Supplemental material for ShapeCoder: Discovering Abstractions for Visual Programs from Unstructured Primitives

1 DISCOVERED LIBRARIES

We include discovered libraries, along with program datasets rewritten with those libraries in our public code release, which can be found at https://github.com/rkjones4/ShapeCoder/. For each of these datasets, we also include simple visualizations of how shapes in the dataset make use of these abstractions.

2 TOY 2D GRAMMAR EXPERIMENTS

Before moving to 3D domains, we evaluated ShapeCoder's ability to discover abstractions on a more basic 2D dataset. We designed a 2D shape grammar (described in Appendix A, of the main paper), and manually designed a sampling procedure for this grammar that would produce 'chair-like' output scenes (combinations of rectangles). This grammar is included in our public code release at https://github.com/rkjones4/ShapeCoder/. The sampling procedure was, in fact, implemented as a single abstraction function, that takes a fixed amount of input parameters and outputs a program using functions from the base DSL. These input parameters controlled both shape parameters (e.g. chair height or width) along with control flow decisions (e.g. should the back have vertical or horizontal bars). This paradigm can be considered as an 'oracle' best-case abstraction for this 2D domain, e.g. what a manually designed abstraction would look like. We evaluate how ShapeCoder was able to improve $\mathcal{F}$ on this dataset, compared with this oracle, in the below table.

Method	$\mathcal{F} \Downarrow$
Input Prims	65.3
No Abstraction	48.6
ShapeCoder	27.3
Oracle	22.7

The oracle single abstraction (that takes in 7 categorical variables, and 9 float variables) is able to achieve the best compression metric. However, ShapeCoder is able to come reasonable close to this target on this toy domain, and improves $\mathcal{F}$ significantly over using either the input primitives, or when only the dream+wake phases are used (No Abstraction). Of note, the oracle abstraction function actually has access to DSL components we don't provide to ShapeCoder (control flow `Switch` and `If`/`Else` operators). Currently ShapeCoder is not able to discover abstractions that introduce different control flow decisions, as these types of operators would never be inferred during the wake phase. As mentioned in the conclusion of the main paper, finding ways to discover useful 'probabilistic' or 'conditional' abstractions is an interesting direction for future work.

3 DREAMCODER EXPERIMENTS

DreamCoder [Ellis et al. 2021] is an inspiring system capable of generalizing across many domains. It makes no assumptions over its input data, which creates a difficult program induction problem. It solves this issue by gradually building up a library of discovered abstractions tailored to the input domain. Dreamcoder's program inference step (i.e. its wake phase) performs enumerative search guided by a library version; when solutions to the program induction task are more compact under a 'good' library version, solutions will be found more quickly in this search process. A downside of this framing is that there is an implicit assumption that the input data contains a curriculum of tasks, that is needed to bootstrap this procedure. Specifically, some tasks in the input set need to have relatively high probability under the base DSL: if enumerative search does not find any solutions to the 'simple' tasks, then no abstractions can ever be discovered, that are necessary to help solve the more 'complex' tasks.

Complex visual programming datasets, like manufactured 3D shapes, don't typically contain a curriculum of tasks. In some cases, a curriculum can be created, but this typically requires access to detailed shape annotations (e.g. a semantic part hierarchy). As such, due to the lack of a curriculum, combined with the complexity of the 3D shape program inference problem, when we attempted to run DreamCoder over PartNet data we observed it did not find any solutions.

Beyond this observation, we also argue that DreamCoder, as a general program induction system, is not as well-suited for visual programming domains, compared with ShapeCoder. Critically, DreamCoder has no mechanism that reasons over parametric relationships between continuous variable, which is of great importance for many visual programming domains (including manufactured shapes, where part-to-part relationships are spatially constrained). While DreamCoder does show success on simple 2D visual domains, it discretizes continuous variables and treats parametric operators as standard functions in the base DSL. To test if DreamCoder has an inductive bias to discover abstractions with parametric relationships under these assumptions, we designed a toy experiment, that we explain below.

We design a very simple shape grammar for the toy 2D language. Where between 1 and 3 primitives are combined together, and where each primitive is created by an abstraction that takes in two input parameters (so two degrees of freedom are constrained). We write this grammar as:

$START \rightarrow ABS \mid$
`Union`(ABS, ABS) $\mid$
`Union`(ABS, `Union`(ABS, ABS)) ;
$ABS \rightarrow$ `Move(Rect(a, a+b), b, a-b)`
a,b $\rightarrow r \in (0, 1)$

Where real-values (e.g. a and b) are discretized into 20 values between 0 and 1. The *ABS* function is easily identifiable by ShapeCoder, because it explicitly checks for these types of parametric relationships during the proposal phase, and this relationship is present in every input scene. We ran DreamCoder over a dataset of 100 samples from this grammar with a budget of 24 hours wall-clock time. To match the computational requirements of ShapeCoder, we used a

single workstation with a Intel i7-11700K CPU, and a python-based executor implementation. Under these conditions, DreamCoder did not discover the 'correct' abstraction with the proper parametric pattern. Moreover, even for this simple grammar, DreamCoder only discovered solutions for around 50% of the tasks (and none of the tasks with 3 `Union` operations). While these results might be improved by making better use of computational resources (running enumerative search over a cluster of machines, designing a faster executor, increasing the wall-clock budget), we believe this example illustrates why DreamCoder is not particularly well-suited for complex visual programming domains. Hopefully, some of the insights we develop and uncover in ShapeCoder can prove useful for other, more general programming tasks, that are the main focus of methods like DreamCoder.

4 SHAPEMOD EXPERIMENTS

ShapeMOD [Jones et al. 2021] requires access to a dataset of imperative programs, where the possible line reorderings of each program is known in advance. The programs under our DSL are quite different: they are functional, and sub-program expressions have no known canonical orderings, so in order to compare against ShapeMOD we must constrain and modify these inputs. To turn our functional programs into imperative versions, we linearize each program from our DSL, and modify the grammar, so that each operator first predicts its own parameters (e.g. floats or categorical variables) before taking in any 'shape' typed arguments. As `Union` is the only operator in the language that takes in more than one 'shape' typed argument, we remove it from the grammar, and assume there is an implicit `Union` whenever a new 'shape' typed expression is created. For other 'shape' typed arguments, we always pass in the value created by the subsequent line, and a complete expression has been created (all input arguments have been instantiated). We constrain possible line reoderings of this imperative program to maintain semantic equivalence, by only considering reorderings over the different sub-program expressions. Even this limited set of reorderings creates a possible set of orders that is far too many for ShapeMOD to handle. In ShapeMOD's experiments on ShapeAssembly [Jones et al. 2020] programs, the median number of orders per shape program was less than 5. We therefore pick a random subset of 10 orders of the sub-program expressions in our DSL, linearize them, and pass these as possible program orders into ShapeMOD. Without this step, ShapeMOD took prohibitively long to run, as its integration phase explicitly tries to refactor every single program ordering independently. In contrast ShapeCoder uses e-graphs in its refactor operation, to reason over a larger set of program refactors (including sub-expression reorderings) in a joint process. Under these constraints, ShapeMOD performs suprisingly well, discovering abstractions in a domain and use-case it was not designed for, but clearly, it does not scale as well as ShapeCoder, which is much better suited for domains that lack access to highly structured input programs.

REFERENCES

Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B Tenenbaum. 2021. DreamCoder: Bootstrapping inductive program synthesis with wake-sleep library learning. In *ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (SIGPLAN)*. 835–850.

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

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