Improving Unsupervised Visual Program Inference with Code Rewriting Families

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

Abstract

Programs offer compactness and structure that makes them an attractive representation for visual data. We explore how code rewriting *can be used to improve systems for inferring programs from visual data. We first propose Sparse Intermittent Rewrite Injection (*SIRI*), a framework for unsupervised bootstrapped learning.* SIRI *sparsely applies code rewrite operations over a dataset of training programs, injecting the improved programs back into the training set. We design a family of rewriters for visual programming domains: parameter optimization, code pruning, and code grafting. For three shape programming languages in 2D and 3D, we show that using* SIRI *with our family of rewriters improves performance: better reconstructions and faster convergence rates, compared with bootstrapped learning methods that do not use rewriters or use them naively. Finally, we demonstrate that our family of rewriters can be effectively used at test time to improve the output of* SIRI *predictions. For 2D and 3D CSG, we outperform or match the reconstruction performance of recent domain-specific neural architectures, while producing more parsimonious programs that use significantly fewer primitives.*

1. Introduction

Visual data is often highly structured: manufactured shapes are produced by assembling parts; vector graphics images are built from layers of primitives; detailed textures can be created via intricate compositions of noise functions. *Visual programs*, i.e. programs that produce visual outputs when executed, are a natural approach to capturing this complexity in a structure-aware fashion. Access to well-written visual programs supports downstream applications across visual computing domains, including editing, generative modeling, and structural analysis. But how can we obtain a program which generates a given visual datum?

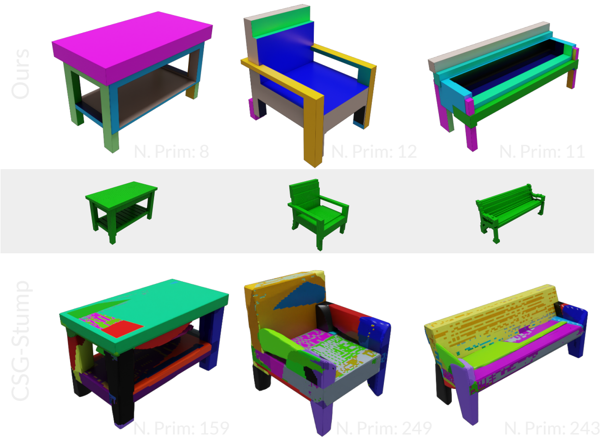


Figure 1. Our method SIRI (top row) generates highly compact yet accurate programs, in contrast to CSG-Stump [24] (bottom row), which generates programs with numerous primitives. Here, we show shapes rendered with colored primitives.

Visual Program Inference (VPI) methods aim to solve this problem by automatically inferring programs that represent visual inputs. Solving this search problem is very difficult: the space of possible programs is often vast, even when constrained by a domain-specific language (DSL). To overcome this challenge, recent works have investigated learning-based solutions, where a neural network is employed to guide the search. When a dataset of visual programs exist, such networks can be trained in a supervised fashion [36, 33, 37, 38, 13]. Unfortunately, most domains lack such data, so recent works have investigated how to train VPI networks in an *unsupervised* fashion.

Learning to infer visual programs without supervision is challenging: programs usually contain discrete and continuous elements, which complicates end-to-end learning. Various solutions have been proposed to work around this issue: end-to-end learning is possible with neural architectures that act as a smooth relaxations of program executors [24], while policy gradient reinforcement learning [26] and bootstrapped learning methods [15] are able to treat

Project page: https://bardofcodes.github.io/coref/

program executors as (potentially non-differentiable) 'black boxes.' These solutions come with downsides: designing differentiable relaxations is challenging (or even impossible) for some domains; reinforcement learning suffers from noisy gradients and slow convergence; bootstrapped learning methods are prone to getting stuck in local minima.

Moreover, a seldom acknowledged limitations of these latter methods is that they treat programs as *just* sequences of tokens. We argue that this view is suboptimal: programs are structured objects that support domain-specific reasoning to meaningfully constrain and guide the VPI search process. One example of such reasoning is the use of domain-specific operations that modify programs toward optimizing an objective—we call such operations *rewrites*. Rewrites have been explored in the context of VPI tasks, but primarily as a test-time optimization, e.g. finding better continuous parameters for a fixed program structure. While such optimization is useful, our claim is that other rewrite operations are similarly useful, especially when used in tandem, and that they can be employed to benefit VPI network learning, not only as test-time optimization schemes.

In this paper, we investigate how to use code rewriting to improve visual program inference. Unlike prior work, we focus on *families* of code rewriters, each of which makes improvements to a program with some goal in mind. We propose *Sparse Intermittent Rewrite Injection* (SIRI), a bootstrapped learning algorithm that sparsely applies rewriters and injects rewritten programs into a search-train loop at intermittent intervals. To realize SIRI, we design a family of rewrites applicable to multiple visual programming DSLs: gradient-based parameter optimization (*Parameter Optimization*), removing spurious sub-programs (*Code Pruning*), and sub-program substitutions from a cache (*Code Grafting*). We also propose a test-time rewriting scheme that searches for improved programs through interleaved rewrites that is well-suited to the types of programs inferred by SIRI-trained networks.

We evaluate SIRI and our family of rewriters (*PO*, *CP*, *CG*) on three shape program DSLs: 2D Constructive Solid Geometry (CSG), 3D CSG, and ShapeAssembly [13]. We compare VPI networks trained with SIRI to VPI networks trained by PLAD, a recently-proposed bootstrapped learning method [15], and find that SIRI both increases reconstruction performance and converges significantly faster. We further show that naively combining our rewrite families with PLAD performs much worse than SIRI, and in some domains even worsens performance compared with PLAD. Finally, we demonstrate that combining SIRI with our test-time rewriting scheme infers visual programs that can match (3D CSG [24]) or surpass (2D CSG [16]) reconstruction performance of domain-specific neural architectures while producing significantly more parsimonious programs (see number of primitives, Figure 1).

In summary, we make the following contributions:

1. *Sparse Intermittent Rewrite Injection*, a framework for unsupervised visual program inference that leverages a family of code rewriters.
2. A family of code rewriters applicable to multiple DSLs that benefit VPI learning methods and can be used in a test-time rewriting scheme.

2. Related Work

Visual program inference (VPI) is a sub-problem within program synthesis. Program synthesis is a storied field, with roots back to the inception of Artificial Intelligence, where the objective is to produce programs that meet some specification [11]. Under our framing, the specification is an input visual datum that the synthesized program should reconstruct when executed. In the rest of this section, we first overview VPI learning paradigms and then summarize prior work that looks at visual-program rewriting.

Learning to infer visual programs: *End-to-end learning* methods train by propagating reconstruction loss gradients directly to a network via differentiable execution. Though such approaches can yield impressive reconstruction accuracy, they either require a soft relaxation of the program execution [16, 24, 41, 40, 25, 4] which is infeasible for many languages, or require training domain-specialized neural executors [29, 12], which can introduce approximation errors. In SIRI, we instead leverage a 'partially' differentiable execution of visual programs, bypassing the need of program relaxation or neural executors.

Reinforcement learning has also been used by prior VPI approaches [26, 8, 30]. Usually, the inference network is treated as an 'agent' maximizing a reward signal tied to its reconstruction accuracy. The high variance of policy gradient methods has limited the application of such techniques to toy datasets, especially for 3D data. Similar in spirit to SIRI, other programmatic RL methods for non-visual domains have explored blends of program optimization and learning [31, 32]. Also related are non-programmatic RL methods that explore episode modification, through episode relabeling or neurally-guided search [1, 19, 10, 18, 23]. Like SIRI, these approaches aim to improve learning targets through local search, but they do so for vastly different domains (often much simpler than complex 3D shape-programs), employ simplistic rewriting techniques, and don't target bootstrapped learning frameworks.

Bootstrapped learning is an attractive alternative that can avoid the pitfalls of RL and end-to-end learning [15, 17]. Such approaches alternate between *Search* phases, that discover 'good' programs, and *Train* phases, that use discovered programs to train a network. While these approaches have demonstrated improvements over RL, they are still limited by treating each program as a sequence of tokens, rather than a structured object. Our work bridges this gap

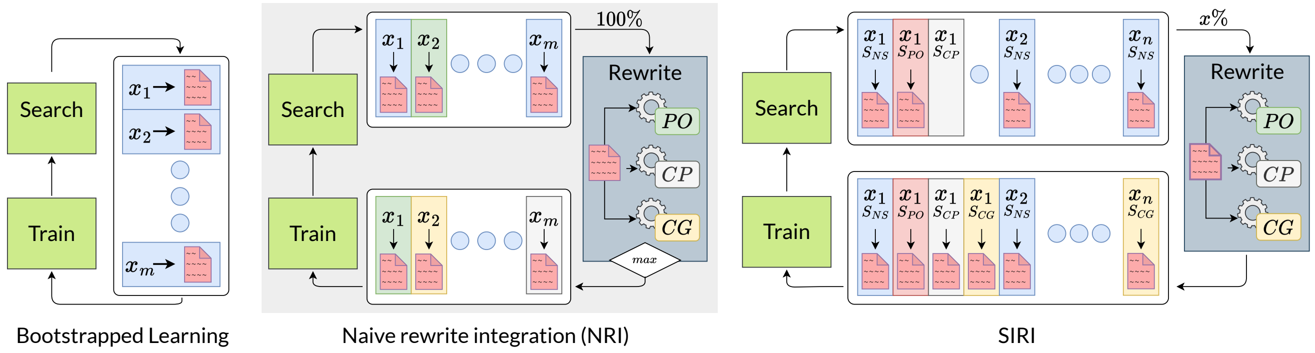


Figure 2. Bootstrapped learning methods [15] store a record of the objective maximizing program for each shape in a Best Program (BP) data-structure. Naive rewrite integration applies a family of rewriters to each entry in BP, and overwrites each entries when a rewrite is successful. Our method SIRI, instead applies the rewriters to a subset of BP entries and overwrites entries only when their sources S_i match.

by offering an effective way of integrating a family of code rewriters into a recent bootstrapped learning paradigm [15].

Rewriting visual programs: *Gradient-based optimization* is a common approach for optimizing visual programs. For neural architectures that serve as a relaxation of the executor, this can be achieved via test-time fine-tuning. While such approaches achieve impressive reconstruction performance, they are expensive to run and often produce messy program structure [41, 40]. This fine-tuning can take prohibitively long to converge, requiring anywhere from 3 minutes [41] to even 30 minutes [24] *per sample*.

Typically, visual program executors can be made piecewise differentiable with respect to parameters of an input program (up to control flow decisions), which supports test-time gradient-based optimization [26, 27, 28]. Yet, as this rewriting scheme does not change program structure, reliance on *only* gradient-based optimization is vulnerable to getting stuck in local minima. We employ a *Parameter Optimization* rewriter (Sec. 4.1) as one member of a rewrite family, where other rewriters can make structural program changes, to avoid getting stuck in these local minima. Critically, our implementation is highly-efficient, and we can apply each member of our rewrite family multiple times in under 20 seconds during test-time optimization.

Gradient-free optimization techniques have been investigated that leverage domain heuristics to develop rewriting strategies that modify the control flow decisions of visual programs. For 3D CAD languages, these operations have been explored for non-learning based reverse-engineering methods [6, 20]. Recently, E-graphs [34] have been employed to efficiently search for rewritten programs that optimize criteria such as program length [21] or fabrication cost [42, 35]. While these methods do not consider *learning* from rewritten programs, it would be possible to integrate these types of techniques into our family of rewriters.

Abstraction discovery is a special form of rewriting, where common subcomputations shared across many programs are factored out into subroutines (i.e. abstractions), and programs are rewritten to use these subroutines. When a dataset of programs is given as input, this step can be decoupled from visual program inference [14, 39, 3, 2]. Alternatively, some methods have investigated how abstraction discovery (AD) phases can improve visual program inference performance [9, 7]. In an iterative procedure, an abstraction phase greedily rewrites a dataset of programs with abstractions, then a recognition model learns on rewritten programs to discover higher-order abstractions and solve more complex inference problems. Such methods are not yet able to scale to the complex 3D shape domains we study in this work, as they employ simplistic recognition models and rely on expensive enumerative search. Furthermore, we find that naively integrating rewriter outputs, as done in past AD approaches, can in fact be detrimental to the bootstrapping process. We instead propose SIRI, a non-deleterious procedure for integrating rewriters under bootstrapped learning paradigms, described in Sec. 3.2, which may prove similarly beneficial for AD approaches.

3. Method

In this section, we explore how families of code rewriters can be employed to improve visual program inference. In section 3.1, we formalize our task specification, objective function, and rewriter assumptions. We then present *Sparse Intermittent Rewrite Injection* (SIRI), an unsupervised learning paradigm for visual program inference (VPI) in Section 3.2. SIRI employs a family of rewriters to improve reconstruction performance for VPI tasks while maintaining a parsimonious program representation. Finally, we describe how these operations can also be used in a test-time rewriting scheme, that is especially well-suited to the outputs of SIRI (Sec. 3.3). We describe the family of rewriters we employ for shape-program domains in Section 4.

3.1. Task Specification

We define the visual program inference task as follows: given a target distribution S of visual inputs (e.g. shapes), we want to learn a model $p_\theta(z|x)$, where $x \in S$, which infers a program z whose execution $E(z)$ reconstructs the input shape x. Following Occam's razor, we seek programs that are parsimonious. More formally, we seek a p_{θ^*} that maximizes our objective function $\mathcal{O}$:

$$\theta^* = \arg\max_{\theta} \mathop{\mathbb{E}}_{x \in S} \left[\sum_{z} \mathcal{O}(x, z) p_\theta(z|x) \right], \quad (1)$$

$$\mathcal{O}(x, z) = \mathcal{R}(x, E(z)) - \alpha|z|, \quad (2)$$

where $\mathcal{R}$ measures the reconstruction accuracy between x and $E(z)$, $|z|$ measures the length of the program, and α modulates the strength of these two terms. Our method takes in a family of rewriters, *RWS*, to help with this task. Though each rewriter may leverage different domain properties, they are all tied to the same objective $\mathcal{O}$. Formally, they aim to perform rewrites *RW* $(x, z) \rightarrow z_i^R$ s.t. $z_i^R \sim \arg\max_{z \in \Omega^i(z)} \mathcal{O}(x, z)$, where $\Omega^i(z)$ represents the set of all programs rewriter i can create from z.

3.2. Sparse Intermittent Rewrite Injection

One way to use rewriters to improve VPI models is to incorporate them into bootstrapped learning schemes. One such recent bootstrapped learning paradigm is proposed by PLAD [15]. PLAD alternates between search and train phases, where a best-program (BP) data-structure modulates how these phases communicate (Figure 2, left). In the search step, a VPI network $p(z|x)$ is fed visual inputs from S and aims to predict programs that optimize $\mathcal{O}$ (e.g through beam-search). The results of this neurally-guided search (NS) are used to populate the fields of BP, where each key corresponds to a unique shape x, and each value corresponds to the best program (in terms of $\mathcal{O}$, for x) seen so far. In the train step, the entries of BP are used to construct paired training sets (X, Z) which are then used to optimize $p_\theta(z|x)$ with maximum likelihood loss.

How can rewrites be integrated into such a framework? We demonstrate a naive approach, termed *Naive Rewrite Integration* (NRI), in Figure 2, middle. In this naive version, a *Rewrite* step takes place between each search and train step, that modifies the entries of BP. Specifically, for every (x, z) entry in BP, each rewriter is applied to z, and if z^R improves the $\mathcal{O}$, then (x, z^R) is entered into BP, overwriting the (x, z) entry. Empirically, we find that this approach can in fact be worse than PLAD. While the rewriters perform a local search to optimize $\mathcal{O}$ for the given (x, z) pair, there is no guarantee that a discovered z^R will make a better training target for $p_\theta(z|x)$ as well. NRI has a deleterious effect on the entries of BP, as only one program value is maintained for each shape key, so indiscriminate applications of rewriters can get easily stuck in local minima with respect to $p(z|x)$. Prior works which incorporate program rewriting strategies [7, 9] update training programs in this way.

Sparse Intermittent Rewrite Injection (SIRI) also employs a *Rewrite* step, but avoids this issue with a more judicious application of rewriters from *RWS* (Figure 2, right). Instead of applying rewriters on each shape, rewrites are only applied to a subset of BP entries. SIRI also adds a source field to each BP key that indicates what produced a given program: either neurally-guided search (NS) or some *RW* $\in$ *RWS* . Critically, this allows SIRI to add programs into BP in a way that can only 'forget' (shape, program) pairs from the same source: NS programs replace NS programs, and each *RW* can only bump out *RW* sourced entries.

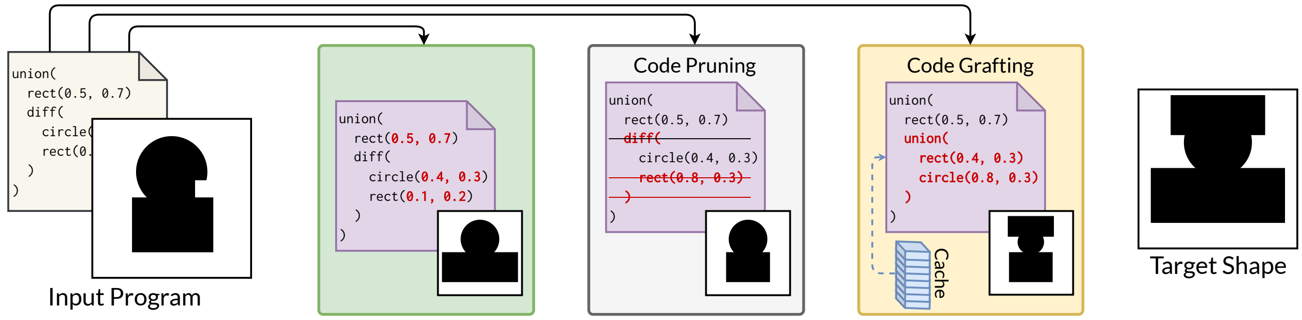
During the search-step, each (x, z) pair produced by the neurally-guided search populates the (x, S_{NS}, z) entry of BP. Then for each *RW* $\in$ *RWS* , SIRI samples a percentage of BP entries, (x, src, z), and adds $(x, \textit{RW}, z^R)$ if the rewriter successfully improves the objective. Note that rewrites can sample inputs from any source, not just S_{NS}. Not only does this scheme ensure that rewriters only overwrite their own previous predictions, but it is also more efficient versus NRI: some rewriters have high computational costs, and excessively applying them can slow down training. Instead, we find that training $p(z|x)$ on BP entries with sparsely rewritten programs, both converges faster and reaches better final end-states, as useful rewriting strategies get amortized by the network weights.

As only a subset of entries are updated in each *Rewrite* phase, some rewrite entries in BP (potentially from previous *Rewrite* phases) may store programs which are worse than the program inferred by $p_\theta(z|x)$ during *Search*. Therefore, before each train step, SIRI purges all stale $(x, \textit{RW}, z^*)$ entries from BP whenever $\mathcal{O}(x, z^*) < \mathcal{O}(x, z)$, where z is the program inferred for x during the *Search* phase.

3.3. Test-time Rewriting

Our family of rewriters, *RWS* can also be employed at inference time to find better programs for a particular visual target. With a slight abuse of notation, given an (x, z) pair, our test-time rewrite (TTR) approach aims to find a sequence seq^* of at-most k rewrite operations from *RWS*, such that $seq^* \sim \arg\max_{seq \in \textit{RWS}^k} \mathcal{O}(x^*, seq(x, z))$, where $z^{ttr} = seq^*(x, z)$ would be the result of applying each rewrite in seq^* to z iteratively.

We realize this formulation with a greedy search. We initialize z^{ttr} as z, and then for k steps, we iterate through each *RW* $\in$ *RWS*, compare $\mathcal{O}$ (x, z^{ttr}) to $\mathcal{O}$ $(x, \textit{RW}(z^{ttr}))$, and replace z^{ttr} with *RW*(z^{ttr}) if $\mathcal{O}$ improves. To avoid redundant work, if a *RW* has already tried to improve a particular z, and failed to do so, we instead pass in the next best observed z (in terms of $\mathcal{O}$) that the *RW* has not operated over. The final z^{ttr} is then returned as the output.



```
union(
  rect(0.5, 0.7)
  diff(
    circle(
    rect(0.
  )
)
```

```
union(
  rect(0.5, 0.7)
  diff(
    circle(0.4, 0.3)
    rect(0.1, 0.2)
  )
)
```

```
union(
  rect(0.5, 0.7)
~~diff(~~
    circle(0.4, 0.3)
~~    rect(0.8, 0.3)~~
~~  )~~
)
```

```
union(
  rect(0.5, 0.7)
  union(
    rect(0.4, 0.3)
    circle(0.8, 0.3)
  )
)
```

Figure 3. SIRI uses *rewrites* to improve VPI networks. We depict three rewrites in action that: optimize continuous parameter values (*Parameter Optimization*), remove extraneous code (*Code Pruning*), and substitute sub-expressions from a cache (*Code Grafting*).

This procedure can in theory be applied to programs produced by any source (e.g. networks that acts as a differentiable language executor [24]). However, we find that there are unique benefits to applying this procedure to the predictions made by a $p_\theta(z|x)$ network trained with SIRI. Some rewrites build a cache of partial results during bootstrapped learning that can be quickly and effectively applied at inference time (*CG*, Section 4.3). Furthermore, some rewrites are too expensive to run on very complex programs, consuming massive amounts of memory, but are well-suited to the parsimonious programs produced by SIRI. For instance, one rewriter (*PO*, Section 4.1) was not able to operate on the highly complex programs output by CSG-Stump [24] (cf. supplemental material).

4. Rewriting Visual Programs

As our method relies on an input family of rewriters, *RWS*, we identify three rewrite operators that generalize across multiple shape-program domains. Figure 3 depicts these rewriters in action. In the rest of this section, we provide a high-level description and motivation for the different rewriters we use during SIRI and test-time rewriting: *Parameter Optimization* (Section 4.1), *Code Pruning* (Section 4.2), and *Code Grafting* (Section 4.3). We provide the implementation details in the supplemental material.

4.1. Parameter Optimization

Visual languages often contain continuous (and differentiable) parameters such as the scale and position of primitives, and discrete parameters such as control flow (e.g. how to combine primitives). While keeping the discrete parameters fixed, *Parameter Optimization* (*PO*) rewriter aims to improve the continuous parameters of a given program using gradient-based optimization. Given a program z_ϕ with continuous parameters ϕ inferred for a shape $x \in S$, *PO* adjusts ϕ to maximizes the reconstruction accuracy $\mathcal{R}$ between x and the program execution $E(z_\phi)$: $\phi^* \sim \arg\max_\phi \mathcal{R}(x, E(z_\phi))$.

To propagate gradients back from a reconstruction metric to the continuous parameters of z_ϕ, *PO* requires that the program executor E is partially differentiable: one or more continuous parameters should have well-defined derivatives for a given program structure (though the program execution may itself be only piecewise continuous). We highlight that *PO* is useful *even* under such constraints, precisely because *PO* is not the only rewriter we consider: other rewriters in *RWS* can influence structural changes, along with SIRI's *Search* phase. *PO*'s requirements on E differ significantly from the differentiable executors employed by neural relaxation architectures. These works [16, 24] often attempt to differentiate through both discrete and continuous decisions, which leads to noisy gradients and poor program quality.

Although the design of executors is domain-dependant, we outline our procedure for converting the outputs of E into an equivalent signed-distance field representation compatible with our reconstruction metric $\mathcal{R}$. For each language we consider, we map outputs from E into a tree-like representation, where each leaf represents a primitive (spheres, etc.) and each intermediate node represents a transformation (position, etc.) or a combinator (union, etc.). For CSG-like languages, we can directly map from program parameters ϕ to this representation. It is also possible to convert the output of more complex program executors that produce collections of primitives into this format [13]. Then, we perform boolean combinations of the parameterized primitives, and apply the transformation operators, to obtain the program's implicit equivalent. With the program's implicit equivalent, we now uniformly sample points $t \in \mathbb{R}^n$, and convert the signed distance at the points into soft-occupancy to yield a differentiable execution of the program. This framing should be extensible to other visual-programming domains of interest: where either this mapping may be explicitly extracted from the input program (SVG) or parsed from primitives created by a more complex executor [22].

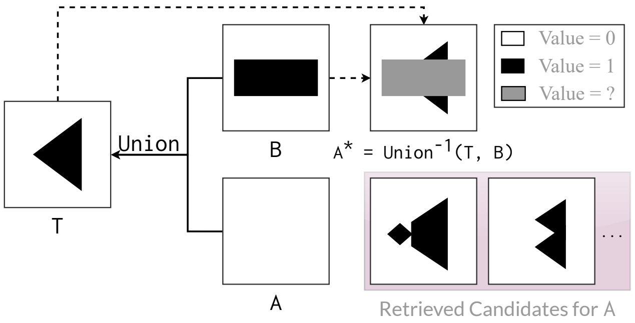


Figure 4. Given a target T, we derive the *desired execution*, A^*, for sub-expression A, through operator inversion. Our *CG* rewriter leverages *desired-executions* to search for replacement candidates.

4.2. Code Pruning

One drawback of bootstrapping techniques is their tendency to reinforce spurious patterns [17]. The *Code Pruning* (*CP*) rewriter mitigates this problem by identifying and removing program fragments that negatively contribute to our objective $\mathcal{O}$. Given a shape x and input z, *CP* rewrites z s.t. $z^R \sim \arg\max_{\Omega^{CP}} \mathcal{O}(x, z)$, where $\Omega^{CP}(z) = \{z * | z* \subseteq z\}$ represents the set of all valid sub-programs of z. A naive *CP* rewriter that considers every valid subexpression would be prohibitively slow. We instead implement a greedy version of *CP* designed for declarative, functional languages. We describe our general approach below, and provide further details in the supplemental material.

Our implementation of *CP* employs two greedy searches, a top-down pass and a bottom-up pass, to approximate z^R. At each pass, we identify and prune nodes which decrease the overall objective score $\mathcal{O}$. The top-down traversal relabels the highest objective scoring node as the root, pruning all but the tree starting at that node. The bottom-up traversal then checked each node's contribution to the final execution and prunes branches with negligible contribution.

4.3. Code Grafting

A key feature of symbolic representations is the ability to perform part-based reasoning, e.g. compose parts from different instances. The *Code Grafting* (*CG*) rewriter exploits this feature to improve programs. Specifically, the *CG* rewriter replaces sub-expressions of a particular program with a better sub-expression (in terms of $\mathcal{O}$). The primary challenge is specifying *what* to replace a sub-expression with, as the space of potential replacements is enormous. To make the search tractable, *CG* builds a program cache populated by sub-expressions discovered while running SIRI, and searches for replacement candidates within the cache.

When the cache is large, searching for good replacement expressions can become expensive. Our *CG* implementation makes this search tractable by focusing on *execution equivalence*, i.e. equivalence is measured by comparing their executions (stored as an n-dimensional occupancy grid). But finding cache entries which would improve reconstruction performance requires some notion of the *desired execution*, i.e. what sub-expression execution would make the program better match the target shape x?

We develop a procedure for calculating such desired execution through masked function inversion; an example is depicted in Figure 4. We provide the high-level insight by walking through this figure; further details are in the supplemental material. In this example, T denotes the target shape (the desired final execution), which is a union of two subexpressions. Suppose we wish to replace subexpression A. Given the current state of its sibling subexpressions (B, in this case), we can invert the Union to produce the *desired-execution* A^*. A^* can be broken into sub-regions: (black and white) areas where the optimal execution behavior is known and (gray) areas where the optimal execution behavior is unknown (for example, due to the non-invertibility of some operators). *CG* uses such ternary *desired-executions* to search for cache entries that are likely to improve reconstruction accuracy and substitutes the most suitable candidate.

5. Results

We evaluate the efficacy of our rewriters over a collection of VPI domains for two tasks: improving bootstrapped learning (Section 5.2), and improving VPI with test-time rewrites (Section 5.3). First, we provide the details concerning our experiments in Section 5.1.

5.1. Experimental Design

Domain-Specific Languages: We consider three VPI domains: 2D Constructive Solid Geometry (CSG), 3D CSG, and ShapeAssembly [13]. Shapes are formed in CSG by declaring primitives such as cylinders, applying transformations, and composition via boolean operations. ShapeAssembly produces hierarchies of cuboid part proxies (which can themselves contain sub-programs) assembled through attachment operations. Please see the supplemental for the complete DSL grammars.

To ease learning, past approaches have used simplified versions of these languages, e.g. restricting CSG to contain only primitive-level transformations or removing hierarchical sub-programs from ShapeAssembly [16, 15, 26]. For fair comparison, we match our DSLs to prior work.

Shape Datasets: We evaluate 2D CSG on the CAD dataset introduced in CSGNet [26]. It contains front and side views of chairs, desks and lamps from the Trimble 3D warehouse. This dataset is divided into 10K training, 3K validation and 3K testing shapes. We evaluate 3D CSG and ShapeAssembly on the 3D CAD dataset released in [15] containing shapes from chair, table, couch, and bench categories of

	Chamfer Distance (↓)			IoU (↑)		
	2D CSG	3D CSG	ShapeAssembly	2D CSG	3D CSG	ShapeAssembly
PLAD [15]	0.24	1.75	1.98	90.8	74.65	63.1
NRI	0.36	1.25	1.53	88.4	74.43	66.89
SIRI	**0.22**	**1.10**	**1.44**	**91.7**	**76.77**	**67.8**

Table 1. We report the Test-set performance across 3 VPI domains. Naively integrating the rewriters into PLAD (NRI) can detoriorate the model's performance (IoU on 2D & 3D CSG). In contrast, SIRI consistently improves over PLAD on all the three VPI domains.

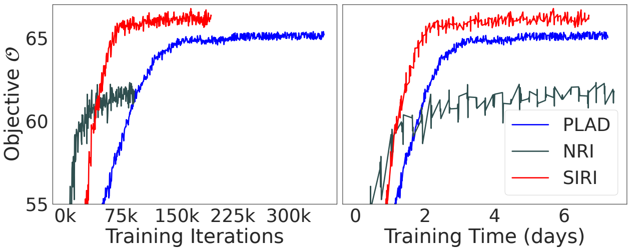


Figure 5. We plot the objective $\mathcal{O}$ (Y-axis) as a function of training time (X-axis), measured by iterations (left) and wall-clock time (including time taken for rewriting) (right) on 3D CSG domain. By amortizing the rewrite-cost, and keeping the training data diverse, SIRI achieves higher performance and does so faster.

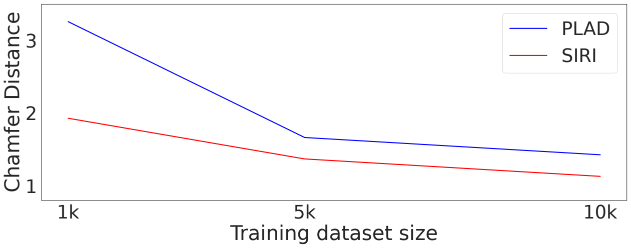


Figure 6. Due to data-independent rewriters, SIRI remains effective at training the network even with a fraction of the data. In comparison, since PLAD's *Search* phase is tied to the inference network's performance, data scarcity deteriorates its performance.

ShapeNet dataset [5] in a voxel grid format. This dataset is split into 10k training, 1k validation and 1k testing shapes.

Model Architecture: Our model $p(z|x)$ synthesizes programs as a sequence of tokens, where each token specifies a command type or its parameters. Numeric parameters are normalized and discretized into 33 bins. Each $p(z|x)$ uses a domain-specific feature extractor (e.g. a 2D or 3D CNN), and a decoder-only transformer module. For 2D languages, the feature extractor takes a 64^2 image as input; for 3D languages, it takes a 32^3 voxel grid. We use the same transformer architecture for all experiments, varying only the last layer output size to model the different number of commands in each language.

Metrics: We measure reconstruction accuracy with two metrics: Intersection Over Union (IoU) and point cloud Chamfer-Distance (CD). We use 64^2 and 32^3 resolution occupancy grid for calculating 2D and 3D IoU respectively. We follow the same methodology as CSGNet [26] for measuring 2D CD; 3D CD is measured between 2048 points sampled on the ground-truth ShapeNet meshes and the meshes produced by executing the inferred programs.

Training details: Following [26, 15, 8], we pretrain our models on a large corpus of synthetically generated programs until it converges. We generate the synthetic programs via the sampling procedure proposed in PLAD [15]. After pretraining, the model is finetuned on the target distribution S, following the procedure outlined in Section 3.2. During each *Rewrite* phase, we apply *PO*, *CP* and *CG* to 50%, 50%, 15% of programs respectively. *CG* is applied to only 15% of data due to its higher computational cost. For our training objective $\mathcal{O}$ (c.f. equation 2), we fix α to 0.015. For 3D we set $\mathcal{R}$ to IoU, and for 2D we set $\mathcal{R}$ to CD. For test-time rewriting (TTR), we perform interleaved application of *each* rewriter thrice, unless specified otherwise (i.e. for Table 4).

5.2. Training with SIRI

We first evaluate the benefit of intermittent rewriting for bootstrapped learning. We compare our method SIRI against 2 baselines, namely PLAD [15], and *Naive Rewrite Integration* (NRI) which naively integrates the rewriters into PLAD (cf. Section 5.2). Note that prior work on integrating code rewriting [7, 9] follow this strategy.

As shown in Table 1, SIRI outperforms both the baselines on all domains. While PLAD performs well on simple domains such as 2D CSG, it is less effective on harder domains such as ShapeAssembly. Naively integrating the rewriters (NRI) can in fact even be detrimental to bootstrapped learning, as can be seen for 2D & 3D CSG (w.r.t. IoU). Excessive use of the rewriters and the lack of training data diversity leads the model trained with NRI to overfit to a local minima, i.e. the *Search* and *Rewrite* phases become ineffective at generating useful better programs to learn from. SIRI resolves this issue by its frugal usage of the rewriters and by training p_θ on a diverse set of programs obtained from both the *Search* and *Rewrite* phases. We see a similar trend on other visual languages, which we discuss

	CD (↓)	N. Prim. (↓)	N. ops. (↓)
CSG-Stump 32	1.90	19.03	5.95
CSG-Stump 256	1.22	154.47	52.85
CSG-Stump 256 (cs)	0.78	191.38	72.52
SIRI	1.101	3.90	2.90
SIRI + TTR	0.83	8.47	7.42

Table 2. SIRI outperforms CSG-Stump 256 with a fraction fewer primitives. Applying test-time-rewrites to SIRI makes its performance comparable to that of class-specific (cs) CSG-Stump while remaining relatively parsimonious.

in the supplemental material. Finally, we note that with 0.22 CD on 2D CSG domain, SIRI out-performs UCSG-Net [16] (0.32 CD), the previous state-of-the art method for 2D CSG.

We also evaluate how SIRI impacts the training convergence in Figure 5, plotting the Objective $\mathcal{O}$ against iterations (left) and wall-clock time (right). Wall-clock time includes the time required to execute the rewriters. Though NRI starts with a high performance, it eventually converges to a lower value of $\mathcal{O}$ despite expending a lot of time for rewriting. In contrast, SIRI is able to achieve a higher performance while also amortizing the cost of rewriting all the programs as the model generalizes the useful patterns present in the rewritten training programs.

Data Scarcity: Often, large datasets of example shapes are not easily available. Thus, we probe the efficacy of SIRI and PLAD under data scarcity. We present our experiment in Figure 6, plotting the validation set CD for PLAD and SIRI. With 100% data, SIRI surpasses PLAD by 0.29 CD, where as at 10% data, SIRI outperforms PLAD by a margin of 1.33 CD. Since PLAD relies solely on neurally-guided search to discover good programs, which is dependant on the dataset size, reduction in dataset size hurts their performance. In contrast, as SIRI employs rewriters such as *PO* and *CP* which are invariant to the training dataset size, it outperforms PLAD in low-data regimes.

5.3. Test-Time Rewriting

We now show how combining SIRI with additional rewriting at test-time allows performance that matches state-of-the-art methods specialized for 3D CSG reconstruction, while producing much more parsimonious programs. Specifically, we now compare SIRI against CSG-Stump [24], a state-of-the-art method with a neural architecture designed specifically for CSG reconstruction.

We train CSG-StumpNet with the author-released code on our dataset. We compare against two versions: CSG-Stump 32 with 32 intersection and union nodes and CSG-Stump 256 with 256 intersection and union nodes. Since the authors originally trained their model independently for each class, we also compare against a class-specific (cs) version of CSG-Stump 256, where we use pretrained class-specific models released by the authors.

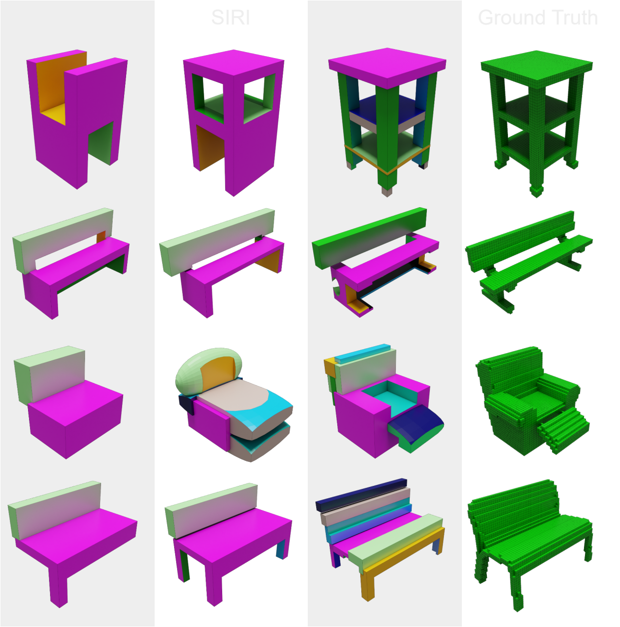


Figure 7. We present qualitative examples of our method and PLAD [15]. SIRI outperforms PLAD and test-time rewriting improves it further.

We show our results in Table 2. We see that SIRI achieves 0.1 CD lower than CSG-Stump 256, despite being domain agnostic. More importantly, it achieves this with a fraction of primitives and operations. Secondly, when accompanied by test-time rewrites, SIRI achieves similar CD to CSG-Stump 256 (cs), which has models trained *individually* for each class. SIRI achieves this while having an order of magnitude fewer primitives and operators (yielding more interpretable and editable programs). In Figure 1, we visualize the difference in inferred program size by rendering their executed shapes with colored primitives.

As the programs inferred by SIRI are parsimonious, applying TTR to the inferred programs takes only 14.6 seconds per shape on an average. In contrast, the over parameterized programs inferred by CSG-Stump are not amenable to fast test-time-optimization (cf. supplemental). Instead, prior-works [41, 40] fine-tune the network itself for each test-shape, which, for CSG-Stump, requires $\sim$ 30 minutes per shape [40]. Moreover, while network fine-tuning can increase reconstruction accuracy, the inferred programs remain incomprehensibly large.

Test-time rewrites are beneficial for PLAD inferred programs as well. However, we find it best to both (i) train with rewriters and (ii) use them at inference time. We compare test-time rewrites on models trained with PLAD and SIRI and report the results in Table 3. For both 3D CSG and Sha-

	3D CSG		ShapeAssembly	
	CD	Length	CD	Length
PLAD + TTR	1.13	**15.80**	1.66	**8.78**
SIRI + TTR	**0.83**	15.96	**1.46**	8.80

Table 3. Using test-time rewrites with SIRI is superior to using it with PLAD across domains.

	Search	only *PO*	only *CP*	only *CG*	1-TTR	3-TTR
IoU	76.8	81.6	76.8	81.98	87.7	90.5
Length	6.81	6.81	6.62	14.6	13.52	15.95
Time (s)	1.44	1.54	0.12	2.81	4.6	14.6

Table 4. Applying only a single rewriter during test-time, though beneficial, has limited value. Interleaved application of the rewriters which we propose results in a larger improvement. Here, n-*TTR* denotes interleaved application of each rewriters n times.

peAssembly, using test-time rewrites on SIRI is more beneficial than using them on PLAD. Apart from yielding better initialization for the inferred programs, training with SIRI also equips the *CG* rewriter with a program cache filled with useful sub-expressions; this cache bolsters *CG*'s efficacy at test-time rewriting. Note that SIRI + TTR programs have only a marginal increase in length over PLAD + TTR.

As described in Section 3.3, we interleave the application of our three rewriters for test-time rewriting, allowing changes to both the structure and continuous parameters of the program. In Table 4, we evaluate the impact of each individual rewriter. Both *PO* and *CG* improve upon the program inferred by beam search while taking only a few seconds. Combining them further improves performance while keeping the programs relatively parsimonious.

6. Conclusion

We introduced *Sparse Intermittent Rewrite Injection* (SIRI), a paradigm for improving unsupervised training of visual-program inference models with code rewriting. We implemented a family of code rewriters that generalized across multiple 2D and 3D shape-program domains. With this family of code rewriters, SIRI learns better VPI networks compared with bootstrap learning methods that ignore rewriters, or use them in a naive fashion. Beyond this, we demonstrated that our rewriters can be employed in a test-time rewriting (TTR) scheme to improve predictions made by SIRI. We found that this SIRI + TTR paradigm is able to match or surpass the reconstruction performance of specially designed neural VPI architectures, while maintaining a much more parsimonious program representation.

In future work, we would like to explore how additional code-rewriting operations could be effectively integrated into our family of rewrites for SIRI + TTR paradigms. While we find SIRI empirically effective for bootstrapped learning, it remains unclear how code rewriting families can best aid RL and end-to-end learning paradigms. Looking forward, we believe that principled use of code-rewriters is a promising way to guide the search of learning-based VPI models, merging domain-specific preferences with neural guidance, and would be a key component of VPI systems designed for complex, real-world domains.

Acknowledgment

We would like to thank the anonymous reviewers for their helpful suggestions. This work was funded in parts by NSF award #1941808 and a Brown University Presidential Fellowship. Daniel Ritchie is an advisor to Geopipe and owns equity in the company. Geopipe is a start-up that is developing 3D technology to build immersive virtual copies of the real world with applications in various fields, including games and architecture.

References

[1] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. In *Advances in Neural Information Processing Systems*, 2017. 2

[2] Matthew Bowers, Theo X. Olausson, Lionel Wong, Gabriel Grand, Joshua B. Tenenbaum, Kevin Ellis, and Armando Solar-Lezama. Top-down synthesis for library learning. *Proc. ACM Program. Lang.*, 7(POPL), 2023. 3

[3] David Cao, Rose Kunkel, Chandrakana Nandi, Max Willsey, Zachary Tatlock, and Nadia Polikarpova. Babble: Learning better abstractions with e-graphs and anti-unification. *Proc. ACM Program. Lang.*, 7(POPL), 2023. 3

[4] Alexandre Carlier, Martin Danelljan, Alexandre Alahi, and Radu Timofte. Deepsvg: A hierarchical generative network for vector graphics animation. In *Advances in Neural Information Processing Systems*, 2020. 2

[5] Angel X. Chang, Thomas Funkhouser, Leonidas Guibas, Pat Hanrahan, Qixing Huang, Zimo Li, Silvio Savarese, Manolis Savva, Shuran Song, Hao Su, Jianxiong Xiao, Li Yi, and Fisher Yu. ShapeNet: An Information-Rich 3D Model Repository. Technical report, Stanford University — Princeton University — Toyota Technological Institute at Chicago, 2015. 7

[6] Tao Du, Jeevana Priya Inala, Yewen Pu, Andrew Spielberg, Adriana Schulz, Daniela Rus, Armando Solar-Lezama, and Wojciech Matusik. InverseCSG: Automatic conversion of 3D models to CSG trees. *ACM TOG*, 2018. 3

[7] Kevin Ellis, Lucas Morales, Mathias Sablé-Meyer, Armando Solar-Lezama, and Joshua B. Tenenbaum. Library Learning for Neurally-Guided Bayesian Program Induction. In *NeurIPS*, 2018. 3, 4, 7

[8] Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. Write, execute, assess: Program synthesis with a repl. In *NeurIPS*, 2019. 2, 7

[9] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. Dreamcoder: Bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021. 3, 4, 7

[10] Meng Fang, Cheng Zhou, Bei Shi, Boqing Gong, Weitao Xi, Tianzhou Wang, Jia Xu, and Tong Zhang. DHER: Hindsight experience replay for dynamic goals. In *International Conference on Learning Representations*, 2019. 2

[11] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 2017. 2

[12] Yiwei Hu, Paul Guerrero, Milos Hasan, Holly Rushmeier, and Valentin Deschaintre. Node graph optimization using differentiable proxies. In *ACM SIGGRAPH Conference Proceedings*, 2022. 2

[13] 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. 1, 2, 5, 6

[14] 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. 3

[15] 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, 2, 3, 4, 6, 7, 8

[16] 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, 5, 6, 8

[17] Chen Lian, Jonathan Berant, Quoc V. Le, Ken Forbus, and Ni Lao. Neural symbolic machines: Learning semantic parsers on freebase with weak supervision. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, 2017. 2, 6

[18] Chen Liang, Mohammad Norouzi, Jonathan Berant, Quoc V Le, and Ni Lao. Memory augmented policy optimization for program synthesis and semantic parsing. In *Advances in Neural Information Processing Systems*. Curran Associates Inc., 2018. 2

[19] Robert McCarthy and Stephen J. Redmond. Imaginary hindsight experience replay: Curious model-based learning for sparse reward tasks, 2021. 2

[20] 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. 3

[21] Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. Synthesizing structured cad models with equality saturation and inverse transformations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, 2020. 3

[22] Ofek Pearl, Itai Lang, Yuhua Hu, Raymond A. Yeh, and Rana Hanocka. Geocode: Interpretable shape programs, 2022. 5

[23] Silviu Pitis, Elliot Creager, and Animesh Garg. Counterfactual data augmentation using locally factored dynamics. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, Red Hook, NY, USA, 2020. Curran Associates Inc. 2

[24] 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, 3, 5, 8

[25] Daxuan Ren, Jianmin Zheng, Jianfei Cai, Jiatong Li, and Junzhe Zhang. Extrudenet: Unsupervised inverse sketch-and-extrude for shape parsing. In *ECCV*, 2022. 2

[26] 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. 1, 2, 3, 6, 7

[27] Gopal Sharma, Difan Liu, Evangelos Kalogerakis, Subhransu Maji, Siddhartha Chaudhuri, and Radomír Měch. Parsenet: A parametric surface fitting network for 3d point clouds, 2020. 3

[28] Liang Shi, Beichen Li, Miloš Hašan, Kalyan Sunkavalli, Tamy Boubekeur, Radomir Mech, and Wojciech Matusik. Match: Differentiable material graphs for procedural material capture. *ACM Trans. Graph.*, 2020. 3

[29] Yonglong Tian, Andrew Luo, Xingyuan Sun, Kevin Ellis, William T. Freeman, Joshua B. Tenenbaum, and Jiajun Wu. Learning to Infer and Execute 3D Shape Programs. In *ICLR*, 2019. 2

[30] Shubham Tulsiani, Hao Su, Leonidas J. Guibas, Alexei A. Efros, and Jitendra Malik. Learning shape abstractions by assembling volumetric primitives. In *Computer Vision and Pattern Regognition (CVPR)*, 2017. 2

[31] Abhinav Verma, Hoang Le, Yisong Yue, and Swarat Chaudhuri. Imitation-projected programmatic reinforcement learning. In *Advances in Neural Information Processing Systems*, 2019. 2

[32] Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically interpretable reinforcement learning. In Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018*, 2018. 2

[33] Karl D. D. Willis, Yewen Pu, Jieliang Luo, Hang Chu, Tao Du, Joseph G. Lambourne, Armando Solar-Lezama, and Wojciech Matusik. Fusion 360 gallery: A dataset and environment for programmatic cad construction from human design sequences. *ACM Trans. Graph.*, 2021. 1

[34] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. Egg: Fast and

extensible equality saturation. *Proc. ACM Program. Lang.*, 5(POPL), 2021. 3

[35] Chenming Wu, Haisen Zhao, Chandrakana Nandi, Jeffrey I Lipton, Zachary Tatlock, and Adriana Schulz. Carpentry compiler. *ACM Transactions on Graphics (TOG)*, 2019. 3

[36] Rundi Wu, Chang Xiao, and Changxi Zheng. Deepcad: A deep generative network for computer-aided design models. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2021. 1

[37] Xianghao Xu, Wenzhe Peng, Chin-Yi Cheng, Karl D. D. Willis, and Daniel Ritchie. Inferring cad modeling sequences using zone graphs. In *CVPR*, 2021. 1

[38] Xiang Xu, Karl DD Willis, Joseph G Lambourne, Chin-Yi Cheng, Pradeep Kumar Jayaraman, and Yasutaka Furukawa. Skexgen: Autoregressive generation of cad construction sequences with disentangled codebooks. In *International Conference on Machine Learning*, 2022. 1

[39] Yuezhi Yang and Hao Pan. Discovering design concepts for CAD sketches. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*, 2022. 3

[40] Fenggen Yu, Qimin Chen, Maham Tanveer, Ali Mahdavi Amiri, and Hao Zhang. Dualcsg: Learning dual csg trees for general and compact cad modeling, 2023. 2, 3, 8

[41] Fenggen Yu, Zhiqin Chen, Manyi Li, Aditya Sanghi, Hooman Shayani, Ali Mahdavi-Amiri, and Hao Zhang. Capri-net: Learning compact cad shapes with adaptive primitive assembly. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022. 2, 3, 8

[42] Haisen Zhao, Max Willsey, Amy Zhu, Chandrakana Nandi, Zachary Tatlock, Justin Solomon, and Adriana Schulz. Co-optimization of design and fabrication plans for carpentry. *ACM Transactions on Graphics (TOG)*, 2022. 3