ShapeCoder: Discovering Abstractions for Visual Programs from Unstructured Primitives

R. KENNY JONES, Brown University, USA
PAUL GUERRERO, Adobe Research, United Kingdom
NILOY J. MITRA, University College London and Adobe Research, United Kingdom
DANIEL RITCHIE, Brown University, USA

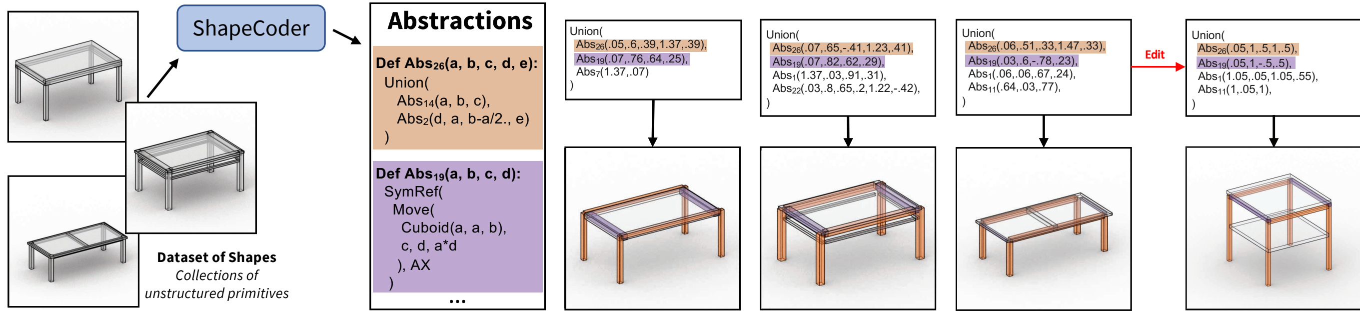

Fig. 1. ShapeCoder automatically discovers abstraction functions, and infers visual programs that use these abstractions, to compactly explain an input dataset of shapes represented with unstructured primitives. For example, the orange abstraction uses only five parameters to encode a distribution of 4-legged table bases with adjoining horizontal support bars.

We introduce ShapeCoder, the first system capable of taking a dataset of shapes, represented with unstructured primitives, and jointly discovering (i) useful *abstraction* functions and (ii) programs that use these abstractions to explain the input shapes. The discovered abstractions capture common patterns (both structural and parametric) across a dataset, so that programs rewritten with these abstractions are more compact, and suppress spurious degrees of freedom. ShapeCoder improves upon previous abstraction discovery methods, finding better abstractions, for more complex inputs, under less stringent input assumptions. This is principally made possible by two methodological advancements: (a) a shape-to-program recognition network that learns to solve sub-problems and (b) the use of e-graphs, augmented with a conditional rewrite scheme, to determine when abstractions with complex parametric expressions can be applied, in a tractable manner. We evaluate ShapeCoder on multiple datasets of 3D shapes, where primitive decompositions are either parsed from manual annotations or produced by an unsupervised cuboid abstraction method. In all domains, ShapeCoder discovers a library of abstractions that captures high-level relationships, removes extraneous degrees of freedom, and achieves better dataset compression compared with alternative approaches. Finally, we investigate how programs rewritten to use discovered abstractions prove useful for downstream tasks.

CCS Concepts: • **Computing methodologies → Shape modeling**.

Authors' addresses: R. Kenny Jones, russell_jones@brown.edu, Brown University, USA; Paul Guerrero, guerrero@adobe.com, Adobe Research, United Kingdom; Niloy J. Mitra, n.mitra@cs.ucl.ac.uk, University College London and Adobe Research, United Kingdom; Daniel Ritchie, daniel_ritchie@brown.edu, Brown University, USA.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
0730-0301/2023/8-ART49 $15.00
https://doi.org/10.1145/3592416

Additional Key Words and Phrases: procedural modeling, visual programs, shape analysis, shape abstraction, library learning, e-graph

ACM Reference Format:
R. Kenny Jones, Paul Guerrero, Niloy J. Mitra, and Daniel Ritchie. 2023. ShapeCoder: Discovering Abstractions for Visual Programs from Unstructured Primitives. *ACM Trans. Graph.* 42, 4, Article 49 (August 2023), 17 pages. https://doi.org/10.1145/3592416

1 INTRODUCTION

Procedural models are an attractive representation for visual data. Visual programs, expressions that produce visual outputs when executed, offer many advantages over alternative representations, such as compactness, interpretability, and editability [Ritchie et al. 2023]. There is a wide-range of domains that fall under the purview of visual programs, and procedural workflows are becoming increasingly common in modeling software [Freiknecht and Effelsberg 2017; SideFx 2014]. A trait that many visual programming domains share, is that their programs often contain both structural diversity and variables that are constrained by complex parametric relationships.

Typically, visual programs are written in domain-specific languages (DSLs) targeted for specific visual applications. 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.

The idea of automatic abstraction discovery has been investigated for general programming domains [Ellis et al. 2021]. Some approaches have also been designed for visual domains [Jones et al. 2021], where programs contain complex parametric relationships that complicate this task. While previous methods have made headway towards solving this task, none offer a complete solution. Two central limitations holding back the applicability of such methods are that they are either designed without visual programs in mind (so fail to find meaningful parametric relationships) or rely on heavy input assumptions that are hard to meet.

In this paper, 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 1). Our approach is inspired by, and improves upon, previous abstraction discovery approaches.

Like DreamCoder [Ellis et al. 2021], a library learning method for general programming domains, we employ 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), which is a limitation of some prior work [Ellis et al. 2021].

Using programs from the wake phase, a proposal phase suggests candidate abstractions, which an integration phase reasons over to find improved library versions. These phases draw inspiration from ShapeMOD [Jones et al. 2021], an abstraction discovery method designed for visual data. ShapeMOD's integration stage relies on enumerative search over a limited, curated subset of possible program line-orderings. ShapeMOD assumes access to a dataset containing hierarchical part annotations, to keep the number of line-orderings small, but this approach scales poorly. To overcome this limitation, we design an integration stage that makes use of e-graphs [Tate et al. 2011; Willsey et al. 2021], a data structure that represents large sets of equivalent programs in a efficient manner through rewrite operations. This enables us to search over a huge space of refactored programs, and allows ShapeCoder to integrate more complex abstractions that better match the input distribution. Critical to making e-graph expansion and extraction tractable for our problem setting, we implement a novel conditional rewrite scheme that lazily evaluates parametric relationships before applying abstraction rewrites, avoiding e-graph size blowup.

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 [Chang et al. 2015]. In this setting, we observe ShapeCoder still discovers interesting, high-level abstractions, even over noisy, inconsistent primitive decompositions.

In summary, our contributions are:

(i) ShapeCoder, a method that learns to infer visual programs that use automatically-discovered abstractions to explain and simplify a collection of shapes represented with unstructured primitives;
(ii) a recognition network capable of inferring visual programs that use discovered abstractions, even without access to a task curriculum; and
(iii) a refactor operation that augments e-graphs with a conditional rewriting scheme to identify matches on complex parametric relationships in a tractable manner.

Our code is available at github.com/rkjones4/ShapeCoder/ .

2 RELATED WORK

Our method is related to a host of prior work that relate in different ways to visual programs: abstraction discovery for non-visual programs, visual program induction (VPI), and visual program generation. We first provide an overview of these areas, and then end this section with a detailed discussion of the two most related works, DreamCoder [Ellis et al. 2021] and ShapeMOD [Jones et al. 2021].

Program abstraction. 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 [Martinovic and Van Gool 2013] or more general grammar types [Hwang et al. 2011; Ritchie et al. 2018; Talton et al. 2012]. Abstraction discovery for more general sets of programs has been explored in the Exploration-Compression algorithm [Dechter et al. 2013] and more recently in DreamCoder [Ellis et al. 2021]. Similar to our approach, these methods find abstractions in multiple rounds that alternate between program induction, where programs for a given set of problems are discovered, and abstraction discovery, where discovered programs are examined to find recurring patterns. We discuss DreamCoder in more detail at the end of this section. Recently, improvements have been proposed for the abstraction step of DreamCoder's algorithm [Bowers et al. 2023]. Most relevant to our work, Babble [Cao et al. 2023], also uses e-graphs [Tate et al. 2009] to identify abstraction applications, but

has no special mechanism for handling rewriters with complex parametric expressions, which allows ShapeCoder to scale to complex 3D visual domains. Babble employs anti-unification over e-graphs to propose abstractions, and it would be interesting to consider how this scheme could be extended to work with our e-graph formulation, where we explicitly avoid expanding parametric operations into e-nodes. A related problem is to discover common patterns in a *single* program, as opposed to a set of programs. This has been explored for L-Systems [Guo et al. 2020] or CAD programs [Nandi et al. 2020; Wu et al. 2019].

Visual program induction. Inferring a visual program that reconstructs a given target is a long-standing problem in computer graphics. Early approaches focused on vegetation [Stava et al. 2014; Xu and Mould 2015], façades [Müller et al. 2007; Wu et al. 2014], and urban landscapes [Demir et al. 2014; Vanegas et al. 2012]. We refer to [Aliaga et al. 2016] for a more complete overview.

In more recent work, neural components are typically employed in key parts of the method. Some of these approaches require an existing program structure to be available and only estimate the parameters of the program to match a given target 3D shape [Michel and Boubekeur 2021; Pearl et al. 2022] or 2D material [Shi et al. 2020]. Other approaches aim to jointly infer both program parameters and program structure. Visual programming domains range from commonly used program types, such as CSG construction sequences [Du et al. 2018; Kania et al. 2020; Ren et al. 2021, 2022; Sharma et al. 2018; Yu et al. 2022], CAD Modelling Sequences [Ganin et al. 2021; Li et al. 2020, 2022; Seff et al. 2022; Xu et al. 2021], SVG shapes [Reddy et al. 2021a,b], and L-Systems [Guo et al. 2020], to custom program domains, like primitive declarations with loops and conditionals in 2D [Ellis et al. 2018] and 3D [Tian et al. 2019], geometry instancing with linear transformations [Deng et al. 2022] and masked procedural noise models for materials [Hu et al. 2022]. A few methods also propose inference methods that apply to diverse types of programs [Ellis et al. 2019; Jones et al. 2022]. Most related to our program domain are ShapeAssembly [Jones et al. 2020] and ShapeMOD [Jones et al. 2021], which output programs that arrange cuboid primitives. All of these methods, excluding ShapeMOD, assume a DSL with a complete set of operators. We discuss ShapeMOD separately at the end of this section.

Visual program generation. Several deep generative models have been proposed to generate visual programs. MatFormer [Guerrero et al. 2022] generates node graphs for materials, several methods propose generative models for SVG images [Carlier et al. 2020; Reddy et al. 2021a], CAD sketches [Ganin et al. 2021; Para et al. 2021; Seff et al. 2022], and 3D CAD Modelling sequences [Li et al. 2022; Wu et al. 2021; Xu et al. 2022]. The ShapeAssembly [Jones et al. 2020] and ShapeMOD [Jones et al. 2021] methods mentioned above can also be used as generative models. Similar to methods for visual program induction, all of these methods, except for ShapeMOD, require a DSL with a full set of operators.

DreamCoder. This work proposes a system that jointly discovers abstractions and performs program induction over arbitrary functional programming languages [Ellis et al. 2021]. 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.

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 supplemental material.

ShapeMOD. In contrast to DreamCoder, ShapeMOD is a system designed for visual datasets, like shape programs. It has been shown to 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, where all possible valid line reorderings are known. In fact, as its intractable to reason over all line reorderings that would lead to the same semantic output, heuristics were employed to limit the orders to a very small set. Applying these heuristics required access to a hierarchical semantic segmentation, which allowed sub-parts to be treated as independent sub-programs. ShapeMOD's integration and proposal stages (analogous to DreamCoder's abstraction phase), relied on these limited program reorderings to both discover candidate macros, and identify when those macros could be applied.

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.

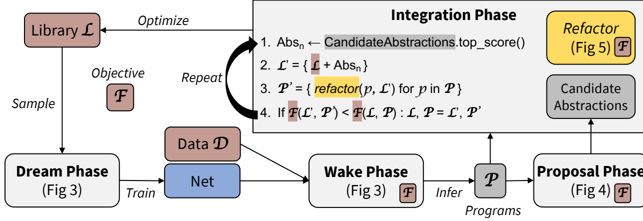

Fig. 2. **Overview.** ShapeCoder consumes an initial library $\mathcal{L}$, an objective $\mathcal{F}$, and a dataset of shapes $\mathcal{D}$ (brown boxes). Each round of the algorithm iterates through a series of phases that progressively add abstractions into $\mathcal{L}$ to improve $\mathcal{F}$. 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 $\mathcal{L}$.

3 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 $\mathcal{L}$, p, such that executing p would recreate d.

ShapeCoder's goal is to minimize $\mathcal{F}$ (Section 3.1), 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 2. The *dream* phase (Section 4.2) samples scenes from $\mathcal{L}$ to train a program recognition network. The *wake* phase (Section 4.3) uses this network to infer programs $\mathcal{P}$ that recreate shapes in $\mathcal{D}$. The *proposal* phase (Section 5.1) consumes $\mathcal{P}$ as input, and generates candidate abstraction functions. Finally, the *integration* phase (Section 5.2) 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 6) 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 A). Further implementation details are provided in Appendix B.

3.1 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 $p \in \mathcal{P}$ 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 $p \in \mathcal{P}$ against its corresponding shape, $d \in \mathcal{D}$. If $err(p, d)$ 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 $(0, \infty)$. 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 $\tau\,(p)$ expresses the number of tokens in p that have type τ, we can express ShapeCoder's objective as:

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

4 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 4.1), trained on randomly sampled programs from $\mathcal{L}$ (dream phase, Section 4.2), to infer $\mathcal{P}$ that minimize $\mathcal{F}$ (wake phase, Section 4.3).

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).

4.1 Recognition Network

We depict ShapeCoder's recognition network on the left side of Figure 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 [Vaswani et al. 2017] 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.

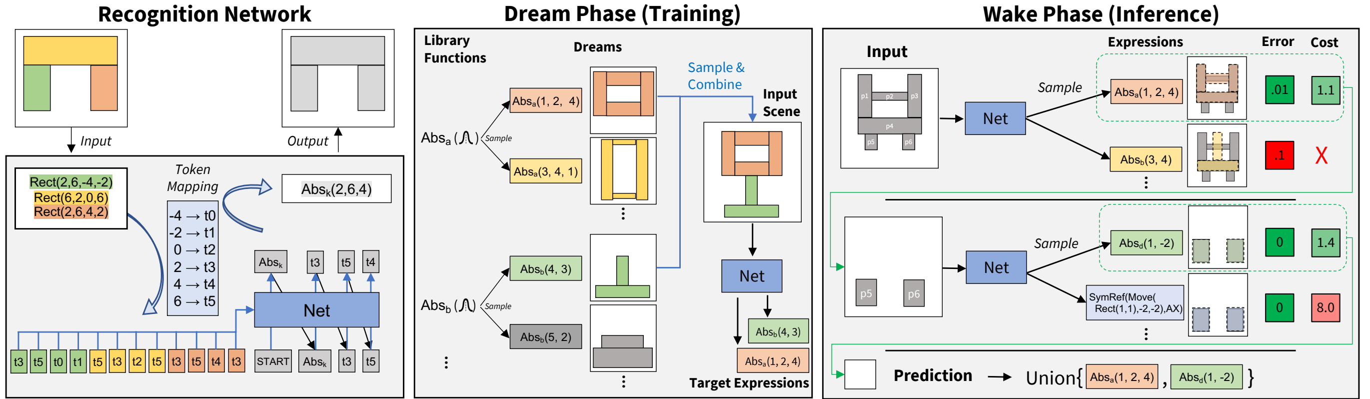

Fig. 3. **Dream and Wake Phases.** *(Left)* ShapeCoder's recognition network is a Transformer decoder that attends over tokenized input primitives and autoregressively predicts functions and parameterizations. *(Middle)* The dream phase trains the recognition network by sampling expressions from library 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.

4.2 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 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 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.

4.3 Wake Phase

The wake phase takes an input shape d and aims to infer a program p that minimizes $\mathcal{F}$ using the recognition network. We depict this process on the right side of Figure 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 p explaining d is formed by applying the combinator operation in $\mathcal{L}$ over each e^* (e.g. the `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*(p, p_r), where *combine* performs a greedy replacement search to optimize $\mathcal{F}$.

5 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 5.1) reasons over $\mathcal{P}$ to suggest candidate abstractions functions, used by the integration phase (Section 5.2) to find library variants that improve $\mathcal{F}$.

5.1 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

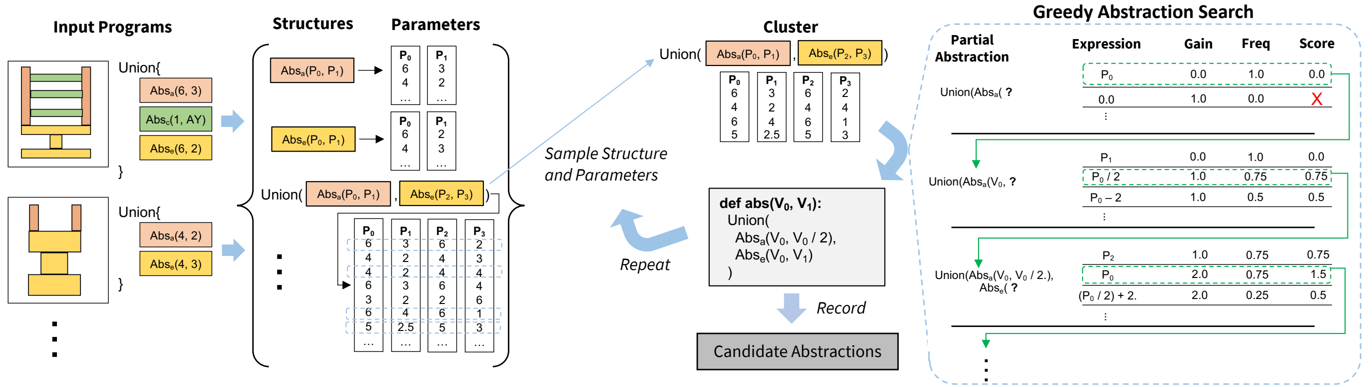

Fig. 4. **Proposal Phase.** The proposal phase consumes a collection of programs and outputs a set of candidate abstractions. First, possible structures and their parameterizations are recorded from the input programs. Then clusters are formed by sampling a structure and a subset of parameterizations. For each cluster, a greedy abstraction search generates a possible abstraction, which is recorded.

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 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 $p \in \mathcal{P}$ is found in the wake phase by combining expressions that solve sub-tasks, so p 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 $p \in \mathcal{P}$. 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 $p \in \mathcal{P}$ 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 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 p, whenever p could be rewritten with a, denoted as p_a. For instance, the proposed abstraction in Figure 4 would remove two float-typed parameters whenever it could be applied, corresponding with slots P_1 and P_3 in the cluster. Using the weighting from $\mathcal{F}$ (Section 3.1), we have:

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

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).

5.2 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 6) 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 $p \in \mathcal{P}$ 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.

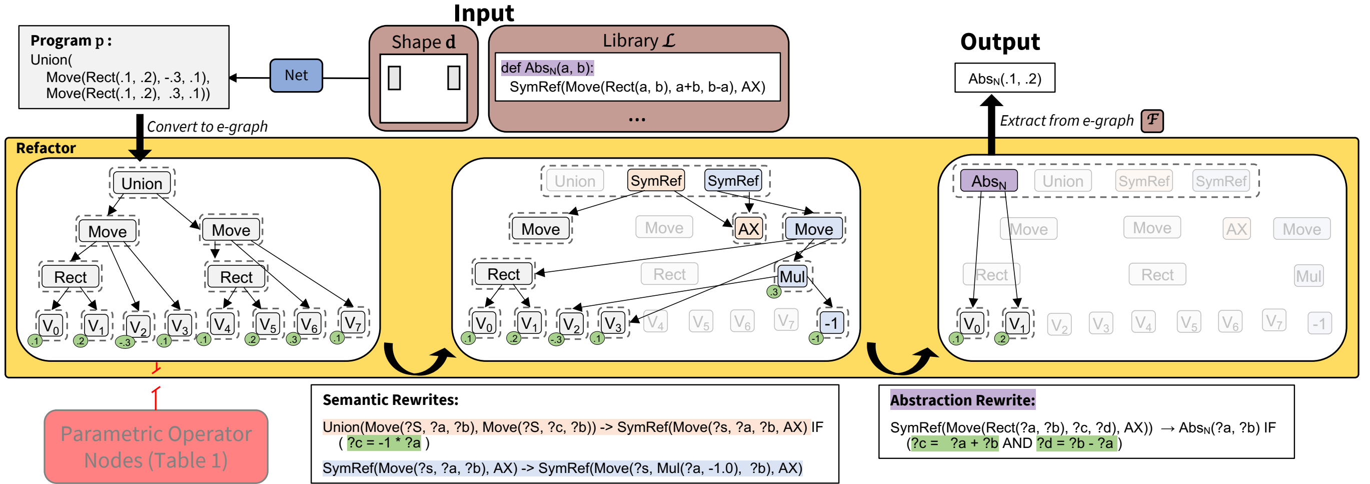

Fig. 5. **Refactor.** The refactor operation uses e-graphs to identify when abstractions can be applied. Input programs are converted into e-graphs, which are expanded with semantic and library-specific rewrites to uncover lower-cost equivalent expressions that can be extracted. We develop a conditional rewrite scheme that reasons over parametric relationships (green highlights) without adding excessive e-nodes for parametric operators (red box).

Evaluating a modified library $\mathcal{L}$' is expensive, as it requires running the *refactor* operation for every $p \in \mathcal{P}$, 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 $p \in \mathcal{P}$ 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.

6 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, p, and aims to find p^*, an equivalent program to p that minimizes $\mathcal{F}$. This is a hard search problem, which we make tractable through the use of e-graphs [Tate et al. 2011] 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 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 p from the wake phase. First, it converts p into an e-graph, as depicted in the left call-out of Figure 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 × ?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 p. 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.

Table 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

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 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 1).

7 RESULTS AND EVALUATION

We run ShapeCoder over distributions of visual shapes represented as collections of unstructured primitives. We describe these domains in Section 7.1. In Section 7.2, we compare how well the abstractions discovered by ShapeCoder improve the objective function compared to alternative approaches. Our main comparison is against ShapeMOD [Jones et al. 2021]. In the main text, we do not include comparisons against DreamCoder [Ellis et al. 2021], as we found it performed poorly on a toy grammar with parametric relationships (see supplemental). In Section 7.3, we analyze properties of the discovered abstractions and investigate their generality with a post hoc inference procedure. In Section 7.4, we run an ablation experiment to investigate the importance of various algorithm components. In Section 7.5 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 [Yang and Chen 2021]. Finally, in Section 7.6 we explore how ShapeCoder's discovered abstractions benefit downstream tasks.

7.1 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 [Mo et al. 2019], 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 space, and run experiments on shapes from the *Chair*, *Table*, and *Storage* categories of PartNet. We perform the cuboid simplification steps outlined by [Jones et al. 2020], so that our starting primitive set is the same as that used by ShapeMOD, except we remove all hierarchy and canonical ordering information.

Table 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

The DSL (Appendix A) 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`).

7.2 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 [Willsey et al. 2021]. 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 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 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 [Nandi et al. 2020]. 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.

7.3 Analysis of Discovered Abstractions

We visualize a subset of abstractions discovered by ShapeCoder when run over PartNet shapes in Figure 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.

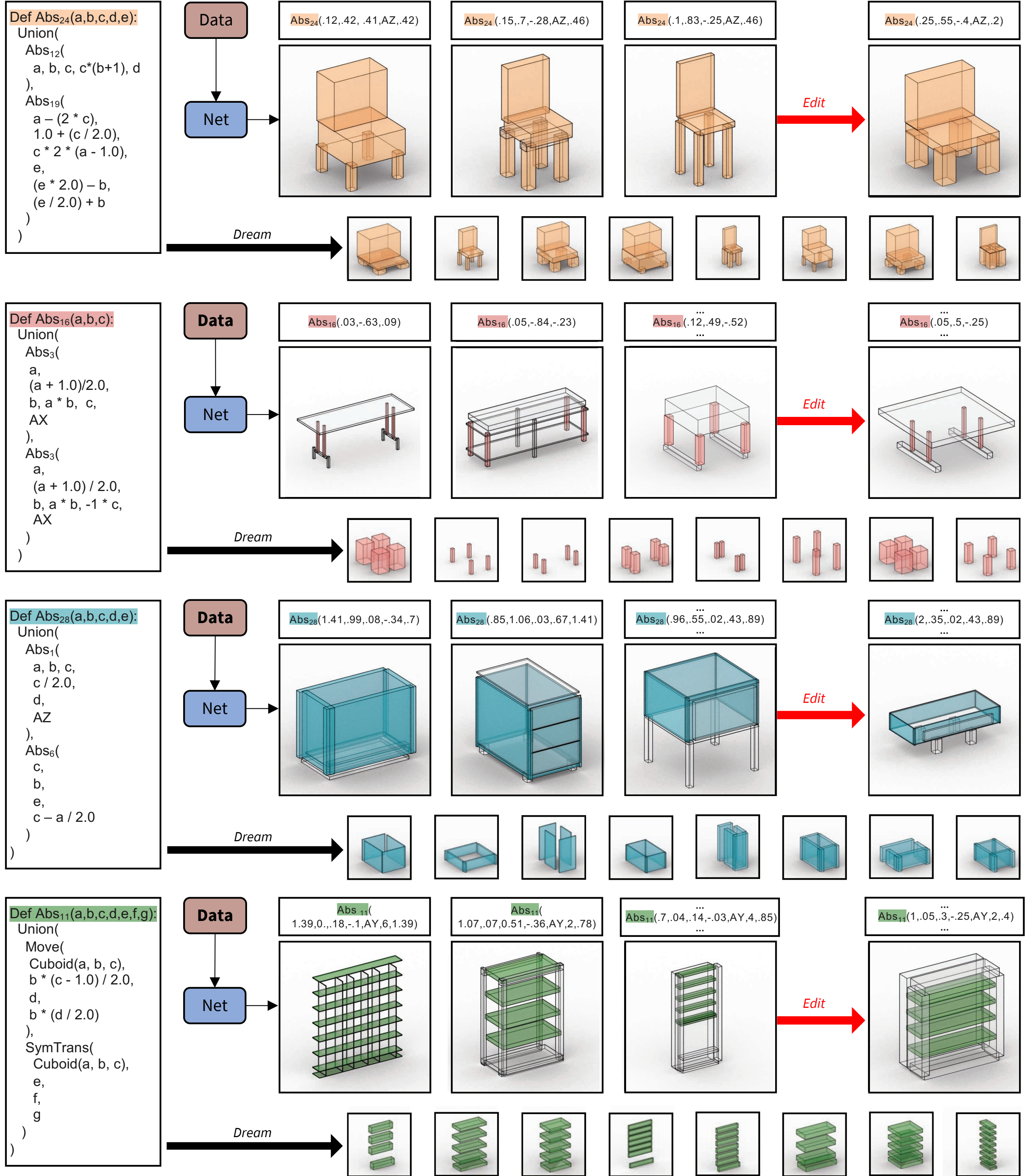

Fig. 6. Qualitative examples of discovered abstractions. We show one abstraction each for *Chair* and *Table*, and two abstractions for *Storage* furniture. The abstraction code is shown on the left, followed by three different usages of the abstraction in our shape dataset discovered by ShapeCoder. 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 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

Table 4. *(Left)* Ablating design decisions of ShapeCoder by tracking objective function improvement (see condition details in Section 7.4). 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**

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 3, for shapes from the *Chair* 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.

7.4 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.
- *No Conditional Rws:* We replace our conditional rewriting scheme with the 'naive' approach described in Section 6.
- *No Abs Preferences:* We remove the preference weighting ω, described in Section 3.1.

We report how these different variants perform in Table 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.

7.5 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 [Yang and Chen 2021]. 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 B.8

Similar to the experiments in Section 7.2, we construct a dataset of 400 shapes, with primitives produced by this unsupervised algorithm. We run ShapeCoder over a dataset of chairs sourced from

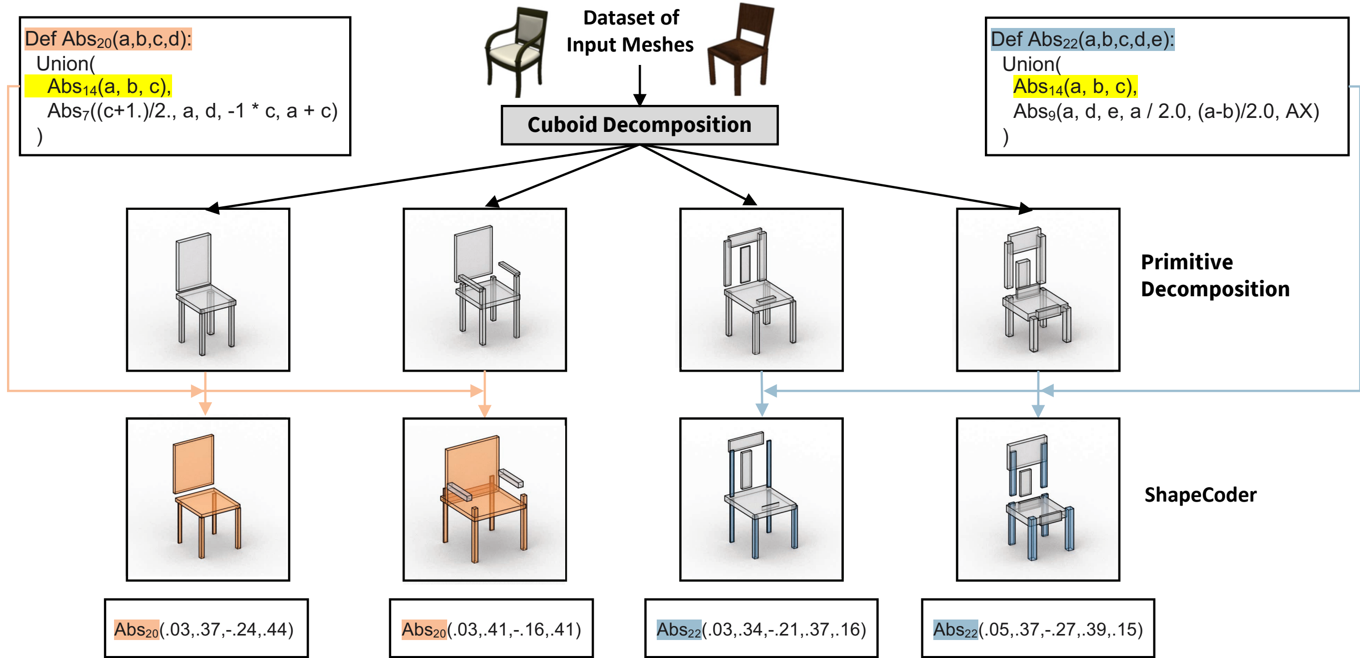

Fig. 7. We leverage an unsupervised primitive decomposition approach [Yang and Chen 2021] to run ShapeCoder over datasets of 3D meshes. Even on these noisy primitive decompositions, our method still finds high-level, useful abstractions that capture meaningful degrees of shape variation. Interestingly, 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.

ShapeNet [Chang et al. 2015] for three rounds and show results of some of the discovered abstractions in Figure 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 6). In fact, over the course of 3 rounds, ShapeCoder improves the objective function score by 61% (140 $\longrightarrow$ 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.

7.6 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 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 7.3) 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 B.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. 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%

Fig. 8. Sampled programs (top) from a generative model that writes programs containing abstractions, along with nearest neighbors (bottom).

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 CONCLUSION

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 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.

8.1 Limitations and Future Work

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. In the future, we want to explore 'conditional' or 'probabilistic' abstractions. For instance, a chair base abstraction could expand into either a pedestal base or a four-leg base, depending on either a discrete input parameter, or a given probability for each variation.

(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. One possibility is to amortize the integration stage with neural components: either by learning rewrites (e.g., using a reward structure in a reinforced learning setup), by learning which parts of the e-graph to expand, or by putting the burden of 'large' rewrites on a learned module, rather than the e-graph. However, training such modules in an unsupervised setup requires further research.

(iii) Bottom-up wake network. ShapeCoder's recognition network (used in the wake phase) solves sub-problems 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. Replacing this bottom-up network with a top-down network would be more challenging. Still, it would open up other possibilities, such as removing the need for the input data to be represented as collections of primitives.

Looking forward, we believe that ShapeCoder should be helpful for many other visual programming domains, beyond the 2D and 3D shape grammars we consider in this report. ShapeCoder requires the following domain attributes: *(a)* the language is functional, *(b)* it contains a combinator operation (e.g. `Union`), and *(c)* visual inputs can be decomposed into primitive types. In fact, properties *(b)* and *(c)* are only needed for the wake phase, so this requirement could be relaxed by using program inference networks that consume 'raw' visual data. Sketches, CSG, SVG, and even shader programs could make good matches for future explorations. For the first time, ShapeCoder provides the ability to perform program abstraction discovery directly on unstructured collections of primitives, reducing the burden of collecting, annotating, and grouping shape categories.

ACKNOWLEDGMENTS

We would like to thank Srinath Sridhar and the anonymous reviewers for their helpful suggestions. Renderings of shape programs were produced using the Blender Cycles renderer. This work was funded in parts by NSF award #1941808, a Brown University Presidential Fellowship, and an ERC grant (SmartGeometry). Daniel Ritchie is an advisor to Geopipe and owns equity in the company. Geopipe is a start-up that is developing 3D technology to build immersive virtual copies of the real world with applications in various fields, including games and architecture.

REFERENCES

Daniel G. Aliaga, Ilke Demir, Bedrich Benes, and Michael Wand. 2016. Inverse Procedural Modeling of 3D Models for Virtual Worlds. In *ACM SIGGRAPH 2016 Courses (SIGGRAPH '16)*. Article 16, 316 pages. https://doi.org/10.1145/2897826.2927323

Matthew Bowers, Theo X. Olausson, Lionel Wong, Gabriel Grand, Joshua B. Tenenbaum, Kevin Ellis, and Armando Solar-Lezama. 2023. Top-Down Synthesis for Library Learning. *Proc. ACM Program. Lang.* 7, POPL, Article 41 (jan 2023), 32 pages. https://doi.org/10.1145/3571234

David Cao, Rose Kunkel, Chandrakana Nandi, Max Willsey, Zachary Tatlock, and Nadia Polikarpova. 2023. Babble: Learning Better Abstractions with E-Graphs and

Anti-Unification. *Proc. ACM Program. Lang.* 7, POPL, Article 14 (jan 2023), 29 pages. https://doi.org/10.1145/3571207

Alexandre Carlier, Martin Danelljan, Alexandre Alahi, and Radu Timofte. 2020. DeepSVG: A Hierarchical Generative Network for Vector Graphics Animation. In *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 33. 16351–16361.

Angel X Chang, Thomas Funkhouser, Leonidas Guibas, Pat Hanrahan, Qixing Huang, Zimo Li, Silvio Savarese, Manolis Savva, Shuran Song, Hao Su, Jianxiong Xiao, Li Yi, and Fisher Yu. 2015. ShapeNet: An information-rich 3D model repository. *arXiv preprint arXiv:1512.03012* (2015).

Eyal Dechter, Jon Malmaud, Ryan P. Adams, and Joshua B. Tenenbaum. 2013. Bootstrap Learning via Modular Concept Discovery. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence* (Beijing, China) *(IJCAI '13)*. AAAI Press, 1302–1309.

Ilke Demir, Daniel G Aliaga, and Bedrich Benes. 2014. Proceduralization of buildings at city scale. In *2014 2nd International Conference on 3D Vision*, Vol. 1. IEEE, 456–463.

Boyang Deng, Sumith Kulal, Zhengyang Dong, Congyue Deng, Yonglong Tian, and Jiajun Wu. 2022. Unsupervised Learning of Shape Programs with Repeatable Implicit Parts. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Tao Du, Jeevana Priya Inala, Yewen Pu, Andrew Spielberg, Adriana Schulz, Daniela Rus, Armando Solar-Lezama, and Wojciech Matusik. 2018. InverseCSG: automatic conversion of 3D models to CSG trees. In *Annual Conference on Computer Graphics and Interactive Techniques Asia (SIGGRAPH Asia)*. ACM.

Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Joshua B. Tenenbaum, and Armando Solar-Lezama. 2019. Write, Execute, Assess: Program Synthesis with a REPL. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. 2018. Learning to Infer Graphics Programs from Hand-Drawn Images. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B Tenenbaum. 2021. 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)*. 835–850.

Jonas Freiknecht and Wolfgang Effelsberg. 2017. A Survey on the Procedural Generation of Virtual Worlds. *Multimodal Technologies and Interaction* 1, 4 (2017). https://doi.org/10.3390/mti1040027

Yaroslav Ganin, Sergey Bartunov, Yujia Li, Ethan Keller, and Stefano Saliceti. 2021. Computer-aided design as language. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Paul Guerrero, Milos Hasan, Kalyan Sunkavalli, Radomir Mech, Tamy Boubekeur, and Niloy Mitra. 2022. MatFormer: A Generative Model for Procedural Materials. *ACM Transactions on Graphics (TOG)* 41, 4, Article 46 (2022).

Jianwei Guo, Haiyong Jiang, Bedrich Benes, Oliver Deussen, Xiaopeng Zhang, Dani Lischinski, and Hui Huang. 2020. Inverse procedural modeling of branching structures by inferring L-systems. *ACM Transactions on Graphics (TOG)* 39, 5 (2020), 1–13.

Yiwei Hu, Chengan He, Valentin Deschaintre, Julie Dorsey, and Holly Rushmeier. 2022. An inverse procedural modeling pipeline for svbrdf maps. *ACM Transactions on Graphics (TOG)* 41, 2 (2022), 1–17.

Irvin Hwang, Andreas Stuhlmüller, and Noah D. Goodman. 2011. Inducing Probabilistic Programs by Bayesian Program Merging. *CoRR* arXiv:1110.5667 (2011).

R. Kenny Jones, Theresa Barton, Xianghao Xu, Kai Wang, Ellen Jiang, Paul Guerrero, Niloy J. Mitra, and Daniel Ritchie. 2020. ShapeAssembly: Learning to Generate Programs for 3D Shape Structure Synthesis. *ACM Transactions on Graphics (TOG), Siggraph Asia 2020* 39, 6 (2020), Article 234.

R. Kenny Jones, David Charatan, Paul Guerrero, Niloy J. Mitra, and Daniel Ritchie. 2021. ShapeMOD: Macro Operation Discovery for 3D Shape Programs. *ACM Transactions on Graphics (TOG), Siggraph 2021* 40, 4 (2021), Article 153.

R. Kenny Jones, Homer Walke, and Daniel Ritchie. 2022. PLAD: Learning to Infer Shape Programs with Pseudo-Labels and Approximate Distributions. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

Kacper Kania, Maciej Zieba, and Tomasz Kajdanowicz. 2020. UCSG-NET - unsupervised discovering of constructive solid geometry tree. In *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 33. 8776–8786.

Changjian Li, Hao Pan, Adrien Bousseau, and Niloy J. Mitra. 2020. Sketch2CAD: Sequential CAD Modeling by Sketching in Context. *ACM Transactions on Graphics (TOG)* 39, 6 (2020), 164:1–164:14.

Changjian Li, Hao Pan, Adrien Bousseau, and Niloy J. Mitra. 2022. Free2CAD: Parsing Freehand Drawings into CAD Commands. *ACM Transactions on Graphics (TOG)* 41, 4 (2022), 93:1–93:16.

Andelo Martinovic and Luc Van Gool. 2013. Bayesian Grammar Learning for Inverse Procedural Modeling. In *2013 IEEE Conference on Computer Vision and Pattern Recognition*. 201–208. https://doi.org/10.1109/CVPR.2013.33

Elie Michel and Tamy Boubekeur. 2021. DAG Amendment for Inverse Control of Parametric Shapes. *ACM Transactions on Graphics* 40, 4 (2021), 173:1–173:14.

Kaichun Mo, Shilin Zhu, Angel X. Chang, Li Yi, Subarna Tripathi, Leonidas J. Guibas, and Hao Su. 2019. 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)*.

Pascal Müller, Gang Zeng, Peter Wonka, and Luc Van Gool. 2007. Image-based procedural modeling of facades. *ACM Trans. Graph.* 26, 3 (2007), 85.

Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. 2020. Synthesizing Structured CAD Models with Equality Saturation and Inverse Transformations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 31–44. https://doi.org/10.1145/3385412.3386012

Wamiq Reyaz Para, Shariq Farooq Bhat, Paul Guerrero, Tom Kelly, Niloy Mitra, Leonidas Guibas, and Peter Wonka. 2021. SketchGen: Generating Constrained CAD Sketches. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Ofek Pearl, Itai Lang, Yuhua Hu, Raymond A. Yeh, and Rana Hanocka. 2022. GeoCode: Interpretable Shape Programs. arXiv:2212.11715 [cs.GR]

Pradyumna Reddy, Michael Gharbi, Michal Lukac, and Niloy J Mitra. 2021a. Im2Vec: Synthesizing vector graphics without vector supervision. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 7342–7351.

Pradyumna Reddy, Zhifei Zhang, Matthew Fisher, Hailin Jin, Zhaowen Wang, and Niloy J Mitra. 2021b. A Multi-Implicit Neural Representation for Fonts. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Daxuan Ren, Jianmin Zheng, Jianfei Cai, Jiatong Li, Haiyong Jiang, Zhongang Cai, Junzhe Zhang, Liang Pan, Mingyuan Zhang, Haiyu Zhao, et al. 2021. CSG-Stump: A Learning Friendly CSG-Like Representation for Interpretable Shape Parsing. In *IEEE/CVF International Conference on Computer Vision (ICCV)*. 12478–12487.

Daxuan Ren, Jianmin Zheng, Jianfei Cai, Jiatong Li, and Junzhe Zhang. 2022. ExtrudeNet: Unsupervised Inverse Sketch-and-Extrude for Shape Parsing. In *European Conference on Computer Vision (ECCV)*.

Daniel Ritchie, Paul Guerrero, R. Kenny Jones, Niloy J. Mitra, Adriana Schulz, Karl D. D. Willis, and Jiajun Wu. 2023. Neurosymbolic Models for Computer Graphics. *Computer Graphics Forum* (2023).

Daniel Ritchie, Sarah Jobalia, and Anna Thomas. 2018. Example-based Authoring of Procedural Modeling Programs with Structural and Continuous Variability. In *EUROGRAPHICS*.

Ari Seff, Wenda Zhou, Nick Richardson, and Ryan P. Adams. 2022. Vitruvion: A Generative Model of Parametric CAD Sketches. In *International Conference on Learning Representations (ICLR)*.

Gopal Sharma, Rishabh Goyal, Difan Liu, Evangelos Kalogerakis, and Subhransu Maji. 2018. CSGNet: Neural Shape Parser for Constructive Solid Geometry. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Liang Shi, Beichen Li, Miloš Hašan, Kalyan Sunkavalli, Tamy Boubekeur, Radomir Mech, and Wojciech Matusik. 2020. MATch: Differentiable Material Graphs for Procedural Material Capture. *ACM Trans. Graph.* 39, 6 (Dec. 2020), 1–15.

SideFx. 2014. Introduction to Procedural Modeling. https://www.sidefx.com/learn/collections/introduction-to-procedural-modeling/. Accessed: 2023-04-18.

Ondrej Stava, Sören Pirk, Julian Kratt, Baoquan Chen, Radomír Měch, Oliver Deussen, and Bedrich Benes. 2014. Inverse procedural modelling of trees. In *Computer Graphics Forum*, Vol. 33. Wiley Online Library, 118–131.

Jerry O. Talton, Lingfeng Yang, Ranjitha Kumar, Maxine Lim, Noah D. Goodman, and Radomír Mech. 2012. Learning design patterns with Bayesian grammar induction. In *UIST*.

Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality Saturation: A New Approach to Optimization *(POPL '09)*. Association for Computing Machinery, New York, NY, USA, 264–276. https://doi.org/10.1145/1480881.1480915

Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2011. Equality Saturation: A New Approach to Optimization. *Logical Methods in Computer Science* Volume 7, Issue 1 (March 2011). https://doi.org/10.2168/LMCS-7(1:10)2011

Yonglong Tian, Andrew Luo, Xingyuan Sun, Kevin Ellis, William T. Freeman, Joshua B. Tenenbaum, and Jiajun Wu. 2019. Learning to Infer and Execute 3D Shape Programs. In *International Conference on Learning Representations (ICLR)*.

Carlos A Vanegas, Ignacio Garcia-Dorado, Daniel G Aliaga, Bedrich Benes, and Paul Waddell. 2012. Inverse design of urban procedural models. *ACM Transactions on Graphics (TOG)* 31, 6 (2012), 1–11.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast and Extensible Equality Saturation. *Proc. ACM Program. Lang.* 5, POPL, Article 23 (Jan. 2021), 29 pages. https://doi.org/10.1145/3434304

Chenming Wu, Haisen Zhao, Chandrakana Nandi, Jeffrey I. Lipton, Zachary Tatlock, and Adriana Schulz. 2019. Carpentry Compiler. *ACM Trans. Graph.* 38, 6, Article 195 (Nov. 2019), 14 pages. https://doi.org/10.1145/3355089.3356518

Fuzhang Wu, Dong-Ming Yan, Weiming Dong, Xiaopeng Zhang, and Peter Wonka. 2014. Inverse Procedural Modeling of Facade Layouts. *ACM Trans. Graph.* 33, 4, Article 121 (jul 2014), 10 pages. https://doi.org/10.1145/2601097.2601162

Rundi Wu, Chang Xiao, and Changxi Zheng. 2021. DeepCAD: A Deep Generative Network for Computer-Aided Design Models. In *IEEE/CVF International Conference on Computer Vision (ICCV)*. 6772–6782.

Ling Xu and David Mould. 2015. Procedural tree modeling with guiding vectors. In *Computer Graphics Forum*, Vol. 34. Wiley Online Library, 47–56.

Xianghao Xu, Wenzhe Peng, Chin-Yi Cheng, Karl D. D. Willis, and Daniel Ritchie. 2021. Inferring CAD Modeling Sequences Using Zone Graphs. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

Xiang Xu, Karl DD Willis, Joseph G Lambourne, Chin-Yi Cheng, Pradeep Kumar Jayaraman, and Yasutaka Furukawa. 2022. SkexGen: Autoregressive Generation of CAD Construction Sequences with Disentangled Codebooks. In *International Conference on Machine Learning (ICML)*.

Kaizhi Yang and Xuejin Chen. 2021. Unsupervised Learning for Cuboid Shape Abstraction via Joint Segmentation from Point Clouds. *ACM Trans. Graph.* 40, 4, Article 152 (jul 2021), 11 pages. https://doi.org/10.1145/3450626.3459873

Fenggen Yu, Zhiqin Chen, Manyi Li, Aditya Sanghi, Hooman Shayani, Ali Mahdavi-Amiri, and Hao Zhang. 2022. CAPRI-Net: Learning Compact CAD Shapes With Adaptive Primitive Assembly. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 11768–11778.

A 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.

B IMPLEMENTATION DETAILS

We provide implementation details for ShapeCoder below. For all experiments in Section 7 we set $N_A = 20$ and $N_D = 10000$.

B.1 Objective Function Weights

We use the following weights for λ in ShapeCoder's objective function (Section 3.1): 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 3.1 and ablated in Section 7.4, 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.

B.2 Geometric Error Function

The objective function (Section 3.1) 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 4.3), 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 5.2), we can modify this approach to check for a *program* p 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 p, and the number of primitives expected in the target shape d, so p 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 .

B.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 4.2) 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).

B.4 Dream Creation

Sampling library functions. During the dream phase (Section 4.2), ShapeCoder randomly samples instantiations of library functions to train the recognition network. Some dreams are visualized in Figure 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.

B.5 Combining Wake Programs

As discussed in Section 4.3, 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 $\mathcal{P}$, p_c. Using a *split* function, that recursively removes combinator operations from a program, we can convert p_c into a set of expressions in $\mathcal{L}$: $split(p_c) = E_c = \{e_c^0, ..., e_c^{|E_c|}\}$. 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, p_r, for d. We decide what program p 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 p_c. *(ii)* Use p_r (note this variant will always be chosen if $r = 0$). *(iii)* Greedily merge p_r into p_c. To do this, we first compute $split(p_r) = E_r = \{e_r^0, ..., e_r^{|E_c|}\}$. 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 p_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 p_c, then the greedy search in *(iv)* is used to find replacement expressions for p_c.

B.6 Preference Ordering of Parametric Relationships

The proposal phase (Section 5.1) 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 4), from which one is chosen according to the *score* function.

B.7 E-graphs

Our refactor operation (Section 6), implements e-graphs using the Egg library [Willsey et al. 2021]. 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 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 7.4).

B.8 Unsupervised Primitive Decomposition

As described in Section 7.5, 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 [Yang and Chen 2021], 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.

B.9 Generative Model for Programs

We provide details for the generative model described in Section 7.6. 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 7.3) over a dataset of 3600 chairs, to form a program dataset $\mathcal{P}$. For each epoch, we randomize expression ordering by applying *split* (Section B.5) to each $p \in \mathcal{P}$, 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).