Unsupervised Machine Translation for Legacy Software Migration

I want to develop methods of program language translation that can be used for legacy software migration and software refactoring projects. I intend to leverage my unique familiarity with tools and techniques from deep learning, programming languages, and software engineering towards making important progress on this research direction.

Background, Motivation, and Introduction

Information technology is an important part of the critical infrastructure of the United States: it forms a backbone for military, homeland security, and civilian operations. Much of this infrastructure is rapidly ageing or already outdated. From a security perspective, legacy systems are vulnerable, because hardware and software may be unsupported by vendors, such that updates, security fixes, and patches are unavailable. This seems to have been the case in the 2015 breaches of the US Office of Personal Management (OPM) in which outdated IT systems caused large security vulnerabilities and compromised information on over 20 million individuals [1]. From a strategic and economic perspective, outdated infrastructure impedes digital agility and is expensive to maintain. For example, in 2019 the Air Force operated a dated system for wartime readiness written in COBOL. The Air Force anticipated annual maintenance costs as high as $35 million per year for the system, and subsequently, the Airforce estimated system migration would enable agility through the adoption of new technologies and could also save as much as $34 million a year in maintenance costs [1].

Large legacy migration projects are highly complex and can often cost hundreds of millions of dollars or more. Technologies that can assist in translating legacy programming languages such as COBOL, Fortran, and Ada to popular modern languages such as Java and C++ can be an effective strategy in assisting developers in migrating large code bases; especially given the fact such systems can consist of millions of lines of code, many of which must be tediously translated into new target languages of choice.

Program translation technologies may not only be useful in migrating the legacy systems of yesterday, but they may be useful in the future if the popular languages of today get replaced with the new languages of tomorrow. Lastly, program translation could also be useful for software refactoring projects on modern languages, for example, it could be used to help translate prototype code written in Python to C++ for performance.

In my current and future research, I intend to focus on three complimentary research directions to improve program translation technologies by (I) committing to experiment on legacy programming languages, (II) simplifying the modeling task of generating programs by treating it as a two-stage sketch and identifier renaming problem, and (III) investigating neurosymbolic methods that leverage the strengths of rule-based and neural approaches to program translation.

Challenges in Traditional Approaches of Program Language Translation

Source-to-source compilers, or a *transcompiler*s have a history dating back to automatic migration of assembly from one architecture to another [2] and are an intuitive choice for translation between high level programming languages. Transcompilers do exist for legacy languages such as GNU's COBOL to C compiler; however, many of these compilers are incapable of producing readable or maintainable code. One reason why the translators are unnatural, is due to large gaps in syntax between programming languages which are non-trivial to translate and often require context [3]. As a result, engineering a rules-based transcompiler can take a team of experts as much time as multiple years to build, and the resulting system may still be unable to handle many edge cases.

Neural sequence models are competitive with human ability on many aspects of language translation performance. Unlike rule-based systems, neural machine translation does a remarkable job of preserving naturalness in its outputs. Nevertheless, traditional methods in machine translation rely on

supervised learning and require hundreds of thousands to millions of human-translated language pairs. While parallel corpora exist for many spoken languages, they generally do not exist between pairs of programming languages and building such resources could be prohibitively costly given the high costs of programming talent.

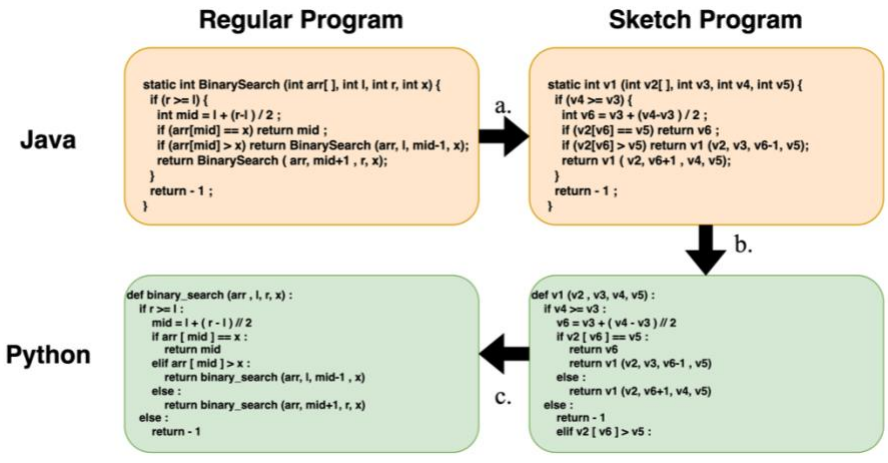
Recently, *unsupervised neural machine translation* [4, 5] has been applied the problem of program translation [6, 7, 8]. The methods do not require any parallel data for training, and they have yielded promising results: the best solution via a neural model with a beam search size of 10 achieves a computational accuracy of 69% on a Java to Python benchmark, whereas a commercial rules-based Java to Python solution only achieves a computational accuracy of 35% [6, 8]. Nevertheless, experiments have only been reported on programming languages for which massive amounts of "monolingual" data are available: the Github BigQuery dataset used for these experiments contains 185 GB of Java, 152 GB of Python, and 93 GB of C++ code.

In contrast, legacy languages are not as available as modern languages: using the same Github BigQuery dataset, I have found the amount of open source Fortran code available is less than 5% the amount of available for Python code. In my research, I will not only build on unsupervised machine translation using modern languages like Java, Python, and C++, but *I will experiment with and evaluate my methods on at least one legacy language such as Fortran or Cobol*. Beyond evaluation, if we are to make progress on program translation, specific improvements in training and modeling will need to made with legacy languages in mind.

Factorizing the Prediction Problem Using Executable Program Sketches

The standard neural program translation task is difficult, because the neural network needs to jointly translate functionally important information such as syntax and semantics, which governs how a program is structured and executed, while simultaneously translating more superficial information like identifier names. In order to prioritize the generation of syntactically and functionally correct programs, I propose breaking the problem into two separate learning tasks. One approach is to factorize the prediction task into first generating am executable program template and then filling in the missing components.

Factorizing a program synthesis problem in this manner is referred to as "sketching," and has had success in both program synthesis [9] as well as semantic parsing [10]. However, to the best of my knowledge, no published work has used this concept for program translation or proposed that a sketch itself could be executable. A first approach to implementing this for program translation is to strip all identifiers from the source program and target program, as shown by the "Sketch Programs" in the figure on the right. Then, using the same figure as an exampe, a Java to Python translation pipeline would cycle through (a.) stripping identifiers from the source program, (b.) translating the Java program sketch into a Python sketch, and then (c.) re-filling in identifier names by conditioning on both the Python sketch and the original Java program. By separating the executable sketch translation step from the identifier prediction task, the task of creating a syntactically and functionally correct translation should be easier for a neural network to learn. Additional experiments would explore which modeling techniques are best for identifier prediction:



some methods could involve the use of pointer networks for copying source identifiers and/or treating identifier prediction as a masked language modeling task for each identifier to be predicted.

This general approach could be particularly useful for legacy languages, first, because the sketch prediction task may prevent learning spurious correlations from identifier names, thereby helping with both data efficiency and generalization. Second, unsupervised machine translation is known to struggle the more languages diverge: given that identifiers in legacy languages may differ stylistically from those in modern languages, removing identifiers may be a helpful preprocessing trick that makes languages more similar.

Generalization Through Neurosymbolic Program Translation

Another large thrust of my research on program language translation would be studying how machine learning can leverage the advantages of rule-based methods without the tedious human engineering and brittleness associated with them. Rule based methods can be desirable, because good rules can systematically generalize well and also help with interpretability. This direction of research will investigate two related problems: (i) how translation rules can be automatically induced, and (ii) how translation rules can be leveraged in a neural machine translation system.

Classic statistical machine translation is known to perform well in low-data regimes. Moreover, phrase-based and tree-based statistical machine translation models respectively learn phrase tables and rule tables [11]; rule tables in particular bear some resemblance to the way transcompiler engineers write translation rules for elements in program abstract-syntax trees (ASTs). In order to overcome the challenge of no parallel corpora, recent research on *unsupervised phrase-based machine translation* initializes the learning process by building a bi-lingual lexicon using distributed representations of words [5]. A first step in extending this to program languages could lie in representing programs as a linearized AST and then learning separate distributed representations for AST non-terminals, production rules, and terminals. Using these representations, a separate translation table for each class of AST elements can be inferred to initialize unsupervised phrase-based machine translation, which in turn could produce better translation rules. This technique imposes an inductive bias that there should be some alignment between non-terminals and production rules between program languages instead of treating all program tokens and constructs equally. By using statistical machine translation, we end up with a translation model which first could outperform data-hungry neural models; but equally or more importantly, we also implicitly learn interpretable translation rule tables which can be used in neural models.

A first step in incorporating induced translation rules could be to model them as a modified copy mechanism: for example, when expanding a non-terminal, instead of picking the specific production rule to follow, the pointer network used for a traditional copy mechanism could pick the aligned source production rule, and look up its most likely equivalent in the target language. This is familiar to an approach that has been recently explored in neural machine translation, where a similar copy mechanism was used at the word level to improve neural machine translation and semantic parsing compositional generalization [12]; however, no work has extended this to syntactic elements like non-terminals and production rules nor applied them in program translation tasks.

Beyond this proposed method, numerous other approaches are open to exploration. One highly ambitious direction may lie in treating the entire program translation task as an unsupervised program synthesis task. That is, instead of learning a neural network to translate between programming languages, a machine-learning powered program synthesizer tries to builds a program that can translate between program languages. Such direction could draw inspiration from recent work on program synthesis powered by wake-sleep library learning [13].

Exploring neurosymbolic approaches is important, because they may be an effective strategy for overcoming the scarcity of data existing for legacy languages. While they may be ambitious and complex, they are worth it considering massive upsides that would come along with developing robust and effective program translation technologies.

References

[1] United States Government Accountability Office. “Agencies Need to Develop Modernization Plans for Critical Legacy Systems,” United States GAO 2019.

[2] MCS-86 Assembly Language Converter Operating Instructions For ISIS-II Users. A30/379/10K TL. Intel Corporation, 1978.

[3] Bysiek, Mateusz, Aleksandr Drozd, and Satoshi Matsuoka. "Migrating legacy Fortran to Python while retaining Fortran-level performance through transpilation and type hints." 2016 6th Workshop on Python for High-Performance and Scientific Computing (PyHPC). IEEE, 2016.

[4] Lample, Guillaume et al. “Unsupervised Machine Translation Using Monolingual Corpora Only”. International Conference on Learning Representations, 2018.

[5] Lample, Guillaume et al. “Phrase-Based & Neural Unsupervised Machine Translation”. Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing. Association for Computational Linguistics, 2018.

[6] Roziere, Baptiste et al. “Unsupervised Translation of Programming Languages”. Advances in Neural Information Processing Systems, 2020.

[7] Roziere, Baptiste, et al. "DOBF: A Deobfuscation Pre-Training Objective for Programming Languages." arXiv preprint arXiv:2102.07492 2021.

[8] Roziere, Baptiste, et al. "Leveraging Automated Unit Tests for Unsupervised Code Translation." arXiv preprint arXiv:2110.06773 2021.

[9] Solar-Lezama, Armando. "The sketching approach to program synthesis." Asian Symposium on Programming Languages and Systems. Springer, Berlin, Heidelberg, 2009.

[10] Dong, Li, en Mirella Lapata. “Coarse-to-Fine Decoding for Neural Semantic Parsing”. Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, 2018.

[11] Koehn, Philipp. Statistical Machine Translation. Cambridge University Press, 2009.

[12] Akyürek, Ekin, and Jacob Andreas. "Lexicon Learning for Few-Shot Neural Sequence Modeling” Annual Meeting of the Association for Computational Linguistics, ACL 2021 (2021).

[13] Ellis, Kevin et al. “DreamCoder: Bootstrapping Inductive Program Synthesis with Wake-Sleep Library Learning”. Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. Association for Computing Machinery, 2021.