a reconstruction IoU of 90 on both train and validation synthetic data.

ShapeAssembly We generate synthetic programs for ShapeAssembly with the following procedure. We first sample the number of primitive blocks K (PBlock), where K is randomly chosen between 2 and 8; note that the number of cuboids created can be greater then K, when symmetry operations are applied. Each PBlock is filled in with random samples according to the grammar syntax. First a cuboid is created, then an attach block is applied, then a symmetry block is applied. An attach block can contain either one attach operation, one squeeze operation, or two attach operations. A symmetry block can contain either a reflect operation, a translation operation, or no operation. Command parameters are randomly sampled according to simple heuristics (e.g. reflections are more common than translations) and in order to maintain language semantics (e.g. attaches can only be made to previously instantiated cuboid indices). A final validation step occurs after a complete set of program tokens has been synthetically generated; we execute the synthetic program, and check how many voxels are uniquely occupied by each cuboid in the executed output. If any cuboid uniquely occupies less than 8 voxels, the entire synthetic sample is rejected. We sample 2,000,000 synthetic programs according to this procedure, that are used during supervised pretraining, and we sample

another 1000 synthetic programs that we use as a validation set. We pretrain our model for 26 epochs, where each epoch takes around 40 minutes to complete. At this check-point the model had converged to reconstruction IoU of 70 on both train and validation synthetic data.

B.3 Experiment Hyperparameters

3D Experiments For 3D CSG and ShapeAssembly, we use the following model hyper-parameters.

The encoder for both cases is a 3D CNN that consumes a 32 x 32 x 32 voxel grid. It has four layers of convolution, ReLU, max-pooling, and dropout. Each convolution layer uses kernel size of 4, stride of 1, padding of 2, with channels (32, 64, 128, 256). The output of the CNN is a (2x2x2x256) dimensional vector, which we transform into a (8 x 256) vector. This vector is then sent through a 3-layer MLP with ReLU and dropout to produce a final (8 x 256) vector that acts as an 8-token embedding of the voxel grid.

The decoder for both cases is a Transformer Decoder module [209]. It uses 8 layers and 16 heads, with a hidden dimension size of 256. It attends over the 8-token CNN voxel encoding and up to 100 additional sequence tokens, with an auto-regressive attention mask. We use a learned positional embedding for each sequence position. An embedding layer lifts each token into an embedding space, consumed by the transformer, and a 2-layer MLP converts Transformer outputs into a probability distribution over tokens.

In all cases we set dropout to 0.1 . We use a learning rate of 0.0005 with the Adam optimizer [106] for all training modes, except for RL, where following CSGNet we use SGD with a learning rate of 0.01 . During supervised pretraining we use a batch size of 400. During PLAD method fine-tuning we use batch size of 100. During RL fine-tuning we use a batch size of 4, due to memory limitations (a batch size of 4 takes up 10GB of GPU memory). Early stopping on the validation set is performed to determine when to end each round and when to stop introducing additional rounds. For deciding when to stop introducing additional rounds, we use a patience of 100 epochs. For deciding when to stop each round, we use a patience of 10 epochs. In both cases we employ a patience threshold of 0.001 IoU improvement (e.g. we must see at least this much improvement to reset the patience). Within each round of PLAD training, we check validation set reconstruction performance with a beam size of 3; between rounds of PLAD training we check validation set reconstruction performance with a beam size of 5; final reconstruction performance of converged models is computed with a beam size of 10.

For RL runs, we make a gradient update after every 10 batches, following CSGNet. For runs that involve VAE training (all Wake-Sleep runs), we add an additional module in-between the encoder and the decoder.

Table B.1: Different ways to update P^{BEST} data structure. In the "Per round" row, the data structure is cleared in between rounds. In the "All-time" row, the data structure maintains the best program for each input shape across multiple rounds.

P^{BEST} mode	ST	LEST	LEST+ST	LEST+ST+WS
Per round	0.881	1.011	0.853	0.845
All-time	**0.841**	**0.976**	**0.829**	**0.811**

This module uses an MLP to convert the output of the encoder into a 128 x 2 latent vector (representing 128 means and standard deviations). This module then samples an 128 dimensional vector from a normal distribution described by the means and standard deviations, and further lifts this encoding into the dimension that the decoder expects with a sequence of linear layers. For each round of VAE training, we allow the VAE to update for no more than 100 epochs. We perform early-stopping for VAE training with respect to its loss, where the loss is a combination of reconstruction (cross-entropy on token predictions) and KL divergence, both weighed equally.

2D Experiments For 2DCSG, we follow the network architecture and hyper-parameters of CSGNet. All training regimes use a dropout of 0.2 and a batch size of 100. PLAD methods use the Adam optimizer with a learning rate of 0.001. For deciding when to stop introducing additional rounds, we use a patience of 1000 epochs. For deciding when to stop each round, we use a patience of 10 epochs. In both cases we employ a patience threshold of 0.005 CD improvement. The parameters for the RL runs and VAE training are the same as in the 3D Experiments.

B.4 P Best Update mode

During updates to P^{BEST}, we choose to update each entry in P^{BEST} according to which inferred program has achieved the best reconstruction similarity with respect to the input shape. The entries of this data structure are maintained across rounds. There is another framing where the entries of this data structure are reset each round, so that the best program for each shape is reset each epoch. This is similar to traditional self-training framing.

We run experiments on 2D CSG with this variant of P^{BEST} update and present results in Table B.1. When the best program is maintained across rounds (All-time, bottom row) each fine-tuning strategy reaches a better converged reconstruction accuracy compared with when the best program is reset after each round (Per round, top row).

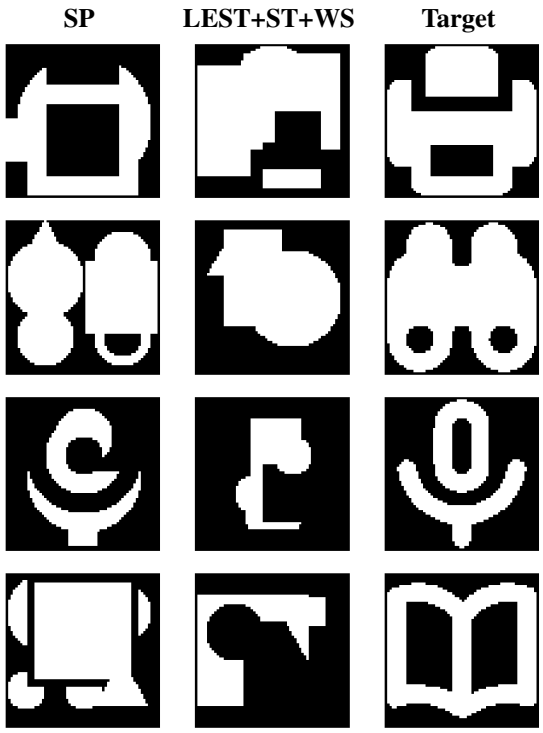

Figure B.1: Qualitative examples of inferring 2D CSG programs for 2D icons. Both SP and LEST+ST+WS fail to infer representative programs, but the reconstructions from LEST+ST+WS are even less accurate than those from SP.

B.5 Failure to generalize beyond S^*

As demonstrated by our experiments, PLAD fine-tuning methods are able to successfully specialize $p(\mathbf{z}|\mathbf{x})$ towards a distribution of interest S^*. Unfortunately, this specialization comes at a cost; the fine-tuned $p(\mathbf{z}|\mathbf{x})$ may actually generalize worse to out of distribution samples. To demonstrate this, we collected a small dataset of 2D icons from the The Noun Project[1]. We tested the shape program inference abilities of the initial $p(\mathbf{z}|\mathbf{x})$ trained under supervised pretraining (SP) and of the fine-tuned $p(\mathbf{z}|\mathbf{x})$ trained under PLAD regimes (LEST+ST+WS) and specialized to CAD shapes. We show qualitative examples of this experiment in Figure B.1. While both methods fail to accurately represent the 2D icons, fine-tuning $p(\mathbf{z}|\mathbf{x})$ on CAD shapes lowers the reconstruction accuracy significantly; the SP variant achieves an average CD of 1.9 while the LEST+ST+WS variant achieves a CD of 4.1 Developing $p(\mathbf{z}|\mathbf{x})$ models capable of out-of-domain generalization is an important area of future research.

[1] `https://thenounproject.com`

B.6 Additional Qualitative Results

We present additional qualitative results comparing various fine-tuning methods in Figure B.2 (2D CSG), Figure B.3 (3D CSG) and Figure B.4 (ShapeAssembly).

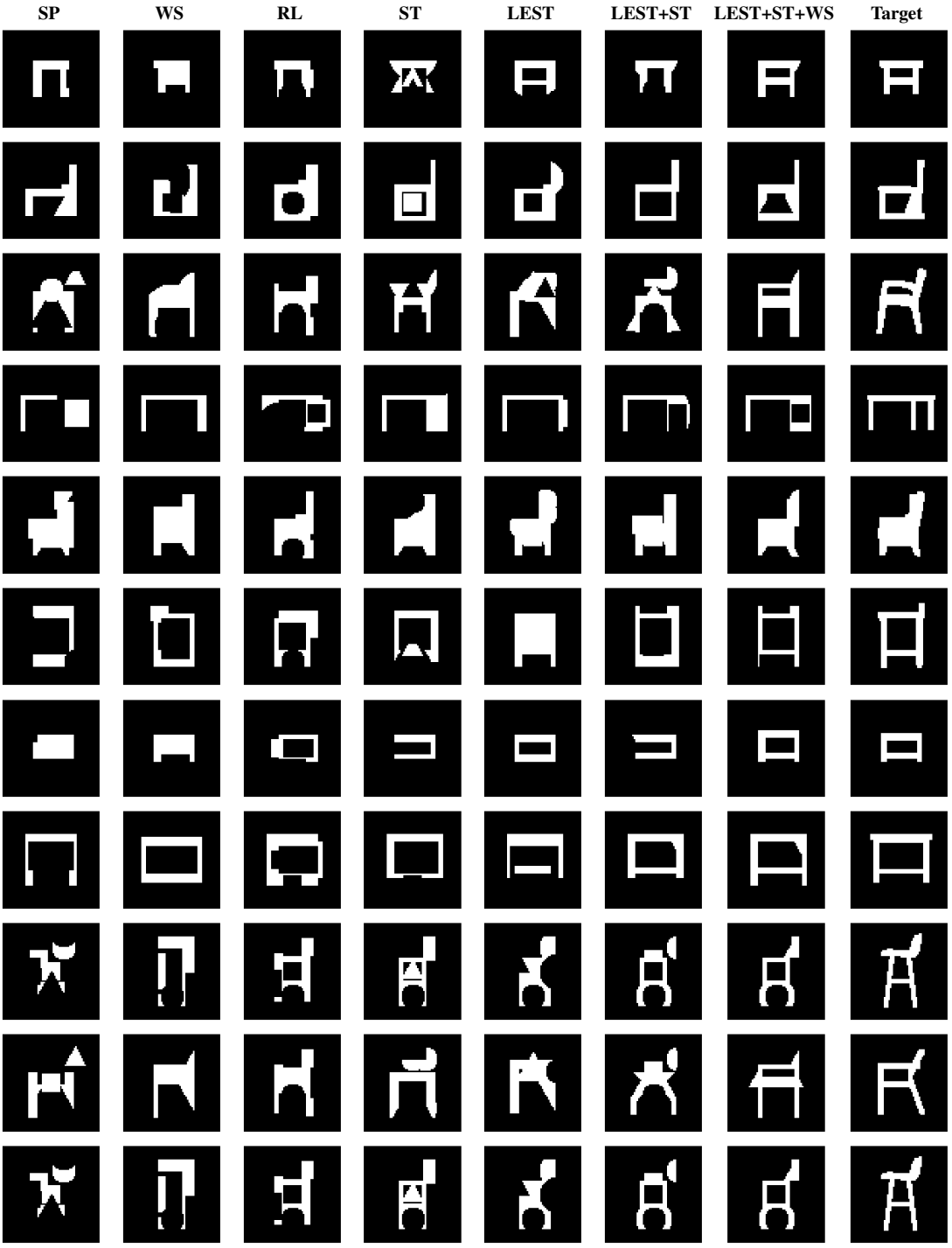

Figure B.2: 2DCSG qualitative examples.

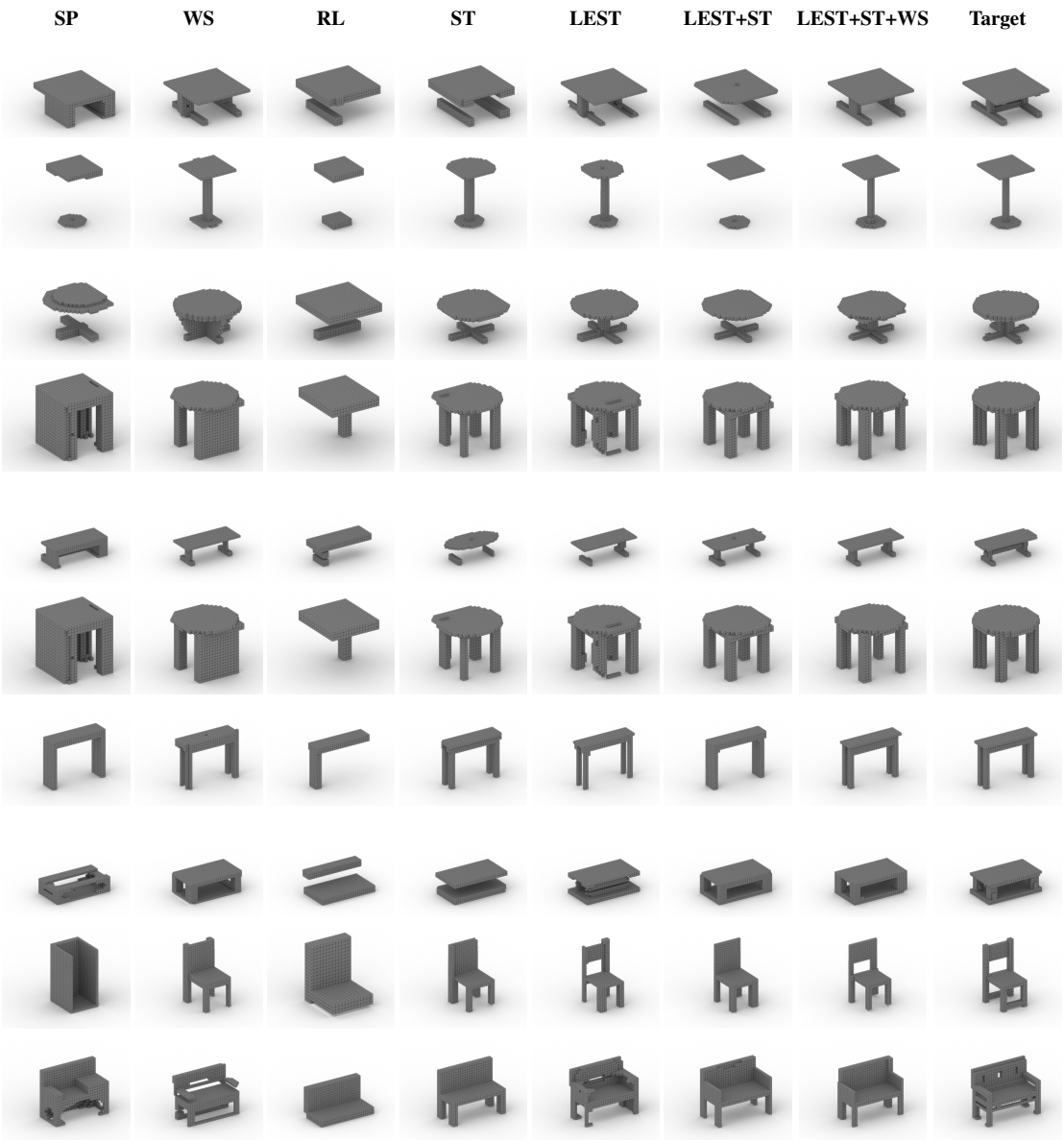

Figure B.3: 3DCSG qualitative examples.

Figure B.4: ShapeAssembly qualitative examples.

Appendix C

Additional Details and Results for VPI-Edit

In Appendix C, we supply additional details for the VPI-Edit method introduced in Chapter 5. In section C.1 we include more experimental results. We then provide additional details on our visual programming domains (Section C.2), on our experimental design (Section C.3), on our editing operations (Section C.4), and on our program corruption experiments (Section C.5).

C.1 Experimental Results

C.1.1 Performance on more challenging tasks

Our formulation employs a self-supervised finetuning scheme that specializes our inference networks towards a target dataset of interest. But how do our networks fare on visual inputs that are outside of these distributions? For instance, one might hypothesize that the performance gap between our joint paradigm and the *one-shot* paradigm might shrink when these approaches are given more challenging problems (e.g. when there is a large distribution gap between training and testing data).

Note though, that as we focus on local edits, our edit networks learn how to solve a local problem: given a current program and some visual target, we task our network with making any edit that would make the current program more similar to the target. Our hypothesis is that this framing should actually scale better

Table C.1: We evaluate reconstruction accuracy for "challenge" tasks that come from concepts or categories not present in the target training set. For both layout and 3D CSG, we observe that our joint paradigm that integrates an edit network with *one-shot* models outperforms the alternative of using only *one-shot* models.

	Layout *cIoU* ⇑	3D CSG *IoU* ⇑
OS Only	75.8	60.8
OS + Edit	**87.6**	**70.9**

than the *one-shot* networks when the target scenes become more complex or when they are further out-of-distribution from the training data.

Our intuition here, is that as the task complexity increases, it becomes more likely that the *one-shot* network will make mistakes. The edit network is able to account for the mistakes of the *one-shot* network and suggest local fixes that make improvements in a goal-directed fashion. When the target is out-of-distribution, even if the edit network has not seen a similar example, it can still compare the current program's execution against the target scene. Reasoning over the differences between the two states admits a more local task (as evidenced by our data efficient learning), and this property can aid in generalization.

To validate the above hypothesis, we set up an experiment to compare how our formulation (which uses a *one-shot* and edit network jointly) performs against using only the *one-shot* network for more challenging tasks in the Layout and 3D CSG domains. For the Layout domain, we evaluate the methods on scenes of new "challenge" concepts (e.g. butterflies / snowmen) that were not seen in the training / validation sets. For 3D CSG, we evaluate the methods on "challenge" shapes from other categories of ShapeNet (airplanes, knives, lamps, laptops, motorbikes, mugs, pistols, skateboards, rifles, vessels) that were not part of the original finetuning training set (chairs, tables, benches, couches).

Using the same models from Section 5.2.2, we compare the reconstruction performance for these challenge tasks. In Table C.1, we report the reconstruction performance over 192 challenge tasks for the layout domain and 100 challenge tasks for the 3D CSG domain. As seen from both the quantitative and qualitative comparisons (Figures C.1 and C.2), it's clear that our approach, which utilizes both the *one-shot* and edit networks, outperforms using only the *one-shot* network for these more challenging program induction tasks, even when they are further *outside* the training distribution.

C.1.2 Comparison to large vision-language models

We ran an experiment to explore how well large vision-language models (e.g. GPT-4v) are able to perform on our visual program induction tasks. We provide some qualitative results of using GPT-4v to predict visual

OS Only *OS + Edit (Ours)* *Target*

Figure C.1: Qualitative reconstructions of ”challenge” tasks for 3D CSG.

programs on examples from our layout domain in Figure C.2. These predictions were made with a relatively straightforward prompt containing: a task-description, a description of the DSL, and the input image that should be reconstructed (zero-shot, col 1). We then tried improving this prompt by adding an in-context example of a (program, image) pair (*one-shot*, col 2). We also experimented with providing GPT-4v with a program predicted from the *one-shot* network, along with this program's execution, and asking it to edit the program to make it more similar to the target image (col 3).

As can be seen, GPT-4v in this setting proved inferior to our proposed method (col 5). While we do not include these results to say that these sorts of large vision-language models will not *ever* be of use for this task, we do believe that these results showcase that this task is not *easily* solved with currently available frontier models.

C.1.3 Method Ablations on 2D CSG domain

In Section 5.2.5 we presented results for an ablation experiment on the layout domain. We include additional ablation results on the 2D CSG domain in Table C.2. Note that while some ablation conditions do come close

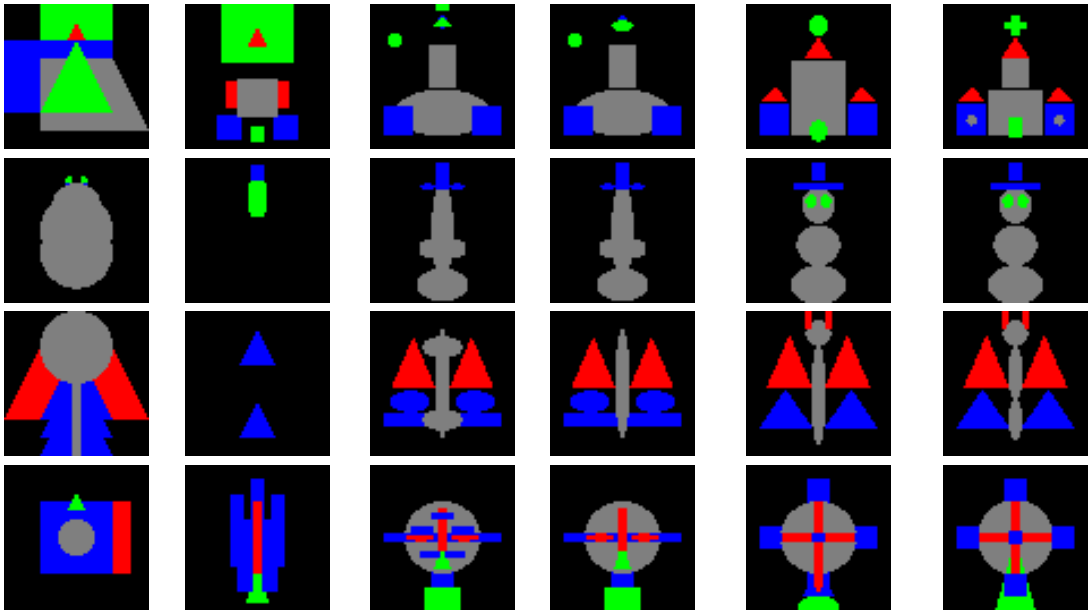

Figure C.2: Qualitative reconstructions of "challenge" tasks for the layout domain. We compare against GPT-4V in a zero-shot setting (column 1), when an in-content example (ICE) is provided in the prompt (column 2), and when the *one-shot* model's predicted program is provided as input (column 3). Our approach (column 5) finds more accurate reconstructions of these out-of-distribution targets (column 6) compared with using only the *one-shot* network (column 4).

Table C.2: Ablation study on our method for the *2D CSG* domain.

Method	Chamfer Distance ⇓
Ours (default)	**0.111**
No FT	0.321
No one-shot FT	0.230
No edit FT	0.123
No edit PT	0.145

to our default performance (e.g. *no edit FT*) these ablation conditions are also made possible by our contributions, as they all use an edit network. When comparing our method against an alternative without an edit network (*OS Only*, Table 5.1) we have consistently seen that our method offers a meaningful improvement. Below we offer some additional commentary on these results.

No edit FT In this ablation condition the edit network is pretrained (with synthetic random data), but is then kept frozen during the joint finetuning. As the task of the edit network is mostly local, we find that the edit network is able to achieve impressive performance even when it does not get to finetune towards data in the

target distribution. That said, the edit network is still very important in this ablation condition (if it's removed then this condition becomes *OS Only*). Even though the edit network remains fixed during finetuning, it still helps to find better solutions during inner-loop inference (Alg 1, line 5), and this better training data leads to a better *one-shot* network. However, once again, the performance of the system is maximized when the edit network is also allowed to update during finetuning.

No one-shot FT This condition does impressively well for the layout domain. This is because even though the *one-shot* network is much worse in this setting, the edit network can overcome almost all of its mistakes, as layout is a relatively easier domain. Consider that for the layout domain, the default approach has a starting cIoU of 0.925 (initialized from the *one-shot* network, which is finetuned) which gets improved to 0.980 through improvements made by the edit network. However, the *one-shot* network of this ablation condition drops the starting cIoU to 0.88 (when it is kept frozen), and yet the edit network is still able to raise this performance all the way to 0.972 (explaining the strong reconstruction score of this condition). That said, when considering the 2D CSG ablation results in Table C.2, we see that for more complex domains it is critical to also finetune the *one-shot* network, as this ablation condition achieves only a Chamfer distance of 0.230 compared with the Chamfer distance of 0.111 achieved by our default approach.

C.2 Domain Details

In this section we detail the domain-specific language used for each visual programming domain.

Layout DSL The layout domain creates scenes by placing colored primitives on a 2D canvas, optionally transforming them, and finally combines them together.

$START \rightarrow UBlock;$

$UBlock \rightarrow \texttt{UNION}(ShBlock, UBlock) \mid ShBlock;$

$ShBlock \rightarrow (SymBlock \mid CBlock \mid MBlock \mid ScBlock); (PBlock \mid UBlock)$

$SymBlock \rightarrow \texttt{SymReflect}(axis) \mid \texttt{SymRotate}(n) \mid \texttt{SymTranslate}(n, x, y)$

$CBlock \rightarrow \texttt{Color}(ctype)$

$MBlock \rightarrow \texttt{Move}(x, y)$

$ScBlock \rightarrow \texttt{Scale}(w, h)$

$PBlock \rightarrow \texttt{Prim}(ptype)$

$axis \rightarrow X \mid Y$

$ctype \rightarrow red \mid green \mid blue$

$ptype \rightarrow square \mid circle \mid triangle$

$n \in (1, 6)$

$x, y, w, h \in [-1, 1]$

In this domain, union is the only combinator operation that combines 'shape'-typed inputs by layering them on top of one another. SymReflect, SymRotate, SymTranslate, Color, Move, Scale are all transformation operations that consume a single 'shape'-typed input and apply some geometric logic to it. Prim is a special command that produces a 'shape'-typed output from only a parameter-type argument.

2D CSG DSL Our 2D constructive solid geometry domain assembles complex shapes using boolean set operations. Following recent work [235] we find it useful to split each program into a set of positive sub-expressions (POS) and negative sub-expressions (NEG). Each sub-expression is allowed to take an arbitrary CSG expression, and then to form the final output all of the positive sub expressions are first unioned together, all of the negative sub expressions are then unioned together, and this second group is differenced out from the first group. This process well-matches typical procedural modeling workflows.

$$START \rightarrow POS, NEG$$

$$POS \rightarrow E, POS \mid \emptyset$$

$$NEG \rightarrow E, NEG \mid \emptyset$$

$$E \rightarrow BEE \mid TE \mid P$$

$$B \rightarrow Union \mid Difference \mid Intersection$$

$$T \rightarrow Move(F, F) \mid Scale(F, F) \mid Rotate(F) \mid Reflect(axis)$$

$$P \rightarrow \texttt{Prim}(ptype)$$

$$ptype \rightarrow square \mid circle \mid triangle$$

$$axis \rightarrow X \mid Y$$

$$F \rightarrow [-1, 1]$$

In this domain, there are three combinator operations that combine multiple 'shape'-typed inputs: union, difference and intersection. Move, scale, rotate and reflect are all transformation functions that consume a single 'shape'-typed input and apply a geometric modification. Once again, Prim is a special command that produces a 'shape'-typed argument from only a parameter-type argument.

3D CSG DSL Our 3D constructive solid geometry domain generalizes the above 2D DSL.

$$START \rightarrow POS, NEG$$

$$POS \rightarrow E, POS \mid \emptyset$$

$$NEG \rightarrow E, NEG \mid \emptyset$$

$$E \rightarrow BEE \mid TE \mid P$$

$$B \rightarrow Union \mid Difference \mid Intersection$$

$$T \rightarrow Move(F,F,F) \mid Scale(F,F,F) \mid Rotate(F,F,F) \mid Reflect(axis)$$

$$P \rightarrow \texttt{Prim}(ptype)$$

$$ptype \rightarrow cuboid \mid sphere \mid cylinder$$

$$axis \rightarrow X \mid Y \mid Z$$

$$F \rightarrow [-1, 1]$$

The split between combinator, transformation and primitive creating functions is the same as in 2D CSG.

Sampling L As previously discussed, we follow prior work and use a synthetic pretraining phase . In this pretraining phase we randomly sample programs from the above grammars. We employ simple rejection criteria to ensure these random samples are useful (e.g. no execution errors, outputs remain within the canvas, etc.), and find it effective to build in some of this rejection logic during the sampling phase (to improve the speed at which we can sample programs). All of the models we evaluate in our experiments train with the same sampling logic.

C.3 Experimental Design Details

Network details For our 2D domains (2D CSG and Layout) we use a 2D CNN. The image size of both domains is 64x64, but in 2D CSG there is only one input feature (occupancy) while in Layout there are three channels (RGB). The network we utilize consists of four layers, each containing convolution, ReLU, max-pooling, and dropout operations. Each convolution layer employs a kernel size of 3, a stride of 1, and padding of 1, with channel dimensions of 32, 64, 128, and 256 respectively. The CNN's output is a (4x4x256)

dimensional vector, which we reshape into a (16x256) vector. This vector is then processed through a 3-layer MLP with ReLU and dropout, resulting in a final (16x256) vector that serves as a 16-token encoding of the visual input. For our 3D CNN model, we adopt a similar convolutional approach by extending all 2D convolutions to 3D. We adjust the kernel size to 4, use padding of size 2. When processing voxel grids of size 32^3, this produces outputs of size (2x2x2x256). We pass these outputs through a 3-layer MLP to generate eight 256-dimensional visual tokens.

Our transformer networks are standard decoder-only variants. We use learned positional encodings and a hidden-dimension size of 256 and dropout of 0.1. We use networks with 8 layers and 16 heads. We set the maximum program sequence length *SL* to 128, 164, 256 for the Layout, 2D CSG, and 3D CSG domains respectively. We set the maximum edit sequence length *EL* to 32, 32, 48 for the Layout, 2D CSG, and 3D CSG domains respectively. Each prediction head (edit type, location, parameters) is modeled with a three-layer MLP with a dropout of 0.1.

Training details We implement all of our networks in PyTorch [158]. All of our experiments are run on NVIDIA GeForce RTX 3090 graphic cards with 24GB of VRAM and consume up to 128GB of RAM (for 3D CSG experiments). We use the Adam optimizer [106] with a learning rate of 1e-4. For $p(z|x)$ pretraining we use a batch size of 128/128/64, for $p(e|z,x)$ pretraining we use a batch size of 128/128/32, for $p(z|x)$ finetuning we use a batch size of 20/20/20, and for $p(e|z,x)$ finetuning we use a batch size of 128/128/32 for Layout / 2D CSG / 3D CSG domains respectively. We pretrain on synthetic programs until convergence with respect to a validation set of synthetic program, for 34 / 17 / 18 million iterations, which takes 6 / 7 / 7 days for $p(z|x)$ and 70 / 30 / 25 million iterations, which takes 7 / 8 / 8 days for $p(e|z,x)$ for the Layout, 2D CSG, and 3D CSG domains respectively. We finetune each method for a maximum of 6 days or until convergence, which took 40 / 40 / 30 bootstrap rounds for the Layout, 2D CSG and 3D CSG domains. For each finetuning run we use a P^G set of size 10000.

Inference Procedure For our test-time inference program search we use the following population size / number of round parameters for each domain: Layout (32, 32), 2D CSG (32, 32), 3D CSG (80, 25). When using the *Os Only* method, we keep the same population / mutation general logic, but each mutation is just a randomly sampled program from $p(z|x)$. In both cases, the best reconstructing program ever seen in any round's population is returned as the 'chosen' program. The settings for this method are: Layout (32, 10), 2D CSG (32, 10), 3D CSG (25, 25). We set these parameters so that the time spent on inference per shape is

even between the two modes (5, 10, 60 seconds for the three domains). For our inner-loop inference step that populates P^{BEST}, we use a less expensive search time budget for both modes, approximately taking (2, 5, 10 seconds for each domain respectively). We sample programs from $p(z|x)$ with top-p (.9) nucleus sampling. We sample edits from $p(e|z, x)$ with a beam search of size 3. Interestingly, we found that this sampling strategy for *Os Only* outperformed a beam-search with a beam size set to the maximum number of tokens in each L.

C.4 Visual Program Edits

C.4.1 Local Edit Operations

As described in Section 5.1, our network predicts local edit operations. We find it useful to constrain the set of possible edit operations as described in Section 5.2.5.

In order to use these local edit operations, we require a few properties of the underlying DSL. We require that it is a functional language, where each valid function has a 'shape' return type. Through a slight abuse-of-notation, we refer to functions that implicitly consume a single 'shape'-typed argument as transformation functions (e.g. *Move*), and we refer to functions that consume multiple 'shape'-typed arguments as combinator functions (e.g. *Union*). Note that as described in Section C.2, there may also be special functions that instantiate 'shape'-types from only non-'shape'-typed arguments (e.g. *Prim* functions).

Specifically, our formulation allows the network to predict one of the following edit operations:

- **Modify parameters** (*MP*): modifies the parameter values of a transform function. Note that this does not modify the function type (unlike *MT*). Requires additional parameter predictions to set the new values.
- **Modify transform** (*MT*): modifies a transformation function, by removing the transform and adding in a new transform with new parameters. Requires additional parameter predictions to set the new function and parameter values.
- **Add transform** (*AT*): adds a transform operator that is applied to the chosen location. Requires additional parameter predictions to specify the new function to be added and its parameters.
- **Remove transform** (*RT*): removes a transform operator and its parameter from the program. Does not require additional parameters

- **Modify Combinator** (*MC*): modifies a combinator function (e.g. changing difference to an intersection). Requires additional parameter predictions to set the new function.
- **Remove Combinator** (*RC*): removes a combinator operator (e.g. union) by specifying one input branch of the function to be completed deleted (to all of this sub-expressions leaf nodes).
- **Add Combinator** (*AC*): adds a combinator operator under the chosen transformation. Adding a combinator (such as union) requires a sequence of additional predictions to fill in one of the 'shape'-typed branches of this operator that was not previously in the program.

We once again note that each of these edit operations has a local effect. For instance, as depicted in Figure 5.1 adding a new transform function inserts a transform node into an already existing tree of functions. Similarly, removing a transform functions simply results in forming a skip connection from the chosen operator's parent function to the chosen operator's child function. Somewhat more arbitrary changes can be enacted by removing or adding combinators, in order to produce or remove entire expression trees, though these are inserted or removed from specific locations. While this framing does focus on local edits, and as such our edit network makes local changes in program space, some of these changes can have dramatic effects in the execution space. For instance, consider changing a boolean operation type in CSG from difference to union.

C.4.2 findEdits Algorithm

Given a starting program and an end program we develop an algorithm that analytically finds a set of edit operations that would transform the starting program into the end program. This algorithm is used to source data for the edit network, as we describe in the next section.

We design our findEdits algorithm to try to find the "minimal cost" set of edit operations that would transform a start program to an end program. Our instantiation of the algorithm works over multiple visual programming domains for the set of edit operations we consider. However, there are many alternative ways this algorithm could be instantiated, and such alterations could prove useful in helping our method adapt for very different domains. As one extreme point, consider that for general purpose programming languages, a simple "git-diff" command could be used to turn a (start, end) program pair into a set of local insert/keep/delete edits.

Our implementation evaluates valid transformations in terms of program semantics (e.g. executions) not just syntax (e.g. token matching), as there are many distinct programs in our domains that will produce

equivalent execution outputs (e.g. note that the argument ordering of union for CSG languages does not change the execution output). We hypothesize that using a findEdit algorithm alternative that does not consider such "semantic-equivalences" would result in a "worse" edit network (as the patterns in the training data would be less consistent), but it would be interesting to explore how different algorithms would effect system performance in future work.

There are two main steps to this algorithm. First considering two sub-expressions a and b, we need to find an approximately minimal set of edit operations such that applying these edit operations to a would recreate the visual output of b. With this logic in hand, we can consider two entire programs A and B, split them into a set of sub-expressions, $A = \{a_0, ..., a_k\}$ and $B = \{b_0, ..., b_m\}$, and then solve a matching problem to see how we should match each a_i to each b_j while accounting for domain-specific ordering requirements.

Finding edits for sub-expressions Given two sub-expression a and b from one of our DSLs, we find a set of edit operations to convert a to b with the following recursive logic. If a and b have no combinator operators or order-dependant transformation functions (e.g. symmetry operations) then we can simply compare the transform functions and their arguments to see which transforms in a need to be modified, added, or removed. If both a and b have a combinator operation, then we recurse this match on the respective sub-programs. If only a has a combinator operation, we know that we need to remove one of a's expression trees, so we check which of the combinator's input expression trees has the better match towards b. If only b has a combinator operation, we know that we need to add an expression tree into a with an AC edit operation. The cost of this edit operation is just the length of all of the tokens of that expression tree; we evaluate the match between a and each of the sub-expression within b to determine which sub-expression to add with the edit operation. Any time an order dependant transform function differs between a and b we will either need to add, remove, or modify this transform. Note that this type of edit operation may also introduce ordering dependencies for later edit operations (which we keep track of).

Finding a minimal matching From the above procedure we know the edit operations and the edit cost of transforming any sub-expression a into another sub-expression b. We design our DSLs so that it is possible to break each program into a series of sub-expressions. For *Layout* this is done by splitting the top-level *UBlock* into the top-level *ShBlock*s. For *CSG* this is done by splitting each *POS* block into *E* blocks and each *NEG* block into *E* blocks. Note that there is some order dependency in this match: for CSG positive sub-expressions must be matched to other positive sub-expressions, while negative sub-expressions must be

matched to other negative sub-expressions. For the *Layout* domain, *Union* is not an order invariant operator as it controls how primitives are layered on the canvas. Therefore we keep the order of *Layout* sub-expressions fixed, although we allow each sub-expression to optionally match to an empty sub-expression $\emptyset$. A match from a_i to $\emptyset$ implies that a_i will be removed with a *RC* edit operation, while a match from $\emptyset$ to b_i implies that b_i will be added with a *AC* edit operation. We consider all valid possible ways to enact this matching by calculating the cost of each sub-expression match and then extracting out a solution with the Hungarian matching algorithm [111].

C.4.3 Converting edits operations to training data

From the above logic we find a set of edit operations ES given input programs A and B. As mentioned, while there may be some ordering dependencies in this set that we keep track of (e.g. adding a transform on top of newly added combinator function) this set of edit operations can be otherwise ordered arbitrarily. While many formulations are possible here we choose to convert this set into paired data for our edit network with the following procedure.

Say ES contains n independent edits. For each i starting at 0 and ending at $n-1$ we first consider all possible ways that we could have chosen i edits from ES. To avoid exponential blow-up, we sub-sample from this set, and choose 5 previous edit sets for each i. Then for each set of previous edits pe_i, for each next edit $e \in ES$ and $e \notin pe_i$, we add the following triplet to the training data for our edit network: the input program is $pe_i(A)$, the target visual target is $E(B)$, and the target edit operation is e.

C.4.4 Generality of our framing

While we designed our edit operations with the task of visual program induction in mind, we believe that these operations are quite flexible. Many other functional DSLs for visual programs (and for other program synthesis tasks) could likely be subsumed directly under our framework, as long as these languages meet the criteria described in Section C.4.1. For instance, this set of edit operations should be able to handle any DSL expressible as a Context Free Grammar.

Under these assumptions, the edit operations we use are quite basic and make limited domain assumptions. For an input functional program, edits to transform functions allow for local edits (delete/insert/modify) that don't affect the branching factor, while edits to combinator functions allow for local edits (delete/insert) that do affect the branching factor. We employ this formulation for a few reasons: (1) it is general enough to

support any program-program transformation (under our assumption set) and (2) applying any of these local edits creates a syntactically complete program that can be immediately executed.

That said, our framework and core contributions are not tied to this specific set of edit operations. Our edit network and proposed training scheme could be easily integrated with any set of local edit operations (assuming an analogous findEdits algorithm can be designed for this new set of edits). So while we believe that the set of edit operations we introduce is quite general (as evidenced by their usefulness across multiple difficult visual programming domains), we are also excited to see how our general learning-to-edit framework could be extended to even more complex DSLs and edit operations.

C.5 Program Corruption

As we mention in Section 5.2.5 there are some high-level connections between the formulation we propose and discrete diffusion models: both do iterative error-correction and learn in a self-supervised manner to 'fix' incorrect targets. To this end, we explored alternative formulations that 'corrupted' programs. As we wanted to maintain the property that each intermediate step of the 'corruption' process is a valid program (e.g. it would not cause an executor error) we designed a domain-specific corruption process for our Layout domain. Unlike unconditional generative diffusion models that need to have strict requirements about the distribution they noise towards, we did not find this necessary in our setting as our iterative error-correcting framing is explicitly goal-directed in the form of a visual target. Specifically, our corruption process starts with an 'end' program and randomly samples 'inverse' edit operations for a random number of corruption steps. We then replace our *findEdits* step in Algorithm 1 with this corruption logic, where the start program is ignored.

While this variant is not as a performant as our default version, it still sources useful training data for our edit network. Our view is that, when possible, it is better to source these edit operations by considering start program and end program pairs, but for domains where such edit difference scripts are hard to analytically find, this corruption variant offers an alternative. While its possible that better corruption processes could close this gap, designing them is non-trivial. Ideally, when we want to combine *one-shot* models and edit networks at inference time, the corruption behavior we want should noise 'end' programs towards those produced by the *one-shot* model – this is exactly the distribution we get access to with the *findEdits* approach that considers program-to-program transformations. Another benefit of this formulation, is that the distribution of edit operations we train over is naturally allowed to evolve and keeps in sync automatically with the finetuned *one-shot* model. Keeping this property with a corruption-based procedure would likely be impractical.

Appendix D

Additional Details and Results for Template Programs

In Appendix D, we supply additional details for the Template Programs method introduced in Chapter 6. In Section D.1 we provide additional experimental results. In Section D.2 we provide more information concerning our various visual domains. In Section D.3 we provide details of our learned models. In Section D.4 we provide details on how we design our training procedure. In Section D.5 we provide further details of our experimental design. Finally, in Section D.6 we describe implementation considerations of each alternative we compare our system with.

D.1 Additional Results

D.1.1 Out-of-distribution Few-shot Generation

As discussed in Section 6.2.5, we designed the Layout domain so that we could evaluate the out-of-distribution generalization capabilities of different approaches. We visualize few-shot generations that different methods make for the layout domain for concepts that gradually get more and more out-of-distribution in Figure D.1. From left-to-right, we present example few-shot generations for an easy, medium and hard concept. The easy concept (a side facing chair) has a set of attributes that have all individually been seen in the training set, but presents them in a new combination. The medium concept (a crab) introduces a new attribute not seen in the training set: extended and vertical arms. The hard concept (a bookshelf) introduces a new meta-concept

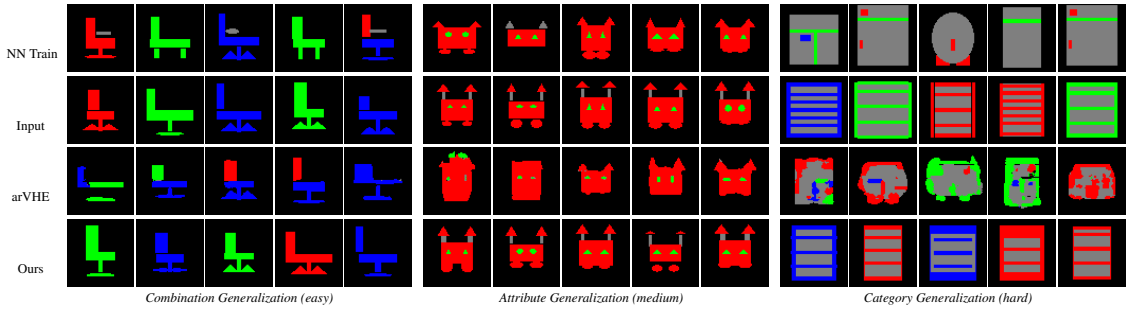

Figure D.1: Qualitative few-shot generation results that demonstrate our method's ability to generalize to out-of-distribution concepts, see Section D.1.1.

that was never seen in the training set. The second row of the figure show the input prompt set, where in the top row we show the nearest neighbor in the training set to each image in the prompt, according to our reconstruction metric. On the third row we show generations produced by the arVHE comparison condition, while on the bottom row we show generations produced by our method. While arVHE does reasonably well on the easy case, as the input prompts get more and more out-of-distribution it begins to generate nonsensical outputs. On the other hand, our approach scales much better to out-of-distribution inputs, even though they don't match any images from the training set.

D.1.2 Method Ablation Study

We run an ablation study to validate different design decisions of our method. We compare our described system against the following variants. *Ours - rel* is a variant of our method where we remove parameter relationships from Template Programs. As by default we only support parameter relationships for argument types that take on discrete values (i.e. categorical variables) we also investigate a variant of our system that adds parameter relations (static assignment and reuse) for float-typed arguments: *Ours + float rels*. We also compare against a version of our method where we remove *HOLE* tokens, so that instantiations from Template Programs always use the same function call sequence: *Ours - HOLE*. Here, we task our network to specify a single program structure that is applicable across the group without using *HOLE* tokens, and it is still responsible for declaring parameter relationships. As there are no *HOLE* tokens, the ExpansionNet will not be used, but the ParamNet will still be used to figure out how the instantiations of the Template Program should be parameterized. Next we compare against a variant where we remove the Structural Expansion step, so the ParamNet must produce a program from the Template Program directly. As it doesn't see the SE intermediary result, it must fill in *HOLE* tokens while figuring out how to predict parameter values. We

Table D.1: Comparing ablated versions of our method to our default settings. Each metric is reported as a percentage, with respect to the performance our default approach achieves. See Section D.1.2 for details.

Method	**FD**	**MMD**	**Cov**	**mIoU**	O
Ours	**100%**	**100%**	**100%**	**100%**	**100%**
Ours - rels	78.5%	92.6%	96.8%	89.3%	95.8%
Ours + float rels	93.9%	96.7%	98.2%	96.7%	95.3%
Ours - *HOLE*	96.3%	98.3%	97.6%	97.4%	98.0%
Ours - SE	80.3%	89.9%	94.3%	86.5%	94.7%
Ours - finetune	57.6%	80.5%	35.0%	70.7%	81.6%

call this variant *Ours* - SE. Finally, we compare against a variant of our base method without any finetuning, where networks only get to train on synthetic data: *Ours - finetune*

We evaluate these ablation conditions on the 2D layout domain, and report results of our experiments in Table D.1. We compare our method against these variants with respect to few-shot generation performance (*FD, MMD, Cov*), co-segmentation performance (*mIoU*), and how well the inferred results optimize our objective O. For ease of interpretation, we report all results as a percentage of the performance reached with respect to our default version (100%).

As shown, our default method achieves the best performance along all of these tracked metrics. The variant without finetuning clearly does the worst, as these networks are not specialized for the target dataset. The results of this experiment validate our parameter relations design: keeping relations for discrete-valued parameters outperforms either no parameter relations or adding relations for float-valued parameters. Using the *HOLE* construct improves performance quantitatively. Moreover this construct is needed to capture complex input concepts that have more than a single expression mode. For instance *HOLE* tokens are required to model the chair concept with armrests and either a regular or pedestal base shown in the bottom left of Figure 6.2. Finally, this ablation experiment demonstrates that our decision to use Structural Expansions simplifies the task of the ParamNet; we hypothesize this result is due to the fact that when attending over a SE, in contrast to attending over the TP, all of the functions and parameter-types that will be used in the end instantiation are known.

D.1.3 Unconditional Concept Generation

As we mention in Section 6.2.5 our Template Program framework is able to sample novel concepts unconditionally. We visualize some concepts that our method is capable of producing in Figure 6.5. To produce these visualizations, we use the networks trained during the wake-sleep phase of our finetuning process, p_{gen}.

Using the version of our TemplateNet from p_{gen} that does not condition on visual information, we first sample a Template Program. Then using the ExpansionNet and ParamNet from p_{gen} that condition only on program inputs, we sample five program instantiations from this Template Program. Each bottom row in the figure shows the executed versions of these five samples, and above each sample we show the nearest neighbor character in the training set according to our reconstruction metric.

D.1.4 Visual Concept Groupings

Typically, past concept learning approaches have assumed access to a dataset that is structured according to visual concepts. For instance, systems like VHE or FSDM require the ability to sample groups of input from the same visual concept during training. This is the same amount of dataset structure that our method requires: during fine-tuning we randomly sample "tasks" according to these visual concept groupings. Note that this requirement is less stringent than many inverse procedural modeling systems, and the BPL and GNS systems, that additionally require per-object structural annotations.

The Omniglot dataset was designed with this kind of visual concept decomposition in mind: each example data-point corresponds with exactly one character type. We design our layout domain in the spirit of Omniglot: each image in the layout domain is associated with a single concept. Following past work, on these domains we always assume "valid" input groups, such that each member is from the same visual concept.

However, this type of clean partition is not as easy to find for 3D shape structures. As there are no known datasets that group shape structures into visual concepts, we propose a heuristic method for forming approximate visual concepts out of shape structures (Appendix D.2.3). The concept groups we find under this formulation have different levels of consistency among their members (where we say a less consistent group forms a "harder" input problem).

For instance consider the examples shown in Figure 6.2. A chair with a regular base and vertical slats (row 7, col 6) could be in one group with only chairs that also have regular bases and vertical slats (row 7, col 7) or (like in the example we show) could also be grouped with chairs that have backs with horizontal slats (row 7, col 8). In our paradigm, the group of visual inputs (along with our objective function) implicitly defines the granularity of the target visual concept. In this case, the latter grouping is considerably harder to handle for concept learning tasks, as it requires a method that is able to reason over input groups that partially mismatch on structures.

Our Template Programs framework is capable of handling even difficult input groups; our partial program

formulation allows our system to explicitly maintain the shared structural aspects of the group while leaving *HOLE* tokens as responsible for representing the aspects of the input group that structurally differ. This design allows us to successfully capture the visual concepts of the two chair groups in Figure 6.2. The left chair group has filled in chair backs, arm-rests, but alternates between regular and pedestal chair bases. The right chair group has regular leg bases, no arm-rests, but differs between chair backs with horizontal slats or vertical slats. As can be seen in the "gen" row, our system is capable of synthesizing novel shape structures that accord with the structural specifications implied by the input visual groups.

D.1.5 Reconstruction Performance

Our system learns how to amortize the difficult inverse search problem of finding a Template Program and instantiations that correspond with a group of visual inputs. This search (our inference procedure) is guided by our networks which are trained on a "training corpus" of visual concepts, separate from those we evaluate on.

The "seg" rows in Figure 6.2 visualize the reconstructions (of the inputs on the top rows) that our method produces. While these reconstructions do not exactly recreate the input, they usually create very good approximations. If reconstruction was our primary goal, it might even be possible to improve the fit through a differentiable execution and refinement procedure.

To explore this phenomenon further, we provide the following reconstruction performance results across our domains in Table D.2. We report the reconstruction fit for both the training set and test set visual concepts. To show the benefits of our learning methodology we compare the reconstruction fit from the pretrained version of our networks (that learn only on synthetic data) to the finetuned versions of our networks (that finetune on visual concepts from training set). The metrics we use are (full descriptions in Appendix D.2):

- **2D Layout**: color-based IoU (higher is better)
- **Omniglot**: edge-based chamfer distance (lower is better)
- **3D Shapes (primitive input)**: structural corner distance (lower is better)
- **3D Shapes (voxel input)**: IoU (higher is better)

As demonstrated, our solution is effective at solving this inverse visual program induction problem. For both the training concepts and the held-out test concepts, our finetuning procedure meaningfully improves

Table D.2: Comparing reconstruction performance across domains, concept sets, and model versions.

Domain	Mode	Train Recon	Test Recon	Test Recon (long)
2D Layout ↑	Pretrain	.822	.808	
	Finetune	.972	.909	.937
Omniglot ↓	Pretrain	.658	.648	
	Finetune	.468	.503	.405
3D Shapes (prim) ↓	Pretrain	.26	.305	
	Finetune	.05	.06	.05
3D Shapes (voxel) ↑	Pretrain	.601	.589	
	Finetune	.865	.83	.851

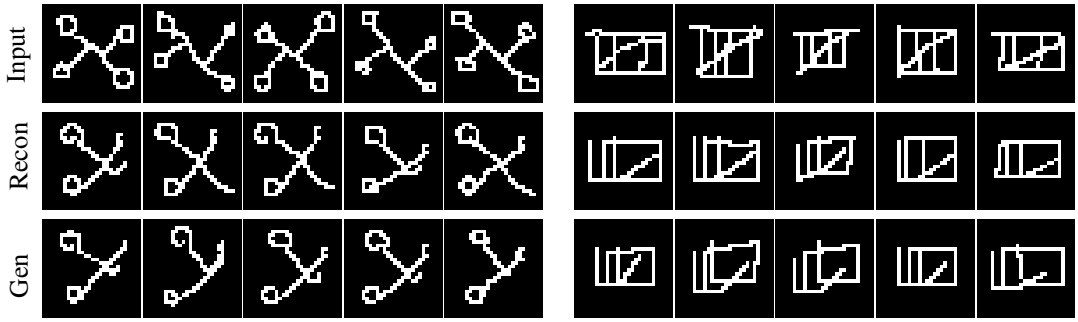

Figure D.2: When our method fails to find good reconstructions of an input concept, downstream task performance worsens.

the reconstruction performance in all cases. For our downstream concept-related tasks we use a more expensive inference procedure ("long" - e.g. increase the beam size, Section 6.1.2) and this gives even better reconstruction results for test-set concepts (see the numbers in the rightmost column of the table).

While our system offers strong reconstruction performance, it is likely that alternative methods could be used to infer single visual programs that better reconstruct an individual visual input. In contrast, our system learns to solve this visual program induction problem over a group of inputs by going through a shared structural intermediary (a Template Program), which allows us to perform concept-related tasks like few-shot generation and co-segmentation (which prior single instance VPI approaches are not suited for).

D.1.6 Failure Modes

Bad reconstruction A possible failure mode is that our inference networks can't find a Template Program whose instantiations well-capture an input visual group with respect to our objective function. In such cases, the few-shot generation and co-segmentation results of our method are typically worse. For instance, consider

Figure D.2. For two Omniglot examples, in the top row we show the input concept groups, in the middle rows we show the reconstructions from our method, and in the bottom rows we show the few-shot generations from our method. Because the same Template Program is used in both the reconstruction and few-shot generation step, failure in one place often means failure in the other. While from one perspective this is a limitation, a positive view of this phenomena is that our method can provide insight into cases where it is "unsure" about its parse. For instance, it could use the objective function score of its reconstruction as a measure of its confidence on how well it will perform on downstream tasks. Moreover, as we show in Table D.2, reconstruction performance can improved by spending more time on inference, which can help to avoid this limitation.

Bad Input Groups How would our system handle 'bad' input groups that contain outliers, or have no commonality among their members? The job of the template network is to consume a group of visual inputs and infer a Template Program that captures the common structure among all members. In such an adversarial setting, it is possible (depending on random sampling) that there are no elements of structure common to all members of the input grouping.

In this case, the "best" result of our system would be to return a "dummy" Template Program that consists of a single *HOLE* token; this *HOLE* token would be able to be expanded into any arbitrary z to explain each individual group member.

For typical visual concept groupings, this degenerate solution is discouraged by our objective function, which penalizes description length differences between "full" programs and their corresponding Template Programs. While finetuning our system with reasonable concept groups we have never observed the system falling-back to this degenerate solution.

Exploring how to extend our framework to handle "noisy" input groupings would be a very interesting direction for future work. This could potentially be approached by (i) extending our objective function to account for outliers (if we want to ignore the distractors) or (ii) adding control flow operators into the DSLs we learn over, which would give the Template Programs an opportunity to account for structural differences without relying solely on *HOLE* tokens.

D.2 Domain Details

In this section we provide additional details on the visual domains we experiment on. We describe the domain-specific languages our method uses and reconstruction metrics that guide our finetuning objective O.

For the 3D shape domain, we additionally provide details on how we produce our target dataset. While we have previously explained how we divide concepts between training and test sets for each of our domains, we have not yet mentioned how we divide training examples into a validation set. We find that a simple approach of taking a subset of training concepts with fixed exemplars as a 'validation' set works well in practice. This validation set controls different early stopping components of our finetuning procedure, but otherwise these concepts are not given special treatment (i.e. they are not removed from the finetuning training set).

D.2.1 Omniglot

DSL We use the following domain-specific language for drawing Omniglot characters, where we present the notation with slight simplifications for ease of understanding.

$$START \rightarrow GBlock;$$

$$GBLock \rightarrow ONBlock \mid OFFBlock \mid MBlock \mid END$$

$$ONBlock \rightarrow \texttt{ON}; SBlock; GBlock$$

$$OFFBlock \rightarrow \texttt{OFF}; SBlock; GBlock$$

$$MBlock \rightarrow \texttt{MOVE}(si, mt, mf); GBlock$$

$$SBlock \rightarrow Stroke \mid \texttt{BOW}(bt, bf);\ Stroke \mid \texttt{EMPTY}$$

$$Stroke \rightarrow \texttt{DRAW}(at, af, dt, df)$$

$$si \in [0, 12]$$

$$dt \in [0, 8]/8$$

$$at \in 360 * [0, 8]/8$$

$$bt \in 90 * [-2, 2]$$

$$mt \in [0, 4]/4$$

$$df \in [-2, 2]/40$$

$$af \in 9 * [-2, 2]$$

$$bf \in 30 * [-1, 1]$$

$$mf \in [-1, 1]/12$$

The ON and OFF commands lift a pen on and off a virtual canvas; each series of strokes begins with one of these commands. The MOVE command brings the pen back to a previous stroke, specified by a stroke index (*si*) and a length along this stroke to travel specified by (*mt, mf*). The DRAW command moves the pen at an angle specified by (*at*, *af*) for a distance of (*dt*, *df*). The trajectory of each DRAW command can be controlled by a BOW command which optionally pushes the trajectory inwards or outwards according to (*bt*, *bf*) parameter. Even if making a curved stroke through the BOW operator, the end location of the pen is

entirely controlled by the parameters of the `DRAW` command.

We draw attention to the fact that each real-valued parameter in this language is represented with a pair of arguments. One member of each pair (those with *t*) controls the coarse behavior, while the other member of the pair (those with *f*) add a fine-grained delta to the initial coarse value (i.e. their values are combined through summation during execution). This representation promotes consistency as close values will match on coarse binning token indices. We further find it useful to treat these 'coarse' real-valued parameters as categorical variables for the purposes of defining parameter relationships in the declaration of Template Programs, but we don't observe similar benefits when fine-grained values are included in this categorization (see ablation in Section D.1.2). *HOLE* tokens are allowed to take place of any function.

Reconstruction Metric For our reconstruction metric M , we use an edge-based Chamfer distance [187]. This allows us train our networks without access to stroke data, as we can compute this metric directly from binary images.

Representational Capacity The maximum complexity of characters that our method is capable of representing is bounded by (i) the maximum number of tokens that our inference networks can handle and (ii) the maximum number of strokes we sample in the synthetic programs used in the pretraining step. This latter value is set to 12 in our sampling scheme, although through the introduction of HOLE tokens in Template Programs, some of the synthetic programs may end up using more than 12 stroke primitives. While the synthetic pretraining distribution will inform the behavior of the inference networks, this distribution will change over the course of bootstrapped fine-tuning and specialize towards "real" Omniglot examples.

While we observe that these settings allow our model to reliably capture the majority of Omniglot characters, there are some very complex characters that might be hard to fit under these constraints with our top-down inference procedure. It should be possible to relax the constraints of both (i) and (ii), although the cost would be a larger GPU memory footprint and more complex pretraining data, which might require more training time and/or inference networks that use more parameters.

D.2.2 2D Primitive Layout

DSL We use the following grammar for creating layouts of 2D colored primitives. We present a slightly simplified representation of this language for clarity.

$START \rightarrow UBlock;$

$UBlock \rightarrow \texttt{UNION}(ShBlock, UBlock) \mid ShBlock;$

$ShBlock \rightarrow (SymBlock \mid CBlock \mid MBlock \mid ScBlock); (PBlock \mid UBlock)$

$SymBlock \rightarrow \texttt{SymReflect}(axis) \mid \texttt{SymRotate}(n) \mid \texttt{SymTranslate}(n, xt, xf, xt, yf)$

$CBlock \rightarrow \texttt{Color}(ctype)$

$MBlock \rightarrow \texttt{Move}(xt, xf, yt, tf)$

$ScBlock \rightarrow \texttt{Scale}(wt, wf, ht, hf)$

$PBlock \rightarrow \texttt{Prim}(ptype)$

$axis \rightarrow X \mid Y$

$ctype \rightarrow red \mid green \mid blue$

$ptype \rightarrow square \mid circle \mid triangle$

$n \in (1, 6)$

$xt \in [-3, 3]/4$

$yt \in [-3, 3]/4$

$wt \in .35 * [1, 6] - .15$

$ht \in .35 * [1, 6] - .15$

$xf \in [-2, 3]/20 - 0.025$

$yf \in [-2, 3]/20 - 0.025$

$wf \in [-3, 3]/20$

$hf \in [-3, 3]/20$

Our language uses a UNION combinator to assemble a collection of primitives on a 2D canvas. Primitives

can take three types: squares, circles and triangles. They are consumed by `MOVE` and `SCALE` operators, where similar to our Omniglot domain, we make a distinction between the coarse and fine parts of each real-valued argument. Once again, we distinguish the coarse values with t endings and the fine values with f endings. Our motivations for adopting this tiered representation for real-values are identical to the Omniglot setting. Instantiated primitives are colored grey, but can change color when passed through a `COLOR` operator. Our DSL also supports symmetry operations: `SymReflect` creates a reflectional symmetry group over a specific axis. `SymRotate` creates n copies of its input argument about the origin. `SymTranslate` creates n copies of its input argument in a direction that is parameterized by a distance in the same way as `MOVE`.

Reconstruction Metric For the layout domain we use a color-based intersection over union metric. Given two images, we first identify all of the occupied pixels, and which of our four colors each occupied pixel is filled in with. We then calculate the 'intersection' numerator between these two images by counting the number of pixels that are both occupied with the same color. We calculate the 'union' denominator between these two images by counting any pixel in either image that is occupied. Our final value M is calculated by dividing the numerator by the denominator. *HOLE* tokens are allowed to take the place of any function.

D.2.3 3D Shape Structures

DSL We use the following domain-specific language for 3D shape structures, which is adapted from ShapeAssembly (Chapter 3). We present a slightly simplified representation of this language for clarity.

$$
\begin{aligned}
&ST ART \rightarrow BBoxBlock; ShapeBlock;\\
&BBoxBlock \rightarrow \text{bbox} = \texttt{Cuboid}(x, x, x)\\
&ShapeBlock \rightarrow (PBlock; ShapeBlock) \mid \texttt{FILL} \mid \texttt{END}\\
&PBlock \rightarrow CBlock; Attach; SBlock\\
&CBlock \rightarrow c_n = \texttt{Cuboid}(x, x, x) \mid c_n = START\\
&Attach \rightarrow \texttt{attach}(cube_n, f, uv, uv)\\
&SBlock \rightarrow Reflect \mid Translate \mid None\\
&Reflect \rightarrow \texttt{Reflect}(\text{axis})\\
&Translate \rightarrow \texttt{Translate}(\text{axis}, m, x)\\
&f \rightarrow right \mid left \mid top \mid bot \mid front \mid back\\
&\text{axis} \rightarrow X \mid Y \mid Z\\
&x \in [0, 40]/40.\\
&uv \in [0, 20]^2/20.\\
&n \in [0, 4]\\
&m \in [1, 5]
\end{aligned}
$$

This DSL creates shape structures by defining cuboids, and arranging them through attachment. Cuboids are instantiated with the `Cuboid` command. Each `Attach` command moves one command to connect to previous part, indicated by $cube_n$ at a location specified by the other parameters of the command. This language supports the creation of reflectional symmetry groups (`Reflect`) and translational symmetry groups `Translate`. Of note, we allow the DSL to expand hierarchically, so that Cuboids can become the bounding volume of their own sub-program (represented above with the return to the *START* block). These nested

sub-programs are allowed to be set to a completely filled mode (`FILL`) or instead expand into empty space if immediately followed by the `END` operator. Differing from other languages, we only allow *HOLE* tokens to replace these *START* tokens that define sub-program structures, to better match the hierarchical processes by which manufactured shapes are commonly modeled.

Recon Metric We employ different metrics for this domain dependant on the visual representation. When we operate over 3D voxel fields, we simply use the voxel occupancy intersection over union as our metric M. When we operate over primitive soups, i.e. unordered collections of primitives, we use the following matching procedure: we first calculate the pairwise distance between each primitive by calculating the bidirectional Chamfer distance on the sets of corner points that form each cuboid. Assuming the two shapes we are comparing contain N and M cuboids, we converted these distances into a NxM array, and find an optimal matching through the Hungarian matching algorithm. Our metric M is then calculated as the mean value of the entries of the matrix that form this assignment. When $\mathrm{N} \; ! = \mathrm{M}$, we convert the distance array into a square matrix using the larger dimension, filling in the 'non-matched' entries with a high default value that penalizes structural mismatch.

Target Data We source input shape structures by leveraging the structural annotations provided in the PartNet dataset [141]. As our DSL supports only axis-aligned parts, we filter out any shape structures that require other kinds of oriented cuboids. We then make use of ShapeAssembly's parsing procedure to heuristically find ShapeAssembly programs, under the original DSL formulation, that correspond with these input shapes. We try converting these programs into our DSL formulation, and check the geometric similarity between this execution and the original PartNet shape, as a sanity check to see if this shape structure *could* be modeled under our procedural language.

At the end of this preprocessing stage, we are left with over 10,000 shapes from the chair, table and storage classes of PartNet. We use the corresponding parsed ShapeAssembly programs to group these shapes into concept groups. We differentiate the internal group consistency along 2 axes: whether or not the group would likely require a *HOLE* token and whether or not the group would have a consistent application of attachment commands. We parse concept groups under all four combinations of these difficulty settings, choosing 25 concept groups from each setting to populate our test set, where each concept is 'formed' according to a grouping of 10 exemplars. We treat all other shapes not in the test set as training shapes, and during finetuning we randomly sample concept groupings from this set according to the same concept identification procedure.

D.3 Model Details

D.3.1 Architecture Details

All of our auto-regressive networks are implemented as standard Transformer decoder-only models [209]. We use learned positional encodings, these cap the maximum sequence lengths for the various networks. There are three positional encodings for various sequences: the Template Program sequence, the Structural Expansion sequences, and parameter instantiation sequences. For the layout domain we cap these at sizes: (64, 16, 72), for the omniglot domain we cap these at sizes (64, 16, 64), for the shape domain we cap these at sizes of (64, 24, 80).

Visual Encoders We employ encoder networks that convert visual inputs into latent codes, see Figure 6.1.

For the layout domain we use a standard CNN that consumes images of size 64x64x3. It has four layers of convolution, ReLU, max-pooling, and dropout. Each convolution layer uses kernel size of 3, stride of 1, padding of 1, with channels (32, 64, 128, 256). The output of the CNN is a (4x4x256) dimensional vector, which we transform into a (16 x 256) vector. This vector is then sent through a 3-layer MLP with ReLU and dropout to produce a final (16 x 256) vector that acts as an 16 token encoding of the visual input. The omniglot CNN is identical, except it uses one fewer convolution layer, a padding size of 2 in the final convolution layer, and its 3-layer MLP consumes features of size (16x128) and transforms them into size (16x256). In this way for Omniglot we also convert each input image into 16 visual tokens.

For the shape domain we have two different encoders depending on the input modality. For our 3D voxel model we follow a similar convolutional paradigm, extending all 2D convolutions to be 3D, changing the kernel size to 4, using padding of size 2, and adding an extra fifth convolution layer. When consuming voxel grids of size 64x64x64 this produces outputs of size (2x2x2x256), we send this through a 3-layer MLP to produce a (8x128) feature, that we reformat to be (4x256) in dimension. In this way, 3D shapes are represented with four visual tokens.

When we consume a primitive soup of input, we use a different architecture based on a Transformer encoder [209]. We assume that each primitive is a cuboid with 6 dimensions that describe its 3D position and size. We linearize these primitive attributes, and lift each of them to dimension 16 with a 2-layer MLP. Following this we add a learned positional encoding to each attribute based on its attribute type. We then have another 'positional encoding' that is produced by concatenating all of the attributes of each primitive

(in the lifted dimension) and sending this feature through a 2-layer MLP that outputs an embedding of the same size as the lifted dimension, which then gets summed back into each attribute. This scheme allows us to avoid worrying about how the primitives are ordered, while still allowing the attention scheme of the network to differentiate which attributes belong to which primitives. We send this tokenized representation through a standard Transformer encoder network, where we prepend the sequence with four 'dummy' tokens. Each token attends to every other token, and we treat the representations output in the indices of the four 'dummy' tokens as the visual tokenization. These dummy tokens build up a representation that attends of the entire input in much the same way as [*CLS*] tokens have been employed. Note that this encoder assumes a maximum number of primitives as input, which we set to 20. If the input scene does not have 20 primitives, we leave these entries as zeros, and then don't attend over those corresponding positions in the sequence while encoding.

D.3.2 Location Encoding scheme

We adopt the location encoding scheme from [166] for predicting how to file in *HOLE* tokens, while predicting each SE , and parameter values, while predicting the complete z. Specifically, we use their notion of 'sentinel' tokens to identify any locations in the linearized function sequence that need to be filled in autoregressively. Then during each autoregressive step, we 'prompt' the network to predict for a specific location by repeating the sentinel token. We depict examples of this process in Figure 6.1. We treat each sentinel token as an independent token in our language, this limits the number of *HOLE* and parameter tokens we can predict. We set the max number of *HOLE* location encoding tokens to be 5, and the max number of parameter location encoding tokens to be 64. Assigning a reuse parameter relationship in the TP also uses similiar location encoding tokens: we allow for up to 4 of these *shared tokens*: when multiple instances of any of these shared tokens appear in the TP , we constrain instantiations of the TP to assign these slots with matching parameter values.

D.3.3 Generative Networks

Unconditional Generative Networks We use unconditional generative networks to produce paired data during our wake-sleep step of fine-tuning. Specifically these networks are unconditional with respect to visual inputs, but they still condition on programmatic elements. These networks can also be used for unconditional concept generation, see Section D.1.3. The networks we use for this process have an identical architecture to

our inference networks. In fact, at the beginning of our fine-tuning process we initialize the weights of these networks with the weights of the inference networks that have undergone supervised pretraining. They differ from the inference networks by simply masking out (i.e. setting to 0) all of the visual latent codes that are used to condition the generation of the Template Program, the Structural Expansion and the final program. In this way, these networks only condition on token sequences, or in the case of the TemplateNet, don't attend over any prefix conditioning information. Our training scheme for these networks uses the same losses as our training scheme for the inference networks, assuming we have paired data

Few-Shot Generative Networks For few-shot generative tasks, we want a network that has conditioning information in between our inference networks (that condition on latent codes specific to visual inputs in an input group) and our unconditional generative networks (that don't condition on visual inputs). To address this point, we train variants of our inference networks that condition on a mean-pooled latent encoding (i.e. we average the 5 visual latent codes that come from an input group). Note that this only affects the ExpansionNet and the ParamNet, as the TemplateNet already is designed to attend over an input visual group. Once we create this mean-pooled latent encoding, the training procedure is undergone in the same fashion, except the shared latent code is used as conditioning information for all of the instances of the (TP^G,Z^G) pair. In this way, we task the network with learning to solve a one-to-many modeling problem: from the same conditioning information, the network has multiple valid targets.

This network is trained on the same paired data as our inference networks (the batches of data created by our ST, LEST and WS procedures). While its possible to train this network during finetuning alongside the inference network, we instead cache all of the training data our inference network consumes during finetuning, and then train this few-shot generative network in a separate process after our inference model has converged. All of the few-shot generative results we demonstrate are sampled from these networks (after a Template Program describing an input group has been inferred).

D.4 Training Details

We implement our networks in PyTorch [158]. We run all experiments on a NVIDIA GeForce RTX 3090 with 24GB of GPU memory, and 64 GB of RAM. During pretraining we set the batch size to max out GPU memory, this amounts to sizes of 32 for the 2D layout domain, 40 for Omniglot domain, 32 for the shape domain with a primitive soup input and 16 for the shape domain with voxel inputs (of size 64^3). Note that

this batch size is effectively multiplied by 5 for the ExpansionNet and ParamNet as we train on visual input groups of size 5. During fine-tuning we set the batch size to 20 for all methods, except for the shape-voxels variant, which we set to 10 to avoid maxing out VRAM.

We use the Adam optimizer to train our networks [106] with a learning rate of 1e-4. We pretrain our networks on synthetic data sampled from each domain until we converge with respect to a validation set of similarly sampled synthetic paired data. This takes approximately $\sim$ 700k batches for the layout domain, $\sim$ 600k batches for the shape domain, and $\sim$ 300k batches for the Omniglot domain.

We finetune our inference networks with the procedure described in Section 6.1.3. For each concept in the training set, we sample a group of visual inputs (at random) from the concept, and record our inference results to produce the LEST and ST dataset. In this way if there are K concepts in X^*, the size of the ST and LEST data on each training step will also be K. Differing from this, in the wake-sleep step of our finetuning procedure we can generate an arbitrarily large number of paired data by sampling our generative model. We find that sampling a large number of 'dreams' is helpful for our finetuning procedure, so we set the number of example TP to sample in each training step to 30,000. This typically takes between 1 and 2 hours, differing slightly for each domain. To encourage the 'dreams' we sample to cover a wide-distribution, we design a negative rejection step where we resample any 'dream' that either creates an already generated TP or X^G. We find this rejection criteria is triggered at relatively infrequent rates ($\sim$5% of the time).

Once we've created the ST, LEST and WS datasets, we use them to finetune our inference networks with cross entropy loss. We train over this datasets for multiple 'epochs', where every 5th epoch we run our updated inference networks over concepts from the validation set. We use the Objective O from this validation inference to decide when to break out of the training step, and return to the inference step. This early stopping inference procedure always backtracks to the version of the inference network that achieved the best O measure on the validation set. We use a patience of 10 epochs, and finetune for at most 50 epochs.

Overall, we run our finetuning procedure to convergence for 25, 17, 32 inference-training loops for the layout, omniglot and shape domains respectively. This corresponded with 565, 450, 620 finetuning 'epochs' for these domains. For the weights of our objective function O, we normalize each reconstruction metric to values typically between 0 and 1, and then we set λ_1 to 1.0 and λ_2 to 0.001. Moreover, when calculating the divergent description length between each Template Program and its respective program instantiations, we discard counting any parameter-types for which we don't support parameter relations. For instance, as we don't allow float variables to use parametric relations (see Section D.1.2), we do not penalize these variables under O, because the TP has no opportunity to constrain them.

D.4.1 Token Sequence Formatting

Given a paired (X^G, TP^G, Z^G) triplet we can produce training data for our inference networks. We train under a teacher-forced autoregressive paradigm, where we make a single pass through the autoregressive network for each training batch. The input for the TemplateNet is a linearized sequence of visual latent codes; these are randomized as we randomly order the visual inputs. The target for the TemplateNet is the linearized sequence of tokens that describe the Template Program, where we use prefix notation to convert expression trees into flat sequences. From TP and z pairs, we can derive targets for the ExpansionNet and the ParamNet. To find targets for the ExpansionNet, we simply identify mismatches in the functions that are used in the TP versus the functions that are used in z: any expression tree in z that is not found in the TP must be the result of filling in a *HOLE* token. Similarly, we scan the TP to identify any parameter relationships that have been defined, either in the form of specifying parameter arguments (static assignment) or using *shared tokens*. As we know the final expression tree of the z from its linearized form, we then use these declarative relationships to reformat the z to replace all free parameters with *sentinel tokens* (Section D.3.2).

D.5 Experiment Details

D.5.1 Few-shot Generation

Task design In the few-shot generation task we employ the following set-up. For each concept in the test set of a particular domain, we take 5 examples from the concept, pass them as input into a method, and then ask the method to synthesize 5 new generations. We then compare these 5 generations to a separate set of 5 examples from same test-set concept (i.e. a reference set). As the layout domain is procedurally generated, we can sample more examples per concept, therefore in this domain we do the above procedure 5 times for each test set concept. In this way for layout, our metrics compare sets of size 25 generations to 25 reference images (where these 25 generations came from 5 prompts).

Metrics We quantitatively evaluate few-shot generative capabilities (Table 6.1) with a series of metrics common to recent generative modeling approaches [2]. Though these metrics are typically designed to operate over much larger sets, we think the trends they exhibit are indicative of few-shot generative performance (and their ordering is largely consistent internally).

Some of these networks directly compare the generated samples to a reference set for each concept.

Frechet Distance (*FD)* [73] measures the distributional similarity between two distributions of encodings. Minimum Matching Distance (*MMD*) measures the average minimum distance of each member of the reference set to any member of the generated set. Coverage (*Cov*) measures the percentage of reference set members who are the nearest neighbors to at least one member of the generated set.

We calculate all of the above metrics with respect to a latent space that is domain-specific. To this end, for each domain, we train a visual auto-encoder to learn how to reconstruct 'random' scenes from that domain. For the layout domain these are randomly placed primitives. For Omniglot, these are randomly placed strokes. For shapes, these are randomly place cuboids primitives. We train each of these networks to convergence on 500,000 random scenes with a small bottleneck layer size (e.g. 100).

For the layout and omniglot domain we train simple classifier networks to learn a K-way classification over all of the concepts present in the domain. For Omniglot, we train on 19 examples from each of the 1623 characters in the dataset, and hold out one example from each concept as a validation set. Our classifier achieves a 82.4% validation accuracy after convergence. For layout, we train over 95 examples from each concept in a 20-way classification task over meta-concepts; we reach 99.9% validation accuracy on a held out set of 5 examples per concept. The class confidence metric (*Conf*) is then computed by taking each generated output, running it through the classifier, and then recording the probability that the classifier predicts for the index of the input concept. Note that this metric is not dependant on the reference set of examples.

D.5.2 Perceptual Study

We design a perceptual study to evaluate our method's few-shot generative capabilities. Our study was designed as a two-alternative forced-choice questionnaire. We recruited 20 participants, who made decisions about which set of few-shot generations better matched a reference concept.

We show an example of our perceptual study interface in Figure D.3. The middle row of each question shows the input prompt examples. The bottom/top row are populated by the few-shot generations of competing methods based on the prompts shown in the middle row. We randomize which method is shown on top vs bottom, and randomize the order of all examples within the row.

Participants were either shown 50 Omniglot character comparisons or 25 shape comparisons. We visualized shape comparisons with a simple rendering style of the primitive outputs produced by each method (for time considerations).

From our 20 participants we record 900 judgements of our method against three other conditions: ours vs arVHE for Omniglot (381 judgements), ours vs GNS for Omniglot (369 judgments) and ours vs arVHE for

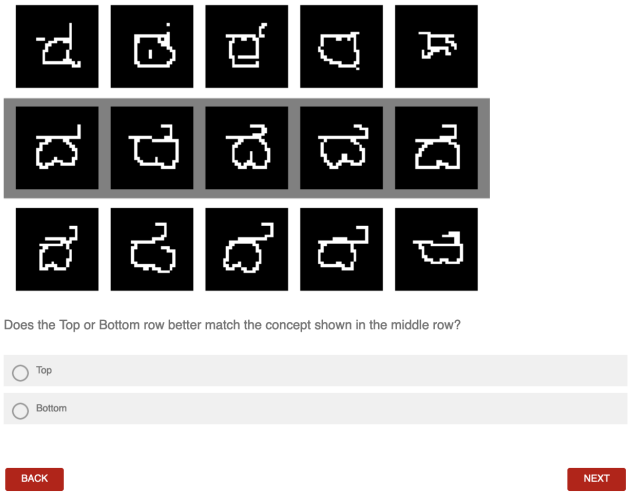

Figure D.3: A visualization of the interface we use in our two-alternative forced-choice perceptual study.

3D shapes (150 judgements). We report the quantitative results from this study in Table 6.2.

D.5.3 Co-segmentation

We formulate the co-segmentation task as follows. We are given 5 examples as input, exactly one of these examples comes with a reference segmentation. The goal of each method is to propagate the labeling from this reference segmentation to the other members of the input group that lack a reference segmentation. We show an example of this task in Figure 6.4.

We compare the produced segmentations against ground-truth annotations for each member of the input set. To quantitatively evaluate performance on this task we use a mean intersection over union metric (mIoU) [141]. This metric calculates the intersection over union for each label that appears in the ground-truth annotation, and then averages these values.

Ground-Truth Segmentations Here we describe how we source ground-truth segmentations for each domain.

For 2D layouts, we produce these as a part of the way we design our meta-procedures. Each primitive group in these specifications is given a semantic label. We evaluate over all concepts in the test set.

For 3D shapes, we record the PartNet hierarchy annotation for each primitive of each shape structure we use [141]. Then within each test-set concept, we search for a group of 5 inputs that use the same semantic parts in their shape structures. If we find such a group, then this is the group from the concept we use during

co-segmentation tasks. From our 100 test set groups, we find such co-segmentation inputs for 94 of them.

We make use of Omniglot stroke data to produce the ground-truth segmentations for characters. We treat each stroke pattern broken by 'BREAK' annotations as a separate segment [116]. Then, as humans vary in the ways that they order strokes to draw characters, for each test set character we run a clustering procedure to try to find valid and consistent segmentation groupings. We first filter for finding groups of characters that use the same number of strokes, and more than a single stroke (otherwise the co-segmentation task is trivial). Then we encode each stroke with a 4 dimensional feature: its length, its angle, its starting x position, and its starting y position. We run an unsupervised clustering algorithm over this feature representation [43], identify if there is any cluster with more than 5 character members, and then take 5 characters from this cluster as a co-segmentation task (where our feature-wise distance creates a correspondence across the strokes of this group). This automatic process generates 306 co-segmentation tasks from the 659 concepts in the Omniglot test set. We manually inspect the generated tasks, and filter out 22 cases where our clustering identified a group that did not have consistent stroke expression. This leaves us with 284 cosegmentation tasks that we use in our experiments.

Group Parsing *Template Programs:* Template Programs support parsing by inferring instantiations from a shared TP that explain a group of visual inputs. As each instantiated program z uses the function call structure specified by a Template Program, we can find correspondences in the visual outputs. We create a corresponding group for each primitive type that the Template Program defines: these are created by the `PRIM` command for the layout domain, the `DRAW` command for Omniglot and the `Cuboid` command for 3D Shapes. Note that *HOLE* tokens are always treated as a construct that creates primitive types. Any command that operates over this primitive type will inherent their corresponding part index (e.g. symmetry operations), excluding combinators like `Union`.

BAE-NET: BAE-NET creates corresponding group parses by performing an argmax over the last layer of an implicit network that is trained to solve occupancy tasks. This implicit network can be run over any spatial position, and assign this input point to one of its part 'slots'.

BPL and GNS: The BPL and GNS methods perform one-shot parsing of input characters into an ordered collection of strokes. This parsing is guided by their learned prior, which models how people produce characters. Conscripting these methods to perform our co-segmentation task is a slight abuse of design, but as their output parses partition space in a consistent fashion, we think it a worthwhile comparison to make. Our method does not learn from any human demonstrations, so we are unable to solve the character parsing task

as it is originally formulated [115, 116].

Label Propagation The parses we get from the above logic are consistent, but might not exactly recreate the input examples (if they do not achieve perfect reconstructions). We thus employ a procedure, on a domain-by-domain basis, that propagates the parse from the reconstruction to its input example. For the layout domain we first take the part index of each occupied pixel to match the primitive that last 'covered it'. Then for any non-occupied pixels, we assign them to the closest instantiated primitive according to the distance from that pixel's center to the primitive. For the shape domain, we take a similar approach, calculating the distance from voxel centers to each cuboid. For any voxel center that is occupied by more than one cuboid, we assign it to the occupying cuboid smallest in volume. For Omniglot, we sample 200 points on each primitive stroke group. Then for five query points evenly spread out within each pixel location, we find the three closest points sampled from any stroke group. We tally up these votes for each pixel, and then each pixel is assigned to the primitive stroke group which recorded the most votes. Note that we employ this same procedure for our method, BPL, and GNS. BAE-NET doesn't need to employ this logic, as its parsing strategy operates over arbitrary input points by construction.

After we have this consistent parse for each region of the input group the procedure is almost done. We use the partitions from the labeled example to assign each parsed region a label. Finally, we propogate this region-to-label mapping to all of the other examples in the input group.

D.6 Comparison Method Details

We provide details on the methods we compare against.

D.6.1 BPL

We use the author's released Matlab implementation: https://github.com/brendenlake/BPL. For five characters from each test-set concept we infer a parse, and use that parse to synthesize 1 new generation (in this way we create 5 few-shot generations from each group of 5). We wrapped this Matlab procedure with a python script, and ran it sequentially on a single machine, which took around 2 weeks.

D.6.2 GNS

We use the author's released implementation at https://github.com/rfeinman/pyBPL. We follow the same procedure as in BPL, inferring a parse for five characters from each test-set concept, and then using each parse to synthesize 1 new generation.

D.6.3 FSDM

We follow the author's implementation, released at: https://github.com/georgosgeorgos/few-shot-diffusion-models. Unfortunately, the provided code was incomplete, and did not work out of the box. We made a best-effort attempt to fix these issues and run the model with the same procedure as described in the technical report. We observed that this model was able to effectively produce few-shot generations for training characters, but struggled greatly on test-set concept generalization.

D.6.4 VHE

We attempted to use the author's implementation, released at: https://github.com/insperatum/vhe. Unfortunately the PixelCNN variant for Omniglot did not converge under training, we reached out to the authors, but they were unable to offer suggestions on how to fix these training issues.

Using the provided code as reference, we re-implemented the system with a simple CNN architecture, following the VAE framing as described in [180]. Though we spent a fair amount of time tuning hyperparameters, as evidenced by the quantitative results in Table 6.1, we were unable to achieve competitive performance.

arVHE In an attempt to improve the performance of our VHE comparison condition, we implemented a related method that combines autoregressive models with the spirit of the VHE approach. Specifically, we break down this few-shot generation modeling task into two separate stages. First we learn a domain-specific discretized representation. For pixel and voxel input representations we use 2D and 3D CNNs in a vector-quantization scheme [208], so that we can convert each visual input into a sequence of discrete tokens.

We list the details of our VQ-VAE training: for Omniglot we convert 28x28x1 images to a 7x7 grid of codes, under a dictionary of 64 codes with hidden dimension of 32. For layout we convert 64x64x3 images to a 7x7 grid of codes, under a dictionary of 200 codes with hidden dimension of 100. For shapes we convert 64x64x64 voxels to a 4x4x4 grid of codes, under a dictionary of 128 codes with hidden dimension of 64. We

try to use the smallest code-book size that can achieve near-perfect reconstructions for each domain.

Once we have trained this VQ-VAE for each domain, we can learn our arVHE model. Like the VHE model, and our system, it learns by sampling random visual groups from the same concept. Following the procedure described in the VHE paper and code, we encode these visual concepts with a visual encoder, take a mean embedding, then use this embedding to condition an autoregressive generation process, where the goal is to predict a sequence of VQ-VAE tokens that correspond to another input example from the same concept. We train this network with cross-entropy loss, on the discretized VQ-VAE tokens. For an apples-to-apples comparison against our method, the arVHE baseline uses the same visual encoders that our method uses (Section D.3). For predicting 3D shapes as a sequence of primitives, we instead just task the VQ-VAE model with predicting discretrized versions of each primitive attribute, where the primitives are randomly ordered (this allows us to skip the VQ-VAE step in this setting).

We note that this arVHE variant is a strong baseline method, outperforming VHE and FSDM in terms of quantitative metrics (Table 6.1).

D.6.5 BAE-NET

We follow the author's implementation released at: https://github.com/czq142857/BAE-NET. We take their architecture and training procedure and adapt it for each of our domains. BAE-NET has model implementations for 2D binary images and 3D voxel grids, so for these settings we directly use the method as described. For the layout domain we have colored images that can adopt 4 color values (red, green, blue, or grey). In the default version of BAE-NET, it uses an MLP where the second to last layer is size NUM_SEGS and the last layer is size 1; this 1 dimensional output learns a binary occupancy prediction for locations in space. We modify the 2D BAE-NET version so that instead, the second to last layer is still size NUM_SEGS, but the last layer is size 4; in this way we task BAE-NET to solve four binary occupancy problems at once, one for each of our colors. In the layout domain, we still take the part segmentation from BAE-NET by choosing the slot in the second to last layer that activates with the highest potential.

Appendix E

Additional Details and Results for ShapeMOD

In Appendix E, we supply additional details for the ShapeMOD method introduced in Chapter 7.

E.1 Modified ShapeAssembly Grammar

Table E.1 shows the modified grammar for ShapeAssembly that we use. We make the following changes from the ShapeAssembly version presented in Chapter 3. Instead of having separate blocks where all cuboids are defined, then all attaches are defined, and then finally all symmetry operators are defined, we interleave the attach / symmetry commands with the cuboids they move. Specifically a program starts with defining a bounding volume, and then is followed with a series of PBlocks. Each PBlock defines a Cuboid, attaches it to at least one previous cuboid (or the bounding volume), and optionally applies a symmetry operation to it. We find that this ordering permits the discovery of more interesting and useful macros, as otherwise macros would mostly be made up of only Cuboid definitions or only attachments (instead of a mix of operators). As a by-product of this new ordering, we assume that all non-Cuboid operators (`attach`, `squeeze`, `reflect`, `translate`) always operate on the last defined cuboid, and so in this way we remove one cuboid index parameter from each of these functions.

Table E.1: Modified ShapeAssembly grammar of ShapeAssembly.

Start → BBoxBlock; ShapeBlock;
BBoxBlock → bbox = `Cuboid`$(w, h, d,$ `True`$)$
ShapeBlock → PBlock ; ShapeBlock | None
PBlock → c_n = `Cuboid`(w, h, d, a) ; ABlock; SBlock
ABlock → Attach | Attach ; Attach | Squeeze
SBlock → Reflect | Translate | None
Attach → `attach`$(c_{n_1}, x_1, y_1, z_1, x_2, y_2, z_2)$
Squeeze → `squeeze`$(c_{n_1}, c_{n_2}, f, u, v)$
Reflect → `reflect`(axis)
Translate → `translate`(axis, m, di)
f → right | left | top | bot | front | back
axis → X | Y | Z
$w, h, d \in \mathbb{R}^+$
$x, y, z, u, v, di \in [0, 1]^2$
$a \in$ [`True`, `False`]
$n, m \in \mathbb{Z}^+$

E.2 Baseline Method for Macro Operator Discovery

Designing a baseline for ShapeMOD is non-trivial, because there do not exist any existing methods that are able to find macro operators over datasets of programs written in imperative languages that contain continuous parameters. Thus, we present a naive single-pass algorithm that mimics a simplified version of ShapeMOD's core logic. It starts by choosing one order for each program in the dataset. Specifically, the most canonical order (Section E.5.3). Then it records all subsequences of functions that appear in the resulting program lines. If any subsequence is observed in more than 10% of programs in the dataset, then it is turned into a macro function. Parameters of this macro function can be converted from free parameters to constants if at least 90 % of the parameterizations of this subsequence across the dataset had the same value (for discrete parameters) or were within .05 range of the mean value (for continuous parameters). Once these macros have been discovered, we use the best program finding step from ShapeMOD to create a dataset of programs expressed with macros discovered by the baseline method. As shown throughout the results section, the macros discovered by ShapeMOD outperform the macros discovered by this baseline method, for every task we consider.

E.3 A Network Architecture for any library

After running our procedure to generate a library L, we want to design a neural network that is able to generate programs using the functions of L. As our procedure is able to produce many different libraries L,

depending on which macro operators it discovers, our network architecture must be flexible enough to model any set of discovered functions. To demonstrate that this is achievable, we generalize the neural network from ShapeAssembly (Chapter 3) so that it is able to learn how to generate programs expressed in any L discovered through our procedure, and validate this works in later experiments.

The base model is a hierarchical sequence VAE. The encoder branch ingests a hierarchical program and embeds it into a high dimensional latent space. The decoder branch converts a code from this latent space into a hierarchical program. Originally, the underlying library was fixed to ShapeAssembly, so the network architecture and input representation could be tailored to one set of functions.

We design a generalized version of this network architecture that is customized based on the library of functions L discovered by our procedure. The parts of the architecture that had to be generalized were the tensor line representation and the sub-networks in the line decoder module.

In our new line representation, the dimension of the line tensor and meaning of each index changes depending on L. The first $|L| + 2$ indices of the tensor correspond to a one hot vector denoting the function type of each line (notice we add special START and STOP tokens). Then for each type of discrete parameter, p_d, we find its number of valid values, p_{d_size}, and maximum number of p_d free parameters in any function of L, p_{d_free}. We then reserve p_{d_free} slots of size p_{d_size} in our tensor for p_d, where each slot corresponds to a one hot vector whenever p_d is required by a function. Finally, for any function $f \in L$ that takes in a set of continuous parameters, f_c, we reserve a slot in our tensor of size $|f_c|$.

The number and structure of sub-networks in our new line decoder model also depends on L. The M_{func} module is responsible for predicting the line's function, and therefore has $|L| + 2$ possible outputs (the functions of L and the special START and STOP tokens. For each $f \in L$, for each of its free discrete parameters f_{d_i}, we add a sub-network $M_{f\_d\_i}$ responsible for predicting the ith discrete parameter of f. Then, for every f that has free continuous parameters, we add a sub-network M_{f_c} for predicting the continuous parameters of f.

We implement each sub-network as a 3 layer MLP. The network is trained in a teacher forcing paradigm with a cross entropy loss for all discrete predictions and an l1 loss for all continuous predictions. Parameter sub-networks are invoked, and tensor slots in each line are filled, depending on the function type predicted in each output line. Otherwise we use the same hyper-parameters as ShapeAssembly.

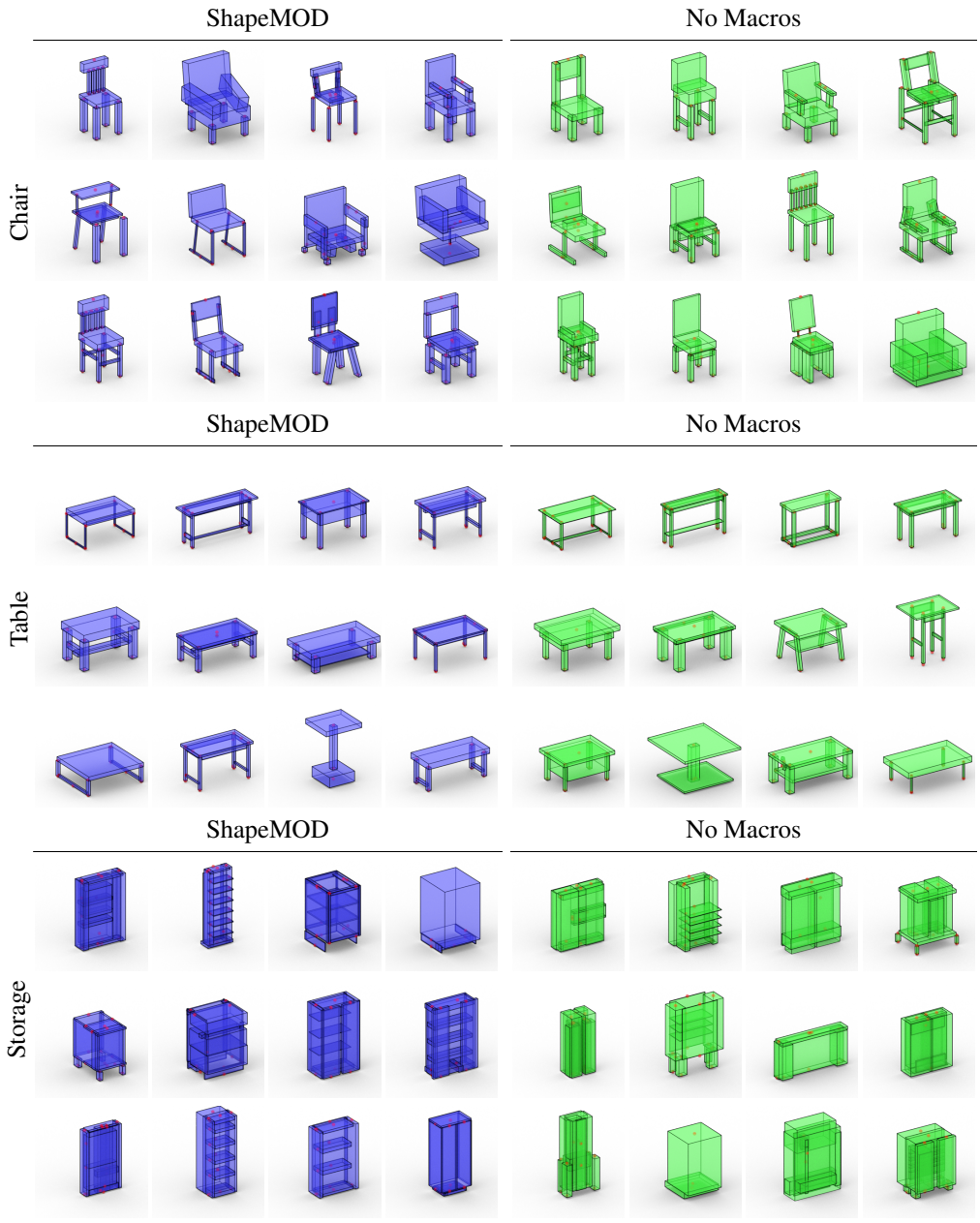

Figure E.1: Samples generated from generative models of ShapeAssembly programs with ShapeMOD macros (blue) and without macros (green).

E.4 Shape Generation Qualitative Comparison

We share some interesting representative shape programs output by learned ShapeAssembly generative models in Figure E.1. Outputs by the model trained with ShapeMOD macros are shown in blue. Outputs by the

model trained on the No Macros programs are shown in green.

These qualitative results enforce the trends of our earlier quantitative experiments from Section 7.4.2. The best generations from Chairs and Tables are qualitatively similar, although across entire shape collections we calculated that programs with ShapeMOD macros were more plausible. For storage, the qualitative difference is more pronounced, as the generations that use ShapeMOD macros are able to create output shapes that are much closer in distribution to the target shape collections.

E.5 Creating A Dataset of Shape Programs

As input, ShapeMOD consumes a dataset of shape programs. To source such a dataset we need find a collection of shape programs composed of program line and orderings of those lines, that when executed, faithfully recreate the target geometry. Instead of working with complete hierarchical ShapeAssembly programs, we instead run ShapeMOD on the flat sub-programs that together compose complete PartNet shapes. For instance, while running ShapeMOD we might have different programs for a single PartNet chair's back, base or root level programs. Re-composing flat programs back into hierarchical ones, after ShapeMOD has discovered the most useful macros, is trivial.

E.5.1 Parsing

Following the parsing method proposed in ShapeAssembly, our method starts with a collection of part graph hierarchies from PartNet [141]. We use a geometric procedure to turn this part graph hierarchies into ShapeAssembly program lines. At the end of this process, for each shape in our dataset of part graph hierarchies, we have all of its cuboid dimensions, know which parts attach together and where, know which attachments could instead be expressed as squeeze operators, and which cuboids should have symmetry operators applied over them. This is enough information to form the program lines required by a ShapeAssembly shape program.

E.5.2 Finding Valid Orderings

Given these lines we still need to figure out how they can be ordered, as not every line ordering will recreate the target geometry, or even result in a valid ShapeAssembly program (i.e. that adheres to the grammar). There are three ordering components that must be considered for ShapeAssembly programs that adhere to the grammar we introduce for ShapeMOD: (i) ordering of cuboids, (ii) orderings of each attach a cuboids

makes, and (iii) deciding which cuboid is making each attach, and which cuboid is being attached to. The combinatorial space of these orderings makes searching over *all* orderings prohibitive; thus we define a series of heuristics to narrow the exploration space.

First, we create a graph where each node is a cuboid and an edge between nodes indicates that they attach. We then find the shortest distance from each node to the bounding box node (call this distance the level of the cuboid). We then enforce (1) that cuboids must only attach to cuboids in a lower level, or the same level and (2) that all cuboids must be declared based on their level ordering (although notice that multiple cuboids can be in the same level still). We further place a strict preference on attachment directions that attach from the center of a face. Whenever a cuboid makes multiple outgoing face attachments we order them by (bot, top, left, right, back, front), and afterwards continue with any non-face attachments.

So far, we have limited the search space we must consider, but we haven't enforced that the resulting ordering will actually recreate the geometry we care about. For an ordering of ShapeAssembly lines to recreate the target geometry, three conditions must hold: (i) it must be a valid program under the grammar, (ii) no cuboid should be moved after it has been attached to and (iii) each cuboid should have enough outgoing attachments to specify its orientation. We satisfy (i) by construction as we only consider lines orderings that would be valid under the grammar. (ii) is satisfied based on the level ordering logic from the above paragraph. For (iii) to be satisfied, different conditions must be met depending on if the part is aligned or not aligned to its parent's bounding box in the target shape. If the part is aligned, it only needs one attachment (as all cubes in ShapeAssembly start off as aligned). If the parts is not aligned, it will require at least two attachments. For any ordering of cuboids/outgoing attachments/attachment directions, if they satisfy (i), (ii) and (iii) we consider them to be a valid ordering. To improve the run-time of the algorithm, we place a strict limit on the number of orderings for any one program to be 10000. This limit is surpassed by around 3% of chair programs, around 5 % of table programs and around 10 % of storage programs. In case a program has more than 10000 valid orderings, we return the first 10000 valid ones ranked by their canonical order (detailed in next section).

E.5.3 Canonical Ordering

At various points in our procedure, multiple program orderings will be 'equally good' and so we will need some criteria to consistently differentiate between them. To accomplish this, we borrow the canonical ordering logic from ShapeAssembly, where during parsing each cuboid is given an index, and then we prefer orders where lower indexed cuboids are defined first, and attached to first (all else equal).

E.6 Details of applying ShapeMOD to ShapeAssembly

E.6.1 Choosing Parameters for an Abstracted Program

When we are finding an abstracted program that can represent an entire cluster of programs, there might be multiple ways to parameterize the abstracted program. Here we detail the preference ordering we iterate over in order to form the abstracted program. Recall that the abstracted program we find must create a valid program for at least $p = 70\%$ of programs in the cluster.

For discrete variables, we first see if the cluster can be explained by a constant, if it can then we use the constant. We next see if the cluster can be explained by the parameter values of a previously used discrete variable (i.e. re-using a free variable in multiple parameter slots), if it can then we re-use the variable. If both of these fail, then we introduce a new free parameter.

For continuous parameters we employ a similar strategy over the following ordered list of expression types. Note that for all macros, we assume that they have access to the dimensions of the bounding box in which they were created.

- select constants that are known to frequently occur - $(0, 0.5, 1.0)$ for ShapeAssembly
- directly using a dimension of the bounding box
- directly using a previously defined free variable
- using a fixed linear combination of 1 and any bounding box dimensions, where the weights are from
  $[(1, 1), (-1, 1), (1, -1)]$
- using a fixed linear combination of 1 and any previously defined free variable, where the weights are from
  $[(1, 1), (-1, 1), (1, -1)]$
- using a fixed linear combination of any previously defined free variable and any bounding box dimension, where the weights are from
  $[(1, 1), (-1, 1), (1, -1)]$
- using a fixed linear combination of any two previously defined free variables, where the weights are from
  $[(1, 1), (-1, 1), (1, -1)]$
- a scaled version of any previously free variable

We find that this preference ordering produced macros for the ShapeAssembly language that were more semantically interpretable. We also tried experimenting with more complex parametric relationships between parameters, such as arbitrary linear combinations of constants / bounding box parameters / past free parameters, but found that this led to slightly worse compression metrics.

E.6.2 Valid Candidate Macro operators

When proposing candidate macros in the proposal phase of ShapeMOD, we choose to optionally make some potential macros ineligible in order to avoid local minima, and limit the number of candidate macros we need to consider for any integration round. The criteria we use are:

- The macro must be four or less lines. Notice that as macros can use macros discovered in previous rounds as sub-routines, a single macro is still able to cover more than four lines of the base ShapeAssembly programs
- A multi-line macro shouldn't end with a Cuboid line.
- A multi-line macro should only include a Cuboid line, if it starts with a Cuboid line. Notice that it can both start with a Cuboid line, and then include subsequent Cuboid lines afterwards

The first requirement helps to speed up the algorithm. The impetus for the last two requirements is that each macro should not cover the partial attributes of any part, it should instead try to build up abstractions that represent an entire part fully, or even a group of parts. We also observed empirically that these requirements helped to encourage macros that developed hierarchical structures.

E.6.3 Candidate Macro Frequencies

The frequency with which a macro was found during the proposal phase influences the gain ranking of that macro during integration. However, whenever we update the library with a new macro, frequencies for all other macros should be updated because the macro that was added might have covered the same program lines (and so double counting these would result in an overly optimistic estimate of the future gain). Thus, whenever M is added into $\mathcal{L}$, we update the line coverage statistics of all other candidate macro operators, so that each line that M covered is no longer used to calculate the frequency score p of future candidate macros during ranking. However if M is not added into $\mathcal{L}$, then we find all reachable candidate macro operators in the the generalization graph starting from the M node, and remove them from future consideration during candidate macro operator ranking (in the current round). This is done so that if a group of macros appear to

have a high gain, but are not actually helpful in decreasing f, we only need to consider one such function in each round of integration.

E.7 Details about Generative Modeling Metrics

- **Rootedness :** We directly use the rootedness calculation from ShapeAssembly
- **Stability :** We use the stability simulator from ShapeAssembly, with one small modification. Instead of the object starting on the ground, and having it be perturbed upwards and to the side, we just drop the object from a small height. We found this to be a more reliable measure of stability that produced stability simulations which more closely matched our intuition.
- **FD:** We use the feature space of a PointNet model trained on a 16 way classification task on shapes from ShapeNet [16]. Each FD calculation takes up to 1000 meshes from two sets, and samples their surfaces to form point clouds with 2500 points. During training, we compare generated meshes against the training set to choose the epoch of the model. For all metrics reported in the results section, we run FD on a held-out set.
- **Realism:** We follow the realism procedure outlined in ShapeAssembly, with a small modification. In ShapeAssembly, the % fool is calculated over a held out portion of shapes from the training set (although not seen by the real versus fake classifier). We instead report % fool statistics calculated over a set of shapes from the validation set.

E.8 Analysis of Variability

To check how macros impact the variability of a generative model we look at the same metrics as in ShapeAssembly, using Chamfer Distance on point cloud samples (1024 points) from the surfaces of generated objects (Table E.2). Generalization measures the nearest neighbor distance from the generated set to the training set. Coverage measures the nearest neighbor distance from the validation set to the generated set. Variety measures the nearest neighbor distance from shapes in the generated set to any other shapes in the generated set. Compared with no macros, ShapeMOD performs slightly worse on the generalization and variety metrics, but slightly better on the coverage metric, across the three categories we look at. The results of baseline macros illustrate how these metrics are tricky to interpret; Baseline macros does the best on Generalization

Table E.2: Using Chamfer Distance (CD) and Program Edit Distance (ED), we check how well generative models generalize from the training set, cover the validation set, and have variability within their own set.

		Generalization	*Coverage*	*Variety*
		NND to Train ⇑	NND from Val ⇓	NND to Self ⇑
Category	**Method**	**CD**	**CD**	**CD**
Chair	No macros	0.114	0.122	0.116
	Baseline macros	**0.115**	**0.121**	**0.115**
	ShapeMOD	0.111	**0.121**	0.114
Table	No macros	0.103	0.106	0.110
	Baseline macros	**0.112**	0.110	**0.115**
	ShapeMOD	0.101	**0.105**	0.108
Storage	No macros	0.132	0.129	0.125
	Baseline macros	**0.144**	0.136	**0.126**
	ShapeMOD	0.127	**0.122**	0.121

Table E.3: Program Dataset Compression

Category	**Method**	f	$\lvert\mathcal{L}\rvert$	**fn**$(\mathcal{P}^*)$	**d**$(\mathcal{P}^*)$	**f**$(\mathcal{P}^*)$	**b**$(\mathcal{P}^*)$
Chair	No macros	411	**5**	29.8	17.8	84.4	11.3
	ShapeMOD	260	17	21.0	**6.4**	**58.1**	**8.6**
	ShapeMOD (CC)	**256**	17	**17.1**	6.9	60.0	9.0
Table	No macros	356	**5**	25.6	16.3	70.7	9.6
	ShapeMOD	214	15	17.4	5.1	**48.7**	**5.6**
	ShapeMOD (CC)	**205**	17	**14.4**	**4.9**	48.4	7.0
Storage	No macros	453	**5**	30.4	21.6	92.2	11.7
	ShapeMOD	283	17	21.1	**7.6**	**68.9**	**4.0**
	ShapeMOD (CC)	**280**	17	**18.2**	7.8	70.	10.0

Table E.4: Metrics comparing samples from learned Generative Models

Category	**Method**	**% fool ⇑**	**FD ⇓**	**# Parts ⇑**	**% rooted ⇑**	**% stable ⇑**
Chair	No Macros	21.2	17.8	7.6	**93.9**	**82.3**
	ShapeMOD	**25.6**	**16.7**	**8.6**	92.7	79.5
	ShapeMOD (CC)	17.8	19.4	**8.8**	**94.5**	**80.8**
Table	No Macros	27.7	26.0	**8.0**	88.8	76.1
	ShapeMOD	**29.2**	23.2	7.8	**93.2**	84.3
	ShapeMOD (CC)	26.8	**21.8**	7.1	**93.2**	**85.2**
Storage	No Macros	4.9	70.0	6.0	92.4	85.5
	ShapeMOD	**11.1**	**38.1**	**7.7**	**95.1**	90.5
	ShapeMOD (CC)	9.3	47.4	**7.7**	94.7	**91.1**

and Variety, but at the expense of high coverage scores. This suggests that baseline macros is doing well on generalization and validation precisely because it is failing to capture the target shape distribution as well as the other two methods.

Table E.5: Metrics on program inference task from a point cloud

Category	Method	CD ⇓	F-Score ⇑	% rooted ⇑	% stable ⇑
Chair	No macros	44.2	54.8	93.7	83.6
	ShapeMOD	**41.7**	**56.1**	**96.9**	88.0
	ShapeMOD (CC)	42.1	55.7	95.9	**89.3**
Table	No macros	41.1	64.0	92.8	78.2
	ShapeMOD	**36.7**	**68.7**	95.2	**88.5**
	ShapeMOD (CC)	37.9	67.9	**96.3**	87.1
Storage	No macros	56.5	41.1	95.0	87.7
	ShapeMOD	**47.0**	**53.0**	97.6	**92.6**
	ShapeMOD (CC)	48.7	51.6	**98.1**	90.7

E.9 Additional Cross-category Macro Discovery Results

We experimented with running ShapeMOD on multiple categories of PartNet objects at once. In this section we report the quantitative effect that using the macros discovered through this procedure had, versus using macros that were discovered when running ShapeMOD on a single category.

We refer to this condition as ShapeMOD (CC). Table E.3 show compression statistics. Table E.4 shows generative metrics. Table E.5 shows reconstruction metrics. Interestingly, in terms of program compression ShapeMOD (CC) outperforms ShapeMOD for every category. This does not translate fully to down-stream task performance though, as ShapeMOD's unconditional generations are more plausible then those from ShapeMOD (CC) and using ShapeMOD leads to better reconstruction metrics over ShapeMOD (CC) on our visual program induction task. Shape validity (rootedness/stability) remains close for both methods across these different experiments. On the whole, while this approach performs slightly worse compared with discovering new macros for each individual dataset of shape programs, for the metrics we care most about (physical plausibility and reconstruction accuracy) it still leads to a dramatic improvement over no macros.

Appendix F

Additional Details for ShapeCoder

In Appendix F, we supply additional details for the ShapeCoder method introduced in Chapter 8.

F.1 Shape Grammar

3D Shape Grammar Below we detail our 3D shape grammar:

START → *SHAPE*

SHAPE → `Union`(*SHAPE*, *SHAPE*) |
`SymRef`(*SHAPE*, AXIS) |
`SymTrans`(*SHAPE*, *AXIS*, *INT*, *FLOAT*) |
`Rotate`(*SHAPE*, *AXIS*, *FLOAT*) |
`Move`(*SHAPE*, *FLOAT*, *FLOAT*, *FLOAT*) |
`Cuboid`(*FLOAT*, *FLOAT*, *FLOAT*);

AXIS → `AX` | `AY` | `AZ`;

INT → [1, 6];

FLOAT → Prim_{ij} | -1 | 0 | 1 | 2 |
`Add`(*FLOAT*, *FLOAT*) | `Sub`(*FLOAT*, *FLOAT*)
`Mul`(*FLOAT*, *FLOAT*) | `Div`(*FLOAT*, *FLOAT*);

We italicize all non-terminal parts of the grammar, and explain what the terminal operators in the language do (non-italicized). `Union` combines two sub-shapes together. `SymRef` is a symmetry reflection across an

axis. `SymTrans` is a symmetry translation over an axis, that creates a specified number of copies, up to a specified distance. `Rotate` specifies an Euler angle rotation about an axis. `Move` moves a cuboid by a specified amount. `Cuboid` instantiates a cuboid with the specified dimensions. Axes can be either the X, Y, or Z axis. Ints can be an integer between 1 and 6. Floats can be either be sourced from a primitive parameter of an input scene (Prim_{ij}), be a constant, or the result of a parametric operation.

2D Shape Grammar Below we detail our 2D shape grammar:

START → *SHAPE*

SHAPE → `Union`(*SHAPE*, *SHAPE*) |
`SymRef`(*SHAPE*, AXIS) |
`SymTrans`(*SHAPE*, *AXIS*, *INT*, *FLOAT*) |
`Move`(*SHAPE*, *FLOAT*, *FLOAT*) |
`Rect`(*FLOAT*, *FLOAT*);

AXIS → `AX` | `AY` ;

INT → [1, 4];

FLOAT → Prim_{ij} | -1 | 0 | 1 | 2 |
`Add`(*FLOAT*, *FLOAT*) | `Sub`(*FLOAT*, *FLOAT*)
`Mul`(*FLOAT*, *FLOAT*) | `Div`(*FLOAT*, *FLOAT*);

This is a simplified version of our 3D grammar, where the rotation command has been removed, and all 3D parameterizations are replaced with 2D parameterizations.

F.2 Implementation Details

We provide implementation details for ShapeCoder below. For all experiments in Section 8.5 we set N_A = 20 and N_D = 10000.

F.2.1 Objective Function Weights

We use the following weights for λ in ShapeCoder's objective function (Section 8.1.1): float tokens are 2.0, shape-returning function tokens are 1.0, float-returning function tokens are 0.1 (i.e. parametric operations), and categorical tokens (including integers) are 0.5 . Additionally we set the geometric error weight, λ_e, to be

10.

For the function weighting scheme ω, described in Section 8.1.1 and ablated in Section 8.5.4, ShapeCoder employs the following logic. The base cost of adding a new abstraction f into $\mathcal{L}$ is 0.25, but this value can be modulated within the range of 0.125 to 0.5 based on properties of f. The presence of parametric expressions in f decrease ω. Too many input parameters in f increases ω, where more than 6 parameters starts to incur penalties, and abstractions with more than 10 input parameters are rejected outright. We decrease ω for doubleton abstractions (those that use multiple sub-functions), and increase ω for singleton abstractions that use a single sub-function. Finally, if f is found to be used very infrequently over $\mathcal{P}$, less than 1% observation rate, then we also reject f outright.

F.2.2 Geometric Error Function

The objective function (Section 8.1.1) uses a geometric error function err that compares how closely an executed expression e from $\mathcal{L}$ matches a target shape d. As this error function is used extensively in the wake phase (Section 8.2.3), it checks for partial solutions. Say executing e creates a set of primitives $prim_e$, and d contains primitives $prim_d$. First our geometric error functions finds an optimal mapping from primitives in $prim_e$ to some primitive in $prim_d$. Mechanically, we construct a distance matrix of size $|prim_e| \times |prim_d|$, that calculates a domain-specific distance metric between each pair of input and target primitives (explained later). For any pair of primitives whose distance is above a user-defined maximum error threshold, we set their paired distance to an arbitrarily high value (10000). We use the Hungarian matching algorithm to find an optimal match over this distance matrix. If none of the paired matches between $prim_e$ and $prim_d$ have distance over 10000, then the match is valid, and the total error incurred by e for d is simply the sum of all entries in the distance matrix involved in this optimal match.

During the integration phase (Section 8.3.2), we can modify this approach to check for a *program* $z$that explains d, by enforcing that the distance matrix must be square. Whenever this condition is not met, it means that there is a mismatch in the number of primitives created by z, and the number of primitives expected in the target shape d, so z is invalid.

2D geometric distance Each primitive (rectangle) is represented as 4 parameters: width, height, x position, and y position. To find the distance between two primitives, we take the average of the absolute differences between each parameter slot. The maximum allowable error threshold is set to 0.05.

3D geometric distance Each primitive (cuboid) is represented as 9 parameters: dimensions, position, Euler angle rotations. To find the distance between two primitives, we calculate the corner positions of each cuboid, and record the Hausdorff distance between the two sets of points. The maximum allowable error threshold is set to 0.1 .

F.2.3 Recognition Network

Our recognition network uses a Transformer decoder backbone architecture with causal masking. We allow it to condition on up to 16 primitives (where each primitive will contribute K tokens), and fix its max prediction length to be 32. It uses 2 attention blocks, with 8 heads in each block, and a hidden dimension of 128. Training uses a batch size of 64, dropout of 0.5, and a learning rate of .0001. Each dream phase (Section 8.2.2) trains the recognition network for a maximum of 300 epochs, where early stopping is performed on a validation set of held-out dreams (10% of samples).

F.2.4 Dream Creation

Sampling library functions During the dream phase (Section 8.2.2), ShapeCoder randomly samples instantiations of library functions to train the recognition network. Some dreams are visualized in Figure 8.6. For each discrete decision needed to parameterize a function f, we find all tokens in $\mathcal{L}$ that type-match, and uniformly sample from this distribution. Float-typed tokens are represented as mixtures of Gaussians distributions (max 3 mixture components). These distributions are designed to broadly reflect reasonable values for certain parameter slots in the base DSL. For instance, the first float parameter slot in the 'Move' operator is associated with x-axis positioning, so we design a trimodal mixture distribution with the following properties: it has a 0-centered dominant component, and then two minor components placed to the left and right of the origin. These distributions don't meaningfully change the performance of the recognition model, as it gets to trains on a massive amount of samples, but it does speed up the rate at which we can find valid dreams under our rejection criteria (explained below). When sampling dreams for abstraction functions, the parameter inputs in the abstraction inherent the distributions of their child sub-functions.

Dream rejection criteria We use simple checks to validate that randomly sampled dreams produce meaningful training data, and reject any dreams that don't meet the following criteria. All primitives must have positive dimensions. The corners of all primitives must be within the allotted scene bounding volume $[-1, 1]^n$, with a 10% leniency threshold. At least 50% of each primitives area must be visible (i.e. not contained

within another primitive). Each primitive must be bigger than a specified threshold: 0.005 area of 2D, .00025 volume for 3D. Dreams cannot contain more than 16 primitives. Dreams cannot use redundant operations, for instance, applying two `Move` commands in a row.

Forming composite scenes ShapeCoder's recognition network trains on composite scenes, that are formed by sampling function-specific dreams and combining them together. To form a composite scene, we sample a random integer k from [1,4], sample k functions from the set of all library functions that have not been represent in N_D dreams, and choose a random dream from each chosen function. Additionally, with 50% chance, we add distractor primitives into the composite scene. Distractor primitives are sourced by randomly sub-sampling primitives found in some $d \in \mathcal{D}$. To encourage the recognition network to be position invariant, we optionally sample a `Move` operation (with 50% frequency) and apply it over the primitives created by a function-specific dream. Note that this `Move` operation is not included in the target expression, so the recognition network must become invariant to where the target primitives show up in the composite scene.

F.2.5 Combining Wake Programs

As discussed in Section 8.2.3, programs discovered in round r's wake phase need to be combined with programs discovered in rounds before r. Here we detail how *combine* is implemented.

Assume we are in the wake phase of round r, $r > 0$. For some $d \in \mathcal{D}$ there is currently some program entry in $\mathcal{P}$, z_c. Using a *split* function, that recursively removes combinator operations from a program, we can convert z_c into a set of expressions in $\mathcal{L}$:
$split(z_c) = E_c = \{e_c^0, ..., e_c^{|E_c|}\}$. When executed, each e_c^i will create a set of primitives, $prim_c^i$, that is a subset of the primitives in d. ShapeCoder keeps track of all such previous expressions associated with d in a data-structure Q_d, sourced from either the wake or integration phases.

The wake inference procedure uses the recognition network to prediction a new program in round r, z_r, for d. We decide what program $z$should be kept in $\mathcal{P}$ by constructing 4 program variants, and keeping the one that minimizes $\mathcal{F}$. The variants we consider are as follows. *(i)* Use z_c. *(ii)* Use z_r (note this variant will always be chosen if $r = 0$). *(iii)* Greedily merge z_r into z_c. To do this, we first compute $split(z_r) = E_r = \{e_r^0, ..., e_r^{|E_c|}\}$. Then for each e_r^i, we find $prim_r^i$, and see if there is a set of matching instances in E_c, M, such that $prim_r^i = \{prim_c^j \text{ for } j \in M\}$. If M exists, then we compare the cost under $\mathcal{F}$ of e_r^i versus the sum of each e_c^j (with $|M| - 1$ combinator calls): if e_r^i improves $\mathcal{F}$ then each e_c^j is removed from E_c, and e_r^i is added into E_c. *(iv)* Greedily construct an entirely new program from Q_d. First E_r is

added into Q_d. Then Q_d greedily creates a new program by initializing E_n (to be empty) and repeating the following steps: find the *cost* of each e in Q_d, take the minimum cost expression e^* and add it into E_n, and temporarily remove all other entries of Q_d that have nonzero overlap with $prim_e^*$. This is repeated until E_n contains expressions that cover all primitives in d.

After these four program variants have been created (where in *(iii)* and *(iv)* combinator operations are applied over E_c and E_n respectively), the variant with the minimum score under $\mathcal{F}$ is kept in $\mathcal{P}$. Finally, we note that some extra logic is required to ensure that Q_d and z_c are kept up-to-date. Whenever the integration phase tries removing a function f from $\mathcal{L}$, all expressions in Q_d that use f are temporarily removed. Moreover if f appears in z_c, then the greedy search in *(iv)* is used to find replacement expressions for z_c.

F.2.6 Preference Ordering of Parametric Relationships

The proposal phase (Section 8.3.1) generates candidate abstractions using a greedy search. These candidate abstractions contain parametric expressions. Below we detail the preference ordering we use to search for matching parametric expressions with respect to a sampled cluster.

The choice of which parametric expression to propose is always made in the context of a cluster, that contains a structure and a group of parameterizations. As we are filling in slots for the candidate abstraction, we may have already instantiated free variables that were used in previous slots. To find a possible expression for the current parameter slot, we reason over the free variables previously instantiated. We iterate through a preference ordering that considers increasingly complex parametric expressions over previous variables: expressions with only constants, then one variable expressions, two variable expressions, and finally three variable expressions. The set of all expressions under $\mathcal{L}$ that contain n variables can be found by calculating the cross-product of (i) all parametric operator combinations that would require n variables with (ii) all ordered sequences of n previously instantiated variables. To avoid overfitting, we limit the possible constants we consider (just 0 for our shape grammars). For each expression, we check which members of the cluster are covered by that expression. Once we find a set of expressions that collectively cover all instances within the cluster, we break out of this loop early. This procedure creates a large set of possible expressions (visualized in Figure 8.4), from which one is chosen according to the *score* function.

F.2.7 E-graphs

Our refactor operation (Section 8.4), implements e-graphs using the Egg library [218]. Egg provides support for defining a DSL, rewrite operations, and a cost function, that can be used by an extraction operation. Egg provides an interface for defining rewrites that reason over conditional logic, but they cannot be directly applied for our use case. Our version of conditional rewrites requires that each rewrite has access to a shared e-class-to-real-value mapping, so we build out this feature. Maintaining this mapping requires dummy rewrite operations, that check for structural matches for various parametric operations, and update the mapping, without changing the structure of the e-graph. When we first instantiate an e-graph, we apply dummy rewrites that match on each float variable, V_i, and adds an entry for V_i into the mapping. Then, during each rewrite round, after applying all semantic and abstraction rewrites, we apply all dummy rewrites, to ensure the mapping is up-to-date (this handles the blue `Mul` e-class from Figure 8.5). For each domain, we provide Egg with a set of semantic rewrites that express domain-specific semantic preserving transformations. There are 25 such rewrites for 3D, and 16 such rewrites for 2D. We ablate the importance of including these semantic rewrites in our ablation experiment (Section 8.5.4).

F.2.8 Unsupervised Primitive Decomposition

As described in Section 8.5.5, we make use of an unsupervised cuboid decomposition method, so that we can apply ShapeCoder to shapes from datasets that contain only meshes. We use the approach described by [230], using their released pretrained models to predict cuboid decompositions over chairs from their test set. We compile a dataset of 400 such predictions, and parse these output predictions into a primitive representation compatible with our method. This conversion procedure performs a few minor filtering steps, rejecting scenes that contain more than 12 cuboids (we found these often were noisy predictions) and snapping cuboids to be axis-aligned whenever their Euler angles were within a 0.05 threshold of 0 or 2π.

F.2.9 Generative Model for Programs

We provide details for the generative model described in Section 8.5.6. This model is capable of synthesizing novel 3D shapes. We implement our generative model as a Transformer decoder, with causal masking. It uses a CNN to encode a shape voxelization into an embedding vector, which conditions the Transformer that autoregressively predicts tokens from $\mathcal{L}$. The network starts with a blank scene, iteratively predicts an expression e from $\mathcal{L}$, and adds it back into the scene (which will be encoded by the CNN in the next

time-step). This process is repeated until a special 'STOP' token is predicted.

We source training data for this model by running our post hoc inference procedure (Section 8.5.3) over a dataset of 3600 chairs, to form a program dataset $\mathcal{P}$. For each epoch, we randomize expression ordering by applying *split* (Section F.2.5) to each $z \in \mathcal{P}$, shuffling the expressions found by *split*, and treating every (previous expressions, next expression) tuple as an independent training example. We use teacher-forcing and maximum likelihood updates to train the generative model. We train the model for 4000 epochs. It has 8 Transformer layers, 16 heads, a hidden size of 256. We train with a batch size of 64, dropout of 0.1, and a learning rate of 0.0005 . At inference time, we use nucleus sampling (top 90%) to predict expressions from the networks probabilities. The 'without abstractions' version we compare against has exactly the same setup, except the post-hoc inference procedure was run using the starting $\mathcal{L}$ version (not the one discovered by ShapeCoder).

F.3 Toy 2D Grammar Experiments

Before moving to 3D domains, we evaluated ShapeCoder's ability to discover abstractions on a more basic 2D dataset. We designed a 2D shape grammar (Section F.1) and manually designed a sampling procedure for this grammar that would produce 'chair-like' output scenes (combinations of rectangles) – see our public code release for details. The sampling procedure was, in fact, implemented as a single abstraction function, that takes a fixed amount of input parameters and outputs a program using functions from the base DSL. These input parameters controlled both shape parameters (e.g. chair height or width) along with control flow decisions (e.g. should the back have vertical or horizontal bars). This paradigm can be considered as an 'oracle' best-case abstraction for this 2D domain, e.g. what a manually designed abstraction would look like. We evaluate how ShapeCoder was able to improve $\mathcal{F}$ on this dataset, compared with this oracle, in the below table.

Method	$\mathcal{F} \Downarrow$
Input Prims	65.3
No Abstraction	48.6
ShapeCoder	27.3
Oracle	22.7

The oracle single abstraction (that takes in 7 categorical variables, and 9 float variables) is able to

achieve the best compression metric. However, ShapeCoder is able to come reasonable close to this target on this toy domain, and improves $\mathcal{F}$ significantly over using either the input primitives, or when only the dream+wake phases are used (No Abstraction). Of note, the oracle abstraction function actually has access to DSL components we don't provide to ShapeCoder (control flow `Switch` and `If`/`Else` operators). Currently ShapeCoder is not able to discover abstractions that introduce different control flow decisions, as these types of operators would never be inferred during the wake phase.

F.4 DreamCoder Experiments

DreamCoder [42] is an inspiring system capable of generalizing across many domains. It makes no assumptions over its input data, which creates a difficult program induction problem. It solves this issue by gradually building up a library of discovered abstractions tailored to the input domain. Dreamcoder's program inference step (i.e. its wake phase) performs enumerative search guided by a library version; when solutions to the program induction task are more compact under a 'good' library version, solutions will be found more quickly in this search process. A downside of this framing is that there is an implicit assumption that the input data contains a curriculum of tasks, that is needed to bootstrap this procedure. Specifically, some tasks in the input set need to have relatively high probability under the base DSL: if enumerative search does not find any solutions to the 'simple' tasks, then no abstractions can ever be discovered, that are necessary to help solve the more 'complex' tasks.

Complex visual programming datasets, like manufactured 3D shapes, don't typically contain a curriculum of tasks. In some cases, a curriculum can be created, but this typically requires access to detailed shape annotations (e.g. a semantic part hierarchy). As such, due to the lack of a curriculum, combined with the complexity of the 3D shape program inference problem, when we attempted to run DreamCoder over PartNet data we observed it did not find any solutions.

Beyond this observation, we also argue that DreamCoder, as a general program induction system, is not as well-suited for visual programming domains, compared with ShapeCoder. Critically, DreamCoder has no mechanism that reasons over parametric relationships between continuous variable, which is of great importance for many visual programming domains (including manufactured shapes, where part-to-part relationships are spatially constrained). While DreamCoder does show success on simple 2D visual domains, it discretizes continuous variables and treats parametric operators as standard functions in the base DSL. To test if DreamCoder has an inductive bias to discover abstractions with parametric relationships under these

assumptions, we designed a toy experiment, that we explain below.

We design a very simple shape grammar for the toy 2D language. Where between 1 and 3 primitives are combined together, and where each primitive is created by an abstraction that takes in two input parameters (so two degrees of freedom are constrained). We write this grammar as:

$START \rightarrow ABS \mid$
$\quad$ `Union`(ABS, ABS) $\mid$
$\quad$ `Union`(ABS, `Union`(ABS, ABS)) ;
$ABS \rightarrow$ `Move`(`Rect`(a, a+b), b, a-b)
a,b $\rightarrow r \in (0, 1)$

Where real-values (e.g. a and b) are discretized into 20 values between 0 and 1. The *ABS* function is easily identifiable by ShapeCoder, because it explicitly checks for these types of parametric relationships during the proposal phase, and this relationship is present in every input scene. We ran DreamCoder over a dataset of 100 samples from this grammar with a budget of 24 hours wall-clock time. To match the computational requirements of ShapeCoder, we used a single workstation with a Intel i7-11700K CPU, and a python-based executor implementation. Under these conditions, DreamCoder did not discover the 'correct' abstraction with the proper parametric pattern. Moreover, even for this simple grammar, DreamCoder only discovered solutions for around 50% of the tasks (and none of the tasks with 3 `Union` operations). While these results might be improved by making better use of computational resources (running enumerative search over a cluster of machines, designing a faster executor, increasing the wall-clock budget), we believe this example illustrates why DreamCoder is not particularly well-suited for complex visual programming domains.

Appendix G

Additional Details and Results for ShapeLib

In Appendix G, we supply additional details for the ShapeLib method introduced in Chapter 9.

G.1 Additional Method Details

G.1.1 Objective Function

When searching for programs that explain shapes, we need an objective function to guide the search. We take inspiration from ShapeCoder (Chapter 8) and formulated an objective function as a weighted average of two terms. One of these terms counts up the number of degrees of freedom in the program representation, for simplicity we treat every token in the program as a degree of freedom with the same weight (1.). Another term ensures that the produced geometry does not deviate too far from the target structure. We calculate the geometric error (more on this in the next paragraph), and add that into our objective function with a weight of 10.

The geometric error function we use takes in two sets of unordered primitives. For every pair of primitives from the predicted to target set, we calculate the maximum minimum distance between any two corners from one primitive to the other. We then use a matching algorithm to assign a stable pairing between the two sets. If any of the distances is above a threshold (0.25, where shapes are normalized to lie within the unit sphere), then we say that there is infinite geometric error. Otherwise, the geometric error is an average of the

maximum minimum corner distance (MMCD), calculated according to the best match.

G.1.2 Network Design

We implement all of our networks in PyTorch [158]. All of our experiments are run on NVIDIA GeForce RTX 3090 graphic cards with 24GB of VRAM. We use the Adam optimizer [106] with a learning rate of 1e-4. We implement our recognition network as a Transformer decoder. Our network has 4 layers, 4 heads, model dim of 256, and a full feature dim of 1024.

This network has full attention over the conditioning information: each primitive in the input shape is quantized and treated as a discrete token. We order the primitives according to their x-y-z positions, as we do not know how they should be ordered otherwise. Programs are similarly tokenized, and our network is trained through teacher forcing. We use learned positional encodings, these cap the maximum sequence lengths and primitive amounts our network can reason over: 20 primitives and programs of up to length 64. We train with a batch size of 128. For point cloud inputs, we replace the primitive token encodings with an embedding produced by a PointNet++ [163] network. For voxel inputs, we replace the primitive token encodings with an embedding produced by a 3D-CNN. We train our networks for between 4-12 hours, depending on the category and task.

G.1.3 Synthetic Data Sampler

We perform two rounds of automated feedback for each 'sample_shape' function generated by the *o1* LLM model. This iterative approach aims to refine the sampler's outputs by addressing discrepancies and improving alignment with respect to seed set patterns. In each round of feedback, we evaluate the function by sampling a diverse set of shapes and assessing various aspects of its behavior. We examine whether all functions in the library were used, whether all parameter types were employed, and whether all output structures described in the function's documentation were produced. These checks are performed automatically. Additionally, we analyze the structures generated by the sampled functions and determine their similarity to those observed during the validation stage. If significant deviations are detected, measured in the parameter space of each function, the sampler is instructed to update its logic to produce outputs closer to the expected structures.

G.2 Additional Experimental Details

G.2.1 Cost and Timing

We provide detailed estimates for how expensive it is (from a time and API monetary expense perspective) to use our system to discover libraries of shape abstraction functions. To produce 20 shape descriptions from images using gpt-4o: 10 cents and 1-2 minutes. To create library interfaces from textual descriptions with o1mini: 25 cents, 2-4 minutes. To propose function applications over (20) shapes with (1) o1mini call and (4) gpt-4o calls: $2-3 and 15-25 minutes. To propose (4) implementations for each function with o1mini: $2-4 and 15-30 minutes. To propose a single program sampler with o1: 50 cents and 1 minute. In total, this amounts to $5-8 and 30 minutes to 1 hour.

Notice that by default we use o1mini, but sometimes deviate based on our developmental experience. Making function applications without knowing function implementations is a 'guess-based' exercise, so we are fine with the increased error rate that 4o produces in this step. For the most complex tasks, like implementing a synthetic data sampler, we turn to o1 as we are able to provide enough task guidance and directives to make use of its 'reasoning' capabilities.

G.2.2 Data

Collections of example shapes in the seed set are chosen by an expert user who has a design intent in mind (they also express this intent in natural language in the function descriptions). Specifically, we have the user select 20 partNet shapes and put them in a list, and then we can automatically produce the rest of the structured data from the partNet annotations.

After we have selected these two shapes, we create separate 'training' and 'validation' sets of shapes by randomly splitting up Partnet object instances. We run all experiments over validation shapes, unless otherwise stated, and use the training shapes to get paired data for the visual program induction step that maps from unstructured geometry to a shape abstraction program. The size of these train/val sets is 4000/1000 for chairs, 1216/400 for storage, 4000/1000 for tables, 434/400 for faucet, and 2625/656 for lamps.

G.2.3 LLM-Direct Baseline

The LLM-direct is an ablated version of our method that relies on only the prior of the LLM and the design intent of the expert user in the form of function descriptions. We compare against it to validate the need for

using the seed set of shapes alongside the natural language specification.

This baseline, is equivalent to our method modulo a few critical changes. The interface creation step is exactly the same. After this step though, it immediately implements each function, without using any input/output guidance about how this function should constructed. As it has no seed set, it assumes that the LLM has perfectly implemented each function, and next advances to the synthetic sampler design stage where it prompts the LLM to produce a 'sample_shape' function from its constructed library. Then, like the full ShapeLib system, we can train a recognition network on data produced by this random sampling procedure.

G.2.4 ShapeCoder

In our comparisons against ShapeCoder we use the officially released implementation. The only change we make is removing the rotation operation from the base ShapeCoder language, as we focus on structures of axis-aligned primitives in our experiments. We develop ShapeCoder's library of abstraction over the same seed set of 20 shapes, which is much smaller than the large datasets used in the original ShapeCoder system (400 shapes). Nevertheless, we find that ShapeCoder can generalize (in terms of compression, at least) fairly well even from these 20 shapes.

We experiment with discovering ShapeCoder libraries over a larger seed set of 400 shapes, and find that compression improves slightly on validation shapes, but not by a huge margin (Obj goes from 52.1 to 46.1, while the average library size grows from 19 to 24). Despite learning this library over a large collection of shapes, we still observe that this 'ShapeCoder-400' variant does not find more semantically aligned function applications over validation structures. In fact, its semantic entropy performance worsens (chair: 1.67 to 1.84, table: 1.578 to 2.16, storage: 2.07 to 2.08, lamp: 1.7 to 1.9, faucet: 2.1 to 2.3) We view this result as lending our framing additional support: compression alone (even over a large dataset) is not enough to develop good shape abstraction libraries, top-down semantic guidance is also required.

Bibliography

[1] Ben Abbatematteo, Stefanie Tellex, and George Konidaris. Learning to generalize kinematic models to novel objects. In *Proceedings of the Third Conference on Robot Learning*, 2019.

[2] Panos Achlioptas, Olga Diamanti, Ioannis Mitliagkas, and Leonidas Guibas. Learning representations and generative models for 3d point clouds, 2018.

[3] Adobe. Substance Designer. `https://www.adobe.com/products/substance3d-designer.html`. Accessed: 2022-09-26.

[4] Rio Aguina-Kang, Maxim Gumin, Do Heon Han, Stewart Morris, Seung Jean Yoo, Aditya Ganeshan, R Kenny Jones, Qiuhong Anna Wei, Kailiang Fu, and Daniel Ritchie. Open-universe indoor scene generation using llm program synthesis and uncurated object databases. *arXiv preprint arXiv:2403.09675*, 2024.

[5] Autodesk. Fusion 360. `https://www.autodesk.com/products/fusion-360/`. Accessed: 2022-10-16.

[6] Autodesk Maya Wiki. Hypershade. `https://autodeskmaya.fandom.com/wiki/Hypershade`. Accessed: 2022-10-16.

[7] Matej Balog, Rishabh Singh, Petros Maniatis, and Charles Sutton. Neural program synthesis with a differentiable fixer, 2020.

[8] Harry Barrow, J Tenenbaum, A Hanson, and E Riseman. Recovering intrinsic scene characteristics. *Comput. vis. syst*, 2(3-26):2, 1978.

[9] Blender Foundation. Blender - A 3D Modelling and Rendering Package, 2024. Version 4.0.

[10] Sam Bond-Taylor, Adam Leach, Yang Long, and Chris G. Willcocks. Deep generative modelling: A comparative review of VAEs, GANs, normalizing flows, energy-based and autoregressive models. *IEEE Transactions on Pattern Analysis and Machine intelligence (TPAMI)*, 44(11):7327–7347, 2022.

[11] Matthew Bowers, Theo X. Olausson, Lionel Wong, Gabriel Grand, Joshua B. Tenenbaum, Kevin Ellis, and Armando Solar-Lezama. Top-down synthesis for library learning. *Proc. ACM Program. Lang.*, 7(POPL), jan 2023.

[12] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, Harsha Nori, Hamid Palangi, Marco Tulio Ribeiro, and Yi Zhang. Sparks of artificial general intelligence: Early experiments with gpt-4, 2023.

[13] Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. *International Conference on Learning Representations (ICLR)*, 2018.

[14] David Cao, Rose Kunkel, Chandrakana Nandi, Max Willsey, Zachary Tatlock, and Nadia Polikarpova. Babble: Learning better abstractions with e-graphs and anti-unification. *Proc. ACM Program. Lang.*, 7(POPL), jan 2023.

[15] Alexandre Carlier, Martin Danelljan, Alexandre Alahi, and Radu Timofte. DeepSVG: A hierarchical generative network for vector graphics animation. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 33, pages 16351–16361, 2020.

[16] 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, and Fisher Yu. ShapeNet: An Information-Rich 3D Model Repository. *arXiv:1512.03012*, 2015.

[17] Eugene Charniak. Statistical parsing with a context-free grammar and word statistics. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence*, AAAI'97/IAAI'97, page 598–603. AAAI Press, 1997.

[18] Siddhartha Chaudhuri, Daniel Ritchie, Jiajun Wu, Kai Xu, and Hao Zhang. Learning Generative Models of 3D Structures. *Computer Graphics Forum*, 2020.

[19] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N.

Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. 2021.

[20] Qimin Chen, Zhiqin Chen, Vladimir G Kim, Noam Aigerman, Hao Zhang, and Siddhartha Chaudhuri. Decollage: 3d detailization by controllable, localized, and learned geometry enhancement. In *European Conference on Computer Vision*, 2025.

[21] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *The Twelfth International Conference on Learning Representations*, 2024.

[22] Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. In *International Conference on Learning Representations*, 2019.

[23] Zhiqin Chen, Vladimir G. Kim, Matthew Fisher, Noam Aigerman, Hao Zhang, and Siddhartha Chaudhuri. Decor-gan: 3d shape detailization by conditional refinement. *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2021.

[24] Zhiqin Chen, Andrea Tagliasacchi, and Hao Zhang. Bsp-net: Generating compact meshes via binary space partitioning. *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

[25] Zhiqin Chen, Kangxue Yin, Matthew Fisher, Siddhartha Chaudhuri, and Hao Zhang. Bae-net: Branched autoencoder for shape co-segmentation. *Proceedings of International Conference on Computer Vision (ICCV)*, 2019.

[26] Zhiqin Chen and Hao Zhang. Learning implicit fields for generative shape modeling. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[27] Robert L. Cook. Shade trees. In *Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, page 223–231, 1984.

[28] F. Croitoru, V. Hondru, R. Ionescu, and M. Shah. Diffusion models in vision: A survey. *IEEE Transactions on Pattern Analysis and Machine intelligence (TPAMI)*, 45(09):10850–10869, sep 2023.

[29] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022.

[30] Dassault Systemes. SOLIDWORKS. `https://www.solidworks.com/`. Accessed: 2022-10-16.

[31] Eyal Dechter, Jon Malmaud, Ryan P. Adams, and Joshua B. Tenenbaum. Bootstrap learning via modular concept discovery. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, IJCAI '13, page 1302–1309. AAAI Press, 2013.

[32] M. Deitke, D. Schwenk, J. Salvador, L. Weihs, O. Michel, E. VanderBilt, L. Schmidt, K. Ehsanit, A. Kembhavi, and A. Farhadi. Objaverse: A universe of annotated 3d objects. In *2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 13142–13153, Los Alamitos, CA, USA, jun 2023. IEEE Computer Society.

[33] İ. Demir, D. G. Aliaga, and B. Benes. Proceduralization for editing 3d architectural models. In *2016 Fourth International Conference on 3D Vision (3DV)*, 2016.

[34] Boyang Deng, Sumith Kulal, Zhengyang Deng, Congyue Deng, Yonglong Tian, and Jiajun Wu. Unsupervised learning of shape programs with repeatable implicit parts. In *Advances in Neural Information Processing Systems*, 2022.

[35] Boyang Deng, Sumith Kulal, Zhengyang Dong, Congyue Deng, Yonglong Tian, and Jiajun Wu. Unsupervised learning of shape programs with repeatable implicit parts. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[36] Valentin Deschaintre, Miika Aittala, Frédo Durand, George Drettakis, and Adrien Bousseau. Single-image SVBRDF capture with a rendering-aware deep network. *ACM Transactions on Graphics (TOG)*, 37(128):15, 2018.

[37] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, page 990–998. JMLR.org, 2017.

[38] Tao Du, Jeevana Priya Inala, Yewen Pu, Andrew Spielberg, Adriana Schulz, Daniela Rus, Armando Solar-Lezama, and Wojciech Matusik. Inversecsg: automatic conversion of 3D models to csg trees. In *Annual Conference on Computer Graphics and Interactive Techniques Asia (SIGGRAPH Asia)*. ACM, 2018.

[39] Sutherland Ivan Edward. *SketchPad: A man-machine graphical communication system*. PhD thesis, Massachusetts Institute of Technology, 1963.

[40] Harrison Edwards and Amos Storkey. Towards a neural statistician. In *5th International Conference on Learning Representations (ICLR 2017)*, pages 1–13, Apr. 2017. 5th International Conference on Learning Representations, ICLR 2017 ; Conference date: 24-04-2017 Through 26-04-2017.

[41] Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. Write, execute, assess: Program synthesis with a repl. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[42] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B Tenenbaum. DreamCoder: Bootstrapping inductive program synthesis with wake-sleep library learning. In *ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (SIGPLAN)*, pages 835–850, 2021.

[43] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, page 226–231. AAAI Press, 1996.

[44] Haoqiang Fan, Hao Su, and Leonidas J Guibas. A point set generation network for 3D object reconstruction from a single image. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 605–613, 2017.

[45] Haoqiang Fan, Hao Su, and Leonidas J Guibas. A point set generation network for 3d object reconstruction from a single image. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 605–613, 2017.

[46] Reuben Feinman and Brenden M. Lake. Learning task-general representations with generative neuro-symbolic modeling. In *International Conference on Learning Representations*, 2021.

[47] Weixi Feng, Wanrong Zhu, Tsu-jui Fu, Varun Jampani, Arjun Akula, Xuehai He, Sugato Basu, Xin Eric Wang, and William Yang Wang. Layoutgpt: Compositional visual planning and generation with large language models. *Advances in Neural Information Processing Systems*, 36, 2024.

[48] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1126–1135. PMLR, 06–11 Aug 2017.

[49] Jonas Freiknecht and Wolfgang Effelsberg. A survey on the procedural generation of virtual worlds. *Multimodal Technologies and Interaction*, 1(4), 2017.

[50] Anna Frühstück, Ibraheem Alhashim, and Peter Wonka. TileGAN: Synthesis of large-scale non-homogeneous textures. *ACM Transactions on Graphics (TOG)*, 38(4), 2019.

[51] Matheus Gadelha, Aruni RoyChowdhury, Gopal Sharma, Evangelos Kalogerakis, Liangliang Cao, Erik Learned-Miller, Rui Wang, and Subhransu Maji. Label-efficient learning on point clouds using approximate convex decompositions. In *European Conference on Computer Vision (ECCV)*, 2020.

[52] Aditya Ganeshan, Ryan Huang, Xianghao Xu, R Kenny Jones, and Daniel Ritchie. Parsel: Parameterized shape editing with language. *ACM Transactions on Graphics (TOG)*, 43(6):1–14, 2024.

[53] Aditya Ganeshan, R. Kenny Jones, and Daniel Ritchie. Improving unsupervised visual program inference with code rewriting families. In *Proceedings of the International Conference on Computer Vision (ICCV)*, 2023.

[54] Yaroslav Ganin, Sergey Bartunov, Yujia Li, Ethan Keller, and Stefano Saliceti. Computer-aided design as language. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

[55] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. Image style transfer using convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[56] Sherif Ghali. *Constructive solid geometry*, pages 277–283. Springer, 2008.

[57] Giorgio Giannone, Didrik Nielsen, and Ole Winther. Few-shot diffusion models, 2022.

[58] Gabriel Grand, Lionel Wong, Matthew Bowers, Theo X. Olausson, Muxin Liu, Joshua B. Tenenbaum, and Jacob Andreas. Lilo: Learning interpretable libraries by compressing and documenting code, 2023.

[59] Thibault Groueix, Matthew Fisher, Vladimir G. Kim, Bryan C. Russell, and Mathieu Aubry. AtlasNet: A Papier-Mâché Approach to Learning 3D Surface Generation. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[60] Paul Guerrero, Milos Hasan, Kalyan Sunkavalli, Radomir Mech, Tamy Boubekeur, and Niloy Mitra. MatFormer: A generative model for procedural materials. *ACM Transactions on Graphics (TOG)*, 41(4), 2022.

[61] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (SIGPLAN)*, 2011.

[62] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

[63] Jianwei Guo, Haiyong Jiang, Bedrich Benes, Oliver Deussen, Xiaopeng Zhang, Dani Lischinski, and Hui Huang. Inverse procedural modeling of branching structures by inferring l-systems. *ACM Transactions on Graphics (TOG)*, 39(5):1–13, 2020.

[64] Yu Guo, Cameron Smith, Miloš Hašan, Kalyan Sunkavalli, and Shuang Zhao. MaterialGAN: Reflectance capture using a generative SVBRDF model. *ACM Transactions on Graphics (TOG)*, 39(6):254:1–254:13, 2020.

[65] Anchit Gupta, Wenhan Xiong, Yixin Nie, Ian Jones, and Barlas Oğuz. 3dgen: Triplane latent diffusion for textured mesh generation, 2023.

[66] Kavi Gupta, Peter Ebert Christensen, Xinyun Chen, and Dawn Song. Synthesize, execute and debug: Learning to repair for neural program synthesis. In *Advances in Neural Information Processing Systems*, 2020.

[67] Tanmay Gupta and Aniruddha Kembhavi. Visual programming: Compositional visual reasoning without training. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 14953–14962, June 2023.

[68] David Ha and Douglas Eck. A neural representation of sketch drawings. In *International Conference on Learning Representations*, 2018.

[69] Chi Han, Jiayuan Mao, Chuang Gan, Josh Tenenbaum, and Jiajun Wu. Visual concept-metaconcept learning. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[70] Junxian He, Jiatao Gu, Jiajun Shen, and Marc'Aurelio Ranzato. Revisiting self-training for neural sequence generation. In *International Conference on Learning Representations (ICLR)*, 2020.

[71] Brian Hempel, Justin Lubin, and Ravi Chugh. Sketch-n-Sketch: Output-directed programming for SVG. In *ACM Symposium on User Interface Software and Technology (UIST)*, pages 281–292, 2019.

[72] Philipp Henzler, Valentin Deschaintre, Niloy J Mitra, and Tobias Ritschel. Generative modelling of BRDF textures from flash images. *ACM Transactions on Graphics (TOG)*, 40(6), 2021.

[73] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. Gans trained by a two time-scale update rule converge to a local nash equilibrium. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[74] Luke B Hewitt, Tuan Anh Le, and Joshua B Tenenbaum. Learning to learn generative programs with memoised wake-sleep. In *Uncertainty in Artificial Intelligence*.

[75] Luke B. Hewitt, Maxwell I. Nye, Andreea Gane, Tommi Jaakkola, and Joshua B. Tenenbaum. The variational homoencoder: Learning to learn high capacity generative models from few examples, 2018.

[76] GE Hinton, P Dayan, BJ Frey, and RM Neal. The "wake-sleep" algorithm for unsupervised neural networks. *Science*, 268(5214):1158–1161, 1995.

[77] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models, 2020.

[78] Yicong Hong, Kai Zhang, Jiuxiang Gu, Sai Bi, Yang Zhou, Difan Liu, Feng Liu, Kalyan Sunkavalli, Trung Bui, and Hao Tan. LRM: Large reconstruction model for single image to 3d. In *The Twelfth International Conference on Learning Representations*, 2024.

[79] Yicong Hong, Kai Zhang, Jiuxiang Gu, Sai Bi, Yang Zhou, Difan Liu, Feng Liu, Kalyan Sunkavalli, Trung Bui, and Hao Tan. Lrm: Large reconstruction model for single image to 3d. In *ICLR*, 2024.

[80] Wentao Hu, Jia Zheng, Zixin Zhang, Xiaojun Yuan, Jian Yin, and Zihan Zhou. Plankassembly: Robust 3d reconstruction from three orthographic views with learnt shape programs. In *ICCV*, 2023.

[81] Ziniu Hu, Ahmet Iscen, Aashi Jain, Thomas Kipf, Yisong Yue, David A Ross, Cordelia Schmid, and Alireza Fathi. Scenecraft: An llm agent for synthesizing 3d scenes as blender code. In *Forty-first International Conference on Machine Learning*, 2024.

[82] Ian Huang, Guandao Yang, and Leonidas Guibas. Blenderalchemy: Editing 3d graphics with vision-language models. In *European Conference on Computer Vision*, pages 297–314. Springer, 2024.

[83] Yujia Huang, Adishree Ghatare, Yuanzhe Liu, Ziniu Hu, Qinsheng Zhang, Chandramouli Shama Sastry, Siddharth Gururani, Sageev Oore, and Yisong Yue. Symbolic music generation with non-differentiable rule guided diffusion. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp, editors, *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 19772–19797. PMLR, 21–27 Jul 2024.

[84] Ka-Hei Hui, Ruihui Li, Jingyu Hu, and Chi-Wing Fu. Neural wavelet-domain diffusion for 3d shape generation. December 2022.

[85] Irvin Hwang, Andreas Stuhlmüller, and Noah D. Goodman. Inducing Probabilistic Programs by Bayesian Program Merging. *CoRR*, arXiv:1110.5667, 2011.

[86] IDV, Inc. SpeedTree – 3D vegetation modeling and middleware. `https://store.speedtree.com/`.

[87] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A Efros. Image-to-image translation with conditional adversarial networks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[88] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*, 2024.

[89] Justin Johnson, Bharath Hariharan, Laurens van der Maaten, Li Fei-Fei, C Lawrence Zitnick, and Ross Girshick. Clevr: A diagnostic dataset for compositional language and elementary visual reasoning. In *CVPR*, 2017.

[90] Justin Johnson, Bharath Hariharan, Laurens van der Maaten, Judy Hoffman, Li Fei-Fei, C Lawrence Zitnick, and Ross Girshick. Inferring and executing programs for visual reasoning. In *IEEE International Conference on Computer Vision (ICCV)*, 2017.

[91] R. Kenny Jones, Theresa Barton, Xianghao Xu, Kai Wang, Ellen Jiang, Paul Guerrero, Niloy J. Mitra, and Daniel Ritchie. Shapeassembly: Learning to generate programs for 3d shape structure synthesis. *ACM Transactions on Graphics (TOG)*, 39(6), 2020.

[92] R. Kenny Jones, David Charatan, Paul Guerrero, Niloy J. Mitra, and Daniel Ritchie. Shapemod: Macro operation discovery for 3d shape programs. *ACM Transactions on Graphics (TOG)*, 40(4), 2021.

[93] R. Kenny Jones, Siddhartha Chaudhuri, and Daniel Ritchie. Learning to infer generative template programs for visual concepts. In *International Conference on Machine Learning (ICML)*, 2024.

[94] R. Kenny Jones, Paul Guerrero, Niloy J. Mitra, and Daniel Ritchie. Shapecoder: Discovering abstractions for visual programs from unstructured primitives. *ACM Transactions on Graphics (TOG), Siggraph 2023*, 42(4), 2023.

[95] R Kenny Jones, Paul Guerrero, Niloy J Mitra, and Daniel Ritchie. Shapelib: designing a library of procedural 3d shape abstractions with large language models. *arXiv preprint arXiv:2502.08884*, 2025.

[96] R. Kenny Jones, Aalia Habib, Rana Hanocka, and Daniel Ritchie. The neurally-guided shape parser: Grammar-based labeling of 3d shape regions with approximate inference. *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022.

[97] R. Kenny Jones, Aalia Habib, and Daniel Ritchie. Shred: 3d shape region decomposition with learned local operations. *ACM Transactions on Graphics (TOG)*, 41(6), 2022.

[98] R. Kenny Jones, Homer Walke, and Daniel Ritchie. Plad: Learning to infer shape programs with pseudo-labels and approximate distributions. *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022.

[99] R. Kenny Jones, Renhao Zhang, Aditya Ganeshan, and Daniel Ritchie. Learning to edit visual programs with self-supervision. In *Advances in Neural Information Processing Systems*, 2024.

[100] Jacob Kahn, Ann Lee, and Awni Hannun. Self-training for end-to-end speech recognition. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 7084–7088. IEEE, 2020.

[101] Kacper Kania, Maciej Zieba, and Tomasz Kajdanowicz. UCSG-NET - unsupervised discovering of constructive solid geometry tree. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 33, pages 8776–8786, 2020.

[102] Shreyas Kapur, Erik Jenner, and Stuart Russell. Diffusion on syntax trees for program synthesis. In *The Thirteenth International Conference on Learning Representations*, 2025.

[103] Amlan Kar, Aayush Prakash, Ming-Yu Liu, Eric Cameracci, Justin Yuan, Matt Rusiniak, David Acuna, Antonio Torralba, and Sanja Fidler. Meta-sim: Learning to generate synthetic datasets. In *IEEE International Conference on Computer Vision (ICCV)*, 2019.

[104] Tero Karras, Samuli Laine, and Timo Aila. A style-based generator architecture for generative adversarial networks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[105] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 3d gaussian splatting for real-time radiance field rendering. *ACM Transactions on Graphics*, 42(4), July 2023.

[106] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, 2015.

[107] Diederik P. Kingma and Max Welling. Auto-Encoding Variational Bayes. In *International Conference on Learning Representations (ICLR)*, 2014.

[108] Arno Knapitsch, Jaesik Park, Qian-Yi Zhou, and Vladlen Koltun. Tanks and temples: Benchmarking large-scale scene reconstruction. *ACM Transactions on Graphics*, 36(4), 2017.

[109] Milin Kodnongbua, Benjamin Jones, Maaz Bin Safeer Ahmad, Vladimir Kim, and Adriana Schulz. Reparamcad: Zero-shot cad re-parameterization for interactive manipulation. In *SIGGRAPH Asia 2023 Conference Papers*, pages 1–12, 2023.

[110] Eric Kolve, Roozbeh Mottaghi, Winson Han, Eli VanderBilt, Luca Weihs, Alvaro Herrasti, Daniel Gordon, Yuke Zhu, Abhinav Gupta, and Ali Farhadi. AI2-THOR: An Interactive 3D Environment for Visual AI. *arXiv*, 2017.

[111] Harold W Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

[112] Sumith Kulal, Jiayuan Mao, Alex Aiken, and Jiajun Wu. Programmatic concept learning for human motion description and synthesis. 2022.

[113] Peter Kulits, Haiwen Feng, Weiyang Liu, Victoria Fernandez Abrevaya, and Michael J. Black. Rethinking inverse graphics with large language models. *Transactions on Machine Learning Research*, 2024.

[114] Matt J. Kusner, Brooks Paige, and José Miguel Hernández-Lobato. Grammar variational autoencoder. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, page 1945–1954. JMLR.org, 2017.

[115] Brenden M. Lake, Ruslan Salakhutdinov, and Joshua B. Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015.

[116] Brenden M Lake, Ruslan Salakhutdinov, and Joshua B Tenenbaum. The omniglot challenge: a 3-year progress report. *Current Opinion in Behavioral Sciences*, 29:97–104, 2019. Artificial Intelligence.

[117] Bosheng Li, Jonathan Klein, Dominik L. Michels, Bedrich Benes, Sören Pirk, and Wojtek Pałubicki. Rhizomorph: The coordinated function of shoots and roots. *ACM Trans. Graph.*, 42(4), jul 2023.

[118] Changjian Li, Hao Pan, Adrien Bousseau, and Niloy J. Mitra. Sketch2CAD: Sequential CAD modeling by sketching in context. *ACM Transactions on Graphics (TOG)*, 39(6):164:1–164:14, 2020.

[119] Changjian Li, Hao Pan, Adrien Bousseau, and Niloy J. Mitra. Free2CAD: Parsing freehand drawings into CAD commands. *ACM Transactions on Graphics (TOG)*, 41(4):93:1–93:16, 2022.

[120] Jun Li, Kai Xu, Siddhartha Chaudhuri, Ersin Yumer, Hao Zhang, and Leonidas Guibas. GRASS: Generative recursive autoencoders for shape structures. *ACM Transactions on Graphics (TOG)*, 36(4):1–14, 2017.

[121] Ruihui Li, Xianzhi Li, Ke-Hei Hui, and Chi-Wing Fu. SP-GAN: Sphere-guided 3D shape generation and manipulation. *ACM Transactions on Graphics (TOG)*, 40(4), 2021.

[122] Wen-Ding Li and Kevin Ellis. Is programming by example solved by LLMs? In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[123] Wen-Ding Li, Keya Hu, Carter Larsen, Yuqing Wu, Simon Alford, Caleb Woo, Spencer M. Dunn, Hao Tang, Wei-Long Zheng, Yewen Pu, and Kevin Ellis. Combining induction and transduction for abstract reasoning. In *The Thirteenth International Conference on Learning Representations*, 2025.

[124] Chen Liang, Jonathan Berant, Quoc Le, Kenneth D Forbus, and Ni Lao. Neural symbolic machines: Learning semantic parsers on freebase with weak supervision. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 23–33, 2017.

[125] Chen-Hsuan Lin, Jun Gao, Luming Tang, Towaki Takikawa, Xiaohui Zeng, Xun Huang, Karsten Kreis, Sanja Fidler, Ming-Yu Liu, and Tsung-Yi Lin. Magic3d: High-resolution text-to-3d content creation. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 300–309, 2023.

[126] Gabrielle Littlefair, Niladri Shekhar Dutt, and Niloy J Mitra. Flairgpt: Repurposing llms for interior designs. *arXiv preprint arXiv:2501.04648*, 2025.

[127] Guan-Ting Liu, En-Pei Hu, Pu-Jen Cheng, Hung-Yi Lee, and Shao-Hua Sun. Hierarchical programmatic reinforcement learning via learning to compose programs. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 21672–21697. PMLR, 23–29 Jul 2023.

[128] Matthew Loper, Naureen Mahmood, Javier Romero, Gerard Pons-Moll, and Michael J. Black. SMPL: A skinned multi-person linear model. *ACM Trans. Graphics (Proc. SIGGRAPH Asia)*, 34(6):248:1–248:16, Oct. 2015.

[129] Andrew L. Maas, Awni Y. Hannum, and Andrew Y. Ng. Rectifier nonlinearities improve neural network acoustic models. In *International Conference on Machine Learning (ICML)*, 2013.

[130] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[131] Jiayuan Mao, Chuang Gan, Pushmeet Kohli, Joshua B. Tenenbaum, and Jiajun Wu. The neuro-symbolic concept learner: Interpreting scenes, words, and sentences from natural supervision. In *International Conference on Learning Representations*, 2019.

[132] Andelo Martinovic and Luc Van Gool. Bayesian grammar learning for inverse procedural modeling. In *Proceedings of the 2013 IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '13, page 201–208, USA, 2013. IEEE Computer Society.

[133] A. Martinovic and L. Van Gool. Bayesian Grammar Learning for Inverse Procedural Modeling. In *CVPR*, 2013.

[134] Massive Software. Massive Software. `https://www.massivesoftware.com/`. Accessed: 2022-09-26.

[135] David McClosky, Eugene Charniak, and Mark Johnson. Effective self-training for parsing. In *Proceedings of the Human Language Technology Conference of the NAACL, Main Conference*, pages 152–159, New York City, USA, June 2006. Association for Computational Linguistics.

[136] Mateusz Michalkiewicz, Jhony K. Pontes, Dominic Jack, Mahsa Baktashmotlagh, and Anders P. Eriksson. Deep level sets: Implicit surface representations for 3D shape inference. *CoRR*, abs/1901.06802, 2019.

[137] Elie Michel and Tamy Boubekeur. Dag amendment for inverse control of parametric shapes. *ACM Transactions on Graphics*, 40(4):173:1–173:14, 2021.

[138] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. NeRF: Representing scenes as neural radiance fields for view synthesis. In *European Conference on Computer Vision (ECCV)*, 2020.

[139] Niloy Mitra, Michael Wand, Hao (Richard) Zhang, Daniel Cohen-Or, Vladimir Kim, and Qi-Xing Huang. Structure-aware shape processing. In *SIGGRAPH Asia 2013 Courses*, 2013.

[140] Kaichun Mo, Paul Guerrero, Li Yi, Hao Su, Peter Wonka, Niloy Mitra, and Leonidas Guibas. StructureNet: Hierarchical graph networks for 3D shape generation. In *Annual Conference on Computer Graphics and Interactive Techniques Asia (SIGGRAPH Asia)*, 2019.

[141] Kaichun Mo, Shilin Zhu, Angel X. Chang, Li Yi, Subarna Tripathi, Leonidas J. Guibas, and Hao Su. PartNet: A large-scale benchmark for fine-grained and hierarchical part-level 3D object understanding. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[142] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. In *Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, 2006.

[143] Gregory Murphy. *The big book of concepts*. MIT press, 2004.

[144] Chandrakana Nandi, James R Wilcox, Pavel Panchekha, Taylor Blau, Dan Grossman, and Zachary Tatlock. Functional programming for compiling and decompiling computer-aided design. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2018.

[145] Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. Synthesizing structured cad models with equality saturation and inverse transformations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 31–44, New York, NY, USA, 2020. Association for Computing Machinery.

[146] Charlie Nash, Yaroslav Ganin, SM Ali Eslami, and Peter Battaglia. Polygen: An autoregressive generative model of 3d meshes. In *International conference on machine learning*, pages 7220–7229. PMLR, 2020.

[147] Radford M. Neal and Geoffrey E. Hinton. A new view of the em algorithm that justifies incremental and other variants. In *Learning in Graphical Models*, pages 355–368. Kluwer Academic Publishers, 1993.

[148] Ansong Ni, Miltiadis Allamanis, Arman Cohan, Yinlin Deng, Kensen Shi, Charles Sutton, and Pengcheng Yin. Next: Teaching large language models to reason about code execution, 2024.

[149] Gen Nishida, Adrien Bousseau, and Daniel G. Aliaga. Procedural modeling of a building from a single image. *Computer Graphics Forum (Proceedings of the Eurographics conference)*, 2018.

[150] Gen Nishida, Ignacio Garcia-Dorado, Daniel G. Aliaga, Bedrich Benes, and Adrien Bousseau. Interactive sketching of urban procedural models. *ACM Trans. Graph.*, 35(4), jul 2016.

[151] Maxwell Nye, Yewen Pu, Matthew Bowers, Jacob Andreas, Joshua B. Tenenbaum, and Armando Solar-Lezama. Representing partial programs with blended abstract semantics. In *International Conference on Learning Representations*, 2021.

[152] Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. Is self-repair a silver bullet for code generation? In *International Conference on Learning Representations (ICLR)*, 2024.

[153] Wamiq Reyaz Para, Shariq Farooq Bhat, Paul Guerrero, Tom Kelly, Niloy Mitra, Leonidas Guibas, and Peter Wonka. SketchGen: Generating constrained CAD sketches. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

[154] Yoav I. H. Parish and Pascal Müller. Procedural modeling of cities. In *Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, 2001.

[155] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-Symbolic Program Synthesis. In *International Conference on Learning Representations (ICLR)*, 2017.

[156] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. *International Conference on Learning Representations (ICLR)*, 2017.

[157] Jeong Joon Park, Peter Florence, Julian Straub, Richard Newcombe, and Steven Lovegrove. Deepsdf: Learning continuous signed distance functions for shape representation. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[158] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[159] Ofek Pearl, Itai Lang, Yuhua Hu, Raymond A. Yeh, and Rana Hanocka. Geocode: Interpretable shape programs, 2022.

[160] Przemyslaw Prusinkiewicz, Mark James, and Radomír Měch. Synthetic topiary. In *Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pages 351–358, 1994.

[161] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag, Berlin, Heidelberg, 1996.

[162] Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. PointNet: deep learning on point sets for 3D classification and segmentation. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[163] Charles Ruizhongtai Qi, Li Yi, Hao Su, and Leonidas J Guibas. PointNet++: Deep hierarchical feature learning on point sets in a metric space. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[164] Zeju Qiu, Weiyang Liu, Haiwen Feng, Zhen Liu, Tim Z Xiao, Katherine M Collins, Joshua B Tenenbaum, Adrian Weller, Michael J Black, and Bernhard Schölkopf. Can large language models understand symbolic graphics programs? *arXiv preprint arXiv:2408.08313*, 2024.

[165] Inigo Quilez and Pol Jeremias. Shadertoy. *Retrieved March*, 27:2017, 2017.

[166] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *CoRR*, abs/1910.10683, 2019.

[167] Alexander Raistrick, Lahav Lipson, Zeyu Ma, Lingjie Mei, Mingzhe Wang, Yiming Zuo, Karhan Kayan, Hongyu Wen, Beining Han, Yihan Wang, Alejandro Newell, Hei Law, Ankit Goyal, Kaiyu Yang, and Jia Deng. Infinite photorealistic worlds using procedural generation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12630–12641, 2023.

[168] Alexander Raistrick, Lingjie Mei, Karhan Kayan, David Yan, Yiming Zuo, Beining Han, Hongyu Wen, Meenal Parakh, Stamatis Alexandropoulos, Lahav Lipson, Zeyu Ma, and Jia Deng. Infinigen indoors: Photorealistic indoor scenes using procedural generation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 21783–21794, June 2024.

[169] Aditya Ramesh, Prafulla Dhariwal, Alex Nichol, Casey Chu, and Mark Chen. Hierarchical text-conditional image generation with CLIP latents. *arXiv preprint arXiv:2204.06125*, 2022.

[170] Pradyumna Reddy, Michael Gharbi, Michal Lukac, and Niloy J Mitra. Im2Vec: Synthesizing vector graphics without vector supervision. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 7342–7351, 2021.

[171] Pradyumna Reddy, Zhifei Zhang, Matthew Fisher, Hailin Jin, Zhaowen Wang, and Niloy J Mitra. A multi-implicit neural representation for fonts. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

[172] Machel Reid, Vincent J. Hellendoorn, and Graham Neubig. Diffuser: Discrete diffusion via edit-based reconstruction, 2022.

[173] Danilo Jimenez Rezende, Shakir Mohamed, Ivo Danihelka, Karol Gregor, and Daan Wierstra. One-shot generalization in deep generative models, 2016.

[174] Leo Sampaio Ferraz Ribeiro, Tu Bui, John Collomosse, and Moacir Ponti. Sketchformer: Transformer-based representation for sketched structure. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

[175] Stephan R Richter, Vibhav Vineet, Stefan Roth, and Vladlen Koltun. Playing for data: Ground truth from computer games. In *European Conference on Computer Vision (ECCV)*, pages 102–118. Springer, 2016.

[176] Daniel Ritchie, Paul Guerrero, R. Kenny Jones, Niloy J. Mitra, Adriana Schulz, Kar l D. D. Willis, and Jiajun Wu. Neurosymbolic Models for Computer Graphics. *Computer Graphics Forum*, 2023.

[177] Daniel Ritchie, Sarah Jobalia, and Anna Thomas. Example-based authoring of procedural modeling programs with structural and continuous variability. In *EUROGRAPHICS*, 2018.

[178] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and BjÃ¶rn Ommer. High-resolution image synthesis with latent diffusion models. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022.

[179] Javier Romero, Dimitrios Tzionas, and Michael J. Black. Embodied hands: Modeling and capturing hands and bodies together. *ACM Transactions on Graphics, (Proc. SIGGRAPH Asia)*, 36(6), Nov. 2017.

[180] Oleh Rybkin, Kostas Daniilidis, and Sergey Levine. Simple and effective vae training with calibrated decoders, 2020.

[181] Chitwan Saharia, William Chan, Saurabh Saxena, Lala Li, Jay Whang, Emily Denton, Seyed Kamyar Seyed Ghasemipour, Burcu Karagol Ayan, S Sara Mahdavi, Rapha Gontijo Lopes, et al. Photorealistic text-to-image diffusion models with deep language understanding. *arXiv preprint arXiv:2205.11487*, 2022.

[182] Manolis Savva, Abhishek Kadian, Oleksandr Maksymets, Yili Zhao, Erik Wijmans, Bhavana Jain, Julian Straub, Jia Liu, Vladlen Koltun, Jitendra Malik, Devi Parikh, and Dhruv Batra. Habitat: A Platform for Embodied AI Research. In *The IEEE International Conference on Computer Vision (ICCV)*, 2019.

[183] H. Scudder. Probability of error of some adaptive pattern-recognition machines. *IEEE Transactions on Information Theory*, 11(3):363–371, 1965.

[184] Ari Seff, Wenda Zhou, Nick Richardson, and Ryan P. Adams. Vitruvion: A generative model of parametric CAD sketches. In *International Conference on Learning Representations (ICLR)*, 2022.

[185] Etai Sella, Gal Fiebelman, Noam Atia, and Hadar Averbuch-Elor. Spice· e: Structural priors in 3d diffusion using cross-entity attention. In *ACM SIGGRAPH 2024 Conference Papers*, pages 1–11, 2024.

[186] Pratheba Selvaraju, Mohamed Nabail, Marios Loizou, Maria Maslioukova, Melinos Averkiou, Andreas Andreou, Siddhartha Chaudhuri, and Evangelos Kalogerakis. Buildingnet: Learning to label 3d buildings. In *IEEE/CVF International Conference on Computer Vision (ICCV)*, 2021.

[187] Gopal Sharma, Rishabh Goyal, Difan Liu, Evangelos Kalogerakis, and Subhransu Maji. CSGNet: Neural Shape Parser for Constructive Solid Geometry. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[188] Gopal Sharma, Rishabh Goyal, Difan Liu, Evangelos Kalogerakis, and Subhransu Maji. Neural shape parsers for constructive solid geometry. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2020.

[189] Alexander G Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob R. Gardner, Yiming Yang, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. Learning performance-improving code edits. In *The Twelfth International Conference on Learning Representations*, 2024.

[190] Jake Snell, Kevin Swersky, and Richard Zemel. Prototypical networks for few-shot learning. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[191] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 404–415, 2006.

[192] Olga Sorkine and Marc Alexa. As-rigid-as-possible surface modeling. In *Proceedings of EUROGRAPHICS/ACM SIGGRAPH Symposium on Geometry Processing*, pages 109–116, 2007.

[193] Ondrej Stava, Bedrich Benes, Radomír Mech, Daniel G. Aliaga, and Peter Kristof. Inverse procedural modeling by automatic generation of l-systems. *Computer Graphics Forum (CGF)*, 29:665–674, 2010.

[194] Mark Steedman, Miles Osborne, Anoop Sarkar, Stephen Clark, Rebecca Hwa, Julia Hockenmaier, Paul Ruhlen, Steven Baker, and Jeremiah Crim. Bootstrapping statistical parsers from small datasets. In *10th Conference of the European Chapter of the Association for Computational Linguistics*, Budapest, Hungary, 2003. Association for Computational Linguistics.

[195] Sanjay Subramanian, Medhini Narasimhan, Kushal Khangaonkar, Kevin Yang, Arsha Nagrani, Cordelia Schmid, Andy Zeng, Trevor Darrell, and Dan Klein. Modular visual question answering via code generation. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 747–761, Toronto, Canada, July 2023. Association for Computational Linguistics.

[196] Michael Sun, Alston Lo, Minghao Guo, Jie Chen, Connor W. Coley, and Wojciech Matusik. Procedural synthesis of synthesizable molecules. In *The Thirteenth International Conference on Learning Representations*, 2025.

[197] Shao-Hua Sun, Hyeonwoo Noh, Sriram Somasundaram, and Joseph Lim. Neural program synthesis from diverse demonstration videos. In *Proceedings of the 35th International Conference on Machine Learning*, 2018.

[198] Minhyuk Sung, Hao Su, Vladimir G. Kim, Siddhartha Chaudhuri, and Leonidas Guibas. ComplementMe: Weakly-supervised component suggestions for 3D modeling. *ACM Transactions on Graphics (Proc. of SIGGRAPH Asia)*, 2017.

[199] Dídac Surís, Sachit Menon, and Carl Vondrick. Vipergpt: Visual inference via python execution for reasoning. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 11888–11898, October 2023.

[200] Jerry O. Talton, Lingfeng Yang, Ranjitha Kumar, Maxine Lim, Noah D. Goodman, and Radomír Mech. Learning design patterns with Bayesian grammar induction. In *ACM Symposium on User Interface Software and Technology (UIST)*, 2012.

[201] Hao Tang and Kevin Ellis. From perception to programs: regularize, overparameterize, and amortize. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, MAPS 2022, page 30–39, New York, NY, USA, 2022. Association for Computing Machinery.

[202] Jiapeng Tang, Yinyu Nie, Lev Markhasin, Angela Dai, Justus Thies, and Matthias Nießner. Diffuscene: Denoising diffusion models for generative indoor scene synthesis. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2024.

[203] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: a new approach to optimization. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 264–276, New York, NY, USA, 2009. ACM.

[204] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality Saturation: A New Approach to Optimization. *Logical Methods in Computer Science*, Volume 7, Issue 1, Mar. 2011.

[205] Josh Tenenbaum. Building machines that learn and think like people. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '18, page 5, Richland, SC, 2018. International Foundation for Autonomous Agents and Multiagent Systems.

[206] Yonglong Tian, Andrew Luo, Xingyuan Sun, Kevin Ellis, William T. Freeman, Joshua B. Tenenbaum, and Jiajun Wu. Learning to Infer and Execute 3D Shape Programs. In *International Conference on Learning Representations (ICLR)*, 2019.

[207] Emina Torlak and Rastislav Bodik. Growing solver-aided languages with rosette. In *ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (SIGPLAN)*, pages 135–152, 2013.

[208] Aaron van den Oord, Oriol Vinyals, and koray kavukcuoglu. Neural discrete representation learning. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[209] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[210] Yael Vinker, Ehsan Pajouheshgar, Jessica Y. Bo, Roman Christian Bachmann, Amit Haim Bermano, Daniel Cohen-Or, Amir Zamir, and Ariel Shamir. CLIPasso: Semantically-aware object sketching. In *Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, 2022.

[211] Oriol Vinyals, Charles Blundell, Timothy Lillicrap, koray kavukcuoglu, and Daan Wierstra. Matching networks for one shot learning. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016.

[212] Kai Wang, Yu-An Lin, Ben Weissmann, Manolis Savva, Angel X. Chang, and Daniel Ritchie. Planit: planning and instantiating indoor scenes with relation graph and spatial prior networks. *ACM Trans. Graph.*, 38(4), July 2019.

[213] Kai Wang, Manolis Savva, Angel X. Chang, and Daniel Ritchie. Deep convolutional priors for indoor scene synthesis. *ACM Transactions on Graphics (TOG)*, 37(4), 2018.

[214] Kai Wang, Xianghao Xu, Leon Lei, Selena Ling, Natalie Lindsay, Angel X Chang, Manolis Savva, and Daniel Ritchie. Roominoes: Generating novel 3d floor plans from existing 3d rooms. In *Computer Graphics Forum*, volume 40, pages 57–69. Wiley Online Library, 2021.

[215] Qiuhong Anna Wei, Sijie Ding, Jeong Joon Park, Rahul Sajnani, Adrien Poulenard, Srinath Sridhar, and Leonidas Guibas. Lego-net: Learning regular rearrangements of objects in rooms, 2023.

[216] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8, 1992.

[217] Karl D. D. Willis, Yewen Pu, Jieliang Luo, Hang Chu, Tao Du, Joseph G. Lambourne, Armando Solar-Lezama, and Wojciech Matusik. Fusion 360 Gallery: A dataset and environment for programmatic CAD construction from human design sequences. *ACM Transactions on Graphics (TOG)*, 40(4), 2021.

[218] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.*, 5(POPL), Jan. 2021.

[219] Chenming Wu, Haisen Zhao, Chandrakana Nandi, Jeffrey I. Lipton, Zachary Tatlock, and Adriana Schulz. Carpentry compiler. *ACM Trans. Graph.*, 38(6), Nov. 2019.

[220] Jiajun Wu, Joshua B Tenenbaum, and Pushmeet Kohli. Neural scene de-rendering. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[221] Jiajun Wu, Chengkai Zhang, Tianfan Xue, William T Freeman, and Joshua B Tenenbaum. Learning a probabilistic latent space of object shapes via 3D generative-adversarial modeling. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 82–90, 2016.

[222] Rundi Wu, Chang Xiao, and Changxi Zheng. DeepCAD: A deep generative network for computer-aided design models. In *IEEE International Conference on Computer Vision (ICCV)*, pages 6772–6782, 2021.

[223] Zhirong Wu, Shuran Song, Aditya Khosla, Fisher Yu, Linguang Zhang, Xiaoou Tang, and Jianxiong Xiao. 3d shapenets: A deep representation for volumetric shapes. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1912–1920, 2015.

[224] Fei Xia, Amir R. Zamir, Zhi-Yang He, Alexander Sax, Jitendra Malik, and Silvio Savarese. Gibson env: real-world perception for embodied agents. In *Computer Vision and Pattern Recognition (CVPR), 2018 IEEE Conference on*. IEEE, 2018.

[225] Haozhe Xie, Zhaoxi Chen, Fangzhou Hong, and Ziwei Liu. Citydreamer: Compositional generative model of unbounded 3d cities. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 9666–9675, 2024.

[226] Jiacong Xu, Yi Zhang, Jiawei Peng, Wufei Ma, Artur Jesslen, Pengliang Ji, Qixin Hu, Jiehua Zhang, Qihao Liu, Jiahao Wang, et al. Animal3d: A comprehensive dataset of 3d animal pose and shape. *arXiv preprint arXiv:2308.11737*, 2023.

[227] Ken Xu and Damian Campeanu. Houdini engine: Evolution towards a procedural pipeline. In *Proceedings of the fourth symposium on digital production*, pages 13–18, 2014.

[228] Xianghao Xu, Wenzhe Peng, Chin-Yi Cheng, Karl D. D. Willis, and Daniel Ritchie. Inferring CAD modeling sequences using zone graphs. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2021.

[229] Xiang Xu, Karl DD Willis, Joseph G Lambourne, Chin-Yi Cheng, Pradeep Kumar Jayaraman, and Yasutaka Furukawa. SkexGen: Autoregressive generation of CAD construction sequences with disentangled codebooks. In *International Conference on Machine Learning (ICML)*, 2022.

[230] Kaizhi Yang and Xuejin Chen. Unsupervised learning for cuboid shape abstraction via joint segmentation from point clouds. *ACM Trans. Graph.*, 40(4), jul 2021.

[231] Yue Yang, Fan-Yun Sun, Luca Weihs, Eli VanderBilt, Alvaro Herrasti, Winson Han, Jiajun Wu, Nick Haber, Ranjay Krishna, Lingjie Liu, et al. Holodeck: Language guided generation of 3d embodied ai environments. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 16227–16237, 2024.

[232] David Yarowsky. Unsupervised word sense disambiguation rivaling supervised methods. In *33rd Annual Meeting of the Association for Computational Linguistics*, pages 189–196, Cambridge, Massachusetts, USA, June 1995. Association for Computational Linguistics.

[233] Kexin Yi, Jiajun Wu, Chuang Gan, Antonio Torralba, Pushmeet Kohli, and Joshua B. Tenenbaum. Neural-symbolic vqa: Disentangling reasoning from vision and language understanding. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 1039–1050, Red Hook, NY, USA, 2018. Curran Associates Inc.

[234] Wang Yifan, Noam Aigerman, Vladimir G Kim, Siddhartha Chaudhuri, and Olga Sorkine-Hornung. Neural cages for detail-preserving 3d deformations. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 75–83, 2020.

[235] Fenggen Yu, Qimin Chen, Maham Tanveer, Ali Mahdavi Amiri, and Hao Zhang. D^2CSG: Unsupervised learning of compact csg trees with dual complements and dropouts. *Advances in Neural Information Processing Systems*, 36, 2024.

[236] Fenggen Yu, Zhiqin Chen, Manyi Li, Aditya Sanghi, Hooman Shayani, Ali Mahdavi-Amiri, and Hao Zhang. CAPRI-Net: Learning compact CAD shapes with adaptive primitive assembly. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 11768–11778, 2022.

[237] Alan Yuille and Daniel Kersten. Vision as bayesian inference: analysis by synthesis? *Trends in cognitive sciences*, 10(7):301–308, 2006.

[238] Eric Zhan, Stephan Zheng, Yisong Yue, Long Sha, and Patrick Lucey. Generating multi-agent trajectories using programmatic weak supervision. In *International Conference on Learning Representations*, 2019.

[239] Lvmin Zhang, Anyi Rao, and Maneesh Agrawala. Adding conditional control to text-to-image diffusion models. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 3836–3847, October 2023.

[240] Yunzhi Zhang, Zizhang Li, Matt Zhou, Shangzhe Wu, and Jiajun Wu. The scene language: Representing scenes with programs, words, and embeddings. *arXiv preprint arXiv:2410.16770*, 2024.

[241] Yinda Zhang, Shuran Song, Ersin Yumer, Manolis Savva, Joon-Young Lee, Hailin Jin, and Thomas Funkhouser. Physically-based rendering for indoor scene understanding using convolutional neural networks. *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[242] Guangcong Zheng, Xianpan Zhou, Xuewei Li, Zhongang Qi, Ying Shan, and Xi Li. Layoutdiffusion: Controllable diffusion model for layout-to-image generation, 2024.

[243] Chenghui Zhou, Chun-Liang Li, and Barnabas Poczos. Unsupervised program synthesis for images using tree-structured lstm, 2020.

[244] Chenyang Zhu, Kai Xu, Siddhartha Chaudhuri, Renjiao Yi, and Hao Zhang. SCORES: Shape composition with recursive substructure priors. *ACM Transactions on Graphics (TOG)*, 37(6):211:1–211:14, 2018.

[245] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *IEEE International Conference on Computer Vision (ICCV)*, 2017.

[246] Barret Zoph, Golnaz Ghiasi, Tsung-Yi Lin, Yin Cui, Hanxiao Liu, Ekin D. Cubuk, and Quoc V. Le. Rethinking pre-training and self-training. In *Advances in Neural Information Processing Systems (NeurIPS)*, NIPS'20, Red Hook, NY, USA, 2020. Curran Associates Inc.

[247] Chuhang Zou, Ersin Yumer, Jimei Yang, Duygu Ceylan, and Derek Hoiem. 3D-PRNN: Generating Shape Primitives with Recurrent Neural Networks. In *IEEE International Conference on Computer Vision (ICCV)*, 2017.