
Supplement to: Learning Libraries of Subroutines for Neurally-Guided Bayesian Program Learning

Anonymous Author(s)

Affiliation

Address

email

1 Learning Generative Graphics Programs

2 A natural extension of our work is to consider the problem of learning generative models: here,
3 we would learn programs that generate (either deterministically or probabilistically) structures like
4 images or words. As a first step in this direction, we apply SCC to synthesizing graphics programs.
5 A natural starting DSL for graphics programming is the Logo language (Abelson et al. (1974)), also
6 sometimes called **turtle graphics**. These programs control a pen (sometimes called a “turtle”), and
7 can do things like pick the pen up, move the pen forward, rotate the pen, or trace out a programmat-
8 ically specified arc. We take turtle graphics primitives from prior work Sablé-Meyer & Dehaene
9 (2017). In our setting, we will encapsulate turtle graphics primitives inside of λ -calculus, and seek to
10 infer graphics programs from images: thus the task is to look at an image, and write the program that
11 would have drawn it.

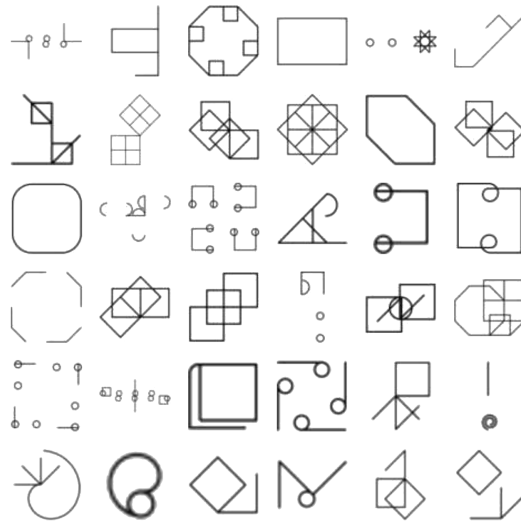


Figure 1: Another example of compiled figures generated by our method.

12 In Figure 1 is another montage with several compiled shapes generated by our method after some
13 training. While not all are that regular, often offer high level structure in the latter run while no
14 structure is to be found at the beginning.
15 The DSL is the following:

16

<i>name</i>	<i>type</i>
Concat	prog → prog → prog
Repeat	var option → prog → prog
Embed	prog → prog
Define	var → prog
Turn	var option → prog
Integrate	var option → bool → var option → var option → prog
True	bool
False	bool
Nothing	var option
Just	var → var option
Unit	var
Name	var
Next	var → var
Prev	var → var
Double	var → var
Half	var → var
Opposite	var → var

17 Some elements of the semantics are common, the others are as follow. Repeat takes a variable
 18 and a prog and repeats said prog n times where n is the evaluation of the variable — if not set it
 19 is defaulted to two. Embed of a prog means that said prog will be executed and then returns to the
 20 current state — leaving what has been drawn in the meantime on the canvas.

21 Integrate is the main instruction and the only one that draws anything:
 22 Integrate(t, p, a, c) takes a time var, a pen bool, an acceleration
 23 var and an angular speed var, and it moves the turtle according to these
 24 parameters — with or without actually drawing depending on the pen
 25 variable p .

26 The default values of these parameters are set such that
 27 Integrate(nothing, true, nothing, nothing) draws a unit
 28 segment, Integrate(nothing, true, nothing, Just(Unit))
 29 draws a circle of unit length, and Integrate(nothing, true,
 30 Just(Unit), Just(unit)) draws the first spire of a spiral. Playing
 31 with the first arguments decides on the duration during which the
 32 arguments are integrated.

33 In the var type, most are self explicit, while Opposite(v) takes the
 34 opposite of v . The behaviour of Name was originally designed to handle
 35 arbitrary variables in a call by name fashion but was latter reduced to a
 36 single storage location, which proves to be enough for the targeted shapes.
 37 One can therefore store elements in that placeholder using Define and
 38 retrieve it through Name.

39 Because of these defaults, the following program draws a cross:

```
Cross = Repeat(Double(Double(Unit)), Concat(Embed(Integrate), Turn(None)))
```

40 Describing highly regular complex shapes in this language is easy to do as a human but quickly
 41 escapes the reach of naïve enumeration search. By using a curriculum of shapes our approach
 42 compresses the search in the corresponding directions — another way to say this is that upon being
 43 given a dataset of shapes, it picks up the simple ones and abstract them as building blocks for latter
 44 staged of search.

45 For example, the simplest way to draw a square in this language is already of length 12. Placing
 46 several around, for example to draw a grid is out of reach of the initial search. However by first

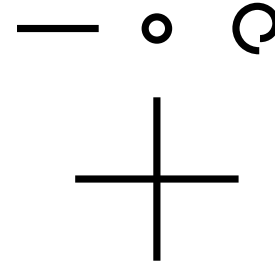


Figure 2: Top: Some defaults for Integrate. Bottom, the result of the Cross example below.

47 abstracting as a primitive the segment — thus reducing the length by four —, then assuming that
 48 after a segment it often needs to draw something else, then abstracting the first arguments of Repeat
 49 — further reducing the length by two — as well as the one of Turn and finally making the square a
 50 primitive on its own once it starts using it often enough, the length of this particular shape drops.

New Primitive	Type	Definition
Segment = f_0	prog	Concat(Nothing, True, Nothing, Nothing)
Right-angle = f_1	prog	Turn(Nothing)
RepeatTwice = f_2	prog \rightarrow prog	$\lambda p. \text{Repeat}(\text{nothing}, p)$
AddToUnitSegment = f_3	prog \rightarrow prog	$\lambda p. \text{Concat}(f_0, p)$
Square = f_4	prog	$f_3(f_3(f_2(f_1)))$

Table 1: How our method compresses the square step by step. On the example given in the main article this was produced by the compressor after the second search phase and leads to the second jump in success, the first one being the abstraction of the Segment. Names are *not* produced by the compressor and are here as indication to help the reader.

51 The underlying hypothesis is that the new primitive distort the space of search toward something
 52 that looks more like what human actually produce and moves away from semantically valid but
 53 meaningless programs — in a sense, learns to care about what *matters* rather than what is *true* in a
 54 very pragmatic-like way.

55 In Figure 3 are listed all the shapes used for this project without particular order.

56 Since this is a first step in broader project of program induction for abstract geometry the likelihood
 57 is currently all-or-none — ongoing work moves this to a neural net based distance function to abstract
 58 away from noise in the shapes.

59 The result presented in the main article describe a sample of tasks and compiled new shapes on the top
 60 row. On the bottom row is a condensed description of a run where are displayed the mean of typical
 61 compiled programs on both sides, once before any training and once after the last iteration — note
 62 how the probability weighting shaped the output space to add structure. In the middle is the learning
 63 rate measured by holding some tasks out, training on all the other, and with each iteration measuring
 64 the success rate on the unseen tasks. In the given example the split was 25% test and 73% train.

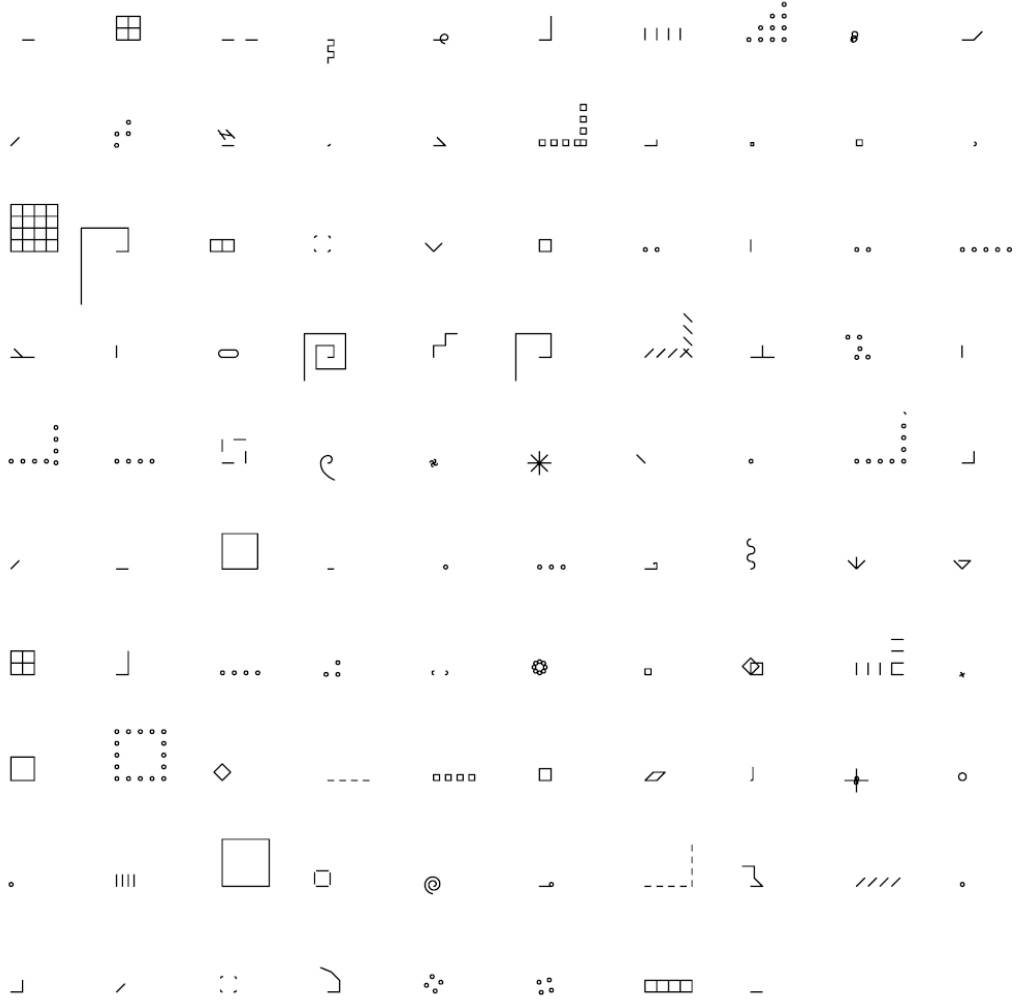


Figure 3: The set of tasks for the geometry domain

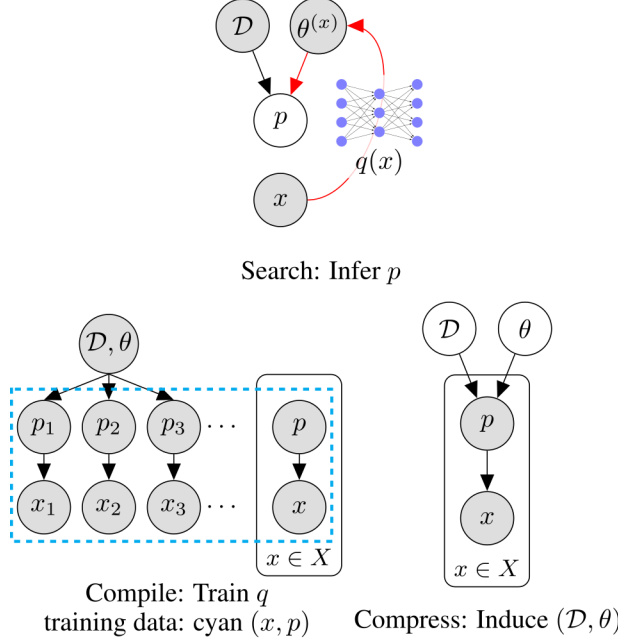
2 An Illustration of the 3 iterations of our algorithm

Below we diagram the iterations employed by our algorithm. At each stage of the algorithm, we have shaded the observed variables in gray and left the unobserved variables white. Black lines correspond to a connection from the top-down generative model, while red lines correspond to connections from the bottom-up recognition model.

3 Program Representation

We choose to represent programs using λ -calculus Pierce (2002). A λ -calculus expression is either:

- A *primitive*, like the number 5 or the function `sum`.
- A *variable*, like x , y , or z .
- A λ -*abstraction*, which creates a new function. λ -abstractions have a variable and a body. The body is a λ -calculus expression. Abstractions are written as $\lambda\text{var}.\text{body}$ or in Lisp syntax as `(lambda (var) body)`.



77 – An *application* of a function to an argument. Both the function and the argument are
 78 λ -calculus expressions. The application of the function f to the argument x is written as
 79 $f\ x$ or as $(f\ x)$.

80 For example, the function which squares the logarithm of a number is $\lambda x.(\text{square } (\log x))$,
 81 and the identity function $f(x) = x$ is $\lambda x.x$. The λ -calculus serves as a spartan but expressive
 82 Turing complete program representation, and distills the essential features of functional programming
 83 languages like Lisp.

84 However, many λ -calculus expressions correspond to ill-typed programs, such as the program that
 85 takes the logarithm of the Boolean `true` (i.e., $\log\ \text{true}$) or which applies the number five to the
 86 identity function (i.e., $5\ (\lambda x.x)$). We use a well-established typing system for λ -calculus called
 87 *Hindley-Milner typing* Pierce (2002), which is used in programming languages like OCaml. The
 88 purpose of the typing system is to ensure that our programs never call a function with a type it
 89 is not expecting (like trying to take the logarithm of `true`). Hindley-Milner has two important
 90 features: Feature 1: It supports *parametric polymorphism*, meaning that types can have variables in
 91 them, called *type variables*. Lowercase Greek letters are conventionally used for type variables. For
 92 example, the type of the identity function is $\alpha \rightarrow \alpha$, meaning it takes something of type α and return
 93 something of type α . A function that returns the first element of a list has the type $[\alpha] \rightarrow \alpha$. Type
 94 variables are not the same as variables introduced by λ -abstractions. Feature 2: Remarkably, there is
 95 a simple algorithm for automatically inferring the polymorphic Hindley-Milner type of a λ -calculus
 96 expression Damas & Milner (1982). Our generative model over programs performs Hindley-Milner
 97 type inference during sampling: *Unify* in the generative model uses the machinery of Hindley-Milner
 98 to ensure that the generated programs have valid polymorphic types. A satisfactory exposition of
 99 Hindley-Milner is beyond the scope of this paper, but Pierce (2002) offers a nice overview of lambda
 100 calculus and typing systems like Hindley-Milner.

101 4 Generative model over the programs

102 Alg. 1 is a procedure for drawing samples from the generative model (\mathcal{D}, θ) . In practice, we enumerate
 103 programs in order of their probability under Alg. 1 rather than sample them.

Algorithm 1 Generative model over programs

```
function sample( $\mathcal{D}, \theta, \mathcal{E}, \tau$ ):  
Input: DSL ( $\mathcal{D}, \theta$ ), environment  $\mathcal{E}$ , type  $\tau$   
Output: a program whose type unifies with  $\tau$   
if  $\tau = \alpha \rightarrow \beta$  then  
    var  $\leftarrow$  an unused variable name  
    body  $\sim$  sample( $\mathcal{D}, \theta, \{\text{var} : \alpha\} \cup \mathcal{E}, \beta$ )  
    return (lambda (var) body)  
end if  
primitives  $\leftarrow \{p | p : \tau' \in \mathcal{D} \cup \mathcal{E}$   
    if  $\tau$  can unify with yield( $\tau'$ ) $\}$   
Draw  $e \sim$  primitives, w.p.  $\propto \theta_e$  if  $e \in \mathcal{D}$   
    w.p.  $\propto \frac{\theta_{var}}{|\text{variables}|}$  if  $e \in \mathcal{E}$   
Unify  $\tau$  with yield( $\tau'$ ).  
 $\{\alpha_k\}_{k=1}^K \leftarrow \text{args}(\tau')$   
for  $k = 1$  to  $K$  do  
     $a_k \sim$  sample( $\mathcal{D}, \theta, \mathcal{E}, \alpha_k$ )  
end for  
return ( $e \ a_1 \ a_2 \ \dots \ a_K$ )  
where:  
yield( $\tau$ ) =  $\begin{cases} \text{yield}(\beta) & \text{if } \tau = \alpha \rightarrow \beta \\ \tau & \text{otherwise.} \end{cases}$   
args( $\tau$ ) =  $\begin{cases} [\alpha] + \text{args}(\beta) & \text{if } \tau = \alpha \rightarrow \beta \\ [] & \text{otherwise.} \end{cases}$ 
```

104 5 Neural Recognition Model Architecture

105 The neural recognition model regresses from an observation (set of input/output pairs: $\{(i_n, o_n)\}_{n \leq N}$)
106 to a $|\mathcal{D}| + 1$ dimensional vector. Each input/output pair is processed by an identical encoder network;
107 the outputs of the encoders are average and passed to an MLP with 1 hidden layer, 32 hidden units,
108 and a ReLU activation:

$$q(x) = \text{MLP} \left(\text{Average} \left(\{\text{encoder}(i_n, o_n)\}_{n \leq N} \right) \right) \quad (1)$$

109 For the string editing and list domains, the inputs and outputs are sequences. Our encoder for these
110 domains is a bidirectional GRU with 64 hidden units that reads each input/output pair; we concatenate
111 the input and output along with a special delimiter symbol between them. We MaxPool the final
112 hidden unit activations in the GRU along both passes of the bidirectional GRU.

113 For symbolic regression, the input/outputs are densely sampled points along the curve of the function.
114 We rendered these points to a graph, and pass the image of the graph to a convolutional network,
115 which acts as the encoder.

116 6 DSL Induction

117 6.1 Structure Learning

118 We use Alg. 3 to search for the structure of the DSL that best explains the frontiers.

Algorithm 3 DSL Induction Algorithm

Input: Set of frontiers $\{\mathcal{F}_x\}$
Hyperparameters: Pseudocounts α , regularization parameter λ
Output: DSL \mathcal{D} , weight vector θ
 Define $L(\mathcal{D}, \theta) = \prod_x \sum_{p \in \mathcal{F}_x} \mathbb{P}[p|\mathcal{D}, \theta]$
 Define $\theta^*(\mathcal{D}) = \arg \max_{\theta} \text{Dir}(\theta|\alpha) L(\mathcal{D}, \theta)$
 Define $\text{score}(\mathcal{D}) = \log \mathbb{P}[\mathcal{D}] + L(\mathcal{D}, \theta^*) - \|\theta\|_0$
 $\mathcal{D} \leftarrow$ every primitive in $\{\mathcal{F}_x\}$
while true **do**
 $N \leftarrow \{\mathcal{D} \cup \{s\} | x \in X, p \in \mathcal{F}_x, s \text{ a fragment of } p\}$
 $\mathcal{D}' \leftarrow \arg \max_{\mathcal{D}' \in N} \text{score}(\mathcal{D}')$
 if $\text{score}(\mathcal{D}') < \text{score}(\mathcal{D})$ **return** $\mathcal{D}, \theta^*(\mathcal{D})$
 $\mathcal{D} \leftarrow \mathcal{D}'$
end while

6.2 Estimating θ

We use an EM algorithm to estimate the continuous parameters of the DSL, e.g. θ . Suppressing dependencies on \mathcal{D} , the EM updates are

$$\theta = \arg \max_{\theta} \log P(\theta) + \sum_x \mathbb{E}_{Q_x} [\log \mathbb{P}[p|\theta]] \quad (2)$$

$$Q_x(p) \propto \mathbb{P}[x|p] \mathbb{P}[p|\theta] \quad (3)$$

In the M step of EM we will update θ by instead maximizing a lower bound on $\log \mathbb{P}[p|\theta]$, making our approach an instance of Generalized EM.

We write $c(e, p)$ to mean the number of times that primitive e was used in program p ; $c(p) = \sum_{e \in \mathcal{D}} c(e, p)$ to mean the total number of primitives used in program p ; $R(p)$ to mean the sequence of types input to sample in Alg. 1 of the main paper. Jensen's inequality gives a lower bound on the likelihood:

$$\begin{aligned}
 \sum_x \mathbb{E}_{Q_x} [\log \mathbb{P}[p|\theta]] &= \\
 \sum_{e \in \mathcal{D}} \log \theta_e \sum_x \mathbb{E} [c(e, p_x)] - \sum_{\tau} \mathbb{E} \left[\sum_x c(\tau, p_x) \right] \log \sum_{\substack{e: \tau' \in \mathcal{D} \\ \text{unify}(\tau, \tau')}} \theta_e \\
 &= \sum_e C(e) \log \theta_e - \beta \sum_{\tau} \frac{\mathbb{E} [\sum_x c(\tau, p_x)]}{\beta} \log \sum_{\substack{e: \tau' \in \mathcal{D} \\ \text{unify}(\tau, \tau')}} \theta_e \\
 &\geq \sum_e C(e) \log \theta_e - \beta \log \sum_{\tau} \frac{\mathbb{E} [\sum_x c(\tau, p_x)]}{\beta} \sum_{\substack{e: \tau' \in \mathcal{D} \\ \text{unify}(\tau, \tau')}} \theta_e \\
 &= \sum_e C(e) \log \theta_e - \beta \log \sum_{\tau} \frac{R(\tau)}{\beta} \sum_{\substack{e: \tau' \in \mathcal{D} \\ \text{unify}(\tau, \tau')}} \theta_e
 \end{aligned}$$

where we have defined

$$\begin{aligned}
 C(e) &\triangleq \sum_x \mathbb{E} [c(e, p_x)] \\
 R(\tau) &\triangleq \mathbb{E} \left[\sum_x c(\tau, p_x) \right] \\
 \beta &\triangleq \sum_{\tau} \mathbb{E} \left[\sum_x c(\tau, p_x) \right]
 \end{aligned}$$

Crucially it was defining β that let us use Jensen’s inequality. Recalling from the main paper that $P(\theta) \triangleq \text{Dir}(\alpha)$, we have the following lower bound on M-step objective:

$$\sum_e (C(e) + \alpha) \log \theta_e - \beta \log \sum_{\tau} \frac{R(\tau)}{\beta} \sum_{\substack{e: \tau' \in \mathcal{D} \\ \text{unify}(\tau, \tau')}} \theta_e \quad (4)$$

Differentiate with respect to θ_e , where $e : \tau$, and set to zero to obtain:

$$\frac{C(e) + \alpha}{\theta_e} \propto \sum_{\tau'} \mathbb{1} [\text{unify}(\tau, \tau')] R(\tau') \quad (5)$$

$$\theta_e \propto \frac{C(e) + \alpha}{\sum_{\tau'} \mathbb{1} [\text{unify}(\tau, \tau')] R(\tau')} \quad (6)$$

The above is our estimator for θ_e . Despite the convoluted derivation, the above estimator has an intuitive interpretation. The quantity $C(e)$ is the expected number of times that we used e . The quantity $\sum_{\tau'} \mathbb{1} [\text{unify}(\tau, \tau')] R(\tau')$ is the expected number of times that we *could have* used e . The hyperparameter α acts as pseudocounts that are added to the number of times that we used each primitive, and are not added to the number of times that we could have used each primitive.

We are only maximizing a lower bound on the log posterior; when is this lower bound tight? This lower bound is tight whenever all of the types of the expressions in the DSL are not polymorphic, in which case our DSL is equivalent to a PCFG and this estimator is equivalent to the inside/outside algorithm. Polymorphism introduces context-sensitivity to the DSL, and exactly maximizing the likelihood with respect to θ becomes intractable, so for domains with polymorphic types we use this estimator.

7 Hyperparameters & Implementation Details

We set structure penalty $\lambda = 1$ (Eq. 5 of the main paper) and smoothness parameter $\alpha = 10$ (Eq. 6 of the main paper) for all experiments. For list processing and text editing we used a search timeout of two hours; because the symbolic regression problems are easier, we used a timeout of only five minutes for these.

Because the frontiers can become very large in later iterations of the algorithm, we only keep around the top 10^4 programs in the frontier \mathcal{F}_x as measured by $\mathbb{P}[x, p | \mathcal{D}, \theta]$.

8 Why not the ELBO Bound?

Our lower bound \mathcal{L} is unconventional, and one might wonder why we do not instead maximize an ELBO-style bound like in a VAE or in the EM algorithm. Surprisingly, maximizing an ELBO-style bound leads to a pathological behavior that causes the model to easily become trapped in local optima.

If we were to maximize the ELBO bound to perform inference in our generative model, then, during DSL induction, we would seek a new $(\mathcal{D}^*, \theta^*)$ maximizing the following lower bound on the likelihood (along with an unimportant regularizing term on the DSL):

$$\sum_{x \in X} \mathbb{E}_{p \sim Q_x} [\log \mathbb{P}[p | \mathcal{D}^*, \theta^*]] \quad (7)$$

$$Q_x(p) \triangleq \mathbb{P}[p | x, \mathcal{D}, \theta] \quad (8)$$

where (\mathcal{D}, θ) is our current estimate of the generative model. These equations fall out of an EM-style derivation, and one could replace $Q_x(p)$ with the recognition model $q(p|x)$, either using importance sampling (so the expectation in Eq. 7 is taken over q and we reweigh using Q_x) or by directly using q as our approximate posterior over the program that solves task x .

We do not maximize a bound of this form because it takes an expectation over the *previous* iteration’s posterior over programs, so the approximate posterior Q_x at the next iteration ends up being very close to previous approximate posterior. Intuitively, we want the DSL induction to be a function *only* of the programs that we have found, and *not* be a function of how the previous generative model

weighed them. In practice, we found that maximizing EM-style bounds, like the ELBO, leads to a kind of hysteresis effect, where the next generative model too closely matches the previous one, causing the algorithm to easily become trapped in local optima.

9 List Processing Data Set

Each list processing tasks we created is in described in Tbl 2.

add-k for $k \in \{0..5\}$ append-index-k for $k \in \{1..5\}$ append-k for $k \in \{0..5\}$ bool-identify-geq-k for $k \in \{0..5\}$ bool-identify-is-mod-k for $k \in \{1..5\}$ bool-identify-is-prime bool-identify-k for $k \in \{0..5\}$ caesar-cipher-k-modulo-n for $k \in \{0..5\}$ and $n \in \{1..5\}$ count-head-in-tail count-k for $k \in \{0..5\}$ drop-k for $k \in \{0..5\}$ dup empty evens fibonacci has-head-in-tail has-k for $k \in \{0..5\}$ head index-head index-k for $k \in \{1..5\}$ is-evens is-mod-k for $k \in \{1..5\}$ is-odds is-primes is-squares keep-eq-k for $k \in \{0..3\}$ keep-gt-k for $k \in \{0..3\}$ keep-mod-head keep-mod-k for $k \in \{1..5\}$ keep-primes keep-squares	kth-largest for $k \in \{1..5\}$ kth-smallest for $k \in \{1..5\}$ last len max min modulo-k for $k \in \{1..5\}$ mult-k for $k \in \{0..5\}$ odds pop pow-k for $k \in \{1..5\}$ prepend-index-k for $k \in \{1..5\}$ prepend-k for $k \in \{0..5\}$ product range remove-empty-lists remove-eq-k for $k \in \{0..3\}$ remove-gt-k for $k \in \{0..3\}$ remove-index-k for $k \in \{1..5\}$ remove-mod-head remove-mod-k for $k \in \{2..5\}$ repeat repeat-k for $k \in \{1..5\}$ repeat-many replace-all-with-index-k for $k \in \{1..5\}$ reverse rotate-k for $k \in \{1..5\}$ slice-k-n for $k \in \{1..5\}$ and $n \in \{1..5\}$ sort sum tail take-k for $k \in \{1..5\}$
---	--

Table 2: Our list processing data set

10 Learned DSLs

Here we present representative DSLs learned by our model. DSL primitives discovered by the algorithm are prefixed with #. Variables are prefixed with \$, and we adopt De Bruijn indices to model bound variables Pierce (2002).

174 10.1 List processing

```

175 #(+ 1 1)
176 #(\ (cdr (cdr $0)))
177 #(\ (foldr $0 1 (\ (\ (* $0 $1))))))
178 #(\ (cons (car $0) nil))
179 #(\ (\ (foldr $0 $1 (\ (\ (cons $1 $0))))))
180 #(\ (\ (foldr $0 (is-nil $0) (\ (\ (if $0 $0 (eq? $3 $1))))))
181 #(\ (map (\ (eq? $1 $0)))
182 #(\ (* $0 (* $0 $0)))
183 #(+ 1 #(+ 1 1))
184 #(\ (map (\ (gt? $0 $1)))
185 #(\ (foldr $0 nil (\ (\ (if (is-square $1) (cons $1 $0) $0))))
186 #(\ (map (\ (eq? $0 (length (range $0)))) $0))
187 #(\ (\ (map (\ (index $0 $1)) (range $1)))
188 #(\ (cdr (#(\ (cdr (cdr $0))) $0)))
189 #(\ (map (\ (index 1 $1)))
190 #(\ (foldr $0 nil (\ (\ (cons $1 (cons $1 $0))))))
191 #(\ (\ (cons (car $0) $1))
192 #(+ 1 #(+ 1 #(+ 1 1)))
193 #(\ (map (\ (gt? 1 (mod $0 $1))))
194 #(\ (\ (map (\ (mod (+ $0 $1) $2))))
195 #(\ (#(\ (\ (foldr $0 $1 (\ (\ (cons $1 $0)))))) (#(\ (\ (foldr $0 $1 (\
196   ↪ (\ (cons $1 $0)))))) $0 $0) $0))
197 #(\ (\ (#(\ (\ (foldr $0 $1 (\ (\ (cons $1 $0)))))) (cons $0 nil) $1)))
198 #(+ #(+ 1 #(+ 1 1)) #(+ 1 1))
199 #(\ (map (\ (+ #(+ 1 #(+ 1 1)) (+ $1 $0))))
200 #(\ (map (\ (+ $0 $1)))
201 #(\ (\ (foldr $0 (is-nil $0) (\ (\ (gt? $1 (#(\ (* $0 (* $0 $0)))
202   ↪ $3))))))
203 #(\ (foldr $0 0 (\ (\ (+ $0 (#(\ (foldr $0 1 (\ (\ (* $0 $1)))) (range
204   ↪ $1))))))
205 #(\ (#(\ (cdr (#(\ (cdr (cdr $0))) $0))) (cdr $0)))
206 #(\ (#(\ (foldr $0 nil (\ (\ (if (is-square $1) (cons $1 $0) $0))))
207   ↪ (map (\ (* $0 (+ $0 $0)) $0)))
208 #(\ (\ (#(\ (\ (foldr $0 $1 (\ (\ (cons $1 $0)))))) (#(\ (cons (car $0)
209   ↪ nil)) $0) $1)))
210 #(\ (\ (length (#(\ (#(\ (foldr $0 nil (\ (\ (if (is-square $1) (cons $1
211   ↪ $0) $0)))))) (map (\ (* $0 (+ $0 $0)) $0))) (map (\ (- $1 $0)
212   ↪ $1))))
213 #(\ (\ (is-square (#(\ (foldr $0 1 (\ (\ (* $0 $1)))) (#(\ (\ (map (\
214   ↪ (mod (+ $0 $1) $2)))))) (length (#(\ (#(\ (\ (foldr $0 $1 (\ (\
215   ↪ (cons $1 $0)))))) (#(\ (\ (foldr $0 $1 (\ (\ (cons $1 $0)))))) $0
216   ↪ $0) $0)) $0)) $1 $0))))
217 #(\ (\ (foldr (cdr $0) $1 (\ (\ (#(\ (\ (#(\ (\ (foldr $0 $1 (\ (\ (cons
218   ↪ $1 $0)))))) (#(\ (cons (car $0) nil)) $0) $1)) (cdr $0) $0))))))
219 #(\ (is-nil (#(\ (#(\ (foldr $0 nil (\ (\ (if (is-square $1) (cons $1
220   ↪ $0) $0)))))) (map (\ (* $0 (+ $0 $0)) $0))) (#(\ (\ (map (\ (mod
221   ↪ (+ $0 $1) $2)))))) #(+ 1 1) 1 $0)))
222 #(\ (\ (gt? (#(\ (\ (length (#(\ (#(\ (foldr $0 nil (\ (\ (if (is-square
223   ↪ $1) (cons $1 $0) $0)))))) (map (\ (* $0 (+ $0 $0)) $0))) (map (\
224   ↪ (- $1 $0) $1)))))) $0 $1) 1)))

```

225 10.2 Text editing

```

226 #(+ 1)
227 #(\ (\ (fold $0 $0 (\ (\ (if (char-eq? $1 $3) nil (cons $1 $0))))))
228 #(\ (\ (fold $0 $0 (\ (\ (cdr (if (char-eq? $1 $3) $2 $0))))))
229 #(\ (\ (fold $0 $1 (\ (\ (cons $1 $0))))))
230 #(\ (\ (#(\ (\ (fold $0 $1 (\ (\ (cons $1 $0)))))) (cons $0 $1)))
231 #(\ (#(\ (\ (\ (cons (car $0) (cons $1 $2)))) (#(\ (\ (\ (cons (car
232   ↪ $0) (cons $1 $2)))))) nil) '.' $0) '.'))
233 #(\ (\ (fold $0 $0 (\ (\ (fold $0 $0 (\ (\ (if (char-eq? $1 $5) (cdr $2)
234   ↪ $0))))))
235 #(\ (\ (map (\ (if (char-eq? $1 $0) $2 $0))))

```

```

236 #(\ (#(\ (\ (fold $0 $1 (\ (\ (cons $1 $0)))))) $0 STRING))
237 #(\ (map (\ (index $0 $1))))
238 #(\ (unfold $0 (\ (nil? $0)) (\ (car $0)) (\ (#(\ (\ (fold $0 $0 (\ (\
239   ↪ (cdr (if (char-eq? $1 $3) $2 $0)))))) SPACE $0))))
240 #(\ (#(\ (\ (\ (cons (car $0) (cons $1 $2)))) nil)

```

241 10.3 Symbolic regression

```

242 #(\ (/ (/ (/ REAL $0) $0))
243 #(\ (+ $0 REAL))
244 #(\ (#(\ (+ $0 REAL)) (* $0 (#(\ (#(\ (+ $0 REAL)) (* (#(\ (#(\ (#(\
245   ↪ (+ $0 REAL)) (* $0 REAL))) (* (#(\ (+ $0 REAL)) $0) $0))) $0)
246   ↪ $0))) $0)))
247 #(\ (/ (#(\ (/ (/ (/ REAL $0) $0)) $0) $0))
248 #(\ (\ (#(/ REAL) (/ (#(\ (+ $0 REAL)) $0) $1))))
249 #(\ (#(\ (+ $0 REAL)) (#(\ (#(/ REAL) (#(\ (+ $0 REAL)) $0))) $0)))
250 #(\ (\ (\ (#(/ REAL) (/ (#(\ (+ $0 REAL)) $0) $1))) (#(\ (/ (#(\ (/
251   ↪ (/ REAL $0) $0) $0) $0) REAL))
252 #(\ (/ (#(\ (#(\ (#(\ (+ $0 REAL)) (* $0 REAL))) (* (#(\ (+ $0
253   ↪ REAL)) $0) $0))) $0) (#(\ (+ $0 REAL)) $0))

```

254 10.4 Geometry

```

255 #(var_half var_name)
256 #(var_double var_name)
257 #(var_next #(var_half var_name))
258 #(concat (turn (just #(var_half var_name))))
259 #(\ (integrate nothing $0 nothing nothing))
260 #(\ (integrate $0 true nothing (just var_name)))
261 #(\ (\ (repeat nothing (concat $0 (turn $1))))
262   #(integrate (just #(var_half var_name)) true nothing nothing)
263   #(\ (\ (integrate $0 true nothing (just var_name))) nothing)
264   #(\ (concat $0 (#(\ (integrate nothing $0 nothing nothing)) true)))
265   #(\ (\ (\ (integrate (just #(var_double var_name)) true $0 (just $1)))
266   #(\ (concat (turn $0) (#(\ (integrate nothing $0 nothing nothing)
267     ↪ true)))
268   #(\ (concat $0 (#(\ (integrate $0 true nothing (just var_name))
269     ↪ nothing)))
270   #(\ (\ (\ (\ (repeat nothing (concat (integrate $0 true nothing $1) $2))))
271   #(\ (\ (concat (turn $0) (#(\ (integrate nothing $0 nothing nothing)
272     ↪ true))) nothing)
273   #(\ (\ (\ (integrate $0 true nothing (just var_name))) (just (var_half
274     ↪ #(var_half var_name))))
275   #(\ (\ (repeat nothing (#(\ (\ (repeat nothing (concat $0 (turn $1))))
276     ↪ nothing $0)))
277   #(\ (concat (#(\ (\ (integrate $0 true nothing (just var_name))) (just
278     ↪ (var_half #(var_half var_name))))
279   #(\ (\ (\ (\ (repeat nothing (#(\ (\ (repeat nothing (concat $0 (turn
280     ↪ $1)))) nothing $0))) (embed $0)))
281   #(\ (\ (repeat nothing (repeat nothing (concat (embed $0) (#(\ (integrate
282     ↪ nothing $0 nothing nothing)) false))))
283   #(\ (embed (#(\ (concat (turn $0) (#(\ (integrate nothing $0 nothing
284     ↪ nothing)) true))) (just #(var_half var_name))))
285   #(\ (\ (\ (\ (\ (repeat nothing (concat (integrate $0 true nothing $1)
286     ↪ $2)))) (turn $0) nothing nothing)
287   #(\ (\ (\ (concat $0 (#(\ (integrate nothing $0 nothing nothing)) true)))
288     ↪ (integrate $0 false nothing nothing))
289   #(\ (\ (\ (concat (#(\ (integrate nothing $0 nothing nothing)) $0)
290     ↪ (integrate $1 true $2 (just var_unit))))
291   #(\ (\ (\ (\ (repeat nothing (concat $0 (turn $1)))) $0 (embed (#(\
292     ↪ (integrate nothing $0 nothing nothing)) true)))
293   #(\ (\ (\ (\ (concat (#(\ (integrate nothing $0 nothing nothing)) $0)
294     ↪ (integrate $1 true $2 (just var_unit)))) nothing)
295   #(\ (concat (concat (#(\ (integrate nothing $0 nothing nothing)) true)
296     ↪ (turn nothing)) (integrate (just $0) true nothing nothing))

```

```

297 #(\ (repeat nothing (#(\ (\ (\ (repeat nothing (concat (integrate $0
298   ↪ true nothing $1) $2)))) (turn nothing) nothing $0)))
299 #(\ (\ (\ (repeat nothing (#(\ (\ (\ (repeat nothing (concat (integrate $0
300   ↪ true nothing $1) $2)))) (turn nothing) nothing $0))) nothing)
301 #(\ (\ (\ (\ (repeat nothing (concat (integrate $0 true nothing $1)
302   ↪ $2)))) (integrate nothing true nothing (just #(var_half
303   ↪ var_name))) nothing)
304 #(\ (\ (\ (\ (#(\ (\ (\ (repeat nothing (concat (integrate $0 true nothing
305   ↪ $1) $2)))) (integrate nothing $0 nothing $1) $2 nothing)))
306 #(\ (\ (\ (repeat nothing (#(\ (\ (\ (repeat nothing (concat $0 (turn $1))))
307   ↪ nothing (concat (#(\ (integrate nothing $0 nothing nothing)) $0)
308   ↪ $1))))
309 #(\ (\ (\ (\ (repeat nothing (concat (integrate $0 true nothing $1)
310   ↪ $2)))) (#(\ (integrate nothing $0 nothing nothing)) false) (just
311   ↪ var_name) nothing)
312 #(\ (\ (\ (repeat nothing (#(\ (\ (\ (repeat nothing (concat (integrate $0
313   ↪ true nothing $1) $2)))) (turn nothing) nothing $0))) (just
314   ↪ #(var_half var_name)))
315 #(\ (\ (\ (concat $0 (#(\ (integrate nothing $0 nothing nothing)) true)))
316   ↪ #(\ (\ (\ (integrate $0 true nothing (just var_name))) (just (var_half
317   ↪ #(var_half var_name))))
318 #(\ (#(\ (\ (repeat nothing (#(\ (\ (\ (repeat nothing (concat (integrate
319   ↪ $0 true nothing $1) $2)))) (turn nothing) nothing $0))) (just
320   ↪ (var_double $0)))
321 #(\ (\ (\ (\ (repeat nothing (#(\ (\ (\ (repeat nothing (concat (integrate $0
322   ↪ true nothing $1) $2)))) (define #(var_half var_name)) (just
323   ↪ #(var_half var_name)) $0)))
324 #(\ (\ (\ (repeat nothing (repeat nothing (concat (embed $0) (#(\ (integrate
325   ↪ nothing $0 nothing nothing)) false)))) (#(\ (integrate $0 true
326   ↪ nothing (just var_name))) nothing))
327 #(\ (#(\ (\ (repeat nothing (#(\ (\ (\ (repeat nothing (concat $0 (turn
328   ↪ $1)))) nothing $0))) (concat (define $0) #(integrate (just
329   ↪ #(var_half var_name)) true nothing nothing)))
330 #(\ (\ (repeat nothing (#(\ (\ (\ (concat (#(\ (integrate nothing $0
331   ↪ nothing nothing)) $0) (integrate $1 true $2 (just var_unit))))))
332   ↪ nothing (just (var_half $0)) false)))
333 #(\ (#(\ (\ (\ (repeat nothing (concat $0 (turn $1)))) $0 #(embed (#(\
334   ↪ (concat (turn $0) (#(\ (integrate nothing $0 nothing nothing))
335   ↪ true))) (just #(var_half var_name))))))
336 #(\ (#(\ (#(\ (\ (repeat nothing (#(\ (\ (\ (repeat nothing (concat $0 (turn
337   ↪ $1)))) nothing $0))) (embed $0))) (#(\ (integrate $0 true nothing
338   ↪ (just var_name)) (just $0)))
339 #(\ (#(\ (\ (\ (repeat nothing (concat $0 (turn $1)))) nothing (#(\
340   ↪ (repeat nothing (repeat nothing (concat (embed $0) (#(\ (integrate
341   ↪ nothing $0 nothing nothing)) false)))) $0)))
342 #(\ (embed (#(\ (#(\ (\ (\ (repeat nothing (concat $0 (turn $1)))) $0 #(embed
343   ↪ (#(\ (concat (turn $0) (#(\ (integrate nothing $0 nothing
344   ↪ nothing)) true))) (just #(var_half var_name)))))) nothing))
345 #(\ (\ (concat (#(\ (integrate nothing $0 nothing nothing)) true) (#(\ (#(\
346   ↪ (\ (repeat nothing (concat $0 (turn $1)))) $0 (embed (#(\
347   ↪ (integrate nothing $0 nothing nothing)) true)))) $0)))
348 #(\ (#(\ (\ (\ (\ (#(\ (\ (\ (repeat nothing (concat (integrate $0 true
349   ↪ nothing $1) $2)))) (integrate nothing $0 nothing $1) $2
350   ↪ nothing))) (just #(var_half var_name)) (just $0) true))
351 #(\ (#(\ (\ (repeat nothing (repeat nothing (concat (embed $0) (#(\
352   ↪ (integrate nothing $0 nothing nothing)) false)))) (#(\ (concat
353   ↪ (turn $0) (#(\ (integrate nothing $0 nothing nothing)) true)))
354   ↪ $0)))
355 #(\ (#(\ (\ (\ (repeat nothing (concat $0 (turn $1)))) $0 (#(\ (repeat
356   ↪ nothing (#(\ (\ (\ (\ (repeat nothing (concat (integrate $0 true
357   ↪ nothing $1) $2)))) (turn nothing) nothing $0))) nothing)))
358 #(\ (#(\ (\ (\ (repeat nothing (concat $0 (turn $1)))) $0 (#(\ (#(\ (\ (\
359   ↪ (repeat nothing (concat (integrate $0 true nothing $1) $2))))
360   ↪ (turn $0) nothing nothing)) (just #(var_half var_name))))

```

```

361  (#(λ (λ (repeat nothing (#(λ (λ (repeat nothing (concat $0 (turn
362    ↪ $1)))))) nothing (concat (#(λ (integrate nothing $0 nothing
363    ↪ nothing)) $0) $1)))) (#(λ (integrate nothing $0 nothing nothing))
364    ↪ false) true)
365  #(repeat nothing (#(λ (repeat nothing (#(λ (λ (concat (#(λ (integrate
366    ↪ nothing $0 nothing nothing)) $0) (integrate $1 true $2 (just
367    ↪ var_unit)))))) nothing (just (var_half $0)) false))) #(var_half
368    ↪ var_name)))
369  (#(λ (λ (repeat nothing (#(λ (λ (repeat nothing (concat $0 (turn
370    ↪ $1)))))) nothing (concat (#(λ (integrate nothing $0 nothing
371    ↪ nothing)) $0) $1)))) (#(λ (integrate $0 true nothing (just
372    ↪ var_name))) nothing) false)
373  (#(λ (#(λ (λ (λ (repeat nothing (concat (integrate $0 true nothing $1)
374    ↪ $2)))))) (#(λ (#(λ (λ (repeat nothing (concat $0 (turn $1)))))) $0
375    ↪ (embed (#(λ (integrate nothing $0 nothing nothing)) true))))
376    ↪ nothing) (just $0) nothing))
377  (#(λ (#(λ (λ (λ (repeat nothing (concat $0 (turn $1)))))) nothing (#(λ
378    ↪ (repeat nothing (repeat nothing (concat (embed $0) (#(λ (integrate
379    ↪ nothing $0 nothing nothing)) false)))) $0))) (#(λ (concat (turn
380    ↪ $0) (#(λ (integrate nothing $0 nothing nothing)) true))) $0)))
381  (#(λ (#(λ (repeat nothing (#(λ (λ (repeat nothing (concat $0 (turn
382    ↪ $1)))))) nothing $0))) (#(λ (λ (λ (repeat nothing (concat
383    ↪ (integrate $0 true nothing $1) $2)))) (#(λ (concat (turn $0)
384    ↪ (#(λ (integrate nothing $0 nothing nothing)) true))) nothing)
385    ↪ nothing $0)))
386  #(repeat nothing (concat (#(λ (repeat nothing (repeat nothing (concat
387    ↪ (embed $0) (#(λ (integrate nothing $0 nothing nothing)) false))))
388    ↪ (#(λ (integrate $0 true nothing (just var_name))) nothing)) (#(λ
389    ↪ (integrate $0 true nothing (just var_name))) (just (var_half
390    ↪ #(var_half var_name))))))
391  (#(λ (repeat (just $0) #(repeat nothing (concat (#(λ (repeat nothing
392    ↪ (repeat nothing (concat (embed $0) (#(λ (integrate nothing $0
393    ↪ nothing nothing)) false)))))) (#(λ (integrate $0 true nothing (just
394    ↪ var_name))) nothing)) (#(λ (integrate $0 true nothing (just
395    ↪ var_name))) (just (var_half #(var_half var_name))))))
396  (#(λ (#(λ (#(λ (repeat nothing (#(λ (λ (repeat nothing (concat $0 (turn
397    ↪ $1)))))) nothing $0))) (embed $0))) (#(λ (#(λ (λ (repeat nothing
398    ↪ (concat $0 (turn $1)))))) $0 (embed (#(λ (concat (turn $0) (#(λ
399    ↪ (integrate nothing $0 nothing nothing)) true))) (just #(var_half
400    ↪ var_name)))))) (just $0)))

```

401 References

- 402 Abelson, Hal, Goodman, Nat, and Rudolph, Lee. Logo manual. 1974.
- 403 Damas, Luis and Milner, Robin. Principal type-schemes for functional programs. In *Proceedings*
404 *of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp.
405 207–212. ACM, 1982.
- 406 Pierce, Benjamin C. *Types and programming languages*. MIT Press, 2002. ISBN 978-0-262-16209-8.
- 407 Sablé-Meyer, Mathias and Dehaene, Stanislas. Visual sequence primitives in humans. Master’s thesis,
408 ENS, 2017. DSL proposal in appendix: A proposal for a Language of Shape.