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

Anonymous Author(s)
Affiliation
Address
email

1 Learning Generative Graphics Programs

A natural extension of our work is to consider the problem of learning generative models: here, we would learn programs that generate (either deterministically or probabilistically) structures like images or words. As a first step in this direction, we apply SCC to synthesizing graphics programs.

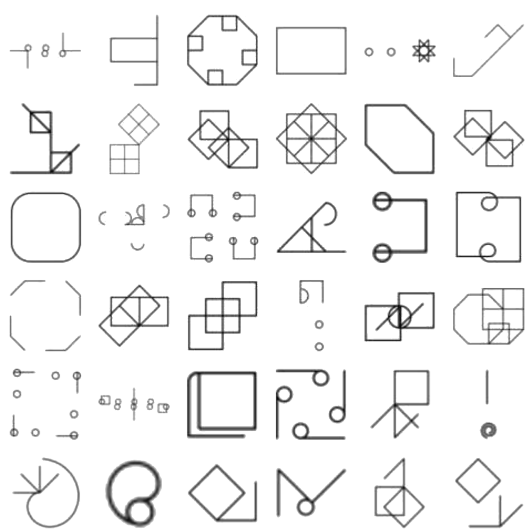


Figure 1: Another example of compiled figures generated by our method.

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

A natural starting DSL for graphics programming is the Logo language (Abelson et al. (1974)), also sometimes called **turtle graphics**. These programs control a pen (sometimes called a "turtle"), and can do things like pick the pen up, move the pen forward, rotate the pen, or trace out a programmatically specified arc. We take turtle graphics primitives from prior work Sablé-Meyer & Dehaene (2017). In our setting, we will encapsulate turtle graphics primitives inside of λ-calculus, and seek to infer graphics programs from images: thus the task is to look at an image, and write the program that would have drawn it.

The DSL is the following:

Submitted to 32nd Conference on Neural Information Processing Systems (NIPS 2018). Do not distribute.

name	*type*
`Concat`	`prog` $\rightarrow$ `prog` $\rightarrow$ `prog`
`Repeat`	`var option` $\rightarrow$ `prog` $\rightarrow$ `prog`
`Embed`	`prog` $\rightarrow$ `prog`
`Define`	`var` $\rightarrow$ `prog`
`Turn`	`var option` $\rightarrow$ `prog`
`Integrate`	`var option` $\rightarrow$ `bool` $\rightarrow$ `var option` $\rightarrow$ `var option` $\rightarrow$ `prog`
`True`	`bool`
`False`	`bool`
`Nothing`	`var option`
`Just`	`var` $\rightarrow$ `var option`
`Unit`	`var`
`Name`	`var`
`Next`	`var` $\rightarrow$ `var`
`Prev`	`var` $\rightarrow$ `var`
`Double`	`var` $\rightarrow$ `var`
`Half`	`var` $\rightarrow$ `var`
`Opposite`	`var` $\rightarrow$ `var`

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

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

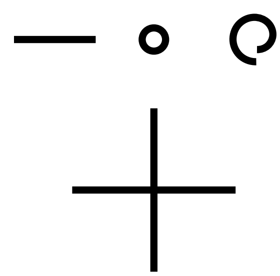


Figure 2: Top: Some defaults for `Integrate`. Bottom, the result of the `Cross` example bellow.

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

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

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

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

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

For example, the simplest way to draw a square in this language is already of length 12. Placing several around, for example to draw a grid is out of reach of the initial search. However by first abstracting as a primitive the segment — thus reducing the length by four —, then assuming that after a segment it often needs to draw something else, then abstracting the first arguments of `Repeat` — further reducing the length by two — as well as the one of `Turn` and finally making the square a 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`	$\lambda p.$ `Repeat(nothing,` p`)`
AddToUnitSegment = f_3	`prog` $\rightarrow$ `prog`	$\lambda p.$ `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.

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

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

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

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

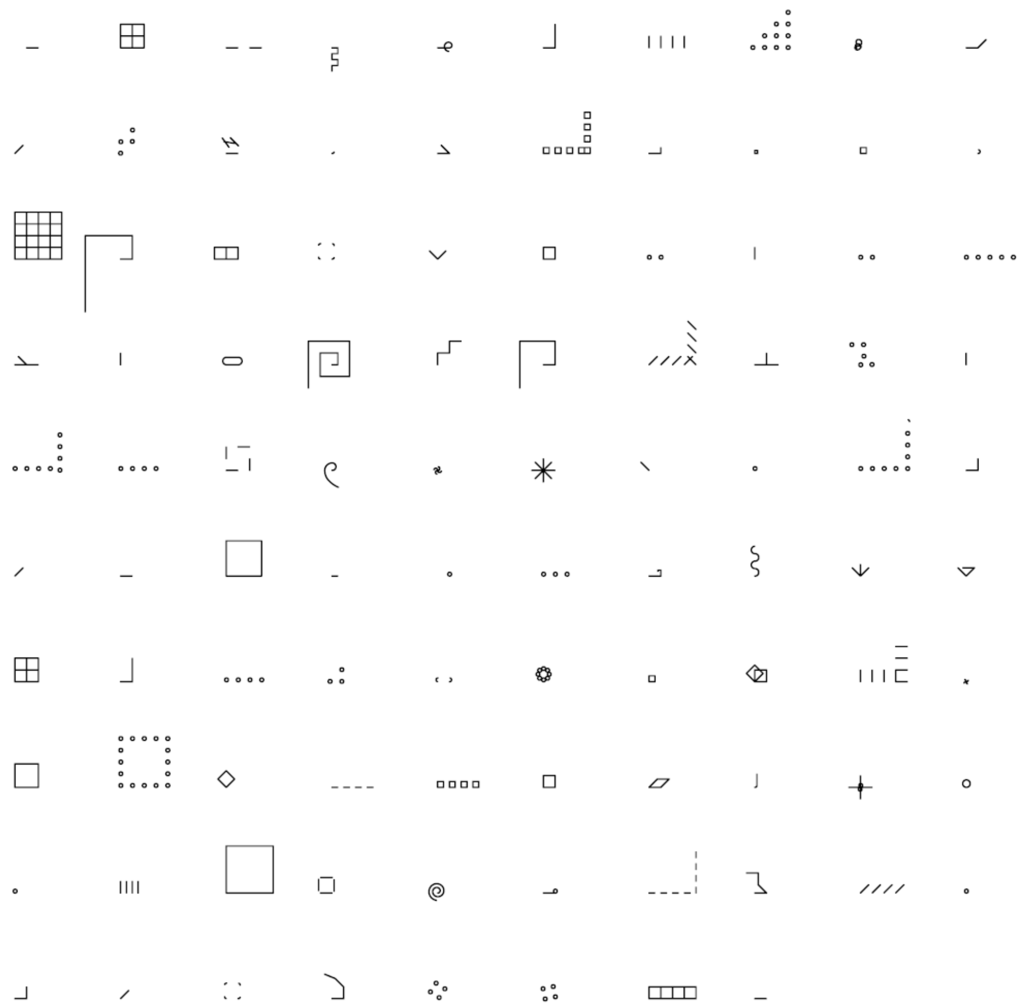


Figure 3: The set of tasks for the geometry domain

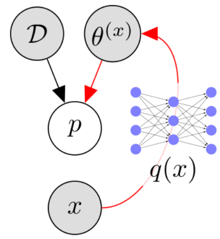
2 An Illustration of the 3 iterations of our algorithm

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.

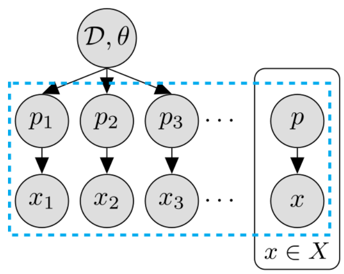
3 Program Representation

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

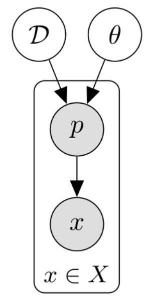
- A *primitive*, like the number 5 or the function `sum`.
- A *variable*, like x, y, or z.
- A λ-*abstraction*, which creates a new function. λ-abstractions have a variable and a body. The body is a λ-calculus expression. Abstractions are written as λvar.body or in Lisp syntax as `(lambda (var) body)`.



Search: Infer p



Compile: Train q
training data: cyan (x, p)



Compress: Induce $(\mathcal{D}, \theta)$

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

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

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

4 Generative model over the programs

Alg. 1 is a procedure for drawing samples from the generative model $(\mathcal{D}, \theta)$. In practice, we enumerate 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 τ
Output: a program whose type unifies with τ
if $\tau = \alpha \to \beta$ **then**
  var $\leftarrow$ an unused variable name
  body $\sim$ sample($\mathcal{D}, \theta, \{$var : $\alpha\} \cup \mathcal{E}, \beta$)
  return `(lambda (var) body)`
end if
primitives $\leftarrow \{p | p : \tau' \in \mathcal{D} \cup \mathcal{E}$
  if τ can unify with yield(τ')$\}$
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 τ with yield(τ').
$\{\alpha_k\}_{k=1}^{K} \leftarrow$ args(τ')
for $k = 1$ **to** K **do**
  $a_k \sim$ sample($\mathcal{D}, \theta, \mathcal{E}, \alpha_k$)
end for
return $(e\ a_1\ a_2\ \cdots\ a_K)$
where:

$$\text{yield}(\tau) = \begin{cases} \text{yield}(\beta) & \text{if } \tau = \alpha \to \beta \\ \tau & \text{otherwise.} \end{cases}$$

$$\text{args}(\tau) = \begin{cases} [\alpha] + \text{args}(\beta) & \text{if } \tau = \alpha \to \beta \\ [] & \text{otherwise.} \end{cases}$$

5 Neural Recognition Model Architecture

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

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

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

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

6 DSL Induction

6.1 Structure Learning

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 $\leftarrow \{\mathcal{D}\cup\{s\}|x\in X, p\in\mathcal{F}_x, s$ 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

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

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

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:

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

where we have defined

$$C(e) \triangleq \sum_x \mathbb{E}\left[c(e,p_x)\right]$$

$$R(\tau) \triangleq \mathbb{E}\left[\sum_x c(\tau,p_x)\right]$$

$$\beta \triangleq \sum_\tau \mathbb{E}\left[\sum_x c(\tau,p_x)\right]$$

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:

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

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

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

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

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 $\sum_{\tau'} \mathbb{1}\left[\text{unify}(\tau, \tau')\right] R(\tau')$ 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 $\mathscr{L}$ 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):

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

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

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

10.1 List processing

```
#(+ 1 1)
#(λ (cdr (cdr $0)))
#(λ (foldr $0 1 (λ (λ (* $0 $1)))))
#(λ (cons (car $0) nil))
#(λ (λ (foldr $0 $1 (λ (λ (cons $1 $0))))))
#(λ (λ (foldr $0 (is-nil $0) (λ (λ (if $0 $0 (eq? $3 $1)))))))
#(λ (map (λ (eq? $1 $0))))
#(λ (* $0 (* $0 $0)))
#(+ 1 #(+ 1 1))
#(λ (map (λ (gt? $0 $1))))
#(λ (foldr $0 nil (λ (λ (if (is-square $1) (cons $1 $0) $0)))))
#(λ (map (λ (eq? $0 (length (range $0)))) $0))
#(λ (λ (map (λ (index $0 $1)) (range $1))))
#(λ (cdr (#(λ (cdr (cdr $0))) $0)))
#(λ (map (λ (index 1 $1))))
#(λ (foldr $0 nil (λ (λ (cons $1 (cons $1 $0))))))
#(λ (λ (cons (car $0) $1)))
#(+ 1 #(+ 1 #(+ 1 1)))
#(λ (map (λ (gt? 1 (mod $0 $1)))))
#(λ (λ (map (λ (mod (+ $0 $1) $2)))))
#(λ (#(λ (λ (foldr $0 $1 (λ (λ (cons $1 $0)))))) (#(λ (λ (foldr $0 $1 (λ
    ↪ (λ (cons $1 $0)))))) $0 $0) $0))
#(λ (λ (#(λ (λ (foldr $0 $1 (λ (λ (cons $1 $0)))))) (cons $0 nil) $1)))
#(+ #(+ 1 #(+ 1 1)) #(+ 1 1))
#(λ (map (λ (+ #(+ 1 #(+ 1 1)) (+ $1 $0)))))
#(λ (map (λ (+ $0 $1))))
#(λ (λ (foldr $0 (is-nil $0) (λ (λ (gt? $1 (#(λ (* $0 (* $0 $0)))
    ↪ $3)))))))
#(λ (foldr $0 0 (λ (λ (+ $0 (#(λ (foldr $0 1 (λ (λ (* $0 $1))))) (range
    ↪ $1)))))))
#(λ (#(λ (cdr (#(λ (cdr (cdr $0))) $0))) (cdr $0)))
#(λ (#(λ (foldr $0 nil (λ (λ (if (is-square $1) (cons $1 $0) $0)))))
    ↪ (map (λ (* $0 (+ $0 $0))) $0)))
#(λ (λ (#(λ (λ (foldr $0 $1 (λ (λ (cons $1 $0)))))) (#(λ (cons (car $0)
    ↪ nil)) $0) $1)))
#(λ (λ (length (#(λ (#(λ (foldr $0 nil (λ (λ (if (is-square $1) (cons $1
    ↪ $0) $0))))) (map (λ (* $0 (+ $0 $0))) $0))) (map (λ (- $1 $0))
    ↪ $1)))))
#(λ (λ (is-square (#(λ (foldr $0 1 (λ (λ (* $0 $1))))) (#(λ (λ (map (λ
    ↪ (mod (+ $0 $1) $2))))) (length (#(λ (#(λ (λ (foldr $0 $1 (λ (λ
    ↪ (cons $1 $0)))))) (#(λ (λ (foldr $0 $1 (λ (λ (cons $1 $0)))))) $0
    ↪ $0) $0)) $0)) $1 $0)))))
#(λ (λ (foldr (cdr $0) $1 (λ (λ (#(λ (λ (#(λ (λ (foldr $0 $1 (λ (λ (cons
    ↪ $1 $0)))))) (#(λ (cons (car $0) nil)) $0) $1))) (cdr $0) $0))))))
#(λ (is-nil (#(λ (#(λ (foldr $0 nil (λ (λ (if (is-square $1) (cons $1
    ↪ $0) $0))))) (map (λ (* $0 (+ $0 $0))) $0))) (#(λ (λ (map (λ (mod
    ↪ (+ $0 $1) $2))))) #(+ 1 1) 1 $0))))
#(λ (λ (gt? (#(λ (λ (length (#(λ (#(λ (foldr $0 nil (λ (λ (if (is-square
    ↪ $1) (cons $1 $0) $0))))) (map (λ (* $0 (+ $0 $0))) $0))) (map (λ
    ↪ (- $1 $0)) $1))))) $0 $1) 1)))
```

10.2 Text editing

```
#(+ 1)
#(λ (λ (fold $0 $0 (λ (λ (if (char-eq? $1 $3) nil (cons $1 $0)))))))
#(λ (λ (fold $0 $0 (λ (λ (cdr (if (char-eq? $1 $3) $2 $0)))))))
#(λ (λ (fold $0 $1 (λ (λ (cons $1 $0))))))
#(λ (λ (#(λ (λ (fold $0 $1 (λ (λ (cons $1 $0)))))) (cons $0 $1))))
#(λ (#(λ (λ (λ (cons (car $0) (cons $1 $2))))) (#(#(λ (λ (λ (cons (car
    ↪ $0) (cons $1 $2))))) nil) '.' $0) '.'))
#(λ (λ (fold $0 $0 (λ (λ (fold $0 $0 (λ (λ (if (char-eq? $1 $5) (cdr $2)
    ↪ $0)))))))))
#(λ (λ (map (λ (if (char-eq? $1 $0) $2 $0)))))
```

```
#(λ (#(λ (λ (fold $0 $1 (λ (λ (cons $1 $0)))))) $0 STRING))
#(λ (map (λ (index $0 $1))))
#(λ (unfold $0 (λ (nil? $0)) (λ (car $0)) (λ (#(λ (λ (fold $0 $0 (λ (λ
    ↪ (cdr (if (char−eq? $1 $3) $2 $0))))))) SPACE $0))))
#(#(λ (λ (λ (cons (car $0) (cons $1 $2))))) nil)
```

10.3 Symbolic regression

```
#(λ (/. (/. REAL $0) $0))
#(λ (+. $0 REAL))
#(λ (#(λ (+. $0 REAL)) (∗. $0 (#(λ (#(λ (+. $0 REAL)) (∗. (#(λ (#(λ (#(λ
    ↪ (+.  $0 REAL)) (∗. $0 REAL))) (∗. (#(λ (+. $0 REAL)) $0) $0))) $0)
    ↪ $0))) $0))))
#(λ (/. (#(λ (/. (/. REAL $0) $0)) $0) $0))
#(λ (λ (#(/. REAL) (/. (#(λ (+. $0 REAL)) $0) $1))))
#(λ (#(λ (+. $0 REAL)) (#(λ (#(/. REAL) (#(λ (+. $0 REAL)) $0))) $0)))
#(#(λ (λ (#(/. REAL) (/. (#(λ (+. $0 REAL)) $0) $1)))) (#(λ (/. (#(λ (/.
    ↪ (/.  REAL $0) $0)) $0) $0)) REAL))
#(λ (/. (#(λ (#(λ (#(λ (+. $0 REAL)) (∗. $0 REAL))) (∗. (#(λ (+. $0
    ↪ REAL)) $0) $0))) $0) (#(λ (+. $0 REAL)) $0)))
```

10.4 Geometry

```
#(var_half var_name)
#(var_double var_name)
#(var_next #(var_half var_name))
#(concat (turn (just #(var_half var_name))))
#(λ (integrate nothing $0 nothing nothing))
#(λ (integrate $0 true nothing (just var_name)))
#(λ (λ (repeat nothing (concat $0 (turn $1)))))
#(integrate (just #(var_half var_name)) true nothing nothing)
#(#(λ (integrate $0 true nothing (just var_name))) nothing)
#(λ (concat $0 (#(λ (integrate nothing $0 nothing nothing)) true)))
#(λ (λ (integrate (just #(var_double var_name)) true $0 (just $1))))
#(λ (concat (turn $0) (#(λ (integrate nothing $0 nothing nothing))
    ↪ true)))
#(λ (concat $0 (#(λ (integrate $0 true nothing (just var_name)))
    ↪ nothing)))
#(λ (λ (λ (repeat nothing (concat (integrate $0 true nothing $1) $2)))))
#(#(λ (concat (turn $0) (#(λ (integrate nothing $0 nothing nothing))
    ↪ true))) nothing)
#(#(λ (integrate $0 true nothing (just var_name))) (just (var_half
    ↪ #(var_half var_name))))
#(λ (repeat nothing (#(λ (λ (repeat nothing (concat $0 (turn $1)))))
    ↪ nothing $0)))
#(concat #(#(λ (integrate $0 true nothing (just var_name))) (just
    ↪ (var_half #(var_half var_name)))))
#(λ (#(λ (repeat nothing (#(λ (λ (repeat nothing (concat $0 (turn
    ↪ $1))))) nothing $0))) (embed $0)))
#(λ (repeat nothing (repeat nothing (concat (embed $0) (#(λ (integrate
    ↪ nothing $0 nothing nothing)) false)))))
#(embed (#(λ (concat (turn $0) (#(λ (integrate nothing $0 nothing
    ↪ nothing)) true))) (just #(var_half var_name))))
#(λ (#(λ (λ (λ (repeat nothing (concat (integrate $0 true nothing $1)
    ↪ $2))))) (turn $0) nothing nothing))
#(λ (#(λ (concat $0 (#(λ (integrate nothing $0 nothing nothing)) true)))
    ↪ (integrate $0 false nothing nothing)))
#(λ (λ (λ (concat (#(λ (integrate nothing $0 nothing nothing)) $0)
    ↪ (integrate $1 true $2 (just var_unit))))))
#(λ (#(λ (λ (repeat nothing (concat $0 (turn $1))))) $0 (embed (#(λ
    ↪ (integrate nothing $0 nothing nothing)) true))))
#(#(λ (λ (λ (concat (#(λ (integrate nothing $0 nothing nothing)) $0)
    ↪ (integrate $1 true $2 (just var_unit)))))) nothing)
#(λ (concat (concat (#(λ (integrate nothing $0 nothing nothing)) true)
    ↪ (turn nothing)) (integrate (just $0) true nothing nothing)))
```

```
#(λ (repeat nothing (#(λ (λ (λ (repeat nothing (concat (integrate $0
    ↪ true nothing $1) $2))))) (turn nothing) nothing $0)))
#(#(λ (repeat nothing (#(λ (λ (λ (repeat nothing (concat (integrate $0
    ↪ true nothing $1) $2))))) (turn nothing) nothing $0))) nothing)
#(#(λ (λ (λ (repeat nothing (concat (integrate $0 true nothing $1)
    ↪ $2))))) (integrate nothing true nothing (just #(var_half
    ↪ var_name))) nothing)
#(λ (λ (λ (#(λ (λ (λ (repeat nothing (concat (integrate $0 true nothing
    ↪ $1) $2))))) (integrate nothing $0 nothing $1) $2 nothing))))
#(λ (λ (repeat nothing (#(λ (λ (repeat nothing (concat $0 (turn $1)))))
    ↪ nothing (concat (#(λ (integrate nothing $0 nothing nothing)) $0)
    ↪ $1)))))
#(#(λ (λ (λ (repeat nothing (concat (integrate $0 true nothing $1)
    ↪ $2))))) (#(λ (integrate nothing $0 nothing nothing)) false) (just
    ↪ var_name) nothing)
#(#(λ (repeat nothing (#(λ (λ (λ (repeat nothing (concat (integrate $0
    ↪ true nothing $1) $2))))) (turn nothing) nothing $0))) (just
    ↪ #(var_half var_name)))
#(#(λ (concat $0 (#(λ (integrate nothing $0 nothing nothing)) true)))
    ↪ #(#(λ (integrate $0 true nothing (just var_name))) (just (var_half
    ↪ #(var_half var_name)))))
#(λ (#(λ (repeat nothing (#(λ (λ (λ (repeat nothing (concat (integrate
    ↪ $0 true nothing $1) $2))))) (turn nothing) nothing $0))) (just
    ↪ (var_double $0))))
#(λ (repeat nothing (#(λ (λ (λ (repeat nothing (concat (integrate $0
    ↪ true nothing $1) $2))))) (define #(var_half var_name)) (just
    ↪ #(var_half var_name)) $0)))
#(#(λ (repeat nothing (repeat nothing (concat (embed $0) (#(λ (integrate
    ↪ nothing $0 nothing nothing)) false))))) (#(λ (integrate $0 true
    ↪ nothing (just var_name))) nothing))
#(λ (#(λ (repeat nothing (#(λ (λ (repeat nothing (concat $0 (turn
    ↪ $1))))) nothing $0))) (concat (define $0) #(integrate (just
    ↪ #(var_half var_name)) true nothing nothing))))
#(λ (repeat nothing (#(λ (λ (λ (concat (#(λ (integrate nothing $0
    ↪ nothing nothing)) $0) (integrate $1 true $2 (just var_unit))))))
    ↪ nothing (just (var_half $0)) false)))
#(λ (#(λ (λ (repeat nothing (concat $0 (turn $1))))) $0 #(embed (#(λ
    ↪ (concat (turn $0) (#(λ (integrate nothing $0 nothing nothing))
    ↪ true))) (just #(var_half var_name))))))
#(λ (#(λ (#(λ (repeat nothing (#(λ (λ (repeat nothing (concat $0 (turn
    ↪ $1))))) nothing $0))) (embed $0))) (#(λ (integrate $0 true nothing
    ↪ (just var_name))) (just $0))))
#(λ (#(λ (λ (repeat nothing (concat $0 (turn $1))))) nothing (#(λ
    ↪ (repeat nothing (repeat nothing (concat (embed $0) (#(λ (integrate
    ↪ nothing $0 nothing nothing)) false))))) $0)))
#(embed (#(λ (#(λ (λ (repeat nothing (concat $0 (turn $1))))) $0 #(embed
    ↪ (#(λ (concat (turn $0) (#(λ (integrate nothing $0 nothing
    ↪ nothing)) true))) (just #(var_half var_name)))))) nothing))
#(λ (concat (#(λ (integrate nothing $0 nothing nothing)) true) (#(λ (#(λ
    ↪ (λ (repeat nothing (concat $0 (turn $1))))) $0 (embed (#(λ
    ↪ (integrate nothing $0 nothing nothing)) true)))) $0)))
#(λ (#(λ (λ (λ (#(λ (λ (λ (repeat nothing (concat (integrate $0 true
    ↪ nothing $1) $2))))) (integrate nothing $0 nothing $1) $2
    ↪ nothing)))) (just #(var_half var_name)) (just $0) true))
#(λ (#(λ (repeat nothing (repeat nothing (concat (embed $0) (#(λ
    ↪ (integrate nothing $0 nothing nothing)) false))))) (#(λ (concat
    ↪ (turn $0) (#(λ (integrate nothing $0 nothing nothing)) true)))
    ↪ $0)))
#(λ (#(λ (λ (repeat nothing (concat $0 (turn $1))))) $0 (#(λ (repeat
    ↪ nothing (#(λ (λ (λ (repeat nothing (concat (integrate $0 true
    ↪ nothing $1) $2))))) (turn nothing) nothing $0))) nothing)))
#(λ (#(λ (λ (repeat nothing (concat $0 (turn $1))))) $0 (#(λ (#(λ (λ (λ
    ↪ (repeat nothing (concat (integrate $0 true nothing $1) $2)))))
    ↪ (turn $0) nothing nothing)) (just #(var_half var_name)))))
```

```
#(#(λ (λ (repeat nothing (#(λ (λ (repeat nothing (concat $0 (turn
    ↪ $1))))) nothing (concat (#(λ (integrate nothing $0 nothing
    ↪ nothing)) $0) $1))))) (#(λ (integrate nothing $0 nothing nothing))
    ↪ false) true)
#(repeat nothing (#(λ (repeat nothing (#(λ (λ (λ (concat (#(λ (integrate
    ↪ nothing $0 nothing nothing)) $0) (integrate $1 true $2 (just
    ↪ var_unit)))))) nothing (just (var_half $0)) false))) #(var_half
    ↪ var_name)))
#(#(λ (λ (repeat nothing (#(λ (λ (repeat nothing (concat $0 (turn
    ↪ $1))))) nothing (concat (#(λ (integrate nothing $0 nothing
    ↪ nothing)) $0) $1))))) (#(λ (integrate $0 true nothing (just
    ↪ var_name))) nothing) false)
#(λ (#(λ (λ (λ (repeat nothing (concat (integrate $0 true nothing $1)
    ↪ $2))))) (#(λ (#(λ (λ (repeat nothing (concat $0 (turn $1))))) $0
    ↪ (embed (#(λ (integrate nothing $0 nothing nothing)) true))))
    ↪ nothing) (just $0) nothing))
#(λ (#(λ (#(λ (λ (repeat nothing (concat $0 (turn $1))))) nothing (#(λ
    ↪ (repeat nothing (repeat nothing (concat (embed $0) (#(λ (integrate
    ↪ nothing $0 nothing nothing)) false))))) $0))) (#(λ (concat (turn
    ↪ $0) (#(λ (integrate nothing $0 nothing nothing)) true))) $0)))
#(λ (#(λ (repeat nothing (#(λ (λ (repeat nothing (concat $0 (turn
    ↪ $1))))) nothing $0))) (#(λ (λ (λ (repeat nothing (concat
    ↪ (integrate $0 true nothing $1) $2))))) #(#(λ (concat (turn $0)
    ↪ (#(λ (integrate nothing $0 nothing nothing)) true))) nothing)
    ↪ nothing $0)))
#(repeat nothing (concat #(#(λ (repeat nothing (repeat nothing (concat
    ↪ (embed $0) (#(λ (integrate nothing $0 nothing nothing)) false)))))
    ↪ (#(λ (integrate $0 true nothing (just var_name))) nothing)) #(#(λ
    ↪ (integrate $0 true nothing (just var_name))) (just (var_half
    ↪ #(var_half var_name))))))
#(λ (repeat (just $0) #(repeat nothing (concat #(#(λ (repeat nothing
    ↪ (repeat nothing (concat (embed $0) (#(λ (integrate nothing $0
    ↪ nothing nothing)) false))))) (#(λ (integrate $0 true nothing (just
    ↪ var_name))) nothing)) #(#(λ (integrate $0 true nothing (just
    ↪ var_name))) (just (var_half #(var_half var_name))))))))
#(λ (#(λ (#(λ (repeat nothing (#(λ (λ (repeat nothing (concat $0 (turn
    ↪ $1))))) nothing $0))) (embed $0))) (#(λ (#(λ (λ (repeat nothing
    ↪ (concat $0 (turn $1))))) $0 #(embed (#(λ (concat (turn $0) (#(λ
    ↪ (integrate nothing $0 nothing nothing)) true))) (just #(var_half
    ↪ var_name)))))) (just $0))))
```

References

Abelson, Hal, Goodman, Nat, and Rudolph, Lee. Logo manual. 1974.

Damas, Luis and Milner, Robin. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 207–212. ACM, 1982.

Pierce, Benjamin C. *Types and programming languages*. MIT Press, 2002. ISBN 978-0-262-16209-8.

Sablé-Meyer, Mathias and Dehaene, Stanislas. Visual sequence primitives in humans. Master's thesis, ENS, 2017. DSL proposal in appendix: A proposal for a Language of Shape.