TREE-TO-TREE NEURAL NETWORKS FOR PROGRAM TRANSLATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Program translation is an important tool to migrate legacy code in one language into an ecosystem built in a different language. In this work, we are the first to consider employing deep neural networks toward tackling this problem. We observe that program translation is a modular procedure, in which one sub-tree in the source is translated into the corresponding target sub-tree at each step. To capture this intuition, we design a tree-to-tree neural network as an encoder-decoder architecture to translate a source tree into a target one. Meanwhile, we develop an attention mechanism for the tree-to-tree model, so that when the decoder expands one non-terminal in the target tree, the attention mechanism locates the corresponding sub-tree in the source to guide the expansion of the decoder. We develop two tasks to evaluate the program translation capability of our tree-to-tree model against two state-of-the-art approaches. We observe that our approach is consistently better than the baselines on all tasks and all metrics with a margin of up to 10%. These results demonstrate that the tree-to-tree model is a promising tool for tackling the program translation problem.

1 INTRODUCTION

Programs are the main tool for building computer applications, the IT industry, and the digital world. Various programming languages have been invented to facilitate programmers to develop programs for different applications. At the same time, the variety of different programming languages also introduces a burden when programmers want to combine programs written in different languages together. Therefore, there is a tremendous need to enable program translation between different programming languages.

Nowadays, to translate programs in different programming languages, typically programmers would manually investigate the correspondence between the grammars of the two languages, then develop a rule-based translator. However, this process can be inefficient and error-prone. In this work, we make the first attempt to examine whether we can leverage deep neural networks to build a translator automatically.

Intuitively, the program translation problem in its format is similar to a natural language translation problem. Some previous work propose to adapt phrase-based statistical machine translation approaches for code migration (Karaivanov et al., 2014; Nguyen et al., 2015). We may also consider applying neural machine translation (NMT) methods, such as sequence-to-sequence-based models (Bahdanau et al., 2015; Cho et al., 2015; Eriguchi et al., 2016; He et al., 2016; Vaswani et al., 2017), to handle the program translation problem. However, a big challenge making a sequence-to-sequence model ineffective is that, unlike human languages, programming languages all have rigorous grammars and are not tolerant to typos and grammatical mistakes. It has been demonstrated that it is very hard for an RNN-based sequence generator to generate syntactically correct programs when the lengths grow large (Karpathy et al., 2015).

In this work, we observe that the main issue of an RNN that makes it hard to produce syntactically correct programs is that it entangles two sub-tasks together: (1) learning the grammar; and (2) aligning the sequence with the grammar. When these two tasks can be handled separately, the performance can typically boost. For example, Dong & Lapata (2016) employs a tree-based decoder to separate the two tasks. In particular, the decoder in (Dong & Lapata, 2016) leverages the tree structural information to (1) generate the nodes at the same depth of the parse tree using an LSTM

decoder; and (2) expand a non-terminal and generate its children in the parse tree. Such an approach has been demonstrated to achieve the state-of-the-art results on several semantic parsing tasks.

Inspired by this observation, we hypothesize that the structural information in both input and output parse trees can be leveraged to enable such a separation. Inspired by this intuition, we propose tree-to-tree neural networks to combine both a tree encoder and a tree decoder. In particular, we observe that in the program translation problem, both input and output programs have their parse trees. In addition, a cross-language compiler typically follows a modular procedure to translate the individual sub-components in the source tree into their corresponding target ones, and then compose them to form the final target tree. Therefore, we design the workflow of a tree-to-tree neural network to align with this procedure: when the decoder expands a non-terminal, it locates the corresponding sub-tree in the source using an attention mechanism, and uses the information of the sub-tree to guide the non-terminal expansion. In particular, a tree encoder is helpful in this scenario, since it can aggregate all information of a sub-tree to the embedding of its root, so that the embedding can be used to guide the non-terminal expansion of the target tree.

We follow the above intuition to design the tree-to-tree model. To the best of our knowledge, this is the first tree-to-tree neural network architecture proposed for translation tasks in the literature. To test our hypothesis, we develop two program translation tasks. The first class of tasks is synthetic, and we use it to evaluate program translation between two different programming paradigms (one imperative and the other functional). The second class of tasks takes advantage of a cross-compiler between two full-fledged languages: CoffeeScript and JavaScript. We use it to evaluate the translation of programs using Python-style succinct syntax into the ones using C-style verbose syntax.

For training and evaluation, we use a pCFG-based generator to generate programs satisfying the grammar of a programming language. This is different from some previous work on code migration, which use open-source projects implemented in both the source and the target programming languages to build the dataset (Karaivanov et al., 2014; Nguyen et al., 2015). While handling real-world programs is also an important direction, randomly generated programs can reach corner cases that may not frequently seen in practice. Therefore, testing on randomly generated programs is an effective way to examine the correctness of the learned model for program translation.

Our evaluation demonstrates that our tree-to-tree model outperforms other state-of-the-art approaches on the program translation tasks, and yields a margin of up to 10% on the token accuracy and up to 5% on the program accuracy. These results demonstrate that our tree-to-tree model is promising toward tackling the program translation problem.

2 PROGRAM TRANSLATION PROBLEM

In this work, we consider the problem of translating a program in one language into another. One approach is to model the problem as a machine translation problem between two languages, and thus numerous neural machine translation approaches can be applied.

For the program translation problem, however, a unique property is that each input program unambiguously corresponds to a unique parse tree. Thus, rather than modeling the input program as a sequence of tokens, we can consider the problem as translating a source tree into a target tree. Note that most modern programming languages are accompanied with a well-developed parser. Thus we can assume the parse trees of both the source and the target programs can be easily computed.

The challenge of the problem in our consideration is that the cross-compiler that translates programs from one language into another typically does not exist. Therefore, even if we assume the existence of parsers for both the source and the target languages, the translation problem itself is still non-trivial. We formally define the problem as follows.

Definition 1 (Program translation)**.** *Given two programming languages $\mathcal{L}_s$ and $\mathcal{L}_t$, each being a set of instances (p_k, T_k), where p_k is a program, and T_k is its corresponding parse tree. We assume that there exists a translation oracle π, which maps instances in $\mathcal{L}_s$ to instances in $\mathcal{L}_t$. Given a dataset of instance pairs (i_s, i_t) such that $i_s \in \mathcal{L}_s, i_t \in \mathcal{L}_t$ and $\pi(i_s) = i_t$, our problem is to learn a function F that maps each $i_s \in \mathcal{L}_s$ into $i_t = \pi(i_s)$.*

Note that this problem definition does not restrict on whether F knows only the program p_k or only the parse tree T_k. Thus it can leverage both information to finish the translation task. Also, in the

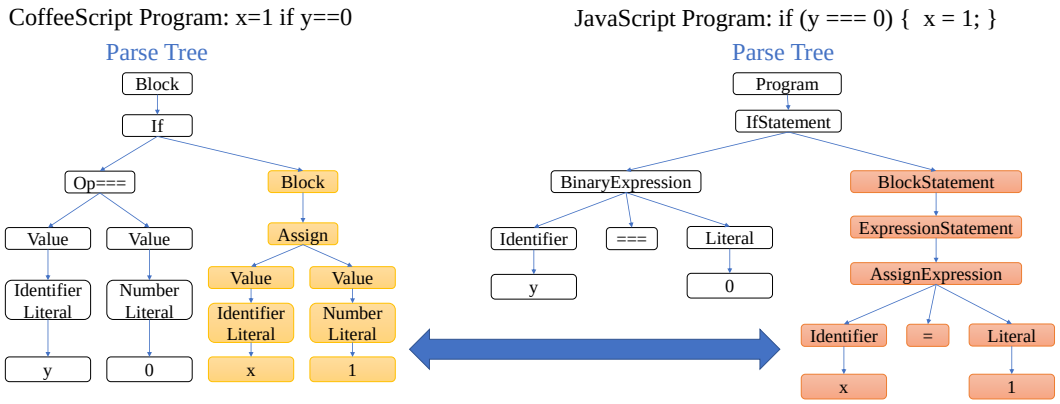

Figure 1: Translating a CoffeeScript program into JavaScript. The sub-component in the CoffeeScript program and its corresponding translation in JavaScript are highlighted.

target language, since a program and its parse tree uniquely matches each other, F only needs to predict one of them to be correct.

In this definition, $\mathcal{L}_s$ and $\mathcal{L}_t$ are the source and target languages respectively, and we use the oracle π to model the goal of the translation. Notice that different tasks can be modeled with different choices of π. For example, besides program translation between two languages, we can also model the compiler optimization by setting $\mathcal{L}_s = \mathcal{L}_t$ and choosing π that maps an un-optimized program into an optimized one.

In this work, we focus on the problem setting that we can get sufficient aligned input-output examples for training. When such an alignment is lacking, the program translation problem can be harder. Several techniques for NMT have been proposed to handle this issue, such as dual learning (He et al., 2016), which have the potential to be extended for the program translation task. We leave these more challenging problem setups as future work.

3 Tree-to-tree Neural Network

In this section, we present our design of the tree-to-tree neural network. We first motivate the design, and then present the details.

3.1 Program translation as a tree-to-tree translation problem

Figure 1 presents an example of translation from CoffeeScript (CoffeeScript, 2009-2017) to JavaScript. We observe several interesting properties of the program translation problem. First, the translation can be modular. The figure highlights a sub-component in the source tree corresponding to `x=1` and its translation in the target tree corresponding to `x=1;`. This correspondence is regardless of other parts of the program. Consider when the program grows longer and this statement may repetitively occur multiple times, it may be hard for a sequence-to-sequence model to capture the correspondence based on only token sequences without structural information.

Second, such a correspondence makes it a natural solution to locate the referenced sub-tree in the source when expanding a non-terminal in the target into a sub-tree. Inspired by this intuition, we design the tree-to-tree neural network with attention, so that when decoding a non-terminal into a (sub-)tree, the model can employ an attention mechanism to locate the referenced source sub-tree. The soft-attention mechanism makes the model differentiable, so that it can be trained end-to-end.

Third, although the corresponding sub-trees are analogues to each other, they differ in two aspects. On the one hand, the non-terminal nodes are different. For example, the non-terminal `IdentifierLiteral` in CoffeeScript is called `Identifier` in JavaScript. Some non-terminals, such as `Value`, may not even have a correspondence. On the other hand, the topology structures of the corresponding sub-trees can be different. For example, the `Assign` node in the

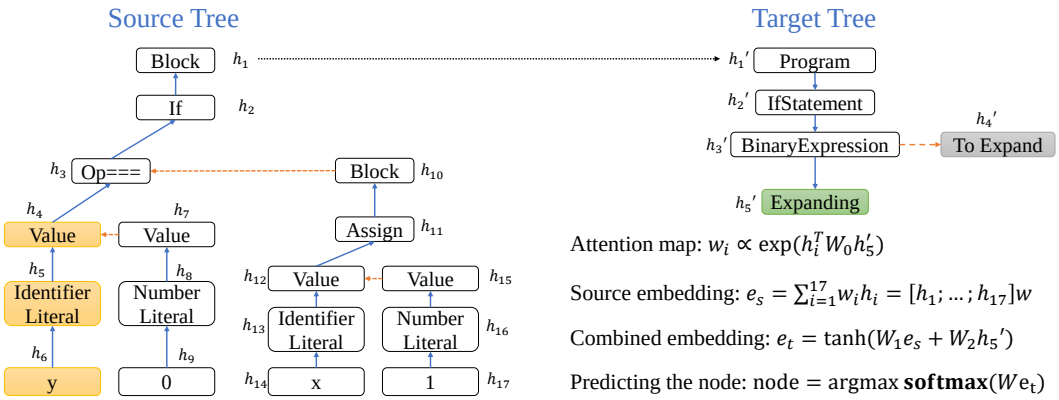

Figure 2: Tree-to-tree workflow: The arrows indicate the computation flow of the encoder-decoder architecture. Blue solid arrows indicate the flow from/to the left child, while orange dashed arrows are for the right child. The black dotted arrow from the source tree root to the target tree root indicates that the LSTM state is copied. The green box denotes the expanding node, and the grey one denotes the node to be expanded in a queue. The sub-tree of the source tree corresponding to the expanding node is highlighted in yellow. The right corner lists the formulas to predict the token for the expanding node.

source tree has two children, while its correspondence in the target tree has three: one additional = is inserted in the middle. These differences can become a challenge to build a program translation model.

3.2 Tree-to-tree Neural Network

A tree-to-tree neural network follows an encoder-decoder architecture to encode the source tree into an embedding, and decode the embedding into the target tree. To capture the intuition of the modular translation process, the decoder employs an (soft) attention model to locate the corresponding source sub-tree when expanding the non-terminal. We illustrate the workflow of a tree-to-tree model in Figure 2. In the following, we present each different component of the model.

Converting a tree into a binary one. Note that the input and output trees may contain multiple branches. Although we can design tree-encoders and tree-decoders to handle trees with arbitrary number of branches, we observe that encoder and decoder for binary trees can be more effective. Thus, the first step is to convert both the source tree and the target tree into a binary tree. To this aim, we employ the Left-child/Next-sibling encoding for this conversion.

Binary Tree Encoder. The encoder part employs a Tree-LSTM (Tai et al. (2015)) to compute embeddings for both the entire tree and each sub-tree. In particular, consider a node N that is attached with a token t_s in its one-hot encoding representation, and it has two children N_L and N_R, which are its left child and right child respectively. The encoder recursively computes the embedding for the N from the bottom up.

Assume that the left child and the right child maintain the LSTM state (h_L, c_L) and (h_R, c_R) respectively. Then the LSTM state (h, c) of N is computed using

$$\begin{aligned}
x &= At_s \\
i &= \sigma(W^{(i)}x + U_L^{(i)}h_L + U_R^{(i)}h_R + b^{(i)}) \\
o &= \sigma(W^{(o)}x + U_L^{(o)}h_L + U_R^{(o)}h_R + b^{(o)}) \\
u &= \mathbf{tanh}(W^{(u)}x + U_L^{(u)}h_L + U_R^{(u)}h_R + b^{(u)}) \\
f_L &= \sigma(W_L^{(f)}x + U_L^{(f)}h_L + b_L^{(f)}) \\
f_R &= \sigma(W_R^{(f)}x + U_R^{(f)}h_R + b_R^{(f)}) \\
c &= i \odot u + f_L \odot c_L + f_R \odot c_R \\
h &= o \odot \mathbf{tanh}(c)
\end{aligned} \tag{1}$$

Here, A is a trainable word embedding matrix of size $d \times V_s$, where d is the embedding dimension and V_s is the vocabulary size of the inputs. All different Ws and Us are different trainable matrix of size $d \times d$, and all different bs are different trainable d-dimensional vectors.

Note that a node may lack one or both of its children. In this case, the encoder sets the LSTM state of the missing child to be zero.

Binary Tree Decoder. The decoder generates the target tree from one single root node. The decoder first copies the LSTM state (h, c) of the root of the source tree, and attaches it to the root node of the target tree. Then the decoder maintains a queue of all nodes to be expanded, and recursively expands each of them. In each iteration, the decoder pops one node from the queue, and expands it. In the following, we call the node being expanded *the expanding node*.

First, the decoder will predict the token of expanding node. To this end, the decoder computes the embedding e_t of the expanding node N, and then feeds it into a softmax regression network for prediction:

$$\mathrm{t_t} = \mathbf{argmax\ softmax}(We_t) \tag{2}$$

Here, W is a trainable matrix of size $V_t \times d$, where V_t is the vocabulary size of the outputs and d is the embedding dimension. Note that e_t is computed using the attention mechanism, which we will explain later.

Each token t_t is one of a non-terminal, a terminal, or a special $\langle \mathrm{EOS} \rangle$ token. If $t_t = \langle \mathrm{EOS} \rangle$, then the decoder finishes expanding this node.

If t_t is a terminal, the decoder does not generate a left child for it. Otherwise, if t_t is a non-terminal, then the decoder generates one new node as the left child of the expanding one. Assuming (h, c) is the LSTM state of the expanding node, then the LSTM state (h', c') of its left child is computed as:

$$(h', c') = \mathrm{LSTM}_L((h, c), Bt_t) \tag{3}$$

Here, B is a trainable word embedding matrix of size $d \times V_t$. This new left child is pushed into the queue of all nodes to be expanded.

If $t_t \neq \langle \mathrm{EOS} \rangle$, then the decoder always generates a right child for the expanding node. The LSTM state (h'', c'') of the right child is computed as:

$$(h'', c'') = \mathrm{LSTM}_R((h, c), Bt_t) \tag{4}$$

Note that here the left child and right child use two different sets of parameters for LSTM_L and LSTM_R respectively. The new right child is also pushed into the queue of all nodes to be expanded.

The entire target tree generation process terminates until the queue is empty.

Attention mechanism to locate the source sub-tree. Now we consider how to compute e_t. One straightforward approach is to compute e_t as h, which is the hidden state attached to the expanding node. However, in doing so, the embedding will soon forget the information about the source tree when generating deep nodes in the target tree, and thus the model yields very poor performance.

To better facilitate the information of the source tree, our tree-to-tree model employs the attention mechanism to locate the source sub-tree corresponding to the sub-tree rooted at the expanding node. In fact, we compute the following probability:

$$P(N_s \text{ is the source sub-tree corresponding to } N_t | N_t)$$

where N_t is the expanding node. We denote this probability as $P(N_s|N_t)$, and we compute it as

$$P(N_s|N_t) \propto \mathbf{exp}(h_s^T W_0 h_t) \tag{5}$$

where W_0 is a trainable matrix of size $d \times d$. To leverage the information from the source tree, the decoder can sample a source node N_s following $P(N_s|N_t)$ or simply predict the most-likely source node $N_s = \mathbf{argmax}_{N_s} P(N_s|N_t)$, and then get its hidden h_s as e_s. This embedding can then be combined with h, the hidden state of the expanding node, to compute e_t as follows:

$$e_t = \mathbf{tanh}(W_1 e_s + W_2 h) \tag{6}$$

where W_1 and W_2 are two trainable matrices of size $d \times d$.

However, this approach suffers a big issue that the entire network is no longer end-to-end differentiable unless explicit supervision is given to train $P(N_s|N_t)$. Unfortunately, this supervision is unavailable. One can rely on policy gradient such as REINFORCE to mitigate the issue, but doing so is expensive and may not be effective.

We solve this problem by changing the above hard-attention scheme to a soft-attention mechanism. Note that in Equation (6), we only need a continuous embedding vector e_s rather than the discrete choice of N_s. Therefore, we can approximate e_s as the expectation of the hidden state value across all N_s conditioning on N_t. In fact, we have

$$e_s = \mathrm{E}[h_{N_s}|N_t] = \sum_{N_s} h_{N_s} \cdot P(N_s|N_t) \tag{7}$$

Note that this final formula (7) coincides with the standard attention formalism. Combining all equations from (1) to (7), the entire tree-to-tree neural network is fully differentiable and can be trained end-to-end.

4 EVALUATION

In this section, we evaluate our tree-to-tree neural network with several baseline approaches on the program translation task. To do so, we first develop two benchmark datasets in Section 4.1 for evaluating different aspects. Then we evaluate our tree-to-tree model with two baseline approaches, which are a sequence-to-sequence model (Bahdanau et al., 2015) and a sequence-to-tree model (Dong & Lapata, 2016). In the following, we start with presenting the details of the benchmark datasets and model details, and then present the results.

4.1 DATASETS

To evaluate different approaches for the program translation problem, we first develop a synthetic translation task, and then leverage a full-fledged program translator from CoffeeScript to JavaScript to develop a more realistic task.

For the synthetic task, we design the source language to be imperative and the target one to be functional. Such a design makes the source and target languages use different programming paradigms, so that the translation can be challenging. Below is an example of the translation:

```
Source program                Target program
for i=1; i<10; i+1 do         letrec f i =
    if x>1 then                   if i<10 then
        y=1                           let _ = if x>1 then let y=1 in () else let y=2 in ()
    else                              in f i+1
        y=2                       else ()
endfor                        in f 1
```

	Seq2seq	Seq2tree	Tree2tree
Batch size	100	20	200
Number of RNN layers	3	1	1
Encoder RNN cell	LSTM	LSTM	Tree LSTM
Decoder RNN cell	LSTM		
Initial learning rate	0.005		
Learning rate decay schedule	Decay the learning rate by a factor of $0.8\times$ when the validation loss does not decrease for 500 mini-batches		
Hidden state size	256		
Embedding size	256		
Dropout rate	0.5		
Gradient clip threshold	5.0		
Weights initialization	Uniformly random from [-0.1, 0.1]		

Table 1: Hyper-parameters chosen for each model

This example demonstrates that a for-loop is translated into a recursive function. We manually implement a translator, which is used to acquire the ground truth. The formal grammars of the two languages can be found in Appendix A.1, and the implementation of the translator between the two languages can be found in Appendix B.

For the CoffeeScript tasks, the programming paradigm is identical. CoffeeScript employs a Python-style succinct syntax, while JavaScript employs a C-style verbose syntax. To control the program length of the training and test data, we develop a pCFG-based program generator and a subset of the core CoffeeScript grammar. We also limit the set of variables to restrict the vocabulary size. We rely on the CoffeeScript compiler to generate the corresponding JavaScript ground truth. The grammar used to generate the programs in our experiments can be found in Appendix A.2.

For both tasks, we are interested in the following two metrics:

- **Token accuracy.** Given a source program, we run the evaluated model to translate it into a target program, and calculate the percentage of the tokens that can match the ground truth.
- **Program accuracy.** Given a set of source programs, we run the evaluated model to translate them into target programs, and calculate the percentage of the predicted programs that can entirely match the ground truth.

While the program accuracy is more meaningful for the program translation task, token accuracy can provide more insight on different models' performance.

Meanwhile, for both tasks, we randomly generate 100,000 pairs of source and target programs for training, 10,000 pairs as the development set, and 10,000 pairs for testing.

4.2 Model details

We evaluate our tree-to-tree model against a sequence-to-sequence model (Bahdanau et al., 2015; Vinyals et al., 2015) and a sequence-to-tree model (Dong & Lapata, 2016). Note that for a sequence-to-sequence model, there can be four variants to handle different input-output formats. For example, given a program, we can simply tokenize them into a sequence of tokens. We call this format as *raw program*, denoted as P. We can also use the parser to parse the program into a parse tree, and then serialize the parse tree as a sequence of tokens. Our serialization of a tree follows its depth-first traversal order, which is the same as (Vinyals et al., 2015). We call this format as *parse tree*, denoted as T. For both input and output formats, we can choose either P or T. Similarly, for a sequence-to-tree model, we have two variants based on its input format being either P or T. Note that the sequence-to-tree model generates a tree as output, and thus requires its output format to be T (unserialized). For the same reason, our tree-to-tree only has one form. Therefore, we have 7 different models in our evaluation.

The hyper-parameters used in different models are summarized in Table 1.

Token accuracy

	Tree2tree		Seq2seq				Seq2tree	
	T→T	T→T (-Attn)	P→P	P→T	T→P	T→T	P→T	T→T
Len=20	**99.95%**	55.60%	99.75%	99.59%	99.90%	99.73%	99.70%	99.51%
Len=50	**96.68%**	34.48%	68.31%	45.28%	67.37%	35.01%	89.96%	90.32%

Program accuracy

	Tree2tree		Seq2seq				Seq2tree	
	T→T	T→T (-Attn)	P→P	P→T	T→P	T→T	P→T	T→T
Len=20	**98.61%**	0%	97.92%	97.35%	98.38%	98.18%	96.14%	98.01%
Len=50	**57.42%**	0%	12.19%	0%	9.19%	0%	55.02%	55.06%

Table 2: Token accuracy and program accuracy of different approaches for the synthetic task. For our tree2tree model, "(-Attn)" means the tree2tree model without attention mechanism.

4.3 RESULTS ON SYNTHETIC TASKS

We evaluate our tree-to-tree models with baseline approaches on the synthetic tasks. We create two datasets for evaluation, where one dataset contains programs of an average length 20, while in the other dataset, the average length of the programs is 50. These lengths correspond to the number of tokens in the source programs.

The experimental results are presented in Table 2. We can observe that our tree2tree model can outperform all baseline models on both tasks and using both metrics. Especially, on the dataset with longer programs, the token accuracy significantly outperforms all seq2seq models by a margin of around 30 points. Its margin over a seq2tree model can also reach more than 6 points. These results demonstrate that tree2tree model is more capable of learning the program translation relationship.

We also evaluate the effectiveness of attention mechanism on our tree2tree model. We observe that the program accuracy goes down to 0%, and the token accuracy also decreases dramatically. This underlines the importance of attention mechanism for our tree2tree model.

We observe that increasing the length of the programs indeed makes the problem harder for all models: the program accuracy dramatically shrinks. However, we observe that the decrease of the tree2tree model on the token accuracy is only around 3 points, and much smaller than other models'. This shows that the tree2tree model has more potential to handle longer inputs.

We observe that seq2seq performs better than seq2tree on shorter programs, but worse on longer ones. We attribute this phenomenon to the fact that for shorter programs, predicting correct structure does not bring many benefits to improve program accuracy; instead, it may hurt the performance, since when generating the ASTs, more tokens are required to be predicted than generating the programs themselves. However, on longer programs, the structure is more crucial to make sure a program is structurally correct, and thus a seq2tree model starts outperforming a seq2seq one.

In addition, we observe that different input formats affect both seq2seq and seq2tree models. For a seq2seq model, outputting target programs makes it more effective than outputting their parse trees. We attribute the reason to that a seq2seq decoder needs to predict more tokens when using the parse tree as its output format, and thus is more likely to make mistakes. However, the input format does not seem to affect too much in these tasks on the token accuracy. This observation is consistent for the seq2tree model as well: we observe that the differences on the choice of input format do not make too many differences on its prediction performance.

4.4 RESULTS ON COFFEESCRIPT TASKS

We now move on to the CoffeeScript dataset. Besides the program length, we want to understand whether the vocabulary size will affect model's performance. Therefore, we created two subset of the CoffeeScript grammar: Coffee1 uses only two variables (i.e., `x,y`) and two literals (i.e., `0,1`), and Coffee2 can choose from a much larger set of variable names (i.e., all alphabetical characters) and literals (i.e., all 1-digit numerical literals). We also generate two datasets for each grammar,

Token accuracy

	Tree2tree		Seq2seq				Seq2tree	
	T→T	T→T (-Attn)	P→P	P→T	T→P	T→T	P→T	T→T
Coffee1-(Len=10)	**99.97%**	56.21%	93.51%	92.30%	95.46%	95.05%	93.29%	95.94%
Coffee2-(Len=10)	**99.98%**	47.54%	99.08%	87.51%	99.11%	96.14%	98.31%	98.09%
Coffee1-(Len=20)	**98.16%**	32.99%	85.84%	25.65%	19.13%	36.18%	88.32%	86.39%
Coffee2-(Len=20)	**99.27%**	31.80%	80.22%	63.49%	87.27%	79.85%	91.19%	94.10%

Program accuracy

	Tree2tree		Seq2seq				Seq2tree	
	T→T	T→T (-Attn)	P→P	P→T	T→P	T→T	P→T	T→T
Coffee1-(Len=10)	**98.80%**	0.09%	90.51%	79.82%	92.73%	89.13%	86.52%	88.50%
Coffee2-(Len=10)	**99.67%**	0%	97.44%	16.26%	98.05%	93.89%	91.97%	88.22%
Coffee1-(Len=20)	**71.52%**	0%	21.04%	0%	0%	0%	68.72%	61.63%
Coffee2-(Len=20)	**78.61%**	0%	19.26%	9.98%	25.35%	42.08%	71.35%	73.66%

Table 3: Token accuracy and program accuracy of different approaches for CoffeeScript to JavaScript translation. For our tree2tree model, "(-Attn)" means the tree2tree model without the attention mechanism.

containing programs of length 10 and 20 respectively. Note that CoffeeScript has a more succinct syntax, and thus a program of 10 or 20 has the same level of complexity of the input programs for our synthetic tasks of length 20 or 50 respectively.

The evaluation results are presented in Table 3. Most of the observations from our synthetic tasks remain: our tree2tree model outperforms all baseline models; without the attention mechanism, the performance of our tree2tree model degrades significantly; all models perform worse on longer inputs; a seq2seq model performs better than a seq2tree model on shorter programs, but worse on longer ones.

However, we observe some additional phenomena. First, using a smaller vocabulary makes both accuracy metrics lower for all models. We attribute this to the lack of variety in the program generator, which makes the generated program may share common sub-clauses or sub-trees. The common sub-tree appears in multiple places in the source program, and they can make the attention mechanism harder to learn to locate the right one when decoding. Since all models employ an attention mechanism, this property has hurt all models. We observe that our tree2tree model can achieve a program accuracy of 98.8% for programs of the length 10, which outperforms the best baseline by 6 points. This shows that our tree2tree model is more robust against this issue.

Second, for a seq2seq model, using the tree input format demonstrates a better performance than using raw programs as the inputs when the programs contain a larger vocabulary. This is different than our observations on the synthetic dataset. We attribute this to the reason that a tree input provides richer information to facilitate the decoder, especially when the attention mechanism is used to locate the attended sub-sequence. Therefore, we argue that employing more structural information can benefit the translation procedure. For the small vocabulary, however, this benefit is offset by the misleading effect discussed above. Thus, the improvement is not significant.

5 Related work

Statistical machine translation. Machine learning models, especially deep neural networks such as sequence-to-sequence based models, have been applied successfully to machine translation tasks between different natural languages (Bahdanau et al., 2015; Cho et al., 2015; Eriguchi et al., 2016; He et al., 2016; Vaswani et al., 2017). Eriguchi et al. (2016) proposes a tree-to-sequence model, which employs a Tree-LSTM (Tai et al., 2015; Zhu et al., 2015) as the encoder, and leverages the parse tree of input sentences to facilitate the translation. However, the parse tree information used in their work may not be accurate. In our work, the parse tree is accurate, so that we can combine both a tree-encoder and a tree-decoder together to achieve a good performance.

Some recent work have applied statistical machine translation techniques to program translation (Allamanis et al., 2017; Karaivanov et al., 2014; Nguyen et al., 2015; 2016). Karaivanov et al. (2014)

and Nguyen et al. (2015) propose to adapt phrase-based statistical machine translation models and leverage grammatical structures of programming languages for code migration. Nguyen et al. (2016) proposes to use Word2Vec vector representation for APIs in libraries used in different programming languages, then learns a transformation matrix for API mapping. On the contrary, our work is the first to employ deep learning techniques for program translation.

Neural networks with tree structures. In recent years, various neural networks with tree structures have been proposed to employ the structured information of the data. (Dong & Lapata, 2016; Rabinovich et al., 2017; Parisotto et al., 2017; Yin & Neubig, 2017; Alvarez-Melis & Jaakkola, 2017; Tai et al., 2015; Zhu et al., 2015; Socher et al., 2011a; Eriguchi et al., 2016; Zhang et al., 2016; Socher et al., 2011b; Kusner et al., 2017). In these work, different tree-structured encoders are proposed for embedding the input data, and different tree-structured decoders are proposed for predicting the output trees. In particular, Socher et al. (2011b) and Kusner et al. (2017) propose tree-structured autoencoders to learn vector representations of trees, and show better performance on tree reconstruction and other tasks such as sentiment analysis. In this work, we are the first to design the tree-to-tree neural network for translation tasks.

Neural networks for parsing. Other work study using neural networks to generate parse trees from input-output examples (Dong & Lapata, 2016; Vinyals et al., 2015; Aharoni & Goldberg, 2017; Rabinovich et al., 2017; Yin & Neubig, 2017; Alvarez-Melis & Jaakkola, 2017; Dyer et al., 2016; Chen et al., 2017; 2016). Dong & Lapata (2016) proposes a seq2tree model that allows the decoder RNN to generate the output tree recursively in a top-down fashion. This approach achieves the state-of-the-art results on many semantic parsing tasks. Some other work incorporate the knowledge about the grammar into the architecture designs (Yin & Neubig, 2017; Rabinovich et al., 2017) to achieve better performance on specific tasks. However, these approaches are harder to generalize to other tasks. Again, none of them is designed for program translation or proposes a tree-to-tree architecture.

Neural networks for code generation. A recent line of research study using neural networks for code generation (Balog et al., 2017; Devlin et al., 2017; Parisotto et al., 2017; Ling et al., 2016; Rabinovich et al., 2017; Yin & Neubig, 2017). In (Ling et al., 2016; Rabinovich et al., 2017; Yin & Neubig, 2017), they study generating code in a DSL from natural language input or programs from another DSL. However, their designs require additional manual efforts to adapt to new DSLs in consideration. In our work, we consider the tree-to-tree model as a generic approach that can be applied to any grammar.

6 Conclusion and future work

In this work, we are the first to consider neural network approaches for the program translation problem, and are the first to propose a tree-to-tree neural network that combines both a tree-RNN encoder and a tree-RNN decoder. Our extensive evaluation demonstrates that our tree-to-tree neural network outperforms several state-of-the-art baselines, including sequence-to-sequence models and sequence-to-tree models. This renders our tree-to-tree model as a promising tool toward tackling the program translation problem.

At the same time, we observe many challenges that existing techniques are not capable to handle. For example, all models' performances degrade when the inputs become longer; it is also unclear how to handle an infinite vocabulary set that may be employed in real applications; further, the training requires a dataset of aligned input-output pairs, which may be lacking in practice. We consider all these problems as important future work in the research agenda toward solving the program translation problem.

References

Roee Aharoni and Yoav Goldberg. Towards string-to-tree neural machine translation. In *ACL*, 2017.

Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *arXiv preprint arXiv:1709.06182*, 2017.

David Alvarez-Melis and Tommi S Jaakkola. Tree-structured decoding with doubly-recurrent neural networks. In *ICLR*, 2017.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *ICLR*, 2015.

Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. In *ICLR*, 2017.

Xinyun Chen, Chang Liu, Eui Chul Shin, Dawn Song, and Mingcheng Chen. Latent attention for if-then program synthesis. In *Advances in Neural Information Processing Systems*, pp. 4574–4582, 2016.

Xinyun Chen, Chang Liu, and Dawn Song. Learning neural programs to parse programs. *arXiv preprint arXiv:1706.01284*, 2017.

Sébastien Jean Kyunghyun Cho, Roland Memisevic, and Yoshua Bengio. On using very large target vocabulary for neural machine translation. In *ACL*, 2015.

CoffeeScript, 2009-2017. URL `http://coffeescript.org/`.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy I/O. *arXiv preprint arXiv:1703.07469*, 2017.

Li Dong and Mirella Lapata. Language to logical form with neural attention. In *ACL*, 2016.

Chris Dyer, Adhiguna Kuncoro, Miguel Ballesteros, and Noah A Smith. Recurrent neural network grammars. In *NAACL*, 2016.

Akiko Eriguchi, Kazuma Hashimoto, and Yoshimasa Tsuruoka. Tree-to-sequence attentional neural machine translation. In *ACL*, 2016.

Di He, Yingce Xia, Tao Qin, Liwei Wang, Nenghai Yu, Tieyan Liu, and Wei-Ying Ma. Dual learning for machine translation. In *Advances in Neural Information Processing Systems*, pp. 820–828, 2016.

Svetoslav Karaivanov, Veselin Raychev, and Martin Vechev. Phrase-based statistical translation of programming languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, pp. 173–184. ACM, 2014.

Andrej Karpathy, Justin Johnson, and Li Fei-Fei. Visualizing and understanding recurrent networks. *arXiv preprint arXiv:1506.02078*, 2015.

Matt J Kusner, Brooks Paige, and José Miguel Hernández-Lobato. Grammar variational autoencoder. *arXiv preprint arXiv:1703.01925*, 2017.

Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiskỳ, Andrew Senior, Fumin Wang, and Phil Blunsom. Latent predictor networks for code generation. In *ACL*, 2016.

Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. Divide-and-conquer approach for multi-phase statistical migration for source code (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pp. 585–596. IEEE, 2015.

Trong Duc Nguyen, Anh Tuan Nguyen, and Tien N Nguyen. Mapping api elements for code migration with vector representations. In *Software Engineering Companion (ICSE-C), IEEE/ACM International Conference on*, pp. 756–758. IEEE, 2016.

Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. In *ICLR*, 2017.

Maxim Rabinovich, Mitchell Stern, and Dan Klein. Abstract syntax networks for code generation and semantic parsing. *arXiv preprint arXiv:1704.07535*, 2017.

Richard Socher, Cliff C Lin, Chris Manning, and Andrew Y Ng. Parsing natural scenes and natural language with recursive neural networks. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pp. 129–136, 2011a.

Richard Socher, Jeffrey Pennington, Eric H Huang, Andrew Y Ng, and Christopher D Manning. Semi-supervised recursive autoencoders for predicting sentiment distributions. In *Proceedings of the conference on empirical methods in natural language processing*, pp. 151–161. Association for Computational Linguistics, 2011b.

Kai Sheng Tai, Richard Socher, and Christopher D Manning. Improved semantic representations from tree-structured long short-term memory networks. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, 2015.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

Oriol Vinyals, Łukasz Kaiser, Terry Koo, Slav Petrov, Ilya Sutskever, and Geoffrey Hinton. Grammar as a foreign language. In *NIPS*, 2015.

Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. In *ACL*, 2017.

Xingxing Zhang, Liang Lu, and Mirella Lapata. Top-down tree long short-term memory networks. In *Proceedings of NAACL-HLT*, pp. 310–320, 2016.

Xiaodan Zhu, Parinaz Sobihani, and Hongyu Guo. Long short-term memory over recursive structures. In *International Conference on Machine Learning*, pp. 1604–1612, 2015.

A GRAMMARS FOR GENERATING THE PROGRAMS IN THE EVALUATION

A.1 GRAMMARS FOR THE SYNTHETIC TASK

Below are the grammar specifications of the source language and the target language used in the synthetic task respectively.

```
      <Expr>  ::=  <Var>
               |   <Const>
               |   <Expr> + <Var>
               |   <Expr> + <Const>
               |   <Expr> − <Var>
               |   <Expr> − <Const>
       <Cmp>  ::=  <Expr> == <Expr>
               |   <Expr> > <Expr>
               |   <Expr> < <Expr>
    <Assign>  ::=  <Var> = <Expr>
        <If>  ::=  if <Cmp> then <statement> else <statement> endif
       <For>  ::=  for <Var> = <Expr> ; <Cmp> ; <Expr> do <Statement> endfor
    <Single>  ::=  <Assign> | <If> | <For>
       <Seq>  ::=  <Single> ; <Single>
               |   <Seq> ; <Single>
 <Statement>  ::=  <Seq> | <Single>
```

```
<Unit>    ::= ()
<Expr>    ::= <Var>
            | <Expr> + <Var>
            | <Expr> − <Var>
<Cmp>     ::= <Expr> == <Expr>
            | <Expr> > <Expr>
            | <Expr> < <Expr>
<Term>    ::= <LetTerm> | <Expr> | <Unit> | <IfTerm>
<LetTerm> ::= let <Var> = <Term> in <Term>
            | letrec <Var> <Var> = <Term> in <Term>
<IfTerm>  ::= if <Cmp> then <Term> else <Term>
```

A.2 Grammar for the CoffeeScript task

Below is the grammar for the CoffeeScript task.

```
<Expr>          ::= <Var>
                  | <Const>
                  | <Expr> + <Var>
                  | <Expr> + <Const>
                  | <Expr> * <Var>
                  | <Expr> * <Const>
                  | <Expr> == <Var>
                  | <Expr> == <Const>
<Simple>        ::= <Var> = <Expr>
                  | <Expr>
<IfShort>       ::= <Simple> if <Expr>
                  | <IfShort> if <Expr>
<WhileShort>    ::= <Simple> while <Expr>
                  | <WhileShort> while <Expr>
<ShortStatement> ::= <Simple> | <IfShort> | <WhileShort>
<Statement>     ::= <ShortStatement>
                  | if <Expr> <br> <indent+> <Block> <indent->
                  | while <Expr> <br> <indent+> <Block> <indent->
                  | if <Expr> <br> <indent+> <Block> <indent-> <br>
                    else <br> <indent+> <Block> <indent->
                  | if <Expr> then <ShortStatement> else <ShortStatement>
<Block>         ::= <Statement>
                  | <Block> <br> <Statement>
```

In the above grammar, <br> denotes the newline character.

B Python implementation of the translator for the synthetic task

```python
def translate_from_for(self, ast):
    if type(ast) == type([]):
        if ast[0] == '<SEQ>':
            t1 = self.translate_from_for(ast[1])
            t2 = self.translate_from_for(ast[2])
            if t1[0] == '<LET>' and t1[-1] == '<UNIT>':
                t1[-1] = t2
                return t1
            else:
                return ['<LET>', 'blank', t1, t2]
        elif ast[0] == '<IF>':
            cmp = ast[1]
```

```
            t1 = self.translate_from_for(ast[2])
            t2 = self.translate_from_for(ast[3])
            return ['<IF>', cmp, t1, t2]
        elif ast[0] == '<FOR>':
            var = self.translate_from_for(ast[1])
            init = self.translate_from_for(ast[2])
            cmp = self.translate_from_for(ast[3])
            inc = self.translate_from_for(ast[4])
            body = self.translate_from_for(ast[5])
            tb = ['<LET>', 'blank', body, ['<APP>', 'func', inc]]
            func_body = ['<IF>', cmp, tb, '<UNIT>']
            translate = ['<LETREC>', 'func', var, func_body, ['<APP>', '
    func', inc]]
            return translate
        elif ast[0] == '<ASSIGN>':
            return ['<LET>', ast[1], ast[2], '<UNIT>']
        elif ast[0] == '<Expr>':
            return ast
        elif ast[0] == '<Op+>':
            return ast
        elif ast[0] == '<Op->':
            return ast
        elif ast[0] == '<CMP>':
            return ast
    else:
        return ast
```