ShapeMOD: Macro Operation Discovery for 3D Shape Programs

R. KENNY JONES, Brown University
DAVID CHARATAN, Brown University
PAUL GUERRERO, Adobe Research
NILOY J. MITRA, University College London and Adobe Research
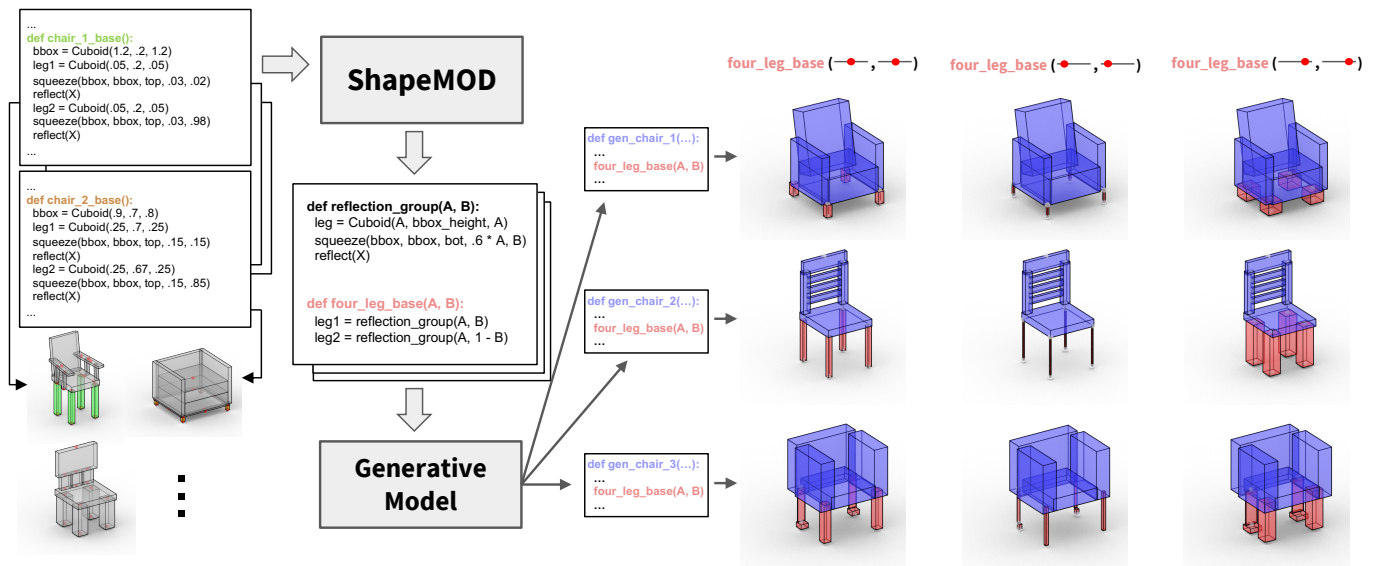DANIEL RITCHIE, Brown University

Fig. 1. We propose ShapeMOD, an algorithm which takes as input a collection of 3D shape programs and makes them more compact by automatically discovering common *macros* which can be re-used across the collection. We apply ShapeMOD to datasets of ShapeAssembly programs and find that generative models which train on refactored programs containing these macros produce more plausible output shapes than those trained on the original programs. The discovered macros also facilitate shape editing by exposing only a small number of meaningful parameters for manipulating shape attributes. For example, the *four_leg_base* macro exposes two parameters (visualized as sliders with red handles); one parameter controls leg size, while the other controls leg spacing.

A popular way to create detailed yet easily controllable 3D shapes is via procedural modeling, i.e. generating geometry using programs. Such programs consist of a series of instructions along with their associated parameter values. To fully realize the benefits of this representation, a shape program should be compact and only expose degrees of freedom that allow for meaningful manipulation of output geometry. One way to achieve this goal is to design higher-level *macro* operators that, when executed, expand into a series of commands from the base shape modeling language. However, manually authoring such macros, much like shape programs themselves, is difficult and largely restricted to domain experts. In this paper, we present ShapeMOD, an algorithm for automatically discovering macros that are useful across large datasets of 3D shape programs. ShapeMOD operates on shape programs expressed in an imperative, statement-based language. It is designed to discover macros that make programs more compact by minimizing the number of function calls and free parameters required to represent an input shape collection. We run ShapeMOD on multiple collections of programs expressed in a domain-specific language for 3D shape structures. We show that it automatically discovers a concise set of macros that abstract out common structural and parametric patterns that generalize over large shape collections. We also demonstrate that the macros found by ShapeMOD improve performance on downstream tasks including shape generative modeling and inferring programs from point clouds. Finally, we conduct a user study that indicates that ShapeMOD's discovered macros make interactive shape editing more efficient.

CCS Concepts: • **Computing methodologies** → **Computer graphics**; **Neural networks**; **Probabilistic reasoning**.

Additional Key Words and Phrases: Shape analysis, shape synthesis, generative models, deep learning, procedural modeling, neurosymbolic models

Authors' addresses: R. Kenny Jones, Brown University; David Charatan, Brown University; Paul Guerrero, Adobe Research; Niloy J. Mitra, University College London and Adobe Research; Daniel Ritchie, Brown University.

1 INTRODUCTION

3D data is important for many applications in graphics, vision, and artificial intelligence: creating content for games and AR/VR experiences, synthetic training data for autonomous agents, etc. Unlike 2D image data, which has a standard representation as a regular grid of pixels, 3D data can be expressed in a variety of representations, each with their own strengths and weaknesses. Point clouds are easy to obtain from the real world via depth sensors, but they lose surface connectivity and detail; meshes are well-supported by rendering and simulation packages, but they are harder to train machine learning models on; implicit functions provide excellent surface detail but are harder to inspect and reason about.

One interesting way to represent a 3D shape is with a program that generates its geometry, or at least its high-level structure. This is particularly appealing for manufactured objects, as such shapes typically *originate* as CAD programs of some form (e.g., connected assemblies of parts, the geometry of which may be specified by lower-level instructions). In computer graphics, the procedural modeling literature offers a rich history demonstrating the value of programs for shape modeling. By using simple domain-specific languages (DSLs) that combine geometric elements, programs (such as shape grammars) can model shapes from a variety of domains, such as plants [Prusinkiewicz and Lindenmayer 1996], buildings [Müller et al. 2006], cities [Parish and Müller 2001], and furniture [Lau et al. 2011]. In addition to enabling rapid creation of a variety of content, programs offer the benefit of being more readable and editable (via the free parameters they expose) than other shape representations.

To maximally realize these benefits, a good shape program should be compact and expressed at a high level while still exposing important degrees of freedom for editing. One way to create such programs is to introduce higher-level functions, or *macros*, into the shape DSL. We define a macro to be a function that, when executed, expands into a series of commands from the base DSL.

In urban procedural modeling, a macro might capture how primitives combine to make a particular class of railing; in furniture modeling, a macro might be used to model a shelving pattern that could be instantiated within different types and sizes of furniture; in plant modeling, a macro might be used to instantiate examples of petal structures across a family of flowers. In past work on modeling 3D shape structures, rudimentary macros proved to be helpful for downstream tasks of interest, such as shape editing and generative modeling [Jones et al. 2020]. However, these macros were carefully (manually) designed by experts and may not generalize to or be useful for other shape modeling domains for which they were not designed. One question naturally follows: *Can useful macros for a given shape modeling domain be designed automatically by an algorithm?*

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

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

We run ShapeMOD on multiple collections of shape programs expressed in the ShapeAssembly DSL [Jones et al. 2020] to discover new libraries of macros. For example, in Figure 1, starting from a set of chair shape programs, ShapeMOD discovers a reusable macro for four leg chair bases which exposes a compact set of associated control sliders. We demonstrate the benefits of working with these discovered macros, by evaluating how adding the discovered macros into the language affects performance on downstream tasks: learning a generative model for shape programs, learning to infer shape programs from unstructured geometry, and goal-directed editing of shapes via their programs. In all cases, task performance is improved by using automatically discovered macros. Finally, we show that ShapeMOD can find useful macros even when trained on a set of ShapeAssembly programs from multiple categories.

In summary, our contributions are:

(i) An algorithm that takes as input a collection of programs expressed in a DSL with imperative functions that may contain continuous parameters, and automatically discovers a concise set of macros that abstract out common structural and parametric patterns within the input programs that generalize over a shape collection.

(ii) Demonstrations on collections of manufactured objects that using discovered macros leads to better performance on important downstream tasks such as novel shape generation, directed shape manipulation, and inferring shape structures from unstructured geometry (i.e., point clouds).

Code can be found at anonymous.link . Our implementation is not specific to ShapeAssembly, making it possible to apply ShapeMOD to collections of programs in other imperative languages with real-valued parameters.

2 BACKGROUND & RELATED WORK

Our work is related to prior work in program synthesis, automatic program abstraction, shape abstraction, and 3D shape generative modeling. We also build heavily upon the ShapeAssembly shape structure modeling DSL [Jones et al. 2020], so we briefly describe its salient features in this section as well.

2.1 Related Prior Work

Finding common abstractions over a set of programs. Our macro-discovery goal is one instance of a more general class of problems:

finding common abstractions within a set of programs. Context-free grammars have been a common class of programs used in prior work on procedural modeling, and grammar induction is one example of common abstraction finding. Prior work has designed induction algorithms for building facade grammars [Martinovic and Van Gool 2013], more general grammars [Talton et al. 2012], and even grammar-like functional programs [Hwang et al. 2011]. One more recent system even supports programs with continuous parameters [Ritchie et al. 2018]. However, these approaches are all limited to context-free languages.

Recently, another line of work has investigated common abstraction discovery for general functional programs: the Exploration-Compression (EC) algorithm [Dechter et al. 2013] and its successor, DreamCoder [Ellis et al. 2020]. Both operate by repeatedly invoking two algorithm phases. In EC, the two phases are "exploration" (trying to find programs that solve input problems) and "compression" (finding abstractions common to these programs). In DreamCoder, these two phases are respectively called "wake" and "sleep." Our method, ShapeMOD, also uses a two-phase approach: "proposal" (finding candidate abstractions) and "integration" (choosing candidate abstractions to add to the language). EC and DreamCoder operate on functional programs, and while they do provide some support for real-valued free parameters, they are unable to discover parametric relationships between continuous variables.

To the best of our knowledge, ShapeMOD is the first approach that discovers macro operations capturing parametric relationships between continuous parameters across a collection of imperative programs. While specialized data structures such as version spaces and E-graphs can efficiently reason about rewrites of functional programs, they cannot efficiently reason over semantic line re-orderings of imperative programs (i.e. maintaining correct execution behavior) and thus are not applicable to languages such as ShapeAssembly.

Rewriting a single program. ShapeMOD discovers macros which are common to a collection of programs. A related, but different, problem is that of finding modifications that improve a *single* program of interest. There has been prior work on this problem in the realm of shape modeling languages. The Szalinski system takes a low-level CAD program as input and searches for a more compact, higher-level program which produces the same output geometry [Nandi et al. 2020]. The Carpentry Compiler is a program optimizer that finds rewrites of low-level instructions to maintain high-level semantics while optimizing to reduce manufacturing cost [Wu et al. 2019]. Our approach can also be seen as related to systems for more general program-rewriting, such as optimizing compilers [Tate et al. 2009].

Shape abstraction. ShapeMOD discovers macros by *abstracting out* common patterns in shape programs. Prior work has investigated how to abstract shapes themselves, either by simplifying the geometry of individual shapes [Mehra et al. 2009], finding common abstractions for collections of shapes [Yumer and Kara 2012], or learning to represent shapes via a union of simple primitives [Deng et al. 2020; Genova et al. 2019; Sun et al. 2019; Tulsiani et al. 2017; Zou et al. 2017]. These methods attempt to abstract the *geometry* of shapes, whereas as we are interested in abstracting the procedural structures of programs that generate shape geometry.

Generative Models of 3D Shapes. One of the major downstream applications we target is learning generative models of 3D shapes. Many such models have been proposed. One category of models learns to generate entire objects as unstructured geometry, representing shapes with occupancy grids [Wu et al. 2016], surface patches [Groueix et al. 2018], point clouds [Achlioptas et al. 2018], or implicit functions [Chen and Zhang 2019]. In contrast, *structure-aware* models learn to generate objects as arrangements of their component parts [Chaudhuri et al. 2020; Gao et al. 2019; Jones et al. 2020; Li et al. 2017; Mitra et al. 2013; Mo et al. 2019a; Yang et al. 2020]. In our results, we show that training a structure-aware generative model on ShapeAssembly programs, refactored with discovered macros, improves novel shape generation.

Visual Program Induction. Another downstream application we target is visual program induction: inferring a program representation of an object given only unstructured sensor input (e.g., an image or a point cloud). Prior work in this area has targeted custom DSLs [Ellis et al. 2018b; Tian et al. 2019], constructive solid geometry (CSG) [Du et al. 2018; Ellis et al. 2019; Sharma et al. 2018; Walke et al. 2020] or other CAD representations [Willis et al. 2020]. We show that when inferring ShapeAssembly programs from point clouds, targeting a macro-enhanced version of the language leads to better reconstruction quality.

2.2 Background: ShapeAssembly

To demonstrate the benefits of ShapeMOD, in this paper we apply it to collections of ShapeAssembly programs [Jones et al. 2020]. ShapeAssembly is a domain-specific language that creates 3D shape structures by defining parts and attaching them together. It is executed imperatively, and its functions use both continuous and discrete parameters. To discover more useful macros, we modify the grammar of ShapeAssembly slightly, as described in Appendix A.

In its original form, ShapeAssembly contains two base functions and three macro functions. The base functions are Cuboid, which creates an cuboid part proxy, and attach, which moves a Cuboid to satisfy the described spatial relationship. The squeeze, reflect and translate commands are expert-defined macros that abstract common patterns of structural and geometric variation. Each of these macros expands into a sequence of lower-level Cuboid and attach commands. ShapeAssembly contains both discrete parameters (cuboid IDs, cuboid faces, symmetry axes, etc.) and continuous parameters (cuboid dimensions, attachment points, etc.). Please refer to the original ShapeAssembly paper [Jones et al. 2020] for further details of the language specification.

3 MACRO OPERATOR DISCOVERY

ShapeMOD's goal is to take a dataset of programs $\mathcal{D}$ and the library of DSL functions used to express them $\mathcal{L}$, and return a new library (with additional macros) which is able to express the programs in $\mathcal{D}$ with fewer free parameters. The motivation here is that macros should remove free parameters that correspond to *extraneous* degrees of freedom, i.e. degrees of freedom that can create implausible output shapes, such as independently changing the length of each leg of a table. At the same time, we want to keep the number of functions in our library relatively small, so as not to
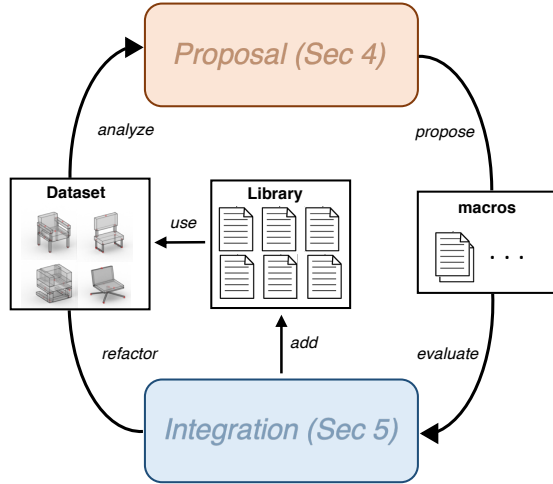
Fig. 2. ShapeMOD consists of two alternating phases: proposing new candidate macros (top) and refactoring programs to use some of the proposed macros (bottom).

Algorithm 1 ShapeMOD

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

```
 1: for num_rounds do
 2:     candidate_macros ← Set()                          {Proposal Phase}
 3:     for num_proposal_steps do
 4:         P, o ← sampleProgAndOrder(𝒟)                    {Sec 4.1}
 5:         𝒫_matches ← findMatchingProgs(𝒟, P, o)
 6:         𝒫_cluster ← sampleByParamSim(𝒫_matches)
 7:         P_abs ← findAbstractProg(𝒫_cluster, ℒ)          {Sec 4.2}
 8:         ℳ ← proposeMacrosForProg(P_abs)                 {Sec 4.3}
 9:         ℳ ← generalize(ℳ)                               {Sec 4.4}
10:         candidate_macros += ℳ
11:     𝒟̃ ← subsample(𝒟)                                   {Integration Phase}
12:     for num_integration_steps do
13:         M ← getTopRankedMacro(candidate_macros)         {Sec 5.1}
14:         ℒ' ← optimize(f, ℒ, ℒ + {M}, 𝒟̃)                {Sec 5.2}
15:         ℒ ← ℒ'; continue
16:         ℳ_infreq ← findInfrequentMacros(𝒟̃, ℒ, ℒ + {M})
17:         continue
18:         ℒ' ← optimize(f, ℒ, ℒ + {M} − ℳ_infreq, 𝒟̃)
19:         ℒ ← ℒ'
20:         for M ∈ ℳ_infreq do
21:             ℒ ← optimize(f, ℒ, ℒ + {M}, 𝒟̃)
22:     for M ∈ ℒ do
23:         ℒ ← optimize(f, ℒ, ℒ − {M}, 𝒟̃)
24:     𝒟 ← filterBadOrders(f, 𝒟, ℒ)                       {Sec 5.3}
25: return ℒ, 𝒫*(𝒟, ℒ)                                    {Sec 3.4}
```

---

remove *necessary* degrees of freedom that can create meaningful shape manipulations. We formalize this trade-off in an objective function f which the algorithm attempts to minimize.

3.1 Overview

The ShapeMOD algorithm has two phases. First, a *proposal phase* (Section 4) finds clusters of similar programs and uses these clusters to propose a set of candidate macros. Then, an *integration phase* (Section 5) greedily iterates through a ranked list of these candidate macros and adds them to the library $\mathcal{L}$ whenever it would improve the objective function f. These phases can be alternated one after the other for multiple rounds, with the output of one phase treated as the input for the next (Fig. 2). By iterating this procedure for multiple rounds, increasingly complex macros can be found; as a macro discovered in round t can use a previously-discovered macro from round $t − 1$ in its definition (Fig. 3).

Working with imperative programs that contain real-valued parameters presents unique challenges. For instance, it is difficult to reason about valid line re-orderings of imperative programs when discovering macros and deciding when they can be applied. ShapeMOD uses a sampling-based approach to discover macros by creating clusters of shapes with shared program structure (Section 4.1) and a beam search procedure to decide how to apply discovered macros to existing programs (Section 3.4). Moreover, when dealing with real-valued parameters, it is challenging to find meaningful (non-spurious) parametric relationships, especially within a single program. To achieve generality, ShapeMOD finds abstracted expressions that simultaneously describe multiple programs from a cluster of related shapes (Section 4.2).

Complete pseudocode for ShapeMOD is shown in Algorithm 1; Sections 4 and 5 explain this procedure in more detail. As input, it takes in a starting library of functions $\mathcal{L}$, a dataset of imperative programs $\mathcal{D}$ and an objective function f to be minimized. Each element of $\mathcal{D}$ is a tuple (P, O_P) containing program lines P and the set of valid orderings for those lines O_P (i.e. re-orderings of the lines which produce the correct output when executed).

3.2 Initialization

In our experiments, the library $\mathcal{L}$ is initialized with the 5 manually designed functions from the ShapeAssembly grammar. Then, starting with a collection of hierarchically-organized 3D cuboid structures from PartNet [Mo et al. 2019b], we use ShapeAssembly's data parsing algorithm to find program lines P which recreate each shape. We then developed a procedure to determine the set of valid orderings O_P for that program (i.e. all orderings which produce the correct output geometry) to form our input dataset $\mathcal{D}$. Further details about the data parsing and valid ordering procedures can be found in the supplemental (Section A.3).

3.3 Objective Function

Our goal is to represent an entire dataset of programs compactly (removing free parameters) while also keeping the number of functions in the library small. Specifically, our objective is to minimize a weighted sum of the number of functions in $\mathcal{L}$ and the number of free parameters needed to represent programs in the dataset $\mathcal{D}$. For ShapeAssembly, free parameters can have multiple types T: Choice of function per line (**fn**), cuboid ID (**cid**), float/continuous (**f**), discrete (**d**), Boolean (**b**). One may care about compressing these types differently, we allow each parameter type to be weighed differently
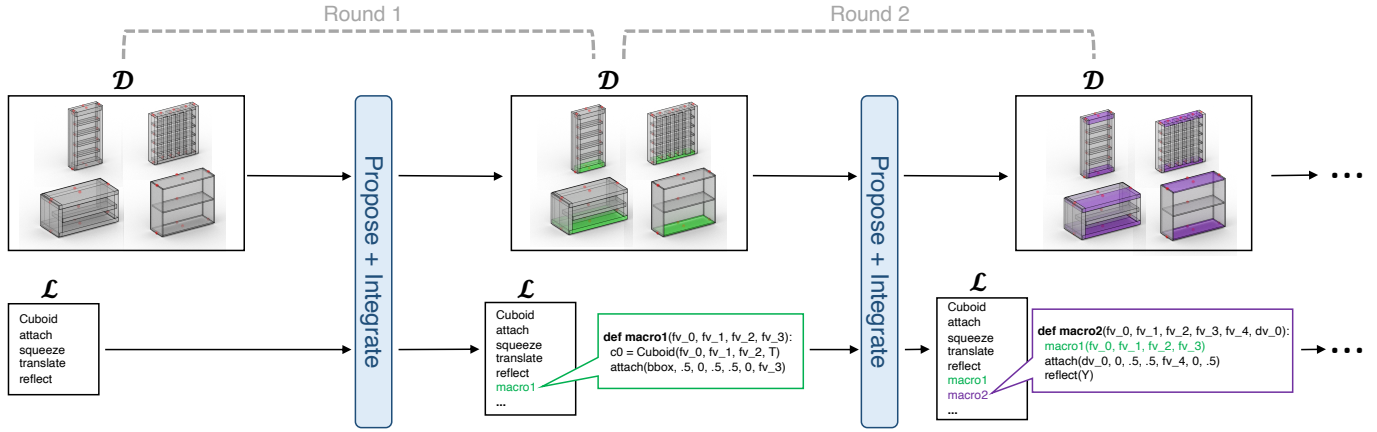
Fig. 3. Running ShapeMOD for multiple rounds allows for discovery of increasingly complex macros. Here, a macro discovered in Round 2 uses a macro previously found in Round 1 as part of its function body.

in the objective defined as,

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

where $\mathcal{P}^*(\mathcal{D}, \mathcal{L})$ returns the best programs for $\mathcal{D}$ using the functions in $\mathcal{L}$ (Section 3.4), $\tau(\mathcal{P})$ returns the set of all τ-typed free parameters in the programs $\mathcal{P}$, and $\epsilon(\tau, \mathcal{D}, \mathcal{P})$ returns the sum of errors in τ-typed parameters incurred by using $\mathcal{P}^*(\mathcal{D}, \mathcal{L})$ in place of the original programs in $\mathcal{D}$. The weights λ_n, $\{\lambda_\tau | \tau \in T\}$ and λ_ϵ can be adjusted to express preferences for the types of macros the algorithm aims to find. In our experiments, we use $\lambda_n = 1$, $\lambda_{\mathbf{fn}} = 8$, $\lambda_{\mathbf{cid}} = 8$, $\lambda_{\mathbf{f}} = 1$, $\lambda_{\mathbf{d}} = 0.5$, $\lambda_{\mathbf{b}} = 0.25$, and $\lambda_\epsilon = 10$.

3.4 Finding the Best Program for a Given Library

Calculating the value of f over a dataset of shapes requires finding the program under $\mathcal{L}$ that minimizes the objective function for each program $(P, O_P) \in \mathcal{D}$. As O_P is a collection of valid orderings of the program lines P, we solve this problem by finding the best scoring program under $\mathcal{L}$ for every $o \in O_P$. Combining an ordering o with program lines P produces a program expressed in terms of base library functions P_o. We then want to find the best program, P^*, that uses the functions in $\mathcal{L}$ (including macros, if $\mathcal{L}$ contains them) to recreate P_o while minimizing f. We implement this procedure with a beam search that iteratively builds partial programs in the beam by adding calls to functions from $\mathcal{L}$ whose expansions cover lines in P_o. For a function expansion to *cover* a sequence of program lines, the expansion must match those lines on command type, the values of the discrete / Boolean parameters must match exactly, and the continuous parameters must differ by an amount no greater than ϵ. We set $\epsilon = 0.05$, finding that larger values lead to abstracted programs with degenerate geometry. We rank partial programs in the beam by their objective value, normalized by the number of lines in P_o it is covering. This search runs until all programs in the beam have no more lines in P_o to cover; the program with lowest objective value is returned as the best program P^*. In the case of ties, we choose the program with the most canonical ordering, as explained in the supplemental material. In our implementation, we

use a beam width of 10. Other search strategies could be applied here; we chose beam search as it was relatively fast and found good solutions.

4 PROPOSAL PHASE

The goal of ShapeMOD's proposal phase is to construct a set of candidate macros which might be useful for compressing the dataset of shape programs $\mathcal{D}$. A schematic overview of the proposal phase is shown in Figure 4. In each proposal round, the algorithm first forms a cluster of similar programs sampled from $\mathcal{D}$ (Section 4.1). Then, using the functions of $\mathcal{L}$, it finds an abstracted program that explains the majority of examples in the cluster while trying to remove free parameters whenever possible (Section 4.2). It converts this abstracted program into a set of candidate macros (Section 4.3) and finds potential generalizations of these macros (Section 4.4). This process is repeated for *num_proposal_steps* (we use 10000) to build up a large collection of candidate macros.

4.1 Form a Program Cluster

The goal of the cluster formation step is to find a set of programs from $\mathcal{D}$ that can be represented by a single abstracted program, i.e. a program with free variables. The blue box in Fig. 4 illustrates the procedure. The algorithm first randomly samples a program P from $\mathcal{D}$ and then randomly samples an order o from the possible valid orderings in O_P (Algorithm 1, line 4). It then finds the set of programs $\mathcal{P}_{\mathbf{matches}}$ from $\mathcal{D}$ that *structurally match* P and also have o as one of their valid orderings in (Algorithm 1, line 5). In ShapeAssembly, two programs structurally match if they use the same set of commands which refer to the same cuboid IDs (though their other parameters may vary).

For each program in $\mathcal{P}_{\mathbf{matches}}$, we record the norm n of the difference of its continuous parameters compared with those in P. We then form a probability distribution over $\mathcal{P}_{\mathbf{matches}}$, where each program is given a weight proportional to $1 - \frac{n}{n*}$, where $n*$ was the maximum observed n. Taking the parameter distance between programs into account results in clusters that are more semantically consistent, which increases the likelihood the abstracted program
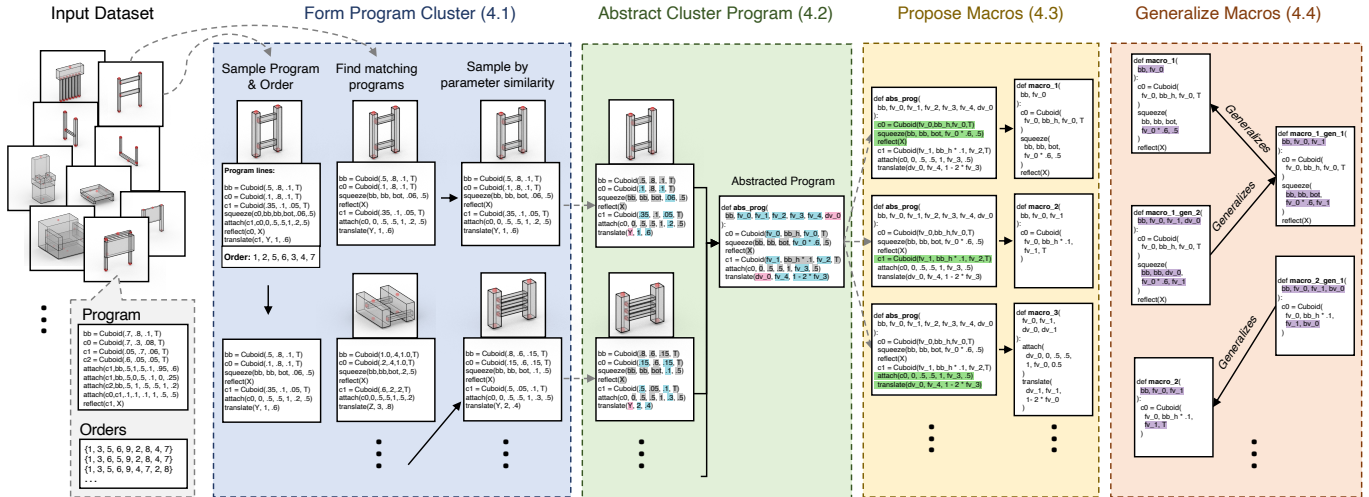
Fig. 4. ShapeMOD's proposal phase, which proposes candidate macros to be added into $\mathcal{L}$. Each round of this phase begins by identifying a cluster of structurally-identical programs with similar parameter values within the input dataset (Section 4.1). It then finds a single abstracted program which subsumes most or all of the programs in this cluster (Section 4.2); here, gray parameter values are abstracted as constants, blue ones as continuous free variables, and pink ones as discrete free variables. Subsequences of lines in this abstracted program (shown in green) are isolated to form potential macros which could be used to re-write the program (Section 4.3). Finally, this set of candidate macros is expanded by including generalizations of the initial set (Section 4.4); purple lines show lines that are generalized. Best viewed on a high-resolution screen.

we produce can discover meaningful parametric relationships. Finally, we sample k programs from $\mathcal{P}_{\mathbf{matches}}$ using this probability distribution in order to form $\mathcal{P}_{\mathbf{cluster}}$ (line 6). We set k to 20.

4.2 Find Abstracted Program for Cluster

Given the cluster of programs $\mathcal{P}_{\mathbf{cluster}}$ identified in the previous section, the next step is to use the library of functions $\mathcal{L}$ to find the most compact program (fewest free parameters) that can represent the majority of programs in $\mathcal{P}_{\mathbf{cluster}}$ (Algorithm 1, line 7). By construction, the sequence of functions and cuboid IDs is the same across all programs in $\mathcal{P}_{\mathbf{cluster}}$. To build up the abstracted program $P_{\mathbf{abs}}$, the algorithm uses a similar procedure to the best-program-finding routine in Section 3.4: covering each line in the cluster by choosing functions from $\mathcal{L}$. However, instead of using a beam search to find the sequence of functions, here we employ a greedy strategy. We create a preference ordering over the functions of $\mathcal{L}$ based on how many free parameters each function constrains (weighted by their respective λ_τ weights). Then, whenever we need to pick a function, we step through this ordering, until we find a function that is able to match the parameters of at least $p = 70\%$ of the next lines from $\mathcal{P}_{\mathbf{cluster}}$.

For each function added to the abstracted program, we iterate through its parameter slots to see if we can remove more degrees of freedom. For discrete parameters, a constant can be used, a previously defined parameter can be used, or a new free parameter can be declared. For continuous parameters, a constant can be used, an expression over previously defined parameters can be used, or a new free parameter can be declared. The details of this logic can be found in the supplemental material (Section B.1). In all cases, the value chosen for each parameter must still be valid for at least p percent of programs in $\mathcal{P}_{\mathbf{cluster}}$. This process iterates until there are

no remaining uncovered lines in the programs of $\mathcal{P}_{\mathbf{cluster}}$. At this point, $P_{\mathbf{abs}}$ is complete. The green box in Fig. 4 shows an example of finding a single abstracted program for two base programs.

4.3 Proposing Candidate Macros

The abstracted program $P_{\mathbf{abs}}$ found in the previous step represents multiple shape programs from our dataset (via leaving some of its parameters as free parameters). Thus, its function body likely contains re-usable shape programming patterns—in other words, it is a good source of potential macros $\mathcal{M}$ (Algorithm 1, line 8). In this next step, the algorithm iterates through the lines of $P_{\mathbf{abs}}$ and finds all line sequences that could be turned into a valid macro (yellow box in Fig. 4). A valid macro M is a sequence of program lines that simplifies the program, i.e. it must remove some degree of freedom from the program lines it aims to cover. For both computational efficiency, and to encourage the creation of more meaningful macros, we impose some additional restrictions on the definition of a valid macro; see the supplemental material (Section B.2). For each created candidate macro, we record what cluster it was found in and the lines of the cluster it covered, in order to calculate frequency statistics used later in the integration phase (Section 5).

4.4 Generalizing Macros

As $P_{\mathbf{abs}}$ is designed to maximally condense all of the programs in $\mathcal{P}_{\mathbf{cluster}}$, the generated candidate macro operators $\mathcal{M}$ may be somewhat overly-specific to the subset of programs in $\mathcal{P}_{\mathbf{cluster}}$. Furthermore, $\mathcal{M}$ may also contain some very similar macros that are treated as distinct. To get around these issues, the proposal phase concludes with a generalization step, where for each discovered candidate macro, we also find all generalizing macros that are within n program edits (Algorithm 1, line 9). We set $n = 2$ due to running
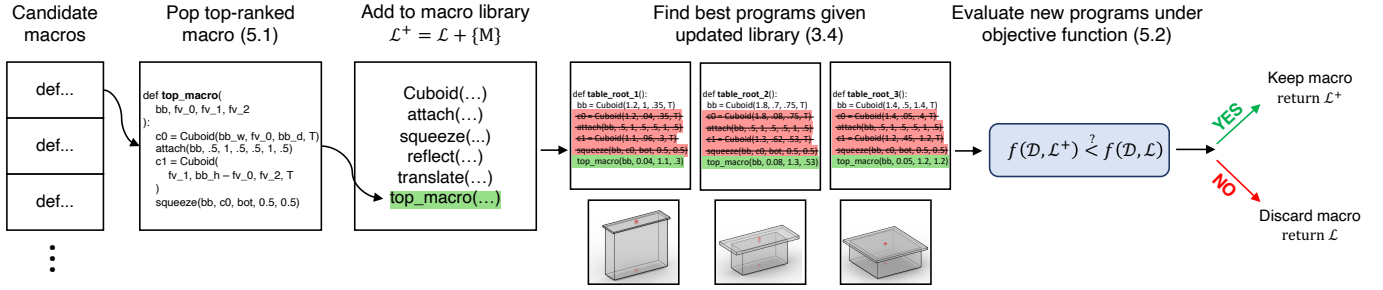
Fig. 5. ShapeMOD's integration phase, which chooses which candidate macros to add to the DSL library $\mathcal{L}$. On each round of this phase, the algorithm heuristically ranks candidate macros based on which are likely to improve program compression, adds the top-ranked macro to the library, then finds the best refactored program for each program in the input dataset $\mathcal{D}$ under this new library. If this refactoring lowers the objective value $f(\mathcal{D}, \mathcal{L})$, then the macro is kept in the library; otherwise, it is discarded.

time constraints; in principle, higher values of n will lead to better solutions. For a given macro M, another macro M' is defined to be generalizing if for every parameterization of M, M' could be parameterized to produce the same output. From this generalization procedure we form a graph where each node is a macro and edges between two nodes indicates a generalizing relationship (orange box in Fig. 4). This graph is used to update frequency statistics (in that generalizing macros also cover all lines covered by macros they generalize) which influences the candidate macro ranking logic used by the integration phase (Section 5).

5 INTEGRATION PHASE

Given candidate macros from the proposal phase, the integration phase chooses which macros to add to the library $\mathcal{L}$ in order to minimize its objective function f. Figure 5 shows an overview. Solving such a subset selection problem optimally is NP-hard, so this phase instead employees a greedy approximation. It iterates through the candidate macro operators, on each iteration taking the highest ranked macro based on expected improvement to f (Section 5.1). It then decides whether to add the macro into the library $\mathcal{L}$ by evaluating its effect on the objective function (Section 5.2).

5.1 Ranking Candidate Macros

The proposal phase can generate tens of thousands of candidate macros; it is computationally intractable to consider all of them. To prioritize which candidate macros to consider within a finite time budget, the algorithm employs a heuristic ranking scheme (Algorithm 1, line 13). The rank of a candidate macro M is based on an estimate of how much using M would improve the score of the objective function. The ranking scheme first calculates the *gain* of the macro over the functions already in $\mathcal{L}$. The gain g of a macro M is the weighted sum of the number of free parameters (weighted by their respective λ_τ weights) that would be removed each time M were used in a program instead of the lowest-cost sequence of functions currently in $\mathcal{L}$ that is equivalent to or generalizes M. Then our ranking scheme calculates the percentage of shapes p that produced M as a candidate macro during the proposal phase. The ranking score of M is then simply $p \cdot g$. This score is a simple estimate of the effect on the actual objective value $f(\mathcal{D}, \mathcal{L} + \{M\})$

that does not require the expensive step of finding the best programs for the whole dataset.

5.2 Evaluating & Selecting Candidate Macros

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

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

5.3 Removing Bad Program Orders

When $\mathcal{L}$ is composed of only original library functions, any valid ordering in O_P for P will lead to a program that produces the same score under f. As macros are added into $\mathcal{L}$, using different line

Table 1. We measure how well different libraries can compress a dataset of shape programs (metric details in Section 6.1). For all compression metrics, lower values are better, as our goal is to find a small collection of functions that remove many degrees of freedom from the underlying shape programs. ShapeMOD operates by attempting to minimize f, and we show that it does in fact improve f compared to the No Macros version.

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

orderings in O_P may result in different scores under f (as some line orders will prohibit certain macros from being applied). As such, after each integration round, the algorithm removes any orders from O_P that lead to objective function scores that are significantly worse (using a threshold of $\tau_o = 1$) then the score produced by the order, o^*; the order that leads to the best objective function score for P (Algorithm 1, line 27). The following proposal rounds will then only be able to use orderings that have not been filtered out of $\mathcal{D}$. Keeping the orderings that perform best during the preceding integration phase produces more accurate heuristic rankings of macros from the proposal phase (Section 5.1). We found this encouraged the discovery of complex macros, e.g. without this step, the 'four leg base' macro was not discovered.

6 RESULTS AND EVALUATION

We experimentally evaluate ShapeMOD's effectiveness at compressing shape programs and at supporting downstream tasks. Our experiments use three categories of manufactured shapes (Chairs, Tables, Storage) from CAD models in PartNet. We use the same data parsing procedure as described in the original ShapeAssembly paper [Jones et al. 2020] to produce 3836 Chair programs, 6536 Table programs, and 1551 Storage programs. In Section 6.1, we examine the properties of ShapeMOD's discovered macros on dataset compression. In Section 6.2, we show that using these macros improves the performance of generative models of 3D shape structures. In Section 6.3, we demonstrate that macros aid in visual program induction tasks. And finally, in Section 6.4, we report the results of a user study comparing performance on goal-directed shape editing tasks with and without discovered macros.

6.1 Discovered Macros

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

the algorithm on a computer with an Intel i9-9900K CPU, which takes 5 hours for Chairs, 12 hours for Storage, and 19 hours for Tables.

Fig. 6 shows examples of some of the macros discovered for Tables; see the supplemental material for complete discovered libraries for all shape categories (Section F). These macros are used by multiple shape programs in our dataset, explaining common patterns and shortening programs that use them. They also better facilitate editing: making edits to a few parameters in macro-refactored programs tends to produce more plausible shape variations than edits to the corresponding parameters of the macro-free program. For instance, discovered macro_1 introduces a relationship that the heights of the table base and the table top should sum to the height of the table bounding box. Without this macro, edits to base ShapeAssembly functions can easily cause the table top to overlap and intersect parts of the table base in an implausible manner (left side of figure).

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

(i) **No Macros**: The base library of functions from ShapeAssembly that is used to initialize our ShapeMOD procedure.

(ii) **Baseline Macros**: A naive single-pass approach for macro discovery that creates macros out of the most common structural sequences present in the dataset and replaces parameters with constants whenever a high percentage of its parameterizations share similar values. See Appendix B for details.

Table 1 compares these baselines to ShapeMOD's discovered language on the task of compressing a dataset of 3D Shape programs. We consider the following metrics:

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

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

The examples shown in Fig. 6 suggest that programs refactored using ShapeMOD macros produce more plausible shapes under variations of their free parameters. We ran an experiment to quantify this behavior. Given a set of ground-truth Chair programs, we run ShapeMOD and our baseline macros procedure on them to create a set of macro-refactored programs. We then perturb the free parameters of both macro refactored and no macro programs by increasingly large perturbations, and we check how distributionally similar the
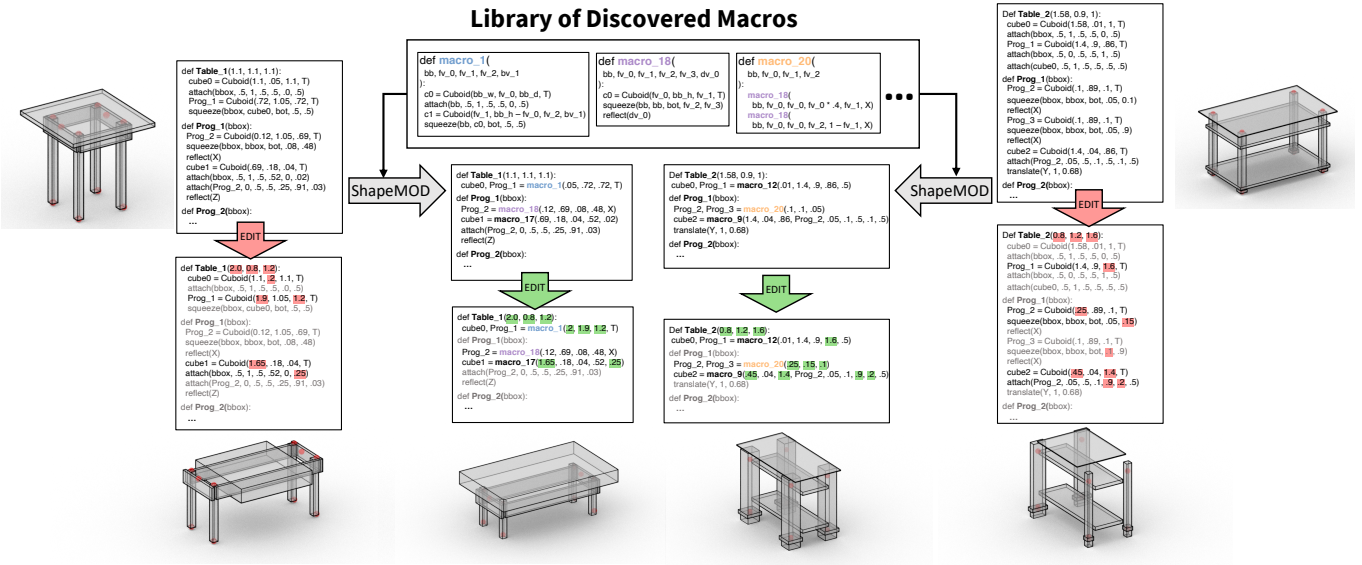
Library of Discovered Macros

Fig. 6. We show some macros (top-middle) that ShapeMOD discovered when run on the Table dataset, and program refactors that use these macros to significantly compress the number of exposed free parameters (ShapeMOD arrows from outside to inside). We show program edits (down arrows) of corresponding parameters in both programs with macros (green) and without macros (red). The discovered macros capture parametric relationships that better preserve shape plausibility under manipulation; for example, all chair legs remain the same size in the third column (macros), while the shape in the fourth column (no macros) becomes disconnected and physically implausible .
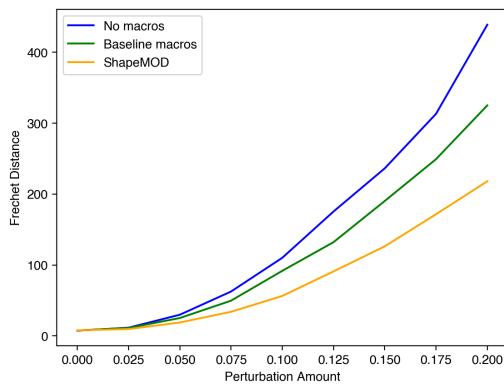

Fig. 7. We measure distributional similarity (Frechet Distance) between a set of reference chairs and a set of chair programs subjected to perturbations. We simulate perturbations by adding noise from a normal distribution (x-axis is σ) to continuous parameters in the programs. Programs with ShapeMOD macros retain more similarity under larger perturbations, suggesting the macros remove degrees of freedom that permit shapes to move outside of their original distribution.

outputs of the perturbed programs are to a held-out validation set of Chair shapes using Frechet Distance [Heusel et al. 2017] in the feature space of a PointNet classifier pre-trained on ShapeNet [Chang et al. 2015; Qi et al. 2017]. Figure 7 plots this distance against the magnitude of parameter perturbation. Frechet Distance increases

more slowly for programs that use macros, and increases the slowest for macros found using ShapeMOD. This indicates that the modes of variation in programs expressed with our method's macros are better at producing plausible output shapes that stay within the distribution that the collection of input programs originally came from. In Section 6.4, we conduct a shape-editing user study to further validate this behavior.

6.2 Generating 3D Shapes

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

For our generative model, we use the variational autoencoder architecture from ShapeAssembly [Jones et al. 2020], modified slightly to support programs that use an arbitrary number of functions as opposed to a fixed, predefined set (see Appendix C for details). We train each model for 5000 epochs with a learning rate of $2e^{-4}$ and a batch size of 64. At the end of training, we choose the model from whichever training epoch produced the lowest Frechet Distance [Heusel et al. 2017] to the training set; we report all other metrics on a held out set. Training was done on a computer with a GeForce RTX 2080 Ti GPU with an Intel i9-9900K CPU, consumed 2GB of GPU memory, and takes approximately 14 hours for Chairs, 22 hours for Tables, and 8 hours for Storage.
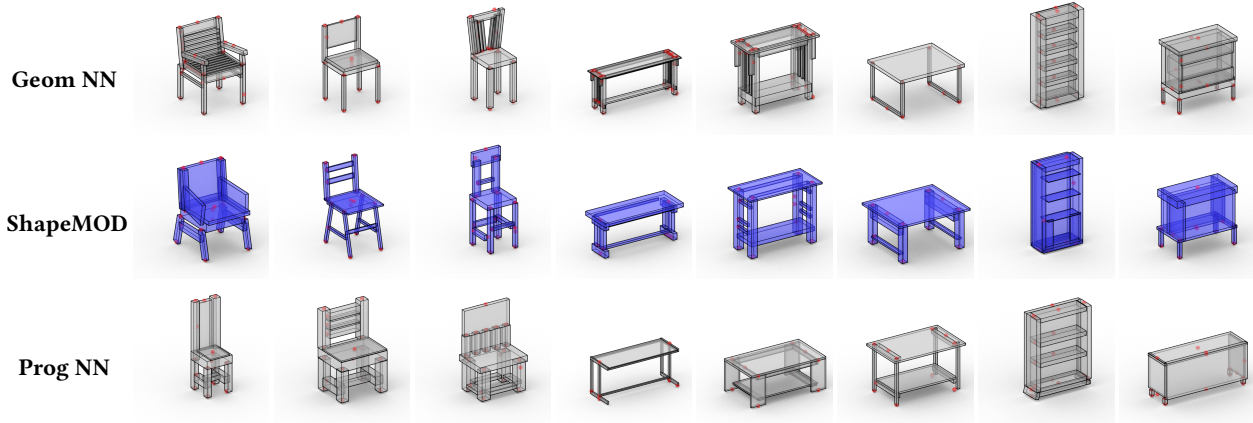
Fig. 8. Some example outputs of generative models trained to produce ShapeAssembly programs expressed with macros discovered by ShapeMOD, along with their training set nearest neighbors (NN) by geometric and program similarity. Each cuboid represents a part proxy bounding volume. Structures are formed through attaching parts to one another (red dots). The generative models produce a variety of plausible structures without memorizing their training data. All corresponding programs can be found in supplemental material.

Fig. 8 shows some examples of novel shapes synthesized by these generative models, as well as their nearest neighbor from the training set according to both program similarity and geometric similarity. The generative models are capable of producing valid, plausible output shapes, and they do not simply memorize their training data.

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

- **Rootedness ⇑ (% rooted):** percentage of shapes whose leaf parts all have a path to the ground.
- **Stability ⇑ (% stable):** percentage of shapes which remain upright when subjected to a small vertical drop.
- **Realism ⇑ (% fool):** percentage of test set shapes classified as "generated" by a PointNet [Qi et al. 2017] trained to distinguish between generated shapes and training set shapes.
- **Frechet Distance ⇓ (FD):** distributional similarity between generated shapes and training set shapes in the feature space of a pre-trained PointNet [Heusel et al. 2017].

Table 2 shows the results of this experiment. Metrics related to realism/plausibility (% fool, FD) are always best for programs that use ShapeMOD macros as opposed to other language variants. Complexity (# Parts) and validity (% rooted, % stable) metrics also generally improve. The simple baseline macros are considerably worse; worse, in fact, than using no macros at all. We provide some qualitative comparisons of generated outputs from ShapeMOD vs No Macros in Appendix E.

6.3 Inferring 3D Shape Structures

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

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

Table 2. Comparing the quality of programs sampled from a learned generative model. Generative models trained on programs with ShapeMOD macros tend to produce more visually plausible, physically valid, and complex shapes than those trained on programs expressed with other libraries.

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

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

Table 3 shows the results of this experiment. As with generative modeling, using ShapeMOD macros results in significantly better performance. Using ShapeMOD macros leads to better reconstruction accuracy, in terms of Chamfer distance and F-score, for all categories (average relative improvement for both is 11%). Moreover, the programs that are inferred with macros also always result in shapes that are more physically valid in terms of stability and rootedness. Fig. 9 shows some example input point clouds and the shapes produced by their inferred programs. Macros help considerably, especially for Storage, which is the most structurally- and

Fig. 9. Example visual program induction results from our point cloud → program inference experiment. ShapeMOD macros are especially helpful for the heterogeneous Storage category. All corresponding programs can be found in the supplemental material.

Table 3. Quantitative results from our visual program induction experiment, where we train encoder-decoder models that learn to infer ShapeAssembly programs from point clouds. ShapeMOD macros regularize the output program space, leading to significant and consistent improvement in both reconstruction accuracy and physical validity. Note: Chamfer Distance (CD) values are multiplied by 1000 for clarity and we use a F-Score threshold of 0.03 [Knapitsch et al. 2017].

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

geometrically-heterogeneous category and thus most likely to cause structured prediction models to output garbage.

6.4 Interactive Shape Editing

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

Editing interface. We designed an interactive editing interface tailored to the goal-directed editing task of modifying a ShapeAssembly program such that its output shape matches a target output shape as closely as possible. Fig. 10 shows our interactive editing interface. The left panel shows the text of the current ShapeAssembly program. The top-right panel shows the current output shape produced by this program; the bottom-right panel shows the target shape.
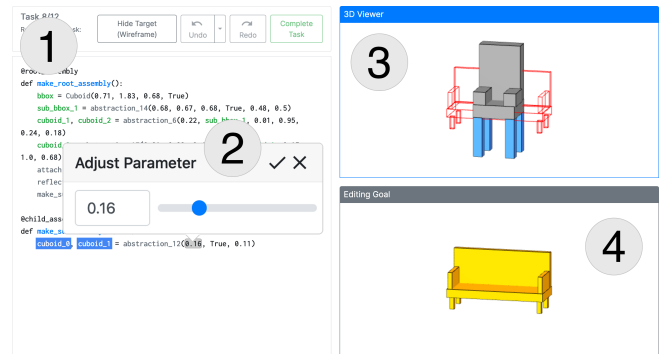

Fig. 10. A screenshot of our editing interface. The key elements are: (1) A view of the ShapeAssembly program's text. (2) Contextual sliders (enlarged in the figure) that allow the user to edit program parameters. (3) A view of the current program's output. Note the optional wireframe of the target shape and the ability to highlight correspondences between cuboids in the text and the 3D viewer (blue highlights shown). (4) The target shape.

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

Experiment design. Our study asked participants to perform a series of goal-directed editing tasks. To ensure that it was possible to complete these tasks, we selected each target shape by finding a program in our dataset that was identical to the input program up to continuous parameters. We recruited 38 participants, all of whom were university students with some programming background. Participants were randomly divided into one of two conditions: editing programs with ShapeMOD macros or programs without them. Participants were not told the meaning of their assigned condition. First, each participant was shown a short tutorial which explained the features of ShapeAssembly and allowed them to become familiar with the editing interface. Then, participants completed six editing tasks (two for each of Chair, Table, and Storage). Participants were given 10 minutes to complete each task. After completing these tasks, participants completed an exit survey which asked them to rate the ease of each task (1-5, with 5 being easiest) as well as to provide qualitative feedback about their experience.

Results. We first ask the question: how long did it take participants to edit the program to produce a close match to the target shape? Fig. 11 plots the running lowest corner distance of the program output to the target shape as a function of task time elapsed, for each of the six study tasks, averaged across participants in each condition. For all tasks, participants using ShapeMOD macros more quickly converged to the target shape.

We also examined the participants' responses to survey questions. Fig. 12 shows the ease rating given to each task, averaged across participants in each condition. For most tasks, participants using ShapeMOD macros rated the task as slightly easier to complete.

6.5 Cross-category Macro Discovery

We also wondered: can one discover useful macros from a dataset consisting of multiple categories of shapes? To answer this question, we ran the ShapeMOD algorithm on the union of our Chair, Table, and Storage datasets, and report full quantitative results in the supplemental material (Section E). Interestingly, the library of functions discovered across multiple categories led to better program compression statistics, but slightly degraded performance on novel shape generation and program inference tasks, compared with libraries discovered by category specific ShapeMOD runs. These experiments show that for downstream tasks it is slightly better to run ShapeMOD on a per-category basis, although the marginal performance gap provides evidence that the discovered macros can generalize.

7 CONCLUSION & FUTURE WORK

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

The abstractions that ShapeMOD currently considers when proposing macros are relatively simple refactorings of free parameters (e.g. into constants or expressions of other variables). The algorithm could be extended to consider other forms of abstraction, for example explaining repeated statements with for loops, use of conditional branches, etc. Such abstractions might allow ShapeMOD to discover even more complex macros, such as new forms of symmetry groups.

As mentioned in Section 5, ShapeMOD's integration step is intractable to solve optimally. But even the greedy approximation we use can be slow for large collections of shape programs. The major computational bottleneck is the cost of finding optimal programs $P^*(\mathcal{D}, \mathcal{L})$. This step could potentially be accelerated via *neurally-guided program search*, i.e. training a neural net which proposes which macros are most promising for explaining which lines of the original programs [Ellis et al. 2018a, 2020; Lu et al. 2019].

ShapeMOD requires programs in a base DSL as input. While some shape datasets exist which are readily convertible into such a format (e.g. PartNet [Mo et al. 2019b]), they are not widely available. It would be interesting to try discovering a shape programming language (complete with macros) from scratch given, for instance, only a set of parts per shape in a shape collection.

While ShapeMOD finds macros that are useful across shape program collections, it does not give them semantic names. In fact, some users in our editing study found the base ShapeAssembly functions easier to work with than the macros for this reason (even though they edited more efficiently with the macros). Finding efficient ways to assign semantic names to automatically-discovered macros would be a fruitful direction for future work.

Finally, while we demonstrated the ShapeMOD algorithm on ShapeAssembly programs in this paper, the method is quite general in principle: it makes no assumptions about its input language, other than that it is imperative and has certain parameter types. Aside from shape modeling, other graphics problems may be expressible in such a language (e.g. shader programming). We are excited to see how ShapeMOD's automatic macro discovery capabilities might find applications elsewhere in computer graphics and beyond.

ACKNOWLEDGMENTS

REFERENCES

Panos Achlioptas, Olga Diamanti, Ioannis Mitliagkas, and Leonidas Guibas. 2018. Learning Representations and Generative Models for 3D Point Clouds. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Jennifer Dy and Andreas Krause (Eds.), Vol. 80. PMLR, Stockholmsmässan, Stockholm Sweden, 40–49. http://proceedings.mlr.press/v80/achlioptas18a.html
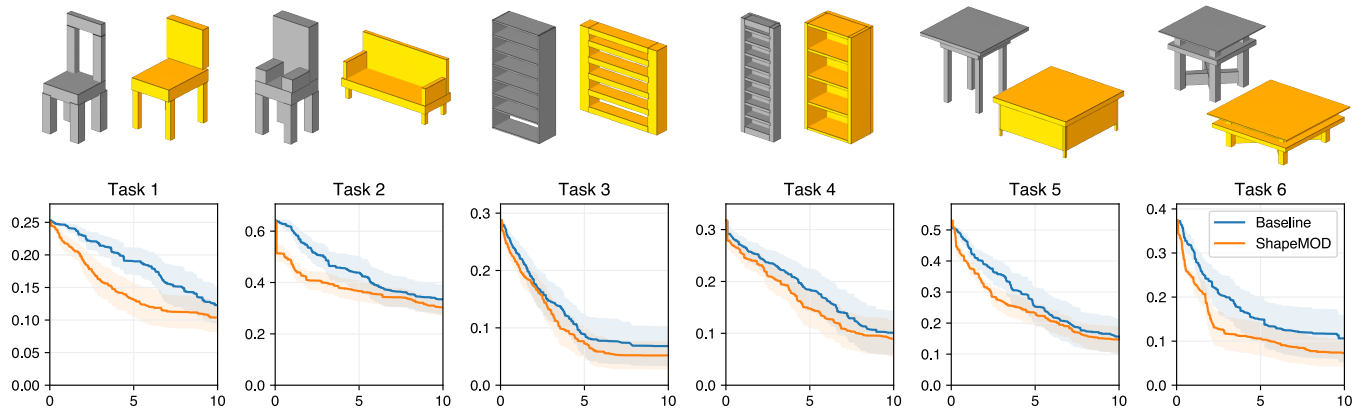
Fig. 11. Top row: the initial program output shape (gray) and target shape (yellow) for each task in our goal-directed editing study. Bottom row: plots of how quickly participants were able to edit a program's parameters to match the target shape, with 95% confidence intervals shown. The x axis is time elapsed in minutes, while the y axis is the mean of the running minimum of each participant's corner distance to the target shape. In general, participants using ShapeMOD macros more quickly converged to the target shape and achieved a closer fit. To allow users to take breaks between tasks, time starts when the user makes their first edit for each task .
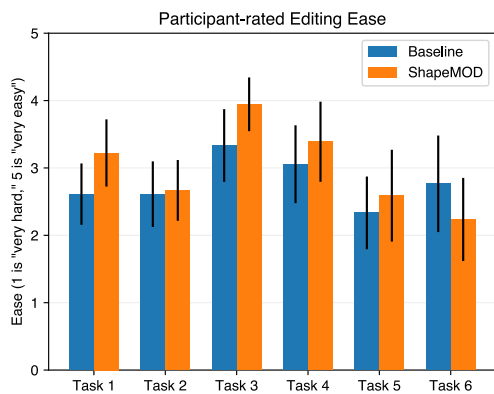

Fig. 12. Participants in our user study rated the ease of completing each task; here, we plot each task's average difficult rating for each condition (5 = very easy, 1 = very difficult) with 95% confidence intervals shown. Participants using ShapeMOD macros generally rated tasks as easier to complete.

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:1512.03012* (2015).

Siddhartha Chaudhuri, Daniel Ritchie, Jiajun Wu, Kai Xu, and Hao Zhang. 2020. Learning Generative Models of 3D Structures. *Computer Graphics Forum* (2020). https://doi.org/10.1111/cgf.14020

Zhiqin Chen and Hao Zhang. 2019. Learning Implicit Fields for Generative Shape Modeling. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

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 (IJCAI '13)*. AAAI Press, 1302–1309.

Boyang Deng, Kyle Genova, Soroosh Yazdani, Sofien Bouaziz, Geoffrey Hinton, and Andrea Tagliasacchi. 2020. CvxNet: Learnable Convex Decomposition. (June 2020).

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. *ACM Trans. Graph.* 37, 6 (Dec. 2018).

Kevin Ellis, Lucas Morales, Mathias Sablé-Meyer, Armando Solar-Lezama, and Joshua B. Tenenbaum. 2018a. Library Learning for Neurally-Guided Bayesian Program Induction. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Josh 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. 2018b. Learning to Infer Graphics Programs from Hand-Drawn Images. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sable-Meyer, Luc Cary, Lucas Morales, Luke Hewitt, Armando Solar-Lezama, and Joshua B. Tenenbaum. 2020. DreamCoder: Growing generalizable, interpretable knowledge with wake-sleep Bayesian program learning. (2020). arXiv:cs.AI/2006.08381

Lin Gao, Jie Yang, Tong Wu, Yu-Jie Yuan, Hongbo Fu, Yu-Kun Lai, and Hao (Richard) Zhang. 2019. SDM-NET: Deep Generative Network for Structured Deformable Mesh. In *SIGGRAPH Asia*.

Kyle Genova, Forrester Cole, Daniel Vlasic, Aaron Sarna, William T Freeman, and Thomas Funkhouser. 2019. Learning shape templates with structured implicit functions. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 7154–7164.

Thibault Groueix, Matthew Fisher, Vladimir G. Kim, Bryan C. Russell, and Mathieu Aubry. 2018. AtlasNet: A Papier-Mâché Approach to Learning 3D Surface Generation. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. 2017. GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium. In *NeurIPS*.

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.

Arno Knapitsch, Jaesik Park, Qian-Yi Zhou, and Vladlen Koltun. 2017. Tanks and Temples: Benchmarking Large-Scale Scene Reconstruction. *ACM Transactions on Graphics* 36, 4 (2017).

Manfred Lau, Akira Ohgawara, Jun Mitani, and Takeo Igarashi. 2011. *Converting 3D Furniture Models to Fabricatable Parts and Connectors*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/1964921.1964980

Jun Li, Kai Xu, Siddhartha Chaudhuri, Ersin Yumer, Hao Zhang, and Leonidas Guibas. 2017. GRASS: Generative recursive autoencoders for shape structures. *ACM Transactions on Graphics (TOG)* 36, 4 (2017), 52.

Sidi Lu, Jiayuan Mao, Joshua B. Tenenbaum, and Jiajun Wu. 2019. Neurally-Guided Structure Inference. In *International Conference on Machine Learning (ICML)*.

A. Martinovic and L. Van Gool. 2013. Bayesian Grammar Learning for Inverse Procedural Modeling. In *CVPR*.

Ravish Mehra, Qingnan Zhou, Jeremy Long, Alla Sheffer, Amy Gooch, and Niloy J. Mitra. 2009. Abstraction of Man-Made Shapes. *ACM Transactions on Graphics* 28, 5 (2009), #137, 1–10.

Niloy Mitra, Michael Wand, Hao (Richard) Zhang, Daniel Cohen-Or, Vladimir Kim, and Qi-Xing Huang. 2013. Structure-Aware Shape Processing. In *SIGGRAPH Asia 2013 Courses.* Article 1, 20 pages.

Kaichun Mo, Paul Guerrero, Li Yi, Hao Su, Peter Wonka, Niloy Mitra, and Leonidas Guibas. 2019a. StructureNet: Hierarchical Graph Networks for 3D Shape Generation. In *SIGGRAPH Asia.*

Kaichun Mo, Shilin Zhu, Angel X. Chang, Li Yi, Subarna Tripathi, Leonidas J. Guibas, and Hao Su. 2019b. 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, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. 2006. Procedural Modeling of Buildings. In *SIGGRAPH.*

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 (PLDI 2020).* Association for Computing Machinery, New York, NY, USA, 31–44. https://doi.org/10.1145/3385412.3386012

Yoav I. H. Parish and Pascal Müller. 2001. Procedural Modeling of Cities. In *SIGGRAPH.*

Przemyslaw Prusinkiewicz and Aristid Lindenmayer. 1996. *The Algorithmic Beauty of Plants.* Springer-Verlag, Berlin, Heidelberg.

Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. 2017. Pointnet: Deep learning on point sets for 3D classification and segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* 652–660.

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

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).*

Chun-Yu Sun, Qian-Fang Zou, Xin Tong, and Yang Liu. 2019. Learning Adaptive Hierarchical Cuboid Abstractions of 3D Shape Collections. *ACM Trans. Graph.* 38, 6, Article 241 (Nov. 2019), 13 pages. https://doi.org/10.1145/3355089.3356529

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

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).*

Shubham Tulsiani, Hao Su, Leonidas J. Guibas, Alexei A. Efros, and Jitendra Malik. 2017. Learning Shape Abstractions by Assembling Volumetric Primitives. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR).*

Homer Walke, R. Kenny Jones, and Daniel Ritchie. 2020. Learning to Infer Shape Programs Using Latent Execution Self Training. (2020). arXiv:cs.CV/2011.13045

Karl D. D. Willis, Yewen Pu, Jieliang Luo, Hang Chu, Tao Du, Joseph G. Lambourne, Armando Solar-Lezama, and Wojciech Matusik. 2020. Fusion 360 Gallery: A Dataset and Environment for Programmatic CAD Reconstruction. *arXiv preprint arXiv:2010.02392* (2020).

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

Jiajun Wu, Chengkai Zhang, Tianfan Xue, William T. Freeman, and Joshua B. Tenenbaum. 2016. Learning a Probabilistic Latent Space of Object Shapes via 3D Generative-Adversarial Modeling. In *Advances in Neural Information Processing Systems (NeurIPS).*

Jie Yang, Kaichun Mo, Yu-Kun Lai, Leonidas J. Guibas, and Lin Gao. 2020. DSM-Net: Disentangled Structured Mesh Net for Controllable Generation of Fine Geometry. (2020). arXiv:cs.GR/2008.05440

Mehmet Ersin Yumer and Levent Burak Kara. 2012. Co-Abstraction of Shape Collections. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2012)* 31 (2012), 166:1–166:11. Issue 6.

Chuhang Zou, Ersin Yumer, Jimei Yang, Duygu Ceylan, and Derek Hoiem. 2017. 3D-PRNN: Generating Shape Primitives with Recurrent Neural Networks. In *IEEE International Conference on Computer Vision (ICCV).*

Table 4. Modified grammar of ShapeAssembly [Jones et al. 2020].

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

A   MODIFIED SHAPEASSEMBLY GRAMMAR

Table 4 shows the modified grammar for ShapeAssembly that we use. We make the following changes from the ShapeAssembly version presented in [Jones et al. 2020]. Instead of having separate blocks where all cuboids are defined, then all attaches are defined, and then finally all symmetry operators are defined, we interleave the attach / symmetry commands with the cuboids they move. Specifically a program starts with defining a bounding volume, and then is followed with a series of PBlocks. Each PBlock defines a Cuboid, attaches it to at least one previous cuboid (or the bounding volume), and optionally applies a symmetry operation to it. We find that this ordering permits the discovery of more interesting and useful macros, as otherwise macros would mostly be made up of only Cuboid definitions or only attachments (instead of a mix of operators). As a by-product of this new ordering, we assume that all non-Cuboid operators (attach, squeeze, reflect, translate) always operate on the last defined cuboid, and so in this way we remove one cuboid index parameter from each of these functions.

B   BASELINE METHOD FOR MACRO OPERATOR DISCOVERY

Designing a baseline for ShapeMOD is non-trivial, because there do not exist any existing methods that are able to find macro operators over datasets of programs written in imperative languages that contain continuous parameters. Thus, we present a naive single-pass algorithm that mimics a simplified version of ShapeMOD's core logic. It starts by choosing one order for each program in the dataset. Specifically, the most canonical order, as defined in the supplemental material (Section A.3). Then it records all subsequences of functions that appear in the resulting program lines. If any subsequence is observed in more than 10% of programs in the dataset, then it is turned into a macro function. Parameters of this macro function can be converted from free parameters to constants if at least 90 % of the parameterizations of this subsequence across the dataset had the same value (for discrete parameters) or were within .05 range of the mean value (for continuous parameters). Once these macros have been discovered, we use the best program finding step from ShapeMOD to create a dataset of programs expressed with macros discovered by the baseline method. As shown throughout the results section, the macros discovered by ShapeMOD outperform

the macros discovered by this baseline method, for every task we consider.

C A NETWORK ARCHITECTURE FOR ANY LIBRARY

After running our procedure to generate a library L, we want to design a neural network that is able to generate programs using the functions of L. As our procedure is able to produce many different libraries L, depending on which macro operators it discovers, our network architecture must be flexible enough to model any set of discovered functions. To demonstrate that this is achievable, we generalize the neural network from [Jones et al. 2020] so that it is able to learn how to generate programs expressed in any L discovered through our procedure, and validate this works in later experiments.

The base model is a hierarchical sequence VAE. The encoder branch ingests a hierarchical program and embeds it into a high dimensional latent space. The decoder branch converts a code from this latent space into a hierarchical program. Originally, the underlying library was fixed to ShapeAssembly, so the network architecture and input representation could be tailored to one set of functions.

We design a generalized version of this network architecture that is customized based on the library of functions L discovered by our procedure. The parts of the architecture that had to be generalized were the tensor line representation and the sub-networks in the line decoder module.

In our new line representation, the dimension of the line tensor and meaning of each index changes depending on L. The first $|L| + 2$ indices of the tensor correspond to a one hot vector denoting the function type of each line (notice we add special START and STOP tokens). Then for each type of discrete parameter, p_d, we find its number of valid values, p_{d_size}, and maximum number of p_d free parameters in any function of L, p_{d_free}. We then reserve p_{d_free} slots of size p_{d_size} in our tensor for p_d, where each slot corresponds to a one hot vector whenever p_d is required by a function. Finally, for any function $f \in L$ that takes in a set of continuous parameters, f_c, we reserve a slot in our tensor of size $|f_c|$.

The number and structure of sub-networks in our new line decoder model also depends on L. The M_{func} module is responsible for predicting the line's function, and therefore has $|L| + 2$ possible outputs (the functions of L and the special START and STOP tokens. For each $f \in L$, for each of its free discrete parameters f_{d_i}, we add a sub-network $M_{f\_d\_i}$ responsible for predicting the ith discrete parameter of f. Then, for every f that has free continuous parameters, we add a sub-network M_{f_c} for predicting the continuous parameters of f.

We implement each sub-network as a 3 layer MLP. The network is trained in a teacher forcing paradigm with a cross entropy loss for all discrete predictions and an l1 loss for all continuous predictions. Parameter sub-networks are invoked, and tensor slots in each line are filled, depending on the function type predicted in each output line. Otherwise we use the same hyper-parameters as in Jones et al. [2020]

Table 5. Evaluating the convergence properties of ShapeMOD under different ablation variants. Lower values of f are better.

Method	$f \Downarrow$
ShapeMOD (5 rounds)	**68.1**
ShapeMOD (3 rounds)	72.0
ShapeMOD (1 round)	82.7
No $\mathcal{P}_{\mathbf{matches}}$ Dist (5 rounds)	68.4
No Beam Search (5 rounds)	71.6
No Bad Order Filter (5 rounds)	73.6
No Generalize Macros (5 rounds)	79.3
No Valid Macros Criteria (5 rounds)	91.5

D SHAPEMOD CONVERGENCE PROPERTIES

To characterize how different design decisions affect the convergence properties of ShapeMOD, we ran an ablation experiment (Table 5). Each variant was tasked with discovering a library of macros for the dataset of chair programs, and we track how well each variant's library was able to compress the dataset according to ShapeMOD's objective function f (Section 3.3). The "$\mathcal{P}_{\mathbf{matches}}$ Dist" variant replaces the sampleByParamSim function (Alg 1, line 6) with a uniform sample over matching programs. The "No Beam Search" variant replaces the beam search step from Section 3.4 with a best-first search approach. The "No Bad Order Filter" variant removes the filterBadOrders function (Alg 1, line 27). The "No Generalize Macros" variant removes the generalize function (Alg 1, line 9). The "No Valid Macro Criteria" variant accepts any proposed macro function, ignoring the validity criteria from Section B.2 of the supplemental. Notice that the full version of ShapeMOD achieves the lowest objective function value compared with all of the other variants, and that the discovered library of macros does improve through multiple rounds of the algorithm (top three rows).

E SHAPE GENERATION QUALITATIVE COMPARISON

We share some interesting representative shape programs output by learned ShapeAssembly generative models in Figure 13. Outputs by the model trained with ShapeMOD macros are shown in blue. Outputs by the model trained on the No Macros programs are shown in green. We include all generated program text in the supplemental materials. These qualitative results enforce the trends of our earlier quantitative experiments from Section 6.2. The best generations from Chairs and Tables are qualitatively similar, although across entire shape collections we calculated that programs with ShapeMOD macros were more plausible. For storage, the qualitative difference is more pronounced, as the generations that use ShapeMOD macros are able to create output shapes that are much closer in distribution to the target shape collections.
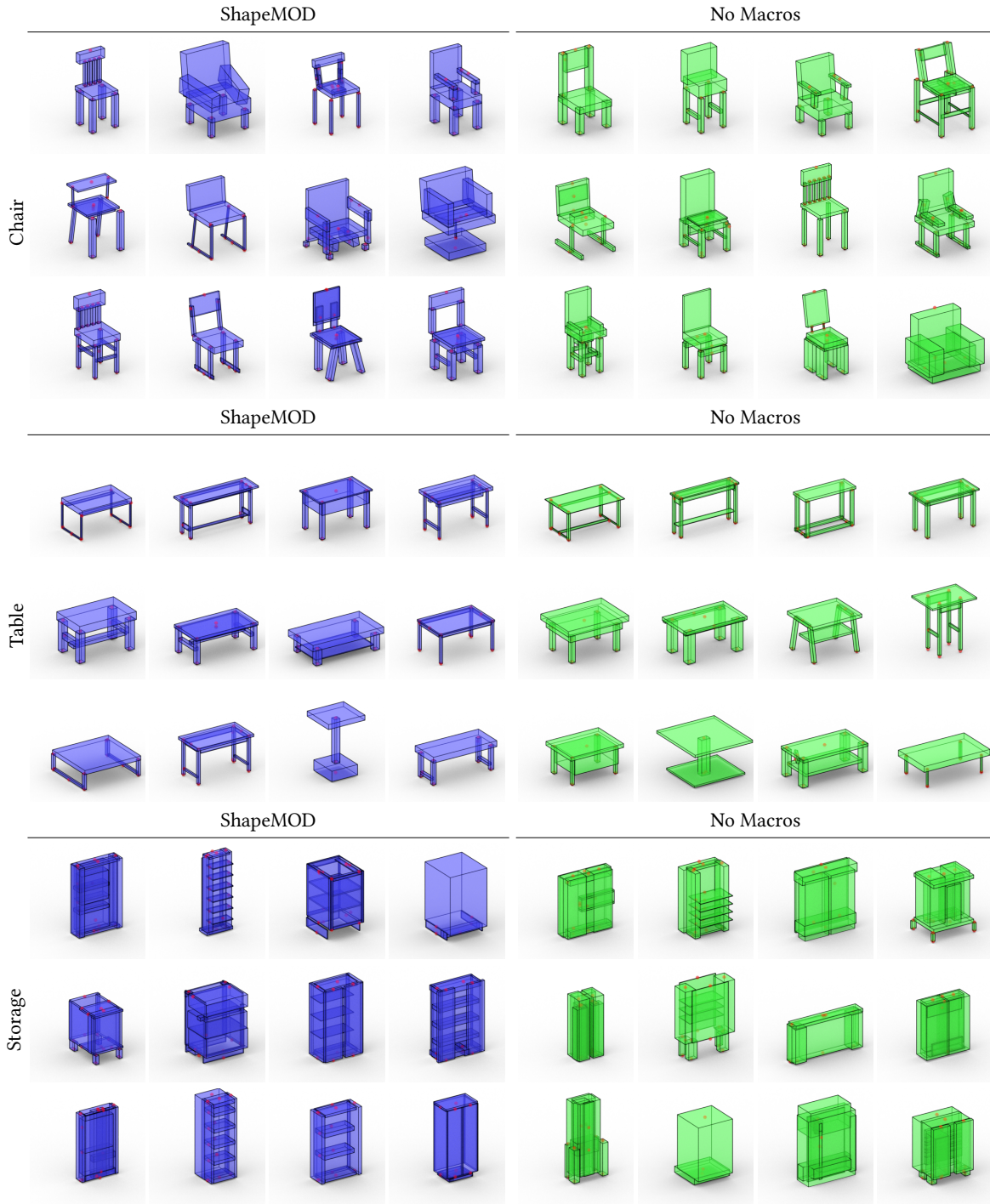
Fig. 13. Samples generated from generative models of ShapeAssembly programs with ShapeMOD macros (blue) and without macros (green).