Attempts on using syntax trees to improve programming language translation quality by machine learning

Dai Feng, Shigeru Chiba

While the number of programming languages is increasing, scenarios of developing programs with same functionality in different programming languages are becoming frequent. Although translation between natural languages has been extensively studied, translation between programming languages has not been well explored yet. In this paper, we present a method to improve translation quality between programming languages by adopting syntax tree information to represent source code. Compared with existing methods, our method pays more attention to structural information in syntax trees, and utilizes the structural information in code representation. We show that our method could improve the translation quality between Java code and Python code in a certain degree. We compare our results to the state-of-the-art results in the field of cross programming language translation, and show that we might be able to achieve better accuracy in some metrics.

1 Introduction

According to [3], the number of programming languages has been growing and the popularity of each programming language also keeps changing, indicating that programmers nowadays have more choices and switch languages more often. In fact, there are many cases in which programmers have to reconstruct a project with another programming language, or implement same function in multiple languages. This is not only time-consuming, but also costly in terms of money.

Therefore, an automatic translator for source code is desirable. Researchers have been looking into this topic a lot, but no general translator is developed until machine learning demonstrates its power in the field of natural language translation. Some researchers have already tried to use machine learning techniques for programming language translation. Work by [8] gives it a try, but we believe there is space for improvement.

In this paper, we present an approach for code representation. This approach utilizes abstract syntax trees for structural information. We use our approach and existing approach to represent source code, and do experiments on programming language translation task. We present the comparison between experiment results by two approaches, and make a discussion about underlying indications.

2 Translation between Programming Languages

Program translation, or code translation, refers to converting a code snippet written in a high-level programming language, such as Java and Python, to another high-level programming language. It is useful and necessary in many scenarios.

Aging software has been increasingly problematic

Dai Feng, Shigeru Chiba, Graduate School of Information Science and Technology, The University of Tokyo.

in recent years. Maintenance for old software, especially that written in old programming languages, is often costly and time-consuming. During the beginning period of Covid-19 pandemic, the Department of Labor of the US government was in urgent demand of programmers for COBOL, which is a programming language developed in the 1950s, because the system was overloaded due to the increasing number of claims of unemployment. This is not an easy job as old programmers have retired while young programmers don't use COBOL anymore. Similarly, the Commonwealth Bank of Australia spent around $750 million and 5 years of work to convert its platform from COBOL to Java. In such scenarios, an automatic tool for code conversion is desirable.

Another reason for the necessity lies in the diversity of programming languages. Usually, programming languages are chosen adapted to the native platforms. Android projects usually require a programming language running on the Java Virtual Machine like Java and Kotlin, while Web projects running in the browser require to be written in JavaScript. Therefore, when developing a project for multiple platforms, writing same logic in different programming languages is necessary, and automatic translation for such logic can save a lot of time.

Many researchers have been looking into this topic. Existing literature regarding this task are mostly rule-based. Rule-based approaches tend to result in translations containing bugs or lacking readability, and can only deal with translation between two specific languages. They cannot translate from an arbitrary programming language to another. Recently, as machine learning has shown its power in many fields, researchers have been considering adopting machine learning in programming language translation. They want to develop a general translator between two arbitrary programming languages without having to write a lot of handcrafted rules.

Among the very few attempts to use machine learning for code translation, the state-of-the-art model is developed by Roziere et al [8]. They adopt unsupervised machine learning techniques from natural language translation, and use the techniques in this task. When representing source code, they follow the similar way to represent natural language text. They tokenize the source code, learn sub-tokens with BPE, and linearize the source code with BPE vocabulary. This is straight forward, but losses the special features of programming languages.

In programming languages, syntax trees are tree representation of source code, and they contain rich information regarding syntactic structure. Therefore, combining syntax tree information in code representation is a potential method to consider special features of programming languages.

Besides, current model requires large amounts of computing resource and data, and is not affordable by everyone. A smaller model that is friendly in resource is desirable.

For the above reasons, we want to do experiments to see whether combining tree information in code representation can contribute to machine learning models, and also whether the model can work when given limited computing resource and data. To the best of our knowledge, no existing literature has examined the idea so far.

3 Model Details

The purpose of this paper is to compare two approaches of code representation, one is our proposal, another is from the paper by Roziere et al., and test the performance on a programming language translation model.

We define our proposal as tree-based code representation, while the approach from Roziere et al.

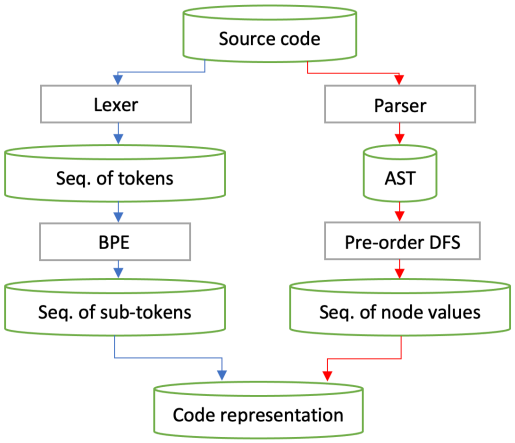

図 1　Comparison of two code representation approaches

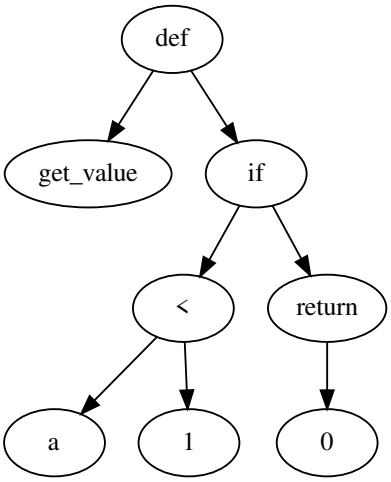

図 2　AST example

is text-based code representation. The overview of two approaches is shown in Fig.1. The blue arrors refer to approach by Roziere et al., and the red arrors refer to our proposal. We describe the details of two code representation approaches in this section.

Besides, we also describe the machine translation model and the training details in this section. We use the same model as in the paper by Roziere et al.

3.1　Tree-based code representation

Tree-based code representation refers to code representation with abstract syntax tree information. Using AST for machine learning input is not our original idea, but we borrow this idea and apply it in the task of programming language translation task.

Tree-based code representation refers to generating abstract syntax trees of source code, and representing source code with such trees. Abstract syntax trees don't preserve all the detailed information such as parentheses, but only programming constructs and relationships between such constructs.

After obtaining these abstract syntax trees, we use pre-order depth-first-search algorithm to get a linearized version of the ASTs.

Consider the following piece of code in Lst.1 and in Fig.2, the tree-based code representation is in Lst.2.

```
def get_value():
    if a < 1:
        return 0
```

Listing 1　Code representation example

```
def, get_value, if, <, a, 1, return, 0
```

Listing 2　Tree-based code representation

3.2　Text-based code representation

Similar as dealing with text in natural languages, we separate source code into individual words for further processing, the separation is also called tokenization.

One traditional way to tokenize source code is by identifiers, built-in key words and punctuation marks. Identifiers include method names and class names, built-in key words include `for`, `if`, `while`,..., and punctuation marks include commas, colons, brackets and so on. The tokens are linearized by

the order of appearance.

Another way is called Byte-Pair Encoding (BPE) [10], which is to learn sub-tokens from the original vocabulary to deal with rare tokens, such as class names and method names. These names are usually concatenated by several words using camel case or snake case. The BPE algorithm iterates the vocabulary to count the most frequent byte pairs, and merge two byte pair tokens into one token. Roziere et al. use BPE algorithm in their paper.

The text-based code representation with traditional tokenization is in Lst.3, and the BPE tokenized code representation is in Lst.4.

```
def, get_value, (, ), :, if, a, <, 1, :,
    return, 0
```

Listing 3 Token-level code representation

```
def, get_@@, value, (, ), :, if, a, <,
    1, :, return, 0
```

Listing 4 Token-level code representation BPE

3.3 Machine translation model

After getting the representation of source code, we will use it as input for machine translation model. It is a typical sequence-to-sequence [11] process, in which we use code representation from previous sections as input sequence, and source code as output sequence.

The basic model for training is Transformer. Transformer [12] is consisted of an encoder and a decoder that both are composed of blocks stacked on top of each other. In each block, there are several modules which are meant for different purposes, mainly multi-head attention modules and feed-forward modules.

The training is consisted of three steps [8], namely cross-lingual masked language model pretraining [5], denoising auto-encoding [13] and back translation [9]. We show the overview of the training pipeline in Fig.3. Briefly, cross-lingual masked

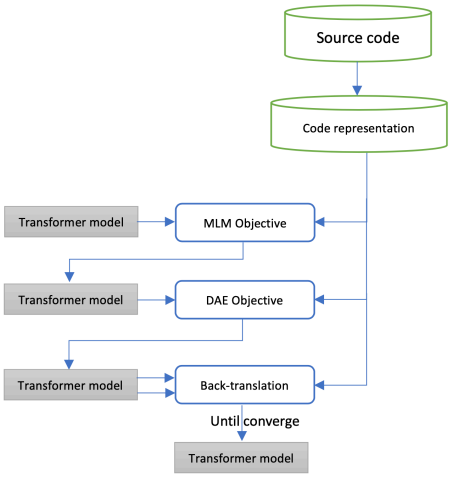

図 3 Model overview

language model pretraining is to map tokens with similar meanings to close positions in vector space. Denoising auto-encoding is to make the model learn how to decode a sequence and be robust to noise. Back translation is to make the model able to generate sequence based on cross-language input representation.

4 Experiments

In this section, we present the designs and results for following experiments. Both experiments are evaluated by translating both ways between Java and Python.

Roziere et al. Text-based representation of source code with BPE tokenization

Our proposal AST-based representation of source code

4.1 Dataset

The training data both in Java and Python is collected from GitHub. We sort the repositories by stars for each language, and download top 100 repositories. In each repository, aside from the files in its main language, there might also be some files in other languages, such as scripts or configure files.

We exclude these files and only keep the files in every repository's main language. After obtaining the source files, we extract the functions and methods in the source code. We use the entire source files for pretraining in order to get meaningful numerical vectors for tokens. However, we only use functions for denoising auto-encoding and back-translation for simplicity and efficiency. After removing files that are too long or contain rare symbols, we finally get 117028 Java source files and 999007 Java methods, and 23518 Python source files and 171538 Python functions.

The evaluation dataset contains 97 parallel implementations of functions both in Java and Python. The data is collected from GitHub and has been manually checked to make sure that parallel functions in a pair implement same logic.

4.2 Configuration

The machine we use is an Nvidia Quadro P6000 GPU card with 24GB GPU memory, an Intel Core i7-6850K CPU with 3.60GHz frequency, 6 cores and 12 threads, and 64GB memory.

For neural network model, we use a hidden embedding size of 256, and train each model for 100 epochs. The training takes an average of 4 days for each model.

As a comparison, Roziere et al. use 32 Nvidia V100 GPU cards, each of them has 32GB GPU memory. They set the hidden embedding dimension to be 1024.

4.3 Results

We will evaluate the models by two kinds of criteria. One is called BLEU score. it considers the similarity between a generated sequence and the ground truth sequence. Another one is about important structures in source code, such as `for` loops and `if` statements.

表 1 BLEU score for translation results

Experiment	Score
Text-based Java to Python	1.28436e-231
Text-based Python to Java	1.62241e-156
Tree-based Java to Python	1.15311e-231
Tree-based Python to Java	1.097212e-231

4.3.1 Translation quality by BLEU score

The BLEU scores of the translation results is shown in Table.1. It can be seen that the absolute values of the scores are very low, and are almost 0. It means the model can barely generate valid translations compared with the ground truth references. We take a look into the results, and find that the model fails to generate consecutive tokens such as 2-grams.

4.3.2 Translation quality by important structures

Loops and conditional statements are important structures in programming. From the translations by two models, we observe that the model with tree-based code representation can generate more structures such as loops or conditional statements than the text-based model. Therefore, we evaluate the translation quality by counting the number of occurrences of such structures.

We define an occurrence of a `for` or a `if` in the ground truth code as a positive case. If the translation successfully generates the `for` loop or the `if` statement, we regard this translation as a true positive, or if the translation does not generate the ground truth `for` or `if`, we regard this translation as a false negative.

In all the cases, there are 80 files containing a `for` loop or a `if` statement. In the text-based model, it successfully generates 5 `if` statements and one `for` loop when translating from Python to Java, and successfully generates only one `for` loop when translating from Java to Python. All the generations are true positive. In the tree-based model,

表 2　Results from text-based model

Type	TP	FP	TN	FN
Java to Python	1	0	17	79
Python to Java	6	0	17	74

表 3　Results from tree-based model

Type	TP	FP	TN	FN
Java to Python	15	6	11	65
Python to Java	13	0	17	67

表 4　Result scores from text-based model

Type	accu	prec	recall	F1-sco
Java to Python	0.186	1	0.0125	0.0247
Python to Java	0.237	1	0.075	0.1395

表 5　Result scores from tree-based model

Type	accu	prec	recall	F1-sco
Java to Python	0.268	0.714	0.1875	0.297
Python to Java	0.309	1	0.1625	0.2796

when translating from Python to Java, we successfully generate 12 `if` statements and one `for` loop, and all the generations are true positive. When translating from Java to Python, we generates 15 `if` statements and 6 `for` loops, however, 6 generations of `if` statements are incorrect.

From Table.2 and Table.3, we can see that tree-based model can generate more important structures than text-based model. However, all the generations by text-based model are correct, while tree-based model may generate some incorrect translations. We calculate the accuracy, the precision and the recall score for each model.

From Table.4 and Table.5, we can see that only the precision score in tree-based model when translating from Java to Python is lower than that of

```
from __future__ import print_function
import hashlib
text = "I am Satoshi Nakamoto"
for nonce in range(20):
    input_data = text + str(nonce)
    hash_data = hashlib.sha256(input_data.encode()).hexdigest()
    print(input_data, '=>', hash_data)
```

図 4　Ground truth code

```
public static final ThreadLocal < byte [ ] > from ( ) {
  checkNotNull ( "I am <unk> <unk>" ) ;
  for ( int nonce = 0 ;
  nonce < 20 ;
  nonce ++ ) {
    byte [ ] bytes = new byte [ nonce + <unk> ( nonce ) ] ;
    byte [ ] bytes = new byte [ nonce . length ] ;
    System . arraycopy ( bytes , bytes , 0 , bytes . length ) ;
    System . arraycopy ( bytes , bytes , 0 , bytes . length ) ;
  }
  return new <unk> ( bytes , bytes ) ;
}
```

図 5　Translation by tree-based model

```
public static String from ( MessageDigest digest ) {
  String text = "I am md5 for nonce " + digest ( range ) ;
  StringBuilder builder = new StringBuilder ( ) ;
  return builder . toString ( ) ;
}
```

図 6　Translation by text-based model

text-based model, and all other scores are higher or equal to text-based model. Therefore, tree-based model does better in generating important structures than text-based model. This could be the evidence to show that embedding AST information in the code representation can contribute to generating more important structures than text-based code representation. To look more deeply into this phenomenon, we provide a case study to see what exactly do the models generate.

4. 4　Case study

We provide a real example of the generated translation by tree-based model and by text-based model, and discuss the indications from this example.

The code we use to translate is shown in Fig.4, and the code generated by tree-based model and text-based model is shown in Fig.5 and in Fig.6.

From the generated translations, we can see that both generations follow the Java code grammar.

However, the generated translations both contain a lot of syntactical mistakes, such as wrong identifiers and false method calls. Such kind of mistakes make the translations impossible to pass the compiler. If we calculate the BLEU score of the generated translations, the absolute value of both scores will be low. However, if we look at the structure of the code, we can see that the translation generated by tree-based model successfully captures the `for`-loop in the ground truth code, which indicates that our translation is slightly more like the ground truth code from a topological viewpoint.

Combined with the results shown in the previous section, we can conclude that tree-based model is better at generating important structures such as loops and conditional statements than text-based model. This is possible because tree-based model embeds abstract syntax tree information in the code representation. To some extend, an abstract syntax tree preserves more structural information than a linear sequence of tokens. Therefore, when feeding the AST-based representation of code into the neural machine translation model, the model might have extra knowledge about the structure of the source code, and this extra knowledge might be able to contribute to final translation quality compared with text-based representation of source code. To prove this, we still need more experiments to find concrete evidence.

5 Related Works

Applying natural language processing techniques to programming language research has been a popular topic for a long time. In the area of code suggestion [1], error detection [2] or code comment generation [4], natural language processing techniques have been of much help.

Translating between programming languages is also not a new research topic. For example, a Python library called 2to3 can help to port Python 2 code to Python 3 [7]. However, these tools are mostly rely on a set of human-defined rules, and creating the rules requires a lot of time. Recently, with the development of machine learning, learning-based translation has been investigated more than before. Nguyen et al. [6] developed a model on a Java-C# corpus using phrase-based statistical machine translation technique. Unfortunately, these attempts are supervised, and rely on parallel data that is not easy to obtain. In 2020, Roziere et al. [8] published a paper discussing the approach to translate source code from one programming language to another in an unsupervised manner. They adopted unsupervised machine translation model to translate between Java, Python and C++ pairwise with high accuracy, proving that unsupervised machine translation could be a useful tool in solving this task.

6 Conclusion

In this paper we present a comparison between text-based code representation and tree-based code representation. Text-based code representation regards source code as plain natural text, while tree-based code representation adopts abstract syntax tree information. The comparison is based on a programming language translation task.

We present the experiment results for tree-based model and text-based model, and show that tree-based model has potential advantages over text-based model in generating important structures of source code. We also provide a case study to discuss the reasons for the results. The results are open for more discussion.

We believe there is space for further improvement, and that part will be our future work.

参 考 文 献

[1] Allamanis, M., Barr, E. T., Bird, C., and Sutton, C.: Learning Natural Coding Conventions, *Pro-*

ceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, New York, NY, USA, Association for Computing Machinery, 2014, pp. 281–293.

[2] Chen, Z., Kommrusch, S. J., Tufano, M., Pouchet, L.-N., Poshyvanyk, D., and Monperrus, M.: SEQUENCER: Sequence-to-Sequence Learning for End-to-End Program Repair, *IEEE Transactions on Software Engineering*, (2021), pp. 1–1.

[3] Frederickson, B.: http://www.benfrederickson.com/ranking-programming-languages-by-github-users/, 2018.

[4] Hu, X., Li, G., Xia, X., Lo, D., and Jin, Z.: Deep Code Comment Generation, *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, New York, NY, USA, Association for Computing Machinery, 2018, pp. 200–210.

[5] Lample, G. and Conneau, A.: Cross-lingual Language Model Pretraining, 2019.

[6] Nguyen, A. T., Nguyen, T. T., and Nguyen, T. N.: Lexical Statistical Machine Translation for Language Migration, *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, New York, NY, USA, Association for Computing Machinery, 2013, pp. 651–654.

[7] Python: https://docs.python.org/2/library/2to3.html.

[8] Roziere, B., Lachaux, M.-A., Chanussot, L., and Lample, G.: Unsupervised translation of programming languages, *Advances in Neural Information Processing Systems*, Vol. 33(2020).

[9] Sennrich, R., Haddow, B., and Birch, A.: Improving Neural Machine Translation Models with Monolingual Data, 2016.

[10] Sennrich, R., Haddow, B., and Birch, A.: Neural Machine Translation of Rare Words with Subword Units, 2016.

[11] Sutskever, I., Vinyals, O., and Le, Q. V.: Sequence to Sequence Learning with Neural Networks, *CoRR*, Vol. abs/1409.3215(2014).

[12] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I.: Attention Is All You Need, 2017.

[13] Vincent, P., Larochelle, H., Bengio, Y., and Manzagol, P.-A.: Extracting and Composing Robust Features with Denoising Autoencoders, *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, New York, NY, USA, Association for Computing Machinery, 2008, pp. 1096 – 1103.