Supplement for: Learning to Infer Program Sketches

1. Architecture Details

Generator:

Our Generator neural network architecture is nearly identical to the Attn-A RobustFill model (Devlin et al., 2017). This is a sequence-to-sequence model which attends over multiple input-output pairs. Our model differs from the Attn-A RobustFill model by adding a learned grammar mask. As in Bunel et al. (2018), we learn a separate LSTM language model over the program syntax. The output probabilities of this LSTM are used to mask the output probabilities of the Generator model, encouraging the model to put less probability mass on grammatically invalid sequences.

For the Algolisp experiments, we did not condition the Generator on input-output examples, instead encoding the natural language descriptions for each program. In these experiments, the Generator is simply a sequence-to-sequence LSTM model with attention, coupled with a learned grammar mask, as above.

For the Generator model, **HOLE** is simple an additional token added to the program DSL. During training, sketches are sampled via Equation 6 in the main text, and are converted to a list of tokens to be processed by the Generator, as is typical with RNN models.

Recognizer:

The recognizer model consists of an LSTM encoder followed by a feed-forward MLP. To encode a specification, each input-output example is separately tokenized (a special `EndOfInput` token is used to separate input from output), and fed into the LSTM encoder. The resulting vectors for each input-output example are averaged. The result is fed into the feed-forward MLP, which terminates in a softmax layer, predicting a distribution over output production probabilities.

This architecture differs slightly from the DeepCoder model (Balog et al., 2016), which encodes inputs and outputs via a feedforward deep network without recurrence. However, the models are similar in functionality; both models encode input-output specs, average the hidden vectors for each example, and decode a distribution over production probabilities. Both models use the resulting distribution to guide a symbolic enumerative search over programs. Our enumeration scheme is equivalent to the depth first search experiments in Balog et al. (2016).

For the Algolisp experiments, we did not condition the Recognizer on input-output examples, instead encoding the natural language descriptions for each program. In these experiments, the LSTM encoder simply encodes the single natural language specification and feeds it to the MLP. As in the other domains, the examples are still used by the synthesizer to determine if enumerated candidate programs satisfy the input-output specification.

2. Experimental details

All code was written in Python, and neural network models were implemented in PyTorch and trained on NVIDIA Tesla-x GPUs. All networks were trained with the Adam optimizer (Kingma & Ba, 2014), with a learning rate of 0.001. Our sketch generator LSTMs used embedding sizes of 128 and hidden sizes of 512. Our recognizer networks used LSTMs with embedding sizes of 128, hidden sizes of 128, and MLPs had a single hidden layer of size 128. For all domains, we trained using a timeout parameter t sampled from $t \sim \text{Exp}(\alpha)$, where $\alpha = 0.25$.

2.1. List Processing

Data: We use the test and training programs from Balog et al. (2016). The test programs are simply all of the length N programs, pruned for redundant or invalid behavior, for which there does not exist a smaller program with identical behavior. We converted these programs into a λ-calculus form to use with our synthesizer.

As in Balog et al. (2016), input-output example pairs were constructed by randomly sampling an input example and running the program on it to determine the corresponding output example. We used simple heuristic constraint propagation code, provided to us by the authors of Balog et al. (2016), to ensure that sampled inputs did not cause errors or out-of-range values when the programs were run on them.

Training: For the sketch generator, we used a batch size of 200. We pretrained all sketch generators on the full programs for 10 epochs, and then trained on our sketch objective for 10 additional epochs. We also note that, we trained the RNN baseline for twice as long, 20 epochs, and observed no difference in performance from the baseline trained for 10 epochs. The Deepcoder-style recognizer net-

work was trained for 50 epochs.

The sketch Generator models had approximately 7 million parameters, and the Recognizer model had about 230,000 parameters.

2.2. String Transformations

Data: As our DSL, we use a modified version of the string transformation language in Devlin et al. (2017). Because our enumerator uses a strongly typed λ-calculus, additional tokens, such as `concat_list`, `concat1`, `expr_n` and `delimiter_to_regex` were added to the DSL to express lists and union types.

Training: Our Generator and Recognizer networks were each trained on 250,000 programs randomly sampled from the DSL. The sketch Generator used a batch size of 50.

2.3. AlgoLisp

Data: We implemented SKETCHADAPT and our baselines for the AlgoLisp DSL in Polosukhin & Skidanov (2018). As in Bednarek et al. (2018), we filter out evaluation tasks for which the reference program does not satisfy the input-output examples.

Training: For the AlgoLisp domain, we used a batch size of 32, and trained our Generator and Recognizer networks until loss values stopped decreasing on the 'dev' dataset, but for no fewer than 1250 training iterations.

3. Additional Experimental Analysis

Training and testing with noisy specification: For the string editing domain—where real-world user input can often be noisy—we conducted an experiment to examine our system's performance when specifications have errors. we injected random noise (insertion, deletion, or substitution) into the training and testing data. We assume that only one of the test examples is corrupted, and measure the number of specs for which we can satisfy at least three out of four test examples. We report accuracy of 53% for SketchAdapt, 52% for "Generator only", and 52% for "Synthesizer only". These results indicate that our system is affected by noise, but can still often recover the desired program from noisy inputs.

Algolisp results using only IO specification: To determine the utility of the natural language descriptions in the Algolisp experiments, we report an additional ablation, in which the description is not used. In these experiments, the Generator and Recognizer networks are conditioned on the input-output examples instead of the program descriptions. Table 1 reports our results for this "IO only" ablation. We observe that without the natural language descriptions, neither SKETCHADAPT or the lesioned baselines are able to synthesize many of the test programs.

Table 1. Algolisp results using only input-output specification

Model	Full dataset		Filtered[1]	
	(Dev)	Test	(Dev)	Test
SKETCHADAPT, IO only	(4.9)	8.3	(5.6)	8.8
Generator, IO only	(5.8)	2.4	(6.4)	2.7
Synthesizer, IO only	(8.7)	7.1	(9.3)	7.8

Evaluation runtime for Algolisp dataset: We report solve time for the Algolisp test data in Table 2. We report 25th percentile, median, and 75th percentile solve times. We note that, despite using both neural beam search and enumerative search, SKETCHADAPT does not find programs significantly slower than the RNN "Generator only" baseline. We also note that the "Synthesizer only" solve times are significantly faster because only a small proportion of the programs were solved.

Breakdown of results: In order to gain further insight into how our system compares to baselines, for each test domain, we examine to what extent problems solved by SKETCHADAPT are not solved by baselines, and visa versa. Figures 3, 4 and 5 report the degree of overlap between problems solved by SKETCHADAPT and the strongest baseline.

For all domains, a large proportion of problems are solved by SKETCHADAPT and not solved by the baseline, while a much smaller proportion of problems are solved by the baseline but not solved by SKETCHADAPT.

We additionally provide samples of programs which were solved by SKETCHADAPT and not solved by the strongest baseline (Figures 1, 2, and 3).

Table 3. Breakdown of results in the list processing domain (train on length 3 programs, test on length 4 programs). We examine the proportion of programs solved after evaluating fewer than 10^4 candidates. We compare SKETCHADAPT to the "Synthesizer only" model, which is the best performing baseline in this domain.

SKETCHADAPT	Synthesizer Only: solved	Synthesizer Only: failed	sum
solved	19%	24%	43%
failed	2%	55%	57%
sum	21%	79%	

[1] As in Bednarek et al. (2018), we filter the test and dev datasets for only those tasks for which reference programs satisfy the given specs.

Table 2. Solve times for Algolisp test programs, in seconds

		Number of training programs used				
		2000	4000	6000	8000	Full dataset
SKETCHADAPT	25th percentile	25.3	30.5	24.8	23.0	34.7
	median	37.3	46.8	38.5	36.6	55.2
	75th percentile	62.7	71.0	55.3	56.1	85.8
Generator only	25th percentile	51.1	31.8	21.8	26.4	28.2
	median	57.8	41.2	33.7	39.2	41.8
	75th percentile	100.6	60.8	49.3	59.2	63.5
Synthesizer only	25th percentile	0.4	0.5	0.6	0.5	0.4
	median	0.8	0.9	1.0	0.9	0.9
	75th percentile	1.3	1.5	2.2	1.6	3.6

Table 4. Breakdown of results in the text editing domain. We compare SKETCHADAPT to the "Generator only" model, which is the best performing baseline in this domain.

		Generator Only		
		solved	failed	sum
SKETCHADAPT	solved	55.2%	7.8%	63.0%
	failed	2.0%	35.0%	37.0%
	sum	57.2%	42.8%	

Table 5. Breakdown of results on Algolisp test data (trained on 6000 programs). We compare SKETCHADAPT to the "Generator only" model, which is the best performing baseline in this domain.

		Generator Only		
		solved	failed	sum
SKETCHADAPT	solved	45.3%	20.8%	66.1%
	failed	7.2%	26.7%	33.9%
	sum	52.5%	47.5%	

Figure 1. Sketches and programs found by SKETCHADAPT in list processing domain

Spec:
[123, -105, 60, 122, 7, -54, 15, 2, 44, 7], [-50, 82, 88, -37, 111, 115, 108, -44, 96, 107] → [-50, -105, 8, -37, 7],
[115, -75, -36, 98, -114, -91, 22, 28, -35, -7], [22, -123, -101, -17, 118, 86, 2, -106, 88, -75] → [22, -123, -101, -34, -114],
...
Sketch:
`(ZIPWITH MIN input1 (ZIPWITH MIN (FILTER <HOLE1> <HOLE2>) input0))`
where
`<HOLE0>` → `isEVEN`
`<HOLE1>` → `(MAP INC input0)`

Spec:
[4, -7, -6, 2, -5, -7, 4, -4, 1, -5], [-4, 1, 7, -3, -2, -7, 1, 5, -2, 7] → [0, 1, -26, -2],
[3, -6, -6, 4, 2, -7, -4, 2, -4, -1], [-5, -6, 4, -7, 0, 7, -7, -5, 4, 3] → [-3, 52, -16, -20],
...
Sketch:
`(ZIPWITH + (FILTER <HOLE0> <HOLE1>) (ZIPWITH * input1 input0))`
where
`<HOLE0>` → `isPOS`
`<HOLE1>` → `(MAP MUL4 input0)`

Spec:
[-1, 5, -6, 1, -4, -7, -3, 6, 4, -1], [-6, -4, 3, 4, 3, -3, 0, 3, 5, -3] → [2, 50, 45, 17, 25, 58, 9, 72, 41, 2],
[-4, 0, -4, 1, 2, -2, 7, 2, -2, -4], [-5, 6, -1, -7, -5, -6, -3, -4, 7, -5] → [32, 36, 17, 2, 8, 8, 98, 8, 53, 32],
...
Sketch:
`(ZIPWITH <HOLE0> (MAP SQR <HOLE1>) (MAP SQR input0))`
where
`<HOLE0>` → `+`
`<HOLE1>` → `(ZIPWITH MAX input1 input0)`

Spec:
[69, -49, 117, 7, -13, 84, -48, -125, 6, -68], [112, -44, 77, -58, -126, -45, 112, 23, -92, 42] → [-9, -21, -7, -15],
[0, -76, -85, 75, 62, -64, 95, -77, -78, -114], [-111, 92, -121, 108, 5, -22, -126, -40, 9, -115] → [-21, -39, -57],
...
Sketch:
`(FILTER <HOLE0> (MAP DIV2 (ZIPWITH MIN input0 <HOLE1>)))`
where
`<HOLE0>` → `isODD`
`<HOLE1>` → `(MAP DIV3 input1)`

Figure 2. Sketches and programs found by SKETCHADAPT in text editing domain. Programs edited for readability.

Spec:
(('Lashanda' → 'Las'), ('Pennsylvania' → 'Pennsyl'), ('California' → 'Calif'), ('Urbana' → 'U'))
Sketch:
`(apply_fn <HOLE1> (SubStr <HOLE2> <HOLE3>))`
where
`<HOLE1>` → `GetTokenWord-1`
`<HOLE2>` → `Position0`
`<HOLE3>` → `Position-5`

Spec:
(('Olague(California' → 'California'), ('621(Seamons' → 'Seamons'), ('Mackenzie(Dr(5(Park' → 'Park'), ('+174(077(Storrs' → 'Storrs'))
Sketch:
`(apply_fn GetFirst_PropCase3 (GetSpan right_paren index-1 <HOLE> Alphanum <HOLE> End))`
where
`<HOLE1>` → `End`
`<HOLE2>` → `Index-1`

Spec:
(('Karrie' → 'Karri'), ('Jeanice' → 'Jeani'), ('Brescia' → 'Bresc'), ('Lango' → 'Lango'))
Sketch:
`(concat_list <HOLE> GetFirst_Lower4)`
where
`<HOLE>` → `GetTokenAlphanum0`

Figure 3. Sketches and programs found by SKETCHADAPT in AlgoLisp dataset

Description:
given numbers a and b , let c be the maximum of a and b , reverse digits in c , compute c
Sketch:
```
(reduce (reverse (digits (max a <HOLE>))) 0 (lambda2 (+ (* arg1 10) arg2)))
```
where
<HOLE> → b

Description:
you are given arrays of numbers a and c and a number b , your task is to compute number of values in a that are less than values on the same index in reverse of values in c bigger than b
Sketch:
```
(reduce (map (range 0 (min (len a) (len (reverse (<HOLE> c (partial1 b >)))))))
(lambda1 (if (< (deref a arg1) (deref (reverse (filter c (partial1 b >))) arg1))
1 0))) 0 +)
```
where
<HOLE> → filter

Description:
consider arrays of numbers a and b and a number c , only keep values in the second half of a , compute sum of first c values among values of a that are also present in b after sorting in ascending order
Sketch:
```
(reduce (slice (sort (filter (slice a (/ (len a) 2) (len a)) (lambda1 (reduce
(map b (partial0 arg1 ==)) false || )))) 0 c) 0 +)
```
where
<HOLE> → reduce

Description:
consider a number a and an array of numbers b , your task is to find the length of the longest subsequence of odd digits of a that is a prefix of b
Sketch:
```
(reduce (<HOLE> a) 0 (lambda2 (if (== arg2 (if (< arg1 (len b)) (deref b arg1)
0)) (+ arg1 1) arg1)))
```
where
<HOLE> → digits

References

Balog, M., Gaunt, A. L., Brockschmidt, M., Nowozin, S., and Tarlow, D. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

Bednarek, J., Piaskowski, K., and Krawiec, K. Ain't nobody got time for coding: Structure-aware program synthesis from natural language. *arXiv preprint arXiv:1810.09717*, 2018.

Bunel, R., Hausknecht, M., Devlin, J., Singh, R., and Kohli, P. Leveraging grammar and reinforcement learning for neural program synthesis. *arXiv preprint arXiv:1805.04276*, 2018.

Devlin, J., Uesato, J., Bhupatiraju, S., Singh, R., Mohamed, A.-r., and Kohli, P. Robustfill: Neural program learning under noisy i/o. *arXiv preprint arXiv:1703.07469*, 2017.

Kingma, D. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Polosukhin, I. and Skidanov, A. Neural program search: Solving programming tasks from description and examples. *arXiv preprint arXiv:1802.04335*, 2018.