Improving Unsupervised Visual Program Inference with Code Rewriting Families –Supplementary Material–

Aditya Ganeshan R. Kenny Jones Daniel Ritchie
Brown University
adityaganeshan@gmail.com

In this document, we provide additional details and experimental results for our proposed method *Sparse Intermittent Rewrite Injection*. The supplementary material is divided into the following sections:

- **Additional Experiments:** We provides additional comparison to CSGStump [9] in section 1.1. We report our experiments on advanced versions of CSG and ShapeAssembly in Section 1.3, and Section 1.4 reports some additional analysis on SIRI.
- **Implementation Details:** In Section 3, we elaborate on the different Domain Specific Languages (DSLs). Additionally, from section 4.1 - 4.3, we provide implementation details of the three proposed rewriters.
- **Qualitative Results:** We present various qualitative comparison between CSGStump [9], PLAD [5] and our method in section 5.

1. Additional Experiments

1.1. Comparison to CSG-Stump

Program Complexity: In Figure 1, we visualize the programs inferred by CSG-Stump 32 and SIRI as Geometry Node graphs in Blender [1], a popular tool for procedural modelling. We can see that CSG-Stump 32, despite being the smallest of the models (vs CSG-Stump 256), still produces highly complex graphs with a myriad of nodes and links. Editing such programs to create shape variations is arduous. Additionally, the program is hard to interpret due to its highly connected nature. Such complex program graphs lose a fundamental advantage of the programmatic representation: interpretability and editability.

Test-time rewriting with CSG-Stump: Our work proposes three rewriters which can improved bootstrapped learning as well as perform test time rewriting (TTR). An important question is whether TTR can improve programs inferred by domain specific networks such as CSG-Stump [9]. To answer this question, we perform test time rewriting with our rewriters on programs inferred by CSG-Stump. We report the results in Table 1. Note that to limit the optimization time, we apply our rewriters on CSG-Stump 32 model (a model with 32 primitives, and 32 intersection nodes).

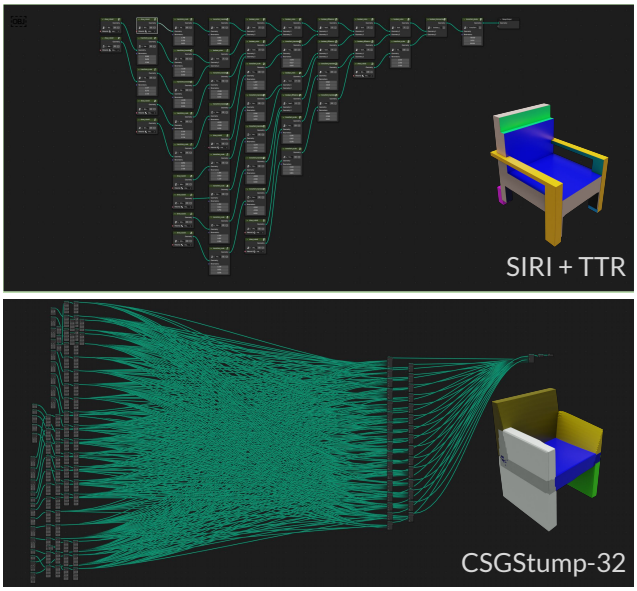

Figure 1. We show the 3D CSG programs inferred by different methods as Geometry Node trees in Blender [1]. Programs inferred by even the smallest CSGStump [9] model (with 32 primitives and 32 intersection nodes) is highly complicated. In comparison, SIRI programs are parsimonious and hence easier to edit.

First, we notice that TTR indeed improves the reconstruction quality (as measured by IoU and median CD) as well as the program length for CSG-Stump as well. However, we do note that mean CD spikes up due to some poorly optimized programs. Second, we note that due to the large programs inferred by CSG-Stump (even for the smallest model), TTR takes an order of magnitude more time than it takes with SIRI inferred programs. Both of these trends in-

TTR	Model	IoU-32 (↑)	IoU-64 (↑)	mean CD (↓)	med. CD (↓)	Length (↓)	Time (↓)
∅	CSG-Stump 32	49.88	39.83	1.88	1.58	98.94	0.05
3	CSG-Stump 32	73.83	54.34	2.56	1.03	63.58	133.88
∅	SIRI	76.77	54.61	1.12	0.69	6.81	1.44
3	SIRI	90.59	60.60	0.83	0.51	15.96	14.60

Table 1. We perform test time rewriting on programs inferred by CSG-Stump 32 [9] with our proposed rewriters. While it improves the reconstruction accuracy and decreases the program length, it still under performs SIRI. Additionally, applying TTR to CSG-Stump 32 inferred programs takes a order of magnitude more time than applying TTR to SIRI inferred programs.

	IoU (↑)	CD (↓)	Length (↓)
SIRI	**78.63**	**1.13**	6.81
No Sparse	77.94	1.28	6.98
Single Rewrite Queue	77.45	1.34	**6.15**
Single Queue	77.63	1.25	6.53
NRI	75.91	1.29	8.98

Table 2. We compare variants of SIRI to validate its design. SIRI outperforms the baselines, and shows that *sparse* rewrite usage, as well as *careful* rewrite injection (via source-separated queues) are essential to integrate rewriters into bootstrapped learning processes.

	IoU (↑)	CD (↓)	Length (↓)
SIRI	**78.63**	**1.13**	6.81
no *PO*	77.80	1.27	6.10
no *CP*	77.31	1.21	6.18
no *CG*	76.21	1.27	**5.58**
PLAD	76.21	1.43	6.39

Table 3. We report reconstruction accuracy as well as code-quality on 3D CSG+ with each rewriters individually removed. Using all the three rewriters (top-row) yields the best reconstruction and code-quality. Using no rewriters is equivalent to PLAD (bottom-row).

dicate that producing sparse programs from start (via bootstrapped methods) may be preferable over producing over-parameterized programs (via domain specific networks).

1.2. Ablation

We now provide an ablative analysis of our method to verify the importance of each component. For consistency, we perform all ablations on the 3D CSG language and report metrics on the validation set.

SIRI ablation: We perform a subtractive analysis on SIRI by changing/removing individual components of the method. We compare SIRI to three alternatives:

1. **No Sparse:** Instead of applying to rewriters to a subset of programs, they are applied to all programs.
2. **Single Rewrite Queue:** Instead of storing separate queues for each rewriter via the source mapping S_{PO}, S_{CP}, S_{CG}, we store a single queue S_R shared between all the rewriters.
3. **Single Queue:** We remove the separate sources $S_{NS}, S_{PO}, S_{CP}, S_{CG}$, and simply store the top-k ($k = 3$) programs for each input shape x.

We report the results in Table 2. SIRI surpasses all the other ablations. At the same time, all the ablations are able to surpass the naive rewriter integration NRI, albeit with a smaller margin. This shows that *sparse* rewrite usage, as well as *careful* rewrite injection (via source-separated queues) help integrating the rewriters into bootstrapped learning processes.

Rewriter Ablation: Next, we test whether using all the three proposed rewriters *PO*, *CP*, *CG* is essential. Table 3 compares SIRI to models trained with one rewriter removed each. We see that using all the three rewriters improves the performance.

1.3. Advanced Languages

To take a step towards supporting languages closer to those used in real-world scenarios, we extend 3D CSG and ShapeAssembly to contain more complex features (described in detail in Section 3), and perform additional experiments on these languages. Note that while the extensions are simple under our general framing, domain-specific networks such as CSG-Stump [9] and UCSG-Net [6] require non-trivial changes in their architecture to support such extensions.

Bootstrapped Learning: We train the unsupervised VPI network with PLAD, NRI and SIRI. We report the results in Table 4. On both the advanced languages, SIRI outperforms the baselines. More importantly, a naive integration of the rewrites (i.e. NRI) is detrimental on both the languages. This experiment emphasizes the complexity of integrating rewrite mechanisms into learning frameworks. By using the rewriters sparsely and carefully injecting rewritten programs into the training data, SIRI is able to improve

	3D CSG+			ShapeAssembly+		
	IoU (↑)	CD (↓)	Length (↓)	IoU (↑)	CD (↓)	Length (↓)
PLAD	70.51	2.35	**9.09**	61.22	2.63	**8.51**
NRI	68.25	2.45	10.57	58.82	2.74	10.31
SIRI	**71.48**	**2.17**	9.99	**63.91**	**2.41**	9.24

Table 4. We report the Test-set performance across advanced versions of 3D CSG (3D CSG+) and ShapeAssembly (ShapeAssembly+). As a result of their higher complexity, we see performance on advanced languages is lower than performance on simple languages. However, that trend observed in the main draft is seen here as well - naively integrating the rewriters into PLAD (NRI) can deteriorate the model's performance, and SIRI consistently outperforms both the baselines.

TTR	3D CSG+			ShapeAssembly+		
	IoU	CD	Length	IoU	CD	Length
PLAD	78.5	1.68	**15.9**	71.82	2.17	**11.35**
SIRI	**82.8**	**1.48**	20.3	**74.65**	**1.92**	11.47

Table 5. We apply test time rewriting for models trained on the advanced languages. Reaffirming the results in the main draft, we see that using test-time rewrites with SIRI remains superior to using it with PLAD.

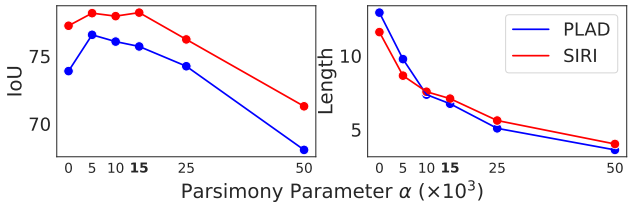

Figure 2. We compare PLAD and SIRI for different values of α (X-axis), measuring the IoU (*left*), and program-length (*right*) on the validation set of the 3D CSG domain. SIRI consistently dominates PLAD, and setting $\alpha = 0$ harms both reconstruction accuracy and conciseness.

bootstrapped learning through the rewriters.

Test Time Rewriting: We perform test time rewriting (TTR) on both 3D CSG+ and ShapeAssembly+ domains with models trained with PLAD and SIRI. Our results are tabulated in Table 5. Similar to the results in the main draft, test time rewriting is more effective on programs inferred by SIRI than by PLAD.

1.4. Other Experiments

Sensitivity to α: A key ingredient in the objective $\mathcal{O}$ is the α factor balancing the weightage between reconstruction accuracy and program length. In figure 2, we compare the validation-set performance of PLAD and SIRI on 3D CSG for different values of the alpha parameter. We note that across the different settings, SIRI consistently outperforms PLAD in reconstruction accuracy (IoU).

Understanding NRI failure: Why does NRI fail? To answer this question, we study additional training statistics, namely the trained network's train-set performance, val-set performance, and quality (as measured by our objective $\mathcal{O}$) of the training data created via the *Search* and *Rewrite* phases. We show the corresponding graphs in Figure 3.

One may assume that NRI performs poorly due to overfitting to the training data. However, we see that this is not the case - inference performance on both the training-set as well as validation-set for NRI is lower than SIRI. As PLAD-finetuning performs early-stopping, with weight-reloading before *Search* phase, it avoids overfitting to the training data. Another hypothesis may be that the programs found for the training shapes might be bad matches. However, we see that this is also not the case from the 'Execution of train shapes' plot. As NRI only trains on the "best programs" seen thus far, the training data quality, as measured by the objective $\mathcal{O}$, is significantly higher than SIRI. However, despite the higher reconstruction matches against their target shapes, the model trained with NRI does not generalize from these training programs and fails to perform well on the validation-set, resulting in performance stagnation for NRI. In contrast, SIRI is able to perform well on the validation set, despite training on programs which have lower reconstruction matches versus their target shapes. This indicates that for bootstrapped learning, simply maximizing the reconstruction accuracy of training programs may not be sufficient: instead it is necessary to find programs for shapes in the training set that both (i) produce good reconstructions, and (ii) help the inference network to learn policies that generalize to held-out shapes from the same-distribution (validation set). We find that the excessive use of rewriters in NRI violates assumption (ii), but we hope to explore this phenomena more in future work.

2. Evaluation details

To infer programs from the networks learned via bootstrapped learning processes (PLAD, NRI, SIRI), we perform neurally guided-beam search with a beam-size of 10. For each input shape, this search generates a candidate pool of programs, from which, the objective maximizing pro-

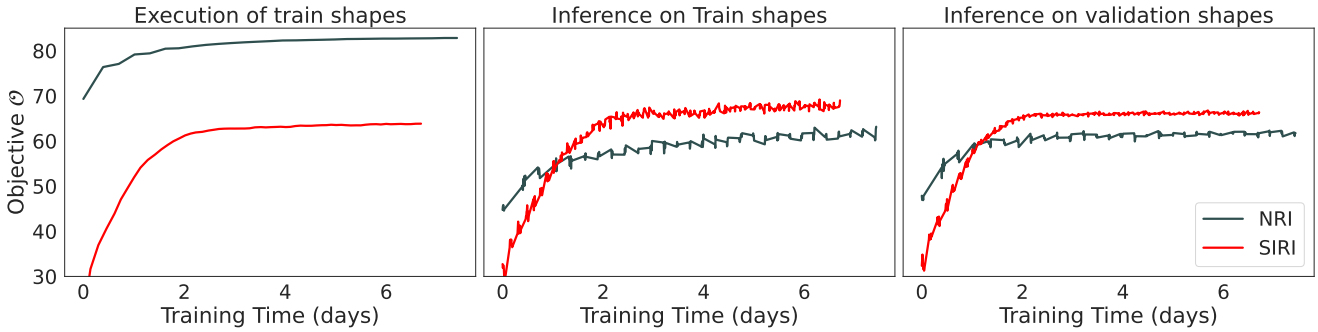

Figure 3. To understand why NRI fails, we analyse the training process for NRI trained and SIRI trained models. We show the (a) execution of train shapes (b) inference on train shapes and (c) inference on validation shapes. NRI does not overfit to the training data, and while its training data is of high quality (i.e. their execution yields high value of objective $\mathcal{O}$), its train and validation inference performance remain low (cf. section 1.4 for more details).

gram is selected as the inferred program. This approach is the same as the one used in PLAD [5]. For CSG-Stump, a forward pass through the network produces a single probabilistic program. The inferred program is then generated by simply replacing each probabilistic parameter with the modal value of its distribution (for instance, the argmax of each categorical distributions).

Given a set of input shapes, we measure the quality of inferred programs along two axis: a) reconstruction accuracy, to measure how well the inferred program's execution reconstructs the input shape, and b) program parsimony, to measure how concise the inferred programs are (motivated by the Occam's razor principle).

Reconstruction metrics: The execution of programs results in a $\mathcal{R}^n$ grid of occupancy values (where $n = 2$ for 2D and $n = 3$ for 3D). As the input is also in the form of a $\mathcal{R}^n$ grid of occupancy values, we can directly measure the Intersection over Union between the inferred occupancy and ground-truth occupancy by comparing them. For measuring Chamfer Distance, on 2D data we follow CSG-Net [10], and on 3D data we follow CSG-Stump [9].

Program Parsimony: Program length is measured as the number of statements/commands in a program. Note that this differs from the number of discrete tokens required to define a command/statement (for instance, a Cuboid command in 3D CSG requires 7 tokens, 1 to specify command type, 3 to specify the 3D position, and 3 to specify the 3D scale).

3. DSL Grammars

3.1. Simple DSLs

2D CSG: For 2D CSG, each primitive is instantiated with 5 parameters, specifying its 2D position, 2D scale and rotation. We specify the grammar as follows:

$$
\begin{aligned}
&S \rightarrow E; \\
&E \rightarrow BEE \mid D(F,F,F,F,F); \\
&B \rightarrow intersect \mid union \mid subtract; \\
&D \rightarrow rectangle \mid ellipse; \\
&F \rightarrow (-1,1);
\end{aligned}
$$

In the $D(F,F,F,F,F)$ command, the first two real numbers F specify the translation, the next two specify scaling, and the last specifies rotation. F is mapped to different ranges for these different operations. For translation, F remains as is, for scaling F is linearly mapped from $(-1,1)$ to $(0.01, 2.01)$, for rotation, we linearly map F from $(-1,1)$ to $(-\pi,\pi)$.

3D CSG: 3D CSG matches the language used in PLAD [5]. For 3D CSG, each primitive is defined by 6 parameters, specifying its 3D position, and 3D scale (no rotation). The range mapping of real number language tokens F for the different operations followed for 2D CSG is applied here as well. We specify the grammar as follows:

$$
\begin{aligned}
&S \rightarrow E; \\
&E \rightarrow BEE \mid D(F,F,F,F,F,F); \\
&B \rightarrow intersect \mid union \mid subtract; \\
&D \rightarrow cuboid \mid ellipsoid\ ; \\
&F \rightarrow [-1,1];
\end{aligned}
$$

ShapeAssembly: ShapeAssembly creates structures by instantiating cuboids, and attaching them to one another. We utilize the version of ShapeAssembly described in PLAD [4] with a slight modification. Specifically, in our version `attach` and `squeeze` commands are defined for 3D points rather than 2D surface points. This makes our version of ShapeAssembly closer to the original defined in ShapeAssembly [3]. Following PLAD, we restrict the bounding

box to always have $l = 1, w = 1$, and remove the axis-alignment flag from cuboid instantiation. Furthermore, for all sub-programs we restrict the bounding box to always have $h = 1$ as well. We specify the grammar for ShapeAssembly as the following:

$$
\begin{aligned}
&Start \rightarrow BBlock; CBlock; ABlock; SBlock;\\
&BBlock \rightarrow \texttt{bbox} = \texttt{Cuboid}(l, h, w)\\
&CBlock \rightarrow c_n = \texttt{Cuboid}(l, w, h); CBlock \mid None\\
&ABlock \rightarrow A; ABlock \mid S; ABlock \mid None\\
&SBlock \rightarrow R; SBlock \mid T; SBlock \mid None\\
&A \rightarrow \texttt{attach}(c_{n_1}, x_1, y_1, z_1, x_2, y_2, z_2)\\
&S \rightarrow \texttt{squeeze}(c_{n_1}, c_{n_2}, f, u, v)\\
&R \rightarrow \texttt{reflect}(\text{axis})\\
&T \rightarrow \texttt{translate}(\text{axis}, m, d)\\
&f \rightarrow right \mid left \mid top \mid bot \mid front \mid back\\
&\text{axis} \rightarrow X \mid Y \mid Z\\
&l, h, w \in \mathbb{R}^+\\
&x, y, z, u, v, d \in [0, 1]^2\\
&n, m \in \mathbb{Z}^+
\end{aligned}
$$

3.2. Advanced DSLs

To ease learning, past approaches have used *simplified* versions of visual languages as described in the previous section. However, real world visual programs offer advanced operations, e.g. real world CSG programs contain hierarchical transformations (rather than primitive level transformations). As a step towards such languages, we also test our method on *advanced* 3D CSG (**3D CSG+**) and ShapeAssembly (**ShapeAssembly+**).

3D CSG+: we introduce two important classes of commands - *hierarchical transformations*, transformations that can be applied to compound shapes, and *axis-aligned reflection*. Further, to reduce parameter redundancy, we instantiate primitives without any parameters, i.e., they are instantiated in a canonical form (origin centered, and unit scale with no rotation). The grammar is defined as follows:

$$
\begin{aligned}
&S \rightarrow E;\\
&E \rightarrow BEE \mid TE \mid D;\\
&B \rightarrow intersect \mid union \mid subtract;\\
&D \rightarrow cuboid \mid ellipsoid \mid cylinder;\\
&T \rightarrow translate(F, F, F) \mid scale(F, F, F) \mid\\
&\quad\ rotate(F, F, F) \mid R;\\
&R \rightarrow reflect(X) \mid reflect(Y) \mid reflect(Z);\\
&F \rightarrow [-1, 1];
\end{aligned}
$$

ShapeAssembly+: We extend the simple ShapeAssembly described previously by allowing hierarchical composition of programs. Specifically, our version allows hierarchical programs, where cuboids in the root program can act as the bounding box for their own ShapeAssembly programs. ShapeAssembly+ follows the same grammar as ShapeAssembly with the following change:

$$
\begin{aligned}
&CBlock \rightarrow c_n = \texttt{Cuboid}(l, w, h, \mathbf{sp}); CBlock \mid None\\
&\mathbf{sp} \in \mathbb{Z}^+
\end{aligned}
$$

During cuboid instantiation, parameter $\mathbf{sp}$ specifies whether the cuboid is empty ($sp = 0$) or contains the sp-th ShapeAssembly program (all the ShapeAssembly programs are decoded by the inference network).

4. Code Rewriters

Our rewriters are formulated to be generally applicable across visual programming languages. In this section, we outline the requirements to apply each rewriter, along with implementation details and a walkthrough example.

4.1. Parameter Optimization (PO)

The *Parameter Optimization* (*PO*) rewriter aims to improve the continuous parameters of a given program while keeping its discrete structural parameters fixed. Furthermore, it aims to use first-order gradient based optimization to update these parameters. Towards this end, *PO* requires that the programs derived from the language grammar must be continuous and piecewise differentiable with respect to ≥ 1 program parameters ϕ.

Additionally, as *PO* performs gradient based optimization, it requires a *partially* differentiable executor for the language (differentiable w.r.t. ϕ at least). By chaining a differentiable reconstruction measure $\mathcal{R}$ (such as L_2-loss w.r.t. a occupancy grid) to the program's differentiable execution, we can then optimize the program parameters ϕ to improve the reconstruction measure $\mathcal{R}$.

Most shape languages instantiate parameterized primitives and combine them with different combinators to create shapes. When languages can map the program parameters p to primitive parameters in a piecewise differentiable manner (as in ShapeAssembly [3], and CSG), we can yield a *partially* differentiable executor D for the language with the procedure outlined in the paper (cf. Section 4.1). We revisit the procedure here with more details.

$D(z)$ maps the execution of program to an equivalent implicit sign-distance function. Given a program z_ϕ, we use the programs executor E to map program parameters ϕ to parameters (position, scale, rotation) of implicit functions representing simple geometric primitives (cuboids, spheres etc.). For CSG, the mapping is 1-to-1, but for

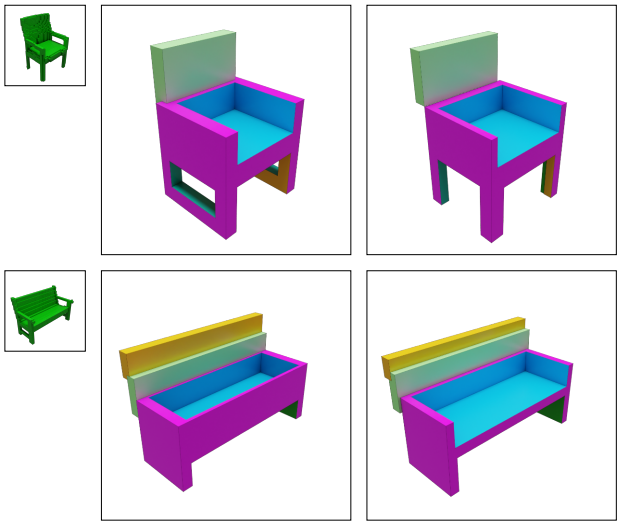

Figure 4. We show the *PO* rewriter applied to 3D CSG programs.

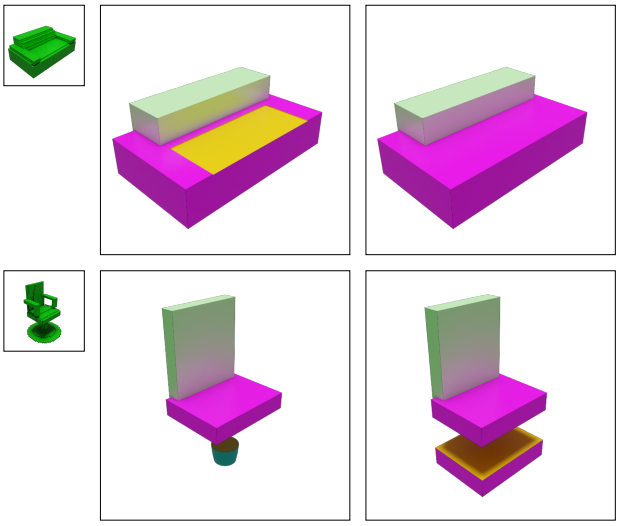

Figure 5. We show the *CP* rewriter applied to 3D CSG programs. Note that in the second figure, *CP* identifies a sub-tree of the input program which in fact obtains higher reconstruction accuracy w.r.t. the target shape.

other languages such as ShapeAssembly, these parameters are derived from the program's execution. Then, we apply boolean combinators of the parameterized primitives to obtain the program's *implicit equivalent*. Armed with the program's implicit equivalent, we uniformly sample points $t \in \mathbb{R}^n$ and convert the signed distance at the points into "soft" occupancy values to yield a differentiable execution of the program.

Optimization: From the program's *implicit equivalent*, the soft-occupancy values $O(z)$ are derived as follows:

$$O(z) = \sigma(-tanh(sdf(z) \times \alpha) \times \alpha), \tag{1}$$

where σ is the sigmoid function, and α is a scalar value. We run our optimization procedure for 250 steps with the Adam [7] optimizer (learning rate is set as 0.01), and we scale α logarithmically from $log(3)$ to $log(10)$. Finally, to ensure that the parameter values remain within range, we perform our optimization on $\theta = tanh^{-1}(\phi/s)$, where s is the range of each parameter, instead of ϕ directly. Figure 4 presents examples of the *PO* rewriting procedure in action.

4.2. Code Pruning (CP)

Given a shape x and program z, *CP* rewrites z s.t. $z^R \sim \arg\max_{\Omega^{CP}} \mathcal{O}(x, z)$, where $\Omega^{CP}(z) = \{\tilde{z} | \tilde{z} \subseteq z\}$ represents the set of all valid sub-programs of z, i.e. *CP* aims to identify and remove program fragments that negatively contribute to our objective $\mathcal{O}$. In Figure 5, we show examples of *CP* rewriter in action and in Algorithm 1 we present a high-level overview of *CP*. We first describe how to apply *CP* to a simply-typed lambda expression. We then describe how we map different languages, such as CSG and ShapeAssembly to such expressions.

Given a simply-typed lambda expression, we perform *CP* via two rewrite passes, namely a bottom-up and a top-down pass to approximate z^R. First, we create a directed acyclic expression-tree comprised of expression-nodes and edges connecting them. Each expression node stores the execution of the sub-expression formed with that node as the root. Now our goal is to identify nodes which can be pruned from the graph. First, we perform a top-down traversal of the graph, and evaluate the objective $\mathcal{O}$ w.r.t. the input shape x at each node. We identify the node with the highest score, and mark it as the root node. Next, during the bottom up traversal, we identify and prune nodes which are extraneous. For CSG language, we detect such extraneous nodes by comparing them to their parent node - if the parent node's execution exactly matches a child node's execution, all the sibling nodes are extraneous and can be removed. We also mark nodes as extraneous if the node's execution is empty (i.e. the node's subexpression evaluates to null). For ShapeAssembly, nodes whose execution does not overlap with the target shape x are marked as extraneous (as ShapeAssembly is a purely additive language). We present an example of the two traversals in *CP* rewriting procedure in Figure 6.

As stated earlier, to apply *CP*, we map the program to a simply typed lambda expression, which is then used to construct an expression tree. For CSG, the program itself is such an expression. For ShapeAssembly, we utilize the program's implicit equivalent (as defined in Section 4.1) to construct the expression. However, since ShapeAssembly is an imperative language, nodes can have interdependency

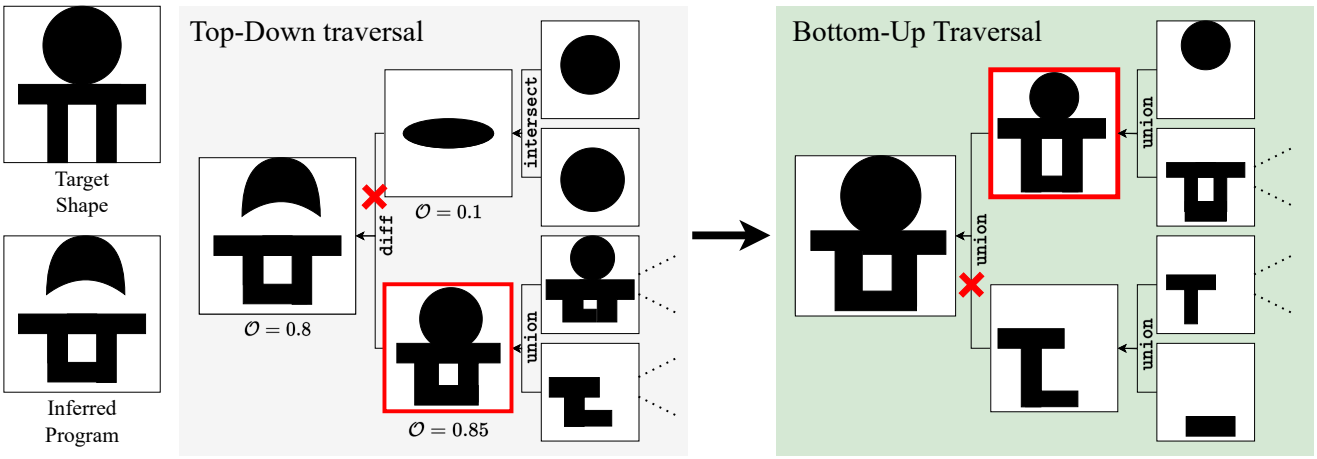

Figure 6. We show the two traversals of *CP* rewriter on 2D CSG on a manually created example. The Top-Down traversal identifies the objective $\mathcal{O}$ maximizing root node, and Bottom-Up traversal prunes extraneous nodes of the sub-tree starting from objective maximizing node. In this example, during the bottom-up traversal, as the top node matches its parent node, we remove its sibling node and simplify the expression.

Algorithm 1 *Code Pruning*

Input: Set of programs Z_x & Executor E.
Hyper-parameters: no. programs to rewrite n
Output: Rewritten programs Z_R

for z_x in random_sample(Z_x, n):
  # top down pass
  z_r = z_x
  for *subtr_node* in extract_subtree_nodes(z_x):
    if $\mathcal{O}(z_{subtr\_node}, x) \geq \mathcal{O}(z_r, x)$:
      $z_r = z_{subtr\_node}$
  # bottom up pass
  for *subtr_node* in extract_subtree_nodes(z_r):
    if removable(z_r, *subtr_node*):
      z_r = remove(z_r, *subtr_node*)
  if $\mathcal{O}(z_r, x) \geq \mathcal{O}(z_x, x)$:
    Z_R.insert(z_r)

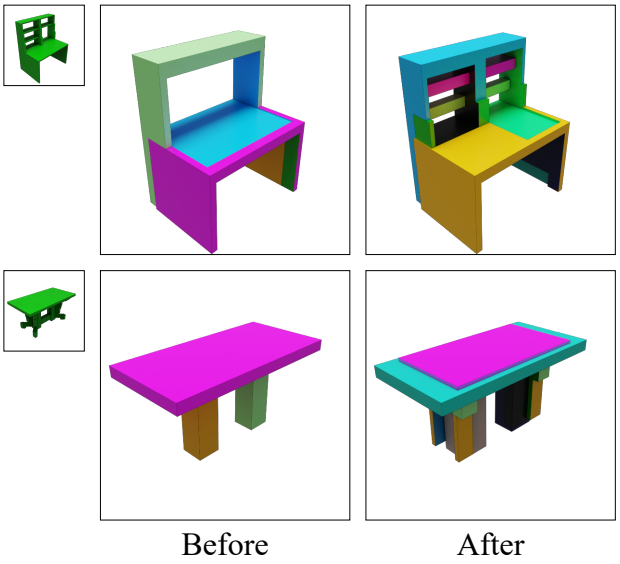

Figure 7. We show the *CG* rewriter applied to 3D CSG programs.

which restricts their removal. Therefore, we additionally extract a partially ordered dependency graph from the program, and only remove nodes which are terminal in the dependency graph. Figure 8 shows the expression tree and dependency graph for a ShapeAssembly program. Additionally, for ShapeAssembly we apply this procedure iteratively, since each pruning operation updates the dependency graph.

4.3. Code Grafting (CG)

The *CG* rewriter aims to replace sub-expressions of the given program with more suitable expressions from a cache of previously discovered sub-expressions. Algorithm 2 provides a high-level overview of *CG*, and Figure 7 shows examples of applying this rewriter.

First, sub-expressions along with their executions (in the form of n-dimensional occupancy fields) are extracted from all the inferred programs. All the sub-expressions are then clustered by their execution using the FAISS [2] library (using IndexBinaryIVF) and stored in a cache. We store only *unique* executions in the cache by identifying sub-expressions with approximately matching executions (hamming distance < 100 for 3D, and < 10 for 2D) and retaining only the shortest 'preferred' subexpression, while rejecting the rest.

This step provides us our first rewrite. For each of the

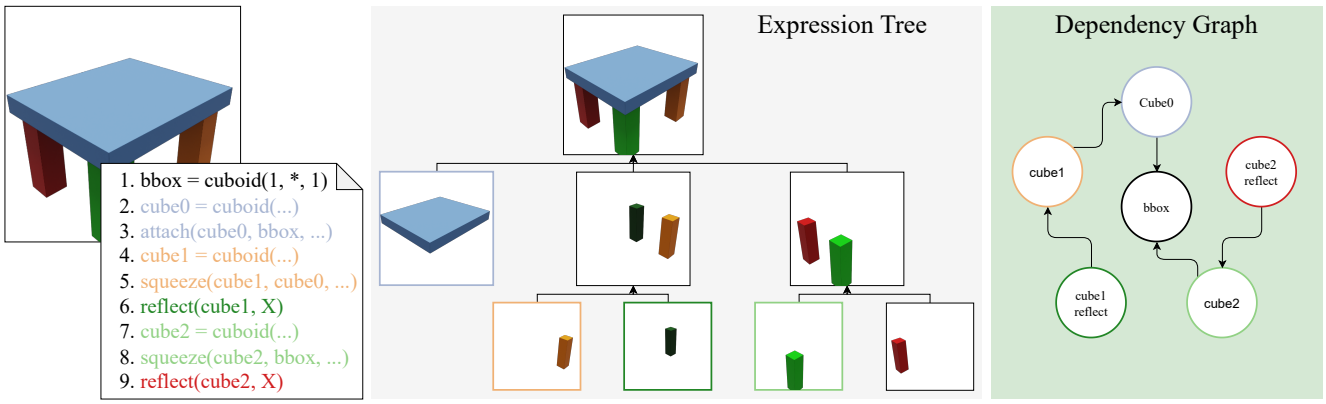

Figure 8. We show the mapping of a ShapeAssembly program to (a) its implicit equivalent expression tree and (b) its dependency graph. The *CP* rewriter prunes extraneous nodes from the expression tree while also ensuring pruned nodes have no incoming dependency. For the illustrated example, only the parts created by reflection operators are prune-able, as they are the only leaves in the dependency graph.

rejected sub-expressions, we retrieve the programs it originated from and insert the 'preferred' subexpression into it. This rewrite results in programs which achieve similar reconstruction accuracy while having a shorter length. Lines 1-12 in Algorithm 2 correspond to these steps.

The second rewrite performed in *Code Grafting* aims to replace sub-expressions in a given program to improve its reconstruction accuracy.

We perform this rewrite in 3 steps:

1. We derive the *desired execution* for each sub-expression in the program by masked function inversion (described in detail ahead). Note that for each sub-expression we only consider the *desired execution* within the bounding box of its execution (line 19).
2. For each sub-expression we retrieve the k-nearest neighbors of its desired execution from the cache as its replacement candidates (line 20).
3. We calculate the objective $\mathcal{O}$ achieved by each replacement, and perform the replacement which yields the highest reconstruction accuracy (lines 21-24).

This process (step 1 to 3) is repeated until none of the replacement candidates improve reconstruction accuracy, or until we perform a fixed number of replacements. Lines 14-24 in Algorithm 2 correspond to these steps.

Masked Function Inversion: Given a sub-expression A, we derive its *desired execution* A^* by masked function inversion. We assume the given expression executes to the given target T, and invert the expression S. The process can be described as follows:

$$T \sim S(A) \tag{2}$$
$$T \sim S_1(S_2(...S_n(A))) \tag{3}$$
$$A^* \sim S^{-1}(T) \tag{4}$$
$$A^* : \sim S_n^{-1}(S_{n-1}^{-1}(...S_1^{-1}(T))) \tag{5}$$

Algorithm 2 *Code Grafting*

Input: Set of programs Z_x, sub-expression cache C & Executor E.
Hyper-parameters: n, $k = 10$, $\tau = 10$.
Output: Rewritten programs Z_R

```
1:  # creating the subexpression cache
2:  for z_x in Z_x:
3:      for subexpr in extract_subexprs(z_x):
4:          match = retrieve(C, E(subexpr))
5:          if match:
6:              C.remove(match)
7:              shorter, longer = compare(subexpr, match)
8:              C.insert(shorter)
9:              z_r = replace(z_longer, shorter)
10:             Z_R.insert(z_r)
11:         else:
12:             C.insert(subexpr)
13: # rewriting programs
14: for z_x in random_sample(Z_x, n):
15:     num_rewrites = 0 & z_r = z_x
16:     while(num_rewrites < τ):
17:         candidates = []
18:         for subexpr in extract_subexprs(z_r):
19:             e* = desired-execution(subexpr, E(z_x))
20:             candidates.extend(C.get_nn(e*, k))
21:         z_best = get_best(candidates)
22:         if O(z_best, x) ≥ O(z_r, x):
23:             z_r = z_best
24:             num_rewrites += 1
25:     if O(z_r, x) ≥ O(z_x, x):
26:         Z_R.insert(z_r)
27: return Z_R
```

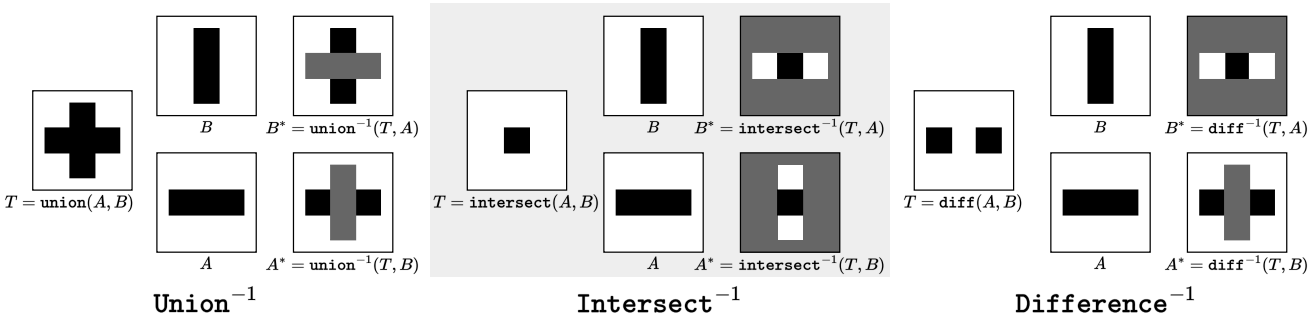

Figure 9. *CG* rewriter derives *desired executions* (A^* and B^*) for each sub-expression (A and B) that can be used to search a cache for potential replacement candidates with respect to a target shape (T). The *desired execution* is derived via masked function inversion, which we show the inversion for the three boolean combinators. For each desired execution, black indicates an area that should be occupied, white indicates an area that should not be occupied, and grey indicates invalid/masked regions.

By defining atomic inversion operations for transforms and combinators, we can simply derive S^{-1} for a sub-expression A by applying the inversions of operators applied on it, to the target T. As S can potentially be composed of non-invertible functions, we use a binary validity *mask* to identify input regions of space ($\mathbb{R}^n$) that cannot be inverted , and perform inversion only for invertible regions. For transforms such as translate, rotate, scale such masked inversions are trivial to define (e.g. $S^{-1}_{translate} = -S_{translate}$). For boolean combinators $U(A, B)$ we define its inversions w.r.t. a child A as a $\{Target, Mask\}$ tuple as follows:

$$\texttt{Union}^{-1}(T, B) = \{T, \overline{\texttt{Intersect}(T, B)}\}, \quad (6)$$

$$\texttt{Intersect}^{-1}(T, B) = \{T, \texttt{Union}(T, B)\}, \quad (7)$$

$$\texttt{Diff}^{-1}(T, B) = \{T, \texttt{Intersect}(T, \overline{B})\}, \quad (8)$$

$$\texttt{Diff}^{-1}(T, A) = \{\tilde{T}, \texttt{Union}(T, A)\}, \quad (9)$$

where $\overline{X}$ stands for the complement of X, and the mask term indicates valid regions. We provide example of the inversions in Figure 9.

Canonical Execution: In CSG domain, all sub-expressions executions are used in *canonical* form - we prepend each subexpression with a *translate* and *scale* command such that its execution is origin-centered and unit scale. Using the canonical form allows us to identify sub-expressions which are equivalent under translation/scale transformations. When the canonical sub-expressions are used for replacement, additional transform commands are prepended to make it fit the target expression's position and scale. For ShapeAssembly domain, all sub-programs are constructed in a unit scale cuboid by construction (their bounding box is fixed to sizes $(1, 1, 1)$). We note that similar canonical forms have been previously used in [8] as well.

Empty Node: During each *CG* rewrite step, we optionally extend the input expression with a union and an empty node, i.e. $expr = \text{Union}(expr, empty)$. This allows us to additionally consider sub-expression from the cache which, when attached to the input expression, improve the objective.

Cache size: We randomly sub-sample the cache as its size grows to curb the growth in memory requirement. We retain 35000 subexpressions, each consisting $32 \times 32 \times 32$ binary values, resulting in only ~ 1 GB memory requirement. Note that the cache entries persist over multiple search-rewrite-train cycles, so that good sub-expressions are retained and propagated once they are discovered.

ShapeAssembly: We apply *CG* to ShapeAssembly+, as it allows hierarchical composition of ShapeAssembly programs. Since the simple ShapeAssembly (used in the main draft) lacks hierarchical composition, we do not apply *CG* to it. First, we apply *CG* to strictly replace or introduce ShapeAssembly sub-programs (rather than just a set of statements). This allows us to treat each ShapeAssembly program's implicit equivalent (as defined in Section 4.1) as an simply-typed (partially invertible) lambda expression. To derive the *desired executions*, we only need to invert simple CSG functions such as `translate` and `union`. Successful expression replacement act as replacement (or introduction) of entire ShapeAssembly subprograms, mapping edited implicit equivalents uniquely to a ShapeAssembly program.

Note that the preconditions on the languages for applying *CG* are fairly simple. It requires a mapping to a typed lambda calculus expression, which then allows us to perform type-matching replacements. Further, our masked inversion procedure requires the existence of masked inverse for each atomic operator.

5. Qualitative Results

We show qualitative comparisons between our method and prior approaches. Our method improves reconstruction accuracy over prior bootstrapping methods [5], and infers programs with higher conciseness than domain specific architectures [9].

Failure Cases: While SIRI achieves better aggregate re-

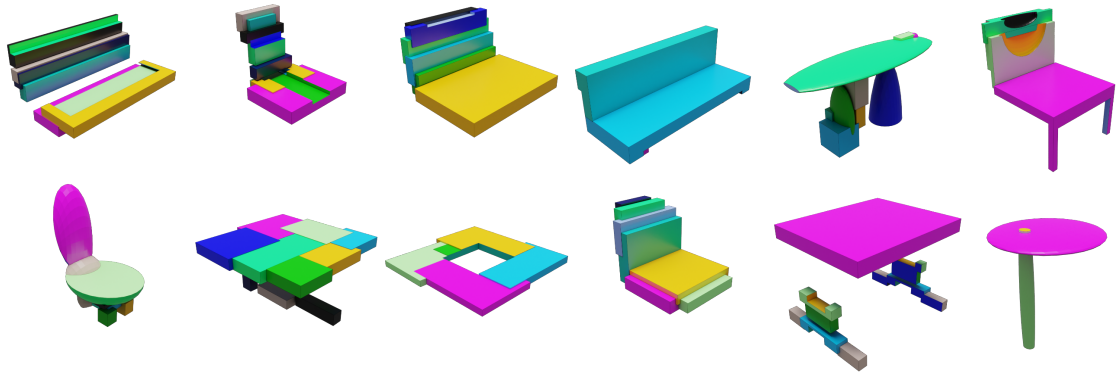

Figure 10. We show instances of program which fail to fully reconstruct the input shape, despite test time rewriting. A common trend we noticed is missing parts of object. Changing the rewrite objective $\mathcal{O}$ may help resolve this issue.

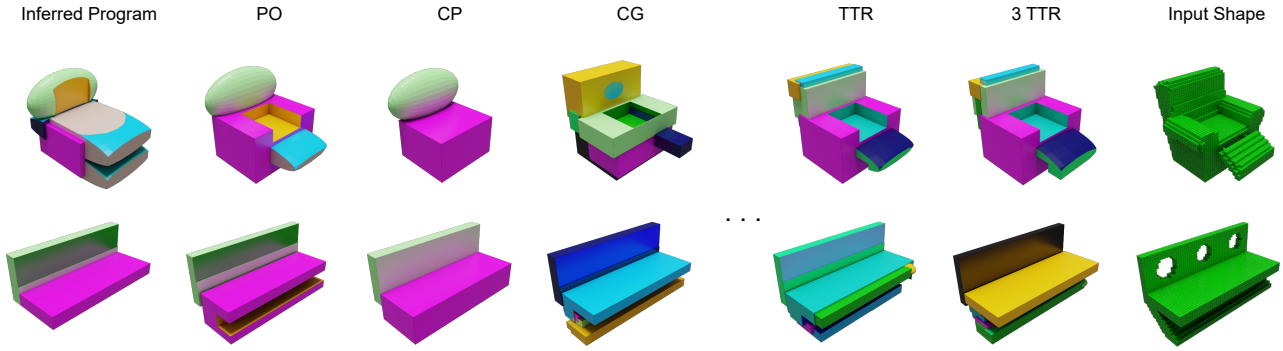

Figure 11. We show examples of the rewriter applied one at a time, and their interleaved application on inferred programs during test-time rewriting. Each rewriter is able to perform iterative improvements on the programs, and interleaved application further improves the program.

construction performance compared with previous bootstrapped learning methods such as PLAD, and matches reconstruction performance of domain-specific architectures such as CSG-Stump, its outputs can still be further improved. One failure mode is that of missing parts, even after test-time rewriting (see Figure 10). We note SIRI is not the only method that falls victim to this failure mode. A careful tuning of the program length weighting parameter α in the objective $\mathcal{O}$, along with a part presence sensitive reconstruction metric (instead of IoU) can help alleviate these challenges. In fact, SIRI's use of rewriters might allow it to uniquely solve this problem, by making use of a new class of rewriters that identifies missing semantic parts of a target shape, and rewrites the program with a sub-expression that covers these missing parts.

Test time rewriting: We visualize test time rewriting in Figure 11. As can be seen, each rewriter refines the program and their iterative application is beneficial.

Comparison to CSGStump: We compare our method to three variants of CSG-Stump to SIRI + TTR in Figure 12. We note that while the class specific CSG-Stump model may surpass the reconstruction accuracy of SIRI + TTR, its output predictions are still overly complex and hard to reason over, as reflected in the renderings with colored primitives. Note that due to the high complexity of CSG Stump programs, we color each intersection node, instead of the primitives. Despite this, we observe that the programs are still greatly over-parameterized.

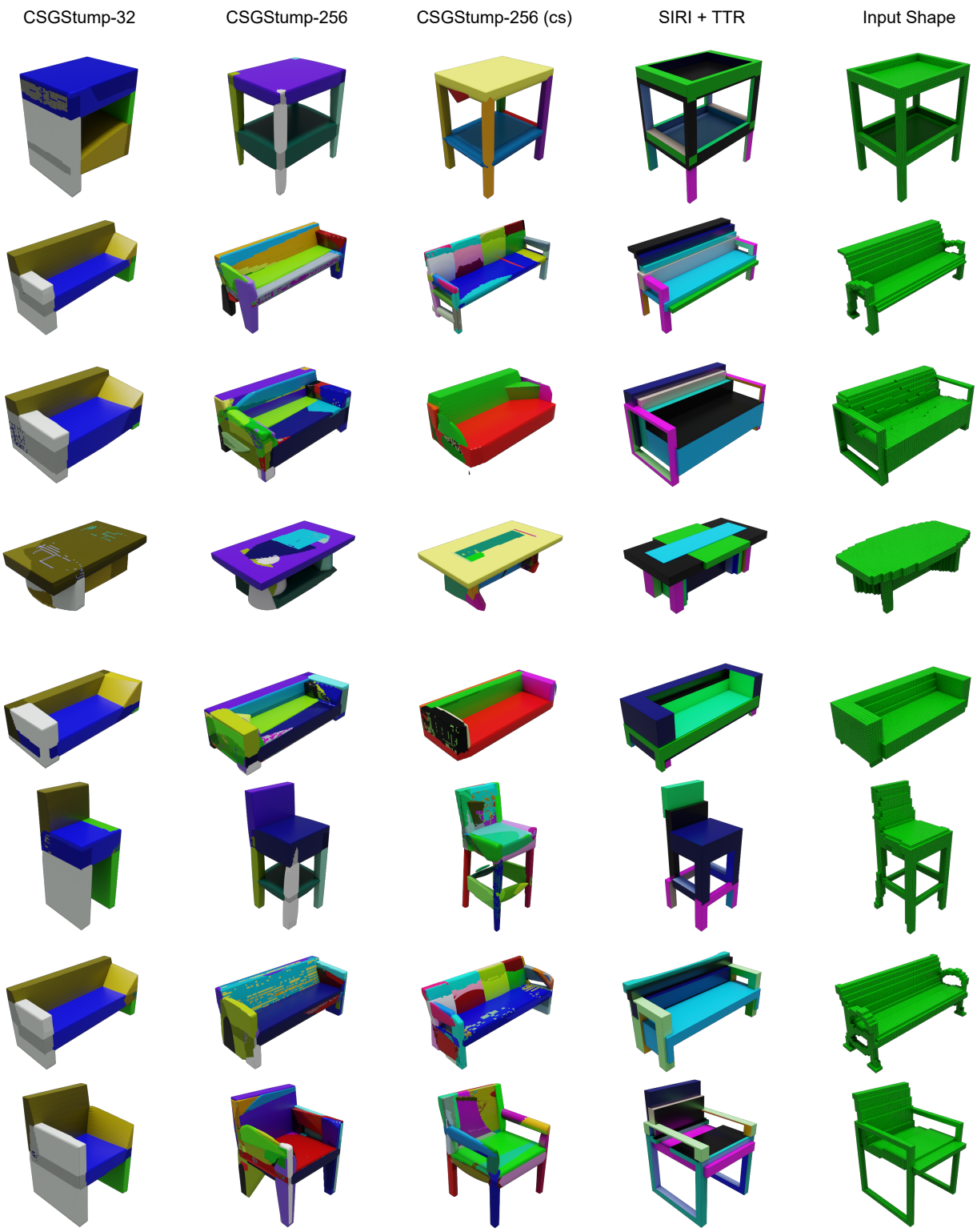

Figure 12. We compare our approach (SIRI + test time rewrite) to three variants of CSGStump [9]: (a) *CSGStump-32* a model trained with 32 primitive intersection nodes, (b) *CSGStump-256* a model trained with 256 primitive and intersection nodes, and (c) *CSGStump-256 (cs)* where a model is trained for each class (we use the pretrained weights provided by the authors).

References

[1] Blender Online Community. *Blender - a 3D modelling and rendering package*. Blender Foundation, Stichting Blender Foundation, Amsterdam, 2018. 1

[2] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547, 2019. 7

[3] R. Kenny Jones, Theresa Barton, Xianghao Xu, Kai Wang, Ellen Jiang, Paul Guerrero, Niloy J. Mitra, and Daniel Ritchie. Shapeassembly: Learning to generate programs for 3d shape structure synthesis. *ACM Transactions on Graphics (TOG), Siggraph Asia 2020*, 2020. 4, 5

[4] R. Kenny Jones, David Charatan, Paul Guerrero, Niloy J. Mitra, and Daniel Ritchie. Shapemod: Macro operation discovery for 3d shape programs. *ACM Transactions on Graphics (TOG), Siggraph 2021*, 2021. 4

[5] R. Kenny Jones, Homer Walke, and Daniel Ritchie. Plad: Learning to infer shape programs with pseudo-labels and approximate distributions. *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022. 1, 4, 9

[6] Kacper Kania, Maciej Zieba, and Tomasz Kajdanowicz. Ucsg-net- unsupervised discovering of constructive solid geometry tree. In *Advances in Neural Information Processing Systems*, 2020. 2

[7] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In *ICLR 2015*, 2015. 6

[8] Chandrakana Nandi, James R. Wilcox, Pavel Panchekha, Taylor Blau, Dan Grossman, and Zachary Tatlock. Functional programming for compiling and decompiling computer-aided design. *Proceedings of the ACM on Programming Languages*, 2018. 9

[9] Daxuan Ren, Jianmin Zheng, Jianfei Cai, Jiatong Li, Haiyong Jiang, Zhongang Cai, Junzhe Zhang, Liang Pan, Mingyuan Zhang, Haiyu Zhao, and Shuai Yi. Csg-stump: A learning friendly csg-like representation for interpretable shape parsing. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2021. 1, 2, 4, 9, 11

[10] Gopal Sharma, Rishabh Goyal, Difan Liu, Evangelos Kalogerakis, and Subhransu Maji. Csgnet: Neural shape parser for constructive solid geometry. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018. 4