Towards Neural Functional Program Evaluation

Torsten Scholak[1]*, **Jonathan Pilault**[2]*, **Joey Velez-Ginorio**[3]
[1]ElementAI, a ServiceNow company, [2]Polytechnique Montreal & Mila, [3]University of Pennsylvania
[1]`torsten.scholak@servicenow.com`, [2]`pilaultj@mila.qc`

Abstract

This paper explores the capabilities of current transformer-based language models for program evaluation of simple functional programming languages. We introduce a new program generation mechanism that allows control over syntactic sugar for semantically equivalent programs. T5 experiments reveal that neural functional program evaluation performs surprisingly well, achieving high 90% exact program match scores for most in-distribution and out-of-distribution tests. Using pretrained T5 weights has significant advantages over random initialization. We present and evaluate on three datasets to study generalization abilities that are specific to functional programs based on: type, function composition, and reduction steps. Code and data are publicly available at https://github.com/ElementAI/neural-interpreters.

1 Introduction

Neural models originally developed for natural language processing show promising performance for modeling computer programming languages (Chen et al., 2021; Austin et al., 2021). This has led to a number of interesting applications, and deep-learning models are now successfully and routinely applied in tools that assist developers in writing and understanding programs and code. For instance, neural language models can synthesize (Gulwani et al., 2017; Ellis et al., 2020), complete (Chen et al., 2021), and summarize programs (Elnaggar et al., 2021), whether they are written in mainstream languages like Python and Java, or in domain-specific languages like SQL and regex.

A less explored application of neural models is program evaluation (Reed and De Freitas, 2015; Zaremba and Sutskever, 2014). Here, the challenge is to predict the output of a program given its input. In software engineering, this problem is solved by program interpreters (Nystrom, 2021), which are implemented as rule-based, non-differentiable systems that take formal program code as input and produce outputs, effects, and/or errors. While these interpreters must reject programs that are syntactically incorrect or contain semantic errors, neural interpreters allow the evaluation of incomplete, informally-, or even incorrectly-specified programs. Developers could use them to predict output or effects of programs before they are fully written, which could aid in their very creation. Previous studies have focused mainly on simple imperative programming languages, e.g., FORTH (Bošnjak et al., 2017) and subsets of Python (Bieber et al., 2020). For these languages, evaluation is complicated by control flow, state, and mutability. State-of-the-art techniques solve these complications with specialized, complex model architectures, but results are not as promising as one might hope (Bieber et al. (2020) report an accuracy of 62.1% on their task and dataset).

This paper studies neural program evaluation for functional programming languages, which share few of the complications of imperative languages (Feser et al., 2016). Our emphasis is on lambda calculi, which sit at the core of modern functional programming languages like Scheme and Haskell (Pierce and Benjamin, 2002). We show that program evaluation can be recast such that it is tractable for a language model based on the standard transformer architecture (Vaswani et al., 2017). Rather than predicting a program's output given an input, we train the model to reduce a program to a form

*Equal contribution.

35th Conference on Neural Information Processing Systems (aiplans 2021), Sydney, Australia.

that cannot be reduced further. This generalizes the notion of program evaluation, as it can also be used to model partial application of functions and programs.

Unlike works that use neural programs to strengthen generalization of models (Chen et al., 2020; Nye et al., 2020) on another downstream task, we study the generalization of neural program evaluation explicitly. We propose three data splits to measure "type generalization", "function compositional generalization", and "reduction steps generalization" of our approach. Furthermore, we investigate the effect of syntax and abstraction on neural program evaluation. We define two lambda calculi that are equally capable of expressing semantically equivalent programs, yet are different in their syntax. We find that the neural program evaluation problem is slightly less tractable for the lambda calculus with the simpler syntax. Lastly, we compare two evaluation strategies for lambda calculus, lazy evaluation and eager evaluation, and find that they are similarly tractable.

2 Evaluation of Lambda Calculus

Our experiments are based on synthetic data for two lambda calculi. The first lambda calculus (1^{st} LC) is untyped and has three syntactic constructs: variable, lambda, and application expressions, i.e.,

$$e = \mathrm{x} \mid \lambda \mathrm{x}.e \mid e\, e \tag{1}$$

The second lambda calculus (2^{nd} LC) adds types and additional syntactic constructs:

$$t = \mathrm{Unit} \mid \mathrm{Bool} \mid \mathrm{List}\, t \mid t \to t \tag{2}$$

$$e = \mathrm{x} \mid \lambda \mathrm{x}.e \mid e\, e \mid () \mid \mathrm{True} \mid \mathrm{False} \mid \mathrm{if}\; e \;\mathrm{then}\; e \;\mathrm{else}\; e \mid \mathrm{Nil} \mid \mathrm{Cons}\, e\, e \mid \mathrm{Foldr}\, e\, e\, e \tag{3}$$

Terms are generated exclusively from the 2^{nd} LC using types to guide the generation. First, the program type is chosen at random from the set of types t. Terms e are then generated according to the type, beginning at the root of the term tree and proceeding down the branches to the leaves. Specifically, production is recursive and top to bottom (root to leaves). Generation starts with the outermost type constructor, and proceeds by sampling from all compatible term constructors. Production stops when none of the resulting leaf nodes contains hole terms, e. Care is taken to ensure that the generated program does not contain any free variables, is well-typed, and that it terminates without error. Terms from the 1^{st} LC are obtained by transforming terms from the 2^{nd} LC. We therefore only consider the normalisable subset of the 1^{st} LC. We use Church encoding to represent syntactic constructs from the 2^{nd} LC that do not appear in the 1^{st} LC (Pierce and Benjamin, 2002). For example, lists are represented by their right fold:

$$\mathrm{Nil} = \lambda \mathrm{c}.\lambda \mathrm{n}.\mathrm{n} \qquad \mathrm{Cons}\, h\, \tau = \lambda \mathrm{c}.\lambda \mathrm{n}.\mathrm{c}\, h\, (\tau\, \mathrm{c}\, \mathrm{n}) \qquad \mathrm{Foldr}\, f\, e\, l = l\, f\, e \tag{4}$$

where h and τ are head and tail terms, and where f, e, and l refer to the combining function, the initial value, and the Church-encoded input list of elements, respectively.

We consider two reduction strategies: lazy and eager evaluation. Lazy evaluation reduces a term to weak head normal form (WHNF), which is a term that is not an application expression. For example, the term $(\lambda \mathrm{x}.\mathrm{x})(\lambda \mathrm{y}.\mathrm{y})$ is reduced to the WHNF $\lambda \mathrm{y}.\mathrm{y}$, while the term $\lambda \mathrm{y}.(\lambda \mathrm{x}.\mathrm{x})\, \mathrm{y}$ is already in WHNF and thus not reduced further. Eager evaluation not only reduces a term to WHNF, but also reduces all subterms of the term. We call this the "deep" reduction and refer to the result as the deep normal form (DNF). The terms $(\lambda \mathrm{x}.\mathrm{x})(\lambda \mathrm{y}.\mathrm{y})$ and $\lambda \mathrm{y}.(\lambda \mathrm{x}.\mathrm{x})\, \mathrm{y}$ are reduced to the same DNF $\lambda \mathrm{y}.\mathrm{y}$.

We define the number of reduction steps of a term to be the number of reductions performed on its root term and all its subterms. Reduction to WHNF is a special case of reduction to DNF, and the number of reduction steps to WHNF is always smaller or equal to the number of reduction steps to DNF. We use the number of reduction steps to denote the degree of evaluation complexity of a term, with more steps translating to higher complexity. This can then be used to evaluate the performance of our neural program evaluation models.

3 Experiments

In our experiments, we train a language model based on the encoder-decoder transformer architecture to reduce programs to their normal forms. Depending on the experiment, the input to the network is a term in either the 1^{st} LC or 2^{nd} LC, and the output is either the WHNF or the DNF of the input term.

Target	Exact Match			
	VR		NVR	
	1^{st} LC	2^{nd} LC	1^{st} LC	2^{nd} LC
T5-Small Pretrained				
WHNF	0.886	0.926	0.598	0.922
DNF	0.698	0.920	—	—
T5-Large Pretrained				
WHNF	0.990	0.996	0.984	0.996
DNF	0.988	0.994	—	—
T5-Large from Scratch				
WHNF	0.530	0.532	0.576	0.581
DNF	0.533	0.536	—	—

Table 1: Results on random splits of the dataset.

Target	Exact Match			
	VR		NVR	
	1^{st} LC	2^{nd} LC	1^{st} LC	2^{nd} LC
T5-Small Pretrained				
WHNF	0.824	0.870	0.502	0.796
DNF	0.464	0.740	—	—
T5-Large Pretrained				
WHNF	0.992	0.986	0.970	0.968
DNF	0.932	0.926	—	—
T5-Large from Scratch				
WHNF	0.519	0.508	0.561	0.552
DNF	0.493	0.480	—	—

Table 2: Results on the split-by-type dataset.

Target	Exact Match			
	VR		NVR	
	1^{st} LC	2^{nd} LC	1^{st} LC	2^{nd} LC
T5-Large Pretrained				
WHNF	0.892	0.938	0.928	0.958
DNF	0.952	0.972	—	—

Table 3: Results on the compositional dataset.

Target	Exact Match			
	VR		NVR	
	1^{st} LC	2^{nd} LC	1^{st} LC	2^{nd} LC
T5-Large Pretrained				
WHNF	0.686	0.930	0.658	0.934
DNF	0.758	0.960	—	—

Table 4: Results for splits by reduction steps.

Terms are encoded as strings, which are then converted to sequences of tokens with ids that are used as inputs to the network. The string representation of a term is obtained by pretty-printing the term in Haskell syntax (Jones, 2003). Variables are encoded as `x0`, `x1`, `x2`, etc. and lambda expressions are encoded as `\x0 -> e`. Application expressions are encoded as `e1 e2`. Where necessary, parentheses are used to denote the precedence of application. Thus, the 1^{st} LC term $(\lambda x.x)(\lambda y.y)$ is encoded as `(\x0 -> x0) (\x1 -> x1)`. In the 2^{nd} LC, Nil becomes `[]`, and $\mathrm{Cons}\, e1\, e2$ becomes `e1 : e2`, while lists of one or more elements are pretty-printed using Haskell's built-in list syntax: `[e1, e2, ...]`. We deviate from the usual Haskell syntax for if-then-else expressions: $\mathrm{if}\, e1\, \mathrm{then}\, e2\, \mathrm{else}\, e3$ is encoded as `ite e1 e2 e3`. The type of a 2^{nd} LC term is not encoded explicitly. Hence, both lambda calculi appear to be untyped to the models except that non-normalisable terms do not appear in the data. Our type-driven generation process avoids generating non-normalising terms. We distinguish evaluation with variable renaming (VR) and evaluation without variable renaming (NVR). In the former case, the variables in the reduced program are freshly generated in the order of their appearance, while in the latter case, the variable names are preserved during reduction, which we expect to be more tractable.

We use a pretrained T5 model (Raffel et al., 2020) as our base model. The number of parameters of the T5-Small and T5-Large models can be found in the appendix section A. We fine-tune the model using the Adafactor optimizer (Shazeer and Stern, 2018) with a maximum learning rate of 10^{-4} in a linear decay schedule and a batch size of 2048. Predictions are made using greedy decoding.

A dataset of 1 million unique examples is generated by sampling from the distribution of terms in the 2^{nd} LC, which are subsequently converted to the 1^{st} LC. We thus use the same dataset in experiments with the 1^{st} LC and the 2^{nd} LC. We chose to generate from 2^{nd} LC so that we could exploit its types for type-driven generation of terms. Terms that are too long to fit within the model's maximum input (512 tokens) and output lengths (256 tokens) are discarded.

Performance is evaluated using the average exact string match metric between the normal forms of the predicted terms and the normal forms of the ground-truth terms. For a given example, if all predicted terms match 100% with ground-truth terms, an exact match is found. The average is taken by dividing the number of exact matches over the total number of examples.

Random Split Our findings on uniform random splits are summarized in Table 1. We report the average exact-match results on a held-out dataset of 500 examples of the best performing models. Those models were trained on 90000 examples for up to 100 epochs. We find that the results for T5-Large are all close to the maximum, and that the results for T5-Small are worse and exhibit more variability: For T5-Small, the average exact-match is around 0.92 for all 2^{nd} LC reduction tasks, while the numbers for the 1$^{\text{st}}$ LC are much lower. The 1$^{\text{st}}$ LC results for WHNF-NVR and DNF-VR are particularly weak, below 0.6 and 0.7, respectively. This pattern is also seen in the results for T5-Large to a lesser degree. Interestingly, if we do not use pretrained weights but instead use Xavier Normal initialization (Glorot and Bengio, 2010), performance drops by close to 40 percentage points[2] across the board. This result shows that pretraining has significant upside benefits for evaluating neural functional programs. We intuit that Natural Language Understanding skills accumulated during pretraining translates well into neural program execution tasks. We will leave it for future work to verify this intuition more thoroughly.

Split by Type In this experiment, reported in Table 2, we split the dataset into two parts based on the types of 2$^{\text{nd}}$ LC terms. The examples are ordered by frequency of type occurrence. The training set contains the most common types representing 80% of the dataset, with the test set containing the remaining 20% comprised of different and less common types.[3] We subsample the training set to 80000 examples and the test set to 500 examples. Despite the changes, the results are similar to the uniform random split in Table 1. For pretrained models, performance decrease is strongest for the T5-Small model, while the T5-Large model performs almost at the level of the uniform random split, except for a 6 percentage-point performance drop on the DNF tasks. We attribute the absence of this drop for the WHNF tasks to the fact that reduction to WHNF tends to be concentrated around the root term, which is well covered by the training set. Similarly to previous experiments, we see that pretraining has a larger impact on performance than model size. The performance drops by around 40 percentage points with training from scratch. In all experiments, we observe that 2$^{\text{nd}}$ LC performs slightly worse (-9 percentage points on average) for large models.

Split by Function Composition For this experiment, we create a new evaluation dataset from the training examples used in the first experiment on random splits. We produce unique examples by composing terms $e1$, $e2$ from the training set using application, $e1\ e2$.[4] The final dataset contains 500 examples of each lambda calculus and evaluation strategy. We use the best performing models on the random splits to evaluate the performance of the new dataset. Our findings, reported in Table 3, show that the performance on the compositional evaluation dataset is lower compared to the random split. The 2$^{\text{nd}}$ LC results are slightly better than the 1$^{\text{st}}$ LC results, and among the 1$^{\text{st}}$ LC results, those on the WHNF-VR task are particularly poor.

We also analyzed the results on the new dataset by token length. 1$^{\text{st}}$ LC input and target programs are both about 2.5 times as long as their 2$^{\text{nd}}$ LC counterparts. When compensated for this difference, the dependence of the exact-match performance on the target length is about the same for both languages: it stays high and approximately constant for targets below 100 tokens for 1$^{\text{st}}$ LC and 50 tokens for 2$^{\text{nd}}$ LC, and then falls off linearly with length. See section 3 for details.

Split by Number of Reduction Steps This last experiment splits the data such that the training examples contain the fewest reduction steps and the test set the most reduction steps. Since larger reduction step counts imply greater complexity, we expect a performance drop compared to the random split. Our results are presented in Table 4.

Consider the WHNF reduction task. The 1$^{\text{st}}$ LC and 2$^{\text{nd}}$ LC have median reduction step counts of 4 and 3, respectively. In order to cover the bulk of the distribution of reduction steps, training examples have therefore up to 6 reduction steps. Larger counts up to 12 steps are in the test set. We subsampled the training and test sets such that the proportion of examples per reduction step count is the same between the two lambda calculi. We find that the 2$^{\text{nd}}$ LC model performs much better than the 1$^{\text{st}}$ LC model on the test set: For 1$^{\text{st}}$ LC, the performance drops by one third compared to the random split, while for 2$^{\text{nd}}$ LC the performance drops by about 7 percentage points. For the DNF reduction task, the 1$^{\text{st}}$ LC and 2$^{\text{nd}}$ LC have median reduction step counts of 6 and 4, respectively. The training examples have up to 8 reductions steps, and examples with between 9 and 32 reduction steps are

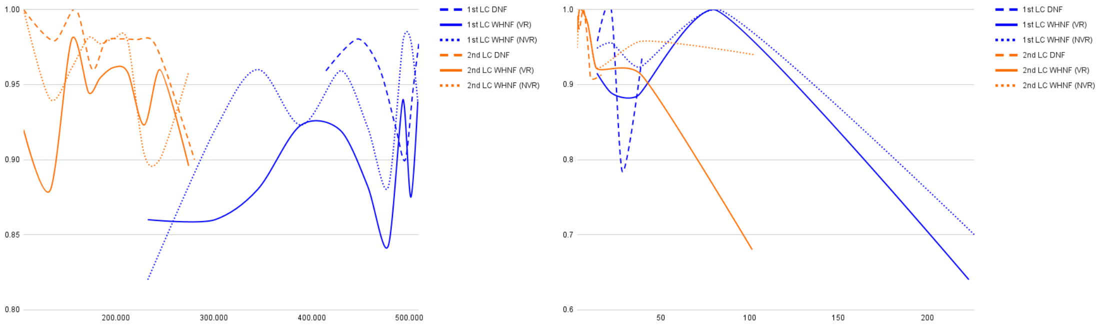
[2] Additional experiments not reported in Table 1 show that training T5 from scratch with default Huggingface initialization reduces the performance of the model. We do not observe numbers above 0.35.

[3] The six most common types are: Bool, Unit, $\mathrm{List\ Bool}$, $\mathrm{List\ Unit}$, $\mathrm{Unit} \to \mathrm{Bool}$, and $\mathrm{Bool} \to \mathrm{Unit}$. These types are among those found in the training set, while the types in the test set are much more nested, e.g., $(\mathrm{Unit} \to \mathrm{List\ List\ Bool} \to \mathrm{List\ Unit}) \to \mathrm{List\ List\ List\ List\ Unit}$.

[4] To improve the dataset's diversity, each term $e1$, $e2$ can only occur up to 3 times. Type checking is performed on the new examples to ensure well-typedness. Input/output maximum lengths are limited to 512/256 tokens.

test examples. Subsampling equivalent to that of the WHNF task was applied. The results for DNF reduction are similar to those obtained for WHNF. Again, the drop in performance is bigger for the 1st LC model than for the 2nd LC model.

Input and Output Program Length To verify if program input and output lengths (number of tokens) may explain differences between 1st LC and 2nd LC, we show exact match performance for increasing program lengths in Figures 1 (a) and 1 (b). We grouped the data in 10 different program length ranges with equal amounts of examples. The graphs plot the average program length of each range (x-axis) against the average exact match performance (y-axis). 2nd LC program inputs are generally smaller. However, for WHNF and NVR, when 2nd LC and 1st LC length intersect, 1st LC exact match is close to 5 percentage points lower. Interestingly, for 1st LC, performance tends to increase with input length as shown in Figure 1 (a). Output program lengths are typically 2.5 times smaller for 2nd LC. Curves in Figure 1 (b) display wave patterns with clear downward trends as program length increases. For the same length, the 1st LC results are not consistently at or below the 2nd LC results. Instead, the 1st LC results appear shifted and scaled up along the length-axis. Overall, we notice that exact-match performance on the output length is about the same for both languages: it stays high and approximately constant for targets below 50 tokens for 1st LC and 100 tokens for 2nd LC and then falls off linearly with length.



(a) Input Program

(b) Output Program

Figure 1: T5-Large Exact Match vs. (a) Input and (b) Output Program Lengths

4 Concluding Remarks

We compare neural execution of programs in two lambda calculi, 1st LC and 2nd LC. Both express semantically equivalent programs, but 1st LC uses a simpler syntax at the cost of longer program length. Our experiments with T5 have shown near tractable performance for synthetic in-distribution 1st LC and 2nd LC data, but stark performance differences are observed for out-of-distribution data. We analyze our models' generalization capability on four splits of the data: uniformly random, by program type, by function composition, and by number of reduction steps. On average, models trained on 1st LC generalize less then models trained on 2nd LC. We observe significant gains from using T5 with pretrained weights compared to randomly initialized models on in-distribution and unseen-type experiments. Input/output program lengths cannot completely account for differences in languages. Our experiments indicate that neural interpretation of functional programs is more tractable when expressed in more "sugared" syntax, but further evidence is needed to completely support this claim. Further, we show that T5-Large is consistently better than T5-Small. Pretraining on the T5 corpus and tasks yields better results than training from scratch. Our analysis of different evaluation strategies show that lazy reduction is slightly easier to learn than eager reduction, but may generalize less on nontrivial splits of the data. Results with and without variable renaming do not show a clear trend. This is unexpected since variable renaming is an additional syntactic operation that complicates the task and is not strictly necessary.

Future work on more difficult tasks and data splits will allow us to better understand the generalization properties of our models as well as the impact of syntactic complexity on performance.

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program Synthesis with Large Language Models. *arXiv preprint arXiv:2108.07732* (2021).

David Bieber, Charles Sutton, Hugo Larochelle, and Daniel Tarlow. 2020. Learning to Execute Programs with Instruction Pointer Attention Graph Neural Networks. *Advances in Neural Information Processing Systems* 33 (2020).

Matko Bošnjak, Tim Rocktäschel, Jason Naradowsky, and Sebastian Riedel. 2017. Programming with a differentiable forth interpreter. In *International conference on machine learning*. PMLR, 547–556.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harri Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

Xinyun Chen, Chen Liang, Adams Wei Yu, Dawn Song, and Denny Zhou. 2020. Compositional Generalization via Neural-Symbolic Stack Machines. *CoRR* abs/2008.06662 (2020). arXiv:2008.06662 https://arxiv.org/abs/2008.06662

Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sable-Meyer, Luc Cary, Lucas Morales, Luke Hewitt, Armando Solar-Lezama, and Joshua B. Tenenbaum. 2020. DreamCoder: Growing generalizable, interpretable knowledge with wake-sleep Bayesian program learning. arXiv:2006.08381 [cs.AI]

Ahmed Elnaggar, Wei Ding, Llion Jones, Tom Gibbs, Tamas Feher, Christoph Angerer, Silvia Severini, Florian Matthes, and Burkhard Rost. 2021. CodeTrans: Towards Cracking the Language of Silicon's Code Through Self-Supervised Deep Learning and High Performance Computing. arXiv:2104.02443 [cs.SE]

John K Feser, Marc Brockschmidt, Alexander L Gaunt, and Daniel Tarlow. 2016. Differentiable functional program interpreters. *arXiv preprint arXiv:1611.01988* (2016).

Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research, Vol. 9)*, Yee Whye Teh and Mike Titterington (Eds.). PMLR, Chia Laguna Resort, Sardinia, Italy, 249–256. https://proceedings.mlr.press/v9/glorot10a.html

Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program Synthesis. *Foundations and Trends® in Programming Languages* 4, 1-2 (2017), 1–119. https://doi.org/10.1561/2500000010

Simon Peyton Jones. 2003. *Haskell 98 language and libraries: the revised report*. Cambridge University Press.

Maxwell I. Nye, Armando Solar-Lezama, Joshua B. Tenenbaum, and Brenden M. Lake. 2020. Learning Compositional Rules via Neural Program Synthesis. *CoRR* abs/2003.05562 (2020). arXiv:2003.05562 https://arxiv.org/abs/2003.05562

R. Nystrom. 2021. *Crafting Interpreters*. Genever Benning. https://books.google.ca/books?id=ySOBzgEACAAJ

Benjamin C Pierce and C Benjamin. 2002. *Types and programming languages*. MIT press.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research* 21 (2020), 1–67.

Scott Reed and Nando De Freitas. 2015. Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279* (2015).

Noam Shazeer and Mitchell Stern. 2018. Adafactor: Adaptive learning rates with sublinear memory cost. In *International Conference on Machine Learning*. PMLR, 4596–4604.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

Wojciech Zaremba and Ilya Sutskever. 2014. Learning to execute. *arXiv preprint arXiv:1410.4615* (2014).

A Model sizes

Model sizes are listed in Table 5.

Model	Params (M)
T5-Small Pretrained	60
T5-Small from Scratch	60
T5-Large Pretrained	770
T5-Large from Scratch	770

Table 5: Number of parameters in millions.

B Example Data

The Table 6 lists parallel examples of 1^{st} LC and 2^{nd} LC terms and their deep normal forms from our dataset.

1st LC	(\x0 -> \x1 -> \x2 -> x1) ((\x3 -> \x4 -> (\x5 -> \x6 -> x6) (\x7 -> (\x8 -> \x9 -> x7) x4) ((\x10 -> \x11 -> \x12 -> \x13 -> x12 x10 (x11 x12 x13)) (\x14 -> x14) (\x15 -> \x16 -> x16))) (\x17 -> \x18 -> x17))
1st LC DNF	\x0 -> \x1 -> x0
2nd LC	(\x0 -> True) ((\x1 -> \x2 -> foldr (\x3 -> (\x4 -> \x5 -> x3) x2) [()] []) True)
2nd LC DNF	True
1st LC	(\x0 -> \x1 -> x1) (\x2 -> \x3 -> \x4 -> \x5 -> \x6 -> x6) (\x7 -> \x8 -> (\x9 -> \x10 -> \x11 -> \x12 -> \x13 -> x13) x8 x7)
1st LC DNF	\x0 -> \x1 -> \x2 -> \x3 -> \x4 -> x4
2nd LC	ite False (\x0 -> \x1 -> \x2 -> []) (\x3 -> \x4 -> (\x5 -> \x6 -> \x7 -> []) x4 x3)
2nd LC DNF	\x0 -> \x1 -> \x2 -> []
1st LC	(\x0 -> \x1 -> x1) (\x2 -> \x3 -> x3) (\x4 -> x4) ((\x5 -> \x6 -> \x7 -> \x8 -> x7 x5 (x6 x7 x8)) ((\x9 -> \x10 -> x10) (\x11 -> x11) (\x12 -> x12)) ((\x13 -> \x14 -> \x15 -> \x16 -> x15 x13 (x14 x15 x16)) ((\x17 -> \x18 -> x18) (\x19 -> x19) (\x20 -> x20) (\x21 -> x21)) ((\x22 -> \x23 -> \x24 -> \x25 -> x24 x22 (x23 x24 x25)) (\x26 -> \x27 -> x26) (\x28 -> \x29 -> x29) (\x30 -> (\x31 -> \x32 -> x32) x30) ((\x33 -> \x34 -> \x35 -> \x36 -> x35 x33 (x34 x35 x36)) (\x37 -> x37) (\x38 -> \x39 -> x39)))) (\x40 -> \x41 -> x41) (\x42 -> x42))
1st LC DNF	\x0 -> x0
2nd LC	foldr (\x0 -> \x1 -> x1) (\x2 -> x2) [] (foldr (\x3 -> \x4 -> x4) () ((:) (ite False () ()) ((:) (ite False (\x5 -> x5) (\x6 -> x6) ()) (foldr (\x7 -> (\x8 -> \x9 -> x9) x7) [()] [True]))))
2nd LC DNF	()
1st LC	(\x0 -> \x1 -> \x2 -> \x3 -> x2 x0 (x1 x2 x3)) ((\x4 -> x4) (\x5 -> x5)) ((\x6 -> \x7 -> \x8 -> x8) ((\x9 -> (\x10 -> \x11 -> \x12 -> \x13 -> x12 x10 (x11 x12 x13)) ((\x14 -> \x15 -> x15) (\x16 -> \x17 -> x17) (\x18 -> \x19 -> x18)) ((\x20 -> \x21 -> \x22 -> x22) x9) (\x23 -> (\x24 -> \x25 -> x24) (\x26 -> x23) (\x27 -> x23)) (\x28 -> \x29 -> x28)) (\x30 -> x30)))
1st LC DNF	\x0 -> \x1 -> x0 (\x2 -> x2) x1
2nd LC	(:) ((\x0 -> x0) ()) ((\x1 -> []) ((\x2 -> foldr (\x3 -> ite True (\x4 -> x3) (\x5 -> x3)) True ((:) (ite False False True) ((\x6 -> []) x2))) ()))
2nd LC DNF	[()]
1st LC	(\x0 -> \x1 -> x1) (\x2 -> (\x3 -> \x4 -> x4) (\x5 -> (\x6 -> (\x7 -> (\x8 -> \x9 -> x9) x7) ((\x10 -> \x11 -> \x12 -> x11) x2)) x2) (\x13 -> x13)) (\x14 -> (\x15 -> \x16 -> x16) (\x17 -> (\x18 -> \x19 -> \x20 -> x20) (\x21 -> x21) (\x22 -> x22)) ((\x23 -> \x24 -> \x25 -> \x26 -> x25 x23 (x24 x25 x26)) ((\x27 -> \x28 -> x28) (\x29 -> \x30 -> x30) (\x31 -> \x32 -> \x33 -> \x34 -> x34) x14) (\x35 -> \x36 -> x36)))
1st LC DNF	\x0 -> \x1 -> \x2 -> x1 (\x3 -> \x4 -> \x5 -> x5) x2
2nd LC	foldr (\x0 -> foldr (\x1 -> (\x2 -> (\x3 -> (\x4 -> \x5 -> x5) x3) ((\x6 -> True) x0)) x0) (\x7 -> x7) []) (\x8 -> foldr (\x9 -> (\x10 -> \x11 -> \x12 -> x12) () ()) [foldr (\x13 -> \x14 -> x14) (\x15 -> \x16 -> \x17 -> ()) [] x8] []) []
2nd LC DNF	\x0 -> [\x1 -> \x2 -> ()]

Table 6: Five program examples. Shown are the equivalent 1st LC and 2nd LC terms and their respective deep normal forms.