Unsupervised Translation of Programming Languages

Baptiste Roziere

PSL UNIVERSITÉ PARIS

Meta AI

THÈSE DE DOCTORAT
DE L'UNIVERSITÉ PSL

Préparée à Université Paris-Dauphine
Dans le cadre d'une collaboration avec Meta AI

Unsupervised Translation of Programming Languages

Soutenue par
Baptiste Roziere
Le 12 juillet 2022

École doctorale n°543
Sciences de la Décision, des Organisations, de la Société et de l'Échange

Spécialité
Informatique

Composition du jury :

Albert COHEN Research Scientist, Google Brain Directeur de recherche, Inria	*Rapporteur* *Président du jury*
François FLEURET Professeur, Université de Genève	*Rapporteur*
Rachel BAWDEN Chargée de recherches, Inria Paris	*Examinatrice*
Ronan COLLOBERT Research Scientist, Apple	*Examinateur*
Elisa FROMONT Professeur, Université de Rennes 1	*Examinatrice*
Tristan CAZENAVE Professeur, Université Paris-Dauphine	*Directeur de thèse*
Olivier TEYTAUD Research Scientist, Meta AI	*Directeur de thèse*

Dauphine | PSL
UNIVERSITÉ PARIS

Remerciements

J'aimerais remercier les membres de mon jury : mes rapporteurs Albert Cohen et François Fleuret ainsi que mes examinateurs Rachel Bawden, Ronan Collobert et Élisa Fromont. Merci d'avoir pris le temps de lire et d'évaluer ce manuscrit, ainsi que ma future soutenance.

Je remercie aussi Tristan, mon directeur de thèse. Merci pour le temps que tu as accordé à mon encadrement, tes conseils avisés, et l'aide que tu m'as apportée.

Merci aussi à mon directeur de thèse à Facebook: Olivier. Merci pour tes conseils, aussi bien professionnels que scientifiques qui m'ont aidé pendant ma thèse et m'aideront encore après. Merci pour la bienveillance dont tu as toujours su faire preuve tout a long de ma thèse. Tu as toujours été attentif et soucieux avant tout de mon bien être et de ma réussite professionnelle.

Je voudrais aussi remercier tout particulièrement Guillaume, qui a été comme un troisième encadrant de thèse pour mes travaux sur le machine learning pour les langages de programmation et m'a appris à m'améliorer au baby-foot lorsque le bureau était encore ouvert. Merci à Marie-Anne, ma principale collaboratrice, et qui faisait partie avec moi des contributeurs principaux pour TransCoder et DOBF. Merci à Marc, avec qui je collabore aujourd'hui. Merci aussi à Gabriel, qui m'a beaucoup appris scientifiquement et sur l'organisation de projets et a été d'une aide précieuse.

Merci aussi à tous mes autres collègues. Le groupe de PhD students, pour toutes les discussions que nous avons eues, que ce soit au bureau, autour d'une table de babyfoot ou au new Jhelum: Léonard, Sablayrolles, Defossez, Pierre, Gautier, Angela, Louis, Rahma, Mathilde, Timothée, Évrard, Jean, Pierre-Alexandre, Laurent, Virginie, Charlotte, Stéphane. Merci à Jérémy, qui m'a aidé à être efficace dès le début de ma thèse. Merci à François pour nos discussions intéressantes et avoir partagé ton expérience. Thanks Hugh for mentoring me, giving me clear and actionable feedback and advice and always cheering me up with his jokes and permanent good mood.

Thanks to all my co-authors: Jie, Mark, Camille, Lowik, Matteo, Alessandro, Mariia, Andry, Nathanaël, Vlad, Hanhe, Fabien, Jialin.

Merci aussi à mes proches. Merci Julia pour ta présence et ton soutien pendant toute ma thèse. Merci à ma sœur Fanny sur qui je sais que je peux toujours compter. Merci aussi bien sûr à mes parents, Paul et Maud, qui m'ont appris à aimer la science et les mathématiques dès ma plus tendre enfance et m'ont permis d'emprunter le chemin qui m'a mené à cette thèse.

Summary

A transcompiler, also known as source-to-source translator, is a system that converts source code from a high-level programming language (such as C++ or Python) to another. Transcompilers are primarily used for interoperability, and to port codebases written in an obsolete or deprecated language (e.g. COBOL, Python 2) to a modern one. They typically rely on handcrafted rewrite rules, applied to the source code abstract syntax tree. Unfortunately, the resulting translations often lack readability, fail to respect the target language conventions, and require manual modifications in order to work properly. The overall translation process is time-consuming and requires expertise in both the source and target languages, making code-translation projects expensive. Although neural models significantly outperform their rule-based counterparts in the context of natural language translation, their applications to transcompilation have been limited due to the scarcity of parallel data in this domain. In this thesis, we propose methods to train effective and fully unsupervised neural transcompilers.

Natural language translators are evaluated with metrics based on token co-occurences between the translation and the reference. e identify that they do not capture the semantics of programming languages. Hence, we build and release a test set composed of 852 parallel functions, along with unit tests to check the semantic correctness of translations. We first leverage objectives designed for natural languages to learn multilingual representations of source code, and train a model to translate, using source code from open source GitHub projects. This model outperforms rule-based methods for translating functions between C++, Java, and Python. Then, we develop an improved pre-training method, which leads the model to learn deeper semantic representations of source code. It results in enhanced performances on several tasks including unsupervised code translation. Finally, we use automated unit tests to automatically create examples of program translations. Training on these examples leads to significant improvements in the performance of our neural transcompilers. Our methods rely exclusively on monolingual source code, require no expertise in the source or target languages, and can easily be generalized to other programming languages.

Résumé

Un transcompilateur est un système qui convertit le code source d'un langage de programmation de haut niveau (tel que C++ ou Python) vers un autre. Les transcompilateurs sont principalement utilisés pour l'interopérabilité et pour transférer des bases de code écrites dans un langage obsolète (par exemple COBOL ou Python 2) vers un langage plus moderne. Ils reposent généralement sur des règles de réécriture manuelles, appliquées à l'arbre de syntaxe abstraite du code source. Malheureusement, les traductions qui en résultent manquent souvent de lisibilité, ne respectent pas les conventions du langage cible et nécessitent des modifications manuelles pour fonctionner correctement. Le processus global de traduction prend du temps et nécessite une expertise à la fois dans les langages source et cible, ce qui rend les projets de traduction de code coûteux. Bien que les modèles neuronaux surpassent considérablement leurs homologues basés sur des règles dans le cadre de la traduction en langues naturelles, leurs applications à la transcompilation ont été limitées en raison de la rareté des données parallèles dans ce domaine. Nous proposons des méthodes pour entraîner des transcompilateurs neuronaux efficaces sans données supervisées.

Les traducteurs de langues naturelles sont évalués avec des métriques basées sur la cooccurrence de tokens entre la traduction et la référence. Nous remarquons que ces métriques ne capturent pas la sémantique des langages de programmation. Nous construisons et publions donc une base de données de tests composée de 852 fonctions parallèles, ainsi que de tests unitaires pour vérifier l'exactitude sémantique des traductions. Nous exploitons d'abord les objectifs conçus pour les langues naturelles afin d'apprendre des représentations multilingues du code source, et entraînons un modèle à traduire, en utilisant seulement le code monolingue de projets open source GitHub. Ce modèle surpasse les méthodes basées sur des règles pour la traduction de fonctions entre C++, Java et Python. Ensuite, nous développons une méthode de pré-entraînement, amenant le modèle à apprendre des représentations sémantiques du code. Cela conduit à des performances améliorées sur plusieurs tâches, y compris la traduction de code non supervisée. Enfin, nous utilisons des tests unitaires automatisés pour créer des exemples de traductions de programmes. Entraîner un modèle sur ces exemples conduit à des améliorations significatives des performances de nos transcompilateurs neuronaux. Nos méthodes reposent exclusivement sur du code source monolingue, ne nécessitent aucune expertise dans les langues source ou cible, et peuvent facilement être généralisées à d'autres langages.

Contents

Chapter 1

Introduction

A transcompiler, transpiler, or source-to-source compiler, is a translator which converts between programming languages that operate at a similar level of abstraction. Transcompilers differ from traditional compilers that translate source code from a high-level to a lower-level programming language (e.g. assembly language) to create an executable. Initially, transcompilers were developed to port source code between different platforms (e.g. convert source code designed for the Intel 8080 processor to make it compatible with the Intel 8086). More recently, new languages have been developed (e.g. CoffeeScript, TypeScript, Dart, Haxe) along with dedicated transcompilers that convert them into a popular or omnipresent language (e.g. JavaScript). These new languages address some shortcomings of the target language by providing new features such as list comprehension (CoffeeScript), object-oriented programming and type checking (TypeScript), while detecting errors and providing optimizations. These languages are designed to be compiled to another high-level programming language with a perfect accuracy (i.e. the compiled language does not require manual adjustments to work properly). In this thesis, we are more interested in the traditional type of transcompilers, where typical use cases are to translate an existing codebase written in an obsolete or deprecated language (e.g. COBOL, Python 2) to a recent one, or to integrate code written in a different language to an existing codebase.

Migrating an existing codebase to a modern or more efficient language like Java or C++ requires expertise in both the source and target languages, and is often costly. For instance, the Commonwealth Bank of Australia spent around $750 million and 5 years of work to convert its platform from COBOL to a more modern language. Using a transcompiler and manually adjusting the output source code may be a faster

and cheaper solution than rewriting the entire codebase from scratch. In natural language, recent advances in neural machine translation have been widely accepted, even among professional translators, who rely more and more on automated machine translation systems. A similar phenomenon could occur in programming language translation in the future.

Translating source code from one Turing-complete language to another is always possible in theory. Unfortunately, building a translator is difficult in practice: different languages can have a different syntax, and rely on different platform APIs and standard-library functions. Currently, the majority of transcompilation tools are rule-based; they essentially tokenize the input source code and convert it into an Abstract Syntax Tree (AST), on which they apply handcrafted rewrite rules. Creating them requires a lot of time, and advanced knowledge in both the source and target languages. Moreover, translating from a dynamically-typed language (e.g. Python) to a statically-typed language (e.g. Java) requires to infer the variable types which is difficult in itself, if not impossible.

The applications of neural machine translation (NMT) to programming languages have long been limited, mainly because of the lack of parallel resources available in this domain. In this thesis, we propose unsupervised machine translation approaches, leveraging a large amount of monolingual source code from GitHub to train a model, to translate between three popular languages: C++, Java and Python. To evaluate our models, we create a test set of 852 parallel functions, along with associated unit tests. Although never provided with parallel data, our models manage to translate functions with a high accuracy, and to properly align functions from the standard libraries across the three languages, outperforming rule-based and commercial baselines by a significant margin. Our approaches require little knowledge in the source or target languages, and can easily be extended to most programming languages with sufficient available data. Although not perfect, our methods could help reduce the amount of work and the level of expertise required to successfully translate a codebase. The main contributions of this thesis are the following:

- We introduce novel approaches to translate functions from a programming language to another, that is purely based on monolingual source code and requires no expertise in either the source or the target languages.

- We show that our methods successfully manage to grasp complex patterns specific to each language, and to translate them to other languages.

- We demonstrate that fully unsupervised methods outperform commercial systems that leverage rule-based methods and advanced programming knowledge.
- Using automatically generated unit tests, we generate tens of thousands of aligned functions, which can substantially improve the performance of unsupervised translation models.
- We build and release a validation and a test set composed of 852 parallel functions in 3 languages, along with unit tests to evaluate the correctness of generated translations.
- Our code and pre-trained models are publicly available [1].

1.1 Thesis Structure

This thesis presents novel unsupervised approaches for source code translation. They brought significant improvements to the state-of-the-art in code translation, which are illustrated in Figure 1.1 and detailed in each chapter.

Chapter 2 In this chapter, we survey related works in sequence modelling, code synthesis, code comprehension and program translation. We also introduce key tools and concepts that influenced our choices and enabled some of our approaches.

Chapter 3: In this chapter, we introduce TransCoder, which learns to translate programming languages using only monolingual data. This model views code as sequences and leverages objective functions designed for Natural Language Processing. It does not use the particularities of source code, except at validation and test time, but still significantly outperforms baselines on source code translation.

Chapter 4: In this chapter, we question the use of the Masked Language Modelling (MLM) and Denoising Auto Encoding (DAE) objectives for pre-training models on source code. These objectives, which were designed for natural languages and mask tokens randomly, often do not force the model to understand the semantics of the code. Hence, we introduce a complementary objective which leverages the particularities of source code: DOBF. It is based on identifier deobfuscation, and leads the model to generate embeddings that represent the semantics of the code. DOBF

[1] https://github.com/facebookresearch/CodeGen

improves the performance of machine learning models on several tasks, including unsupervised code translation.

Chapter 5: The capacity of TransCoder and DOBF to learn multilingual embeddings of source code from monolingual data is essential for their performance for code translation. These embeddings are multilingual due to anchor words such as operators, identifiers, syntax tokens, or keywords that are common to several programming languages. However, it is difficult to learn that the semantics of a given sequence of tokens can differ depending on the programming language (e.g. due to different operator precedence). In this chapter, we present a novel method that leverages automatically generated unit tests to create datasets of aligned functions. It solves the aforementioned issues and substantially improves the performance of the unsupervised source code translation models described in the previous chapters.

Chapter 6: In this final part of the thesis, we review the contributions made in the other chapters and present directions for future research in source code translation and synthesis.

1.2 Publications

Several contributions presented in this thesis were published in peer-reviewed conferences. The content may differ slightly due to small updates of the experimental framework. Additional machine learning works, which are not directly related to the subject of this thesis, are presented briefly in this section.

Machine Learning for Programming Languages.

- TransCoder (Roziere et al., 2020a) use unsupervised methods to translate between programming languages. This work is detailed in chapter 3. Marie-Anne Lachaux is an equal contributor for this work.
- DOBF (Roziere et al., 2021a) provide a new method for pre-training machine learning models for source code, which is presented in chapter 4. Marie-Anne Lachaux is an equal contributor for this work.
- TransCoder-ST (Roziere et al., 2022) use automated unit tests to create datasets of aligned functions, and self-training to improve upon both TransCoder and DOBF. It is detailed in chapter 5.

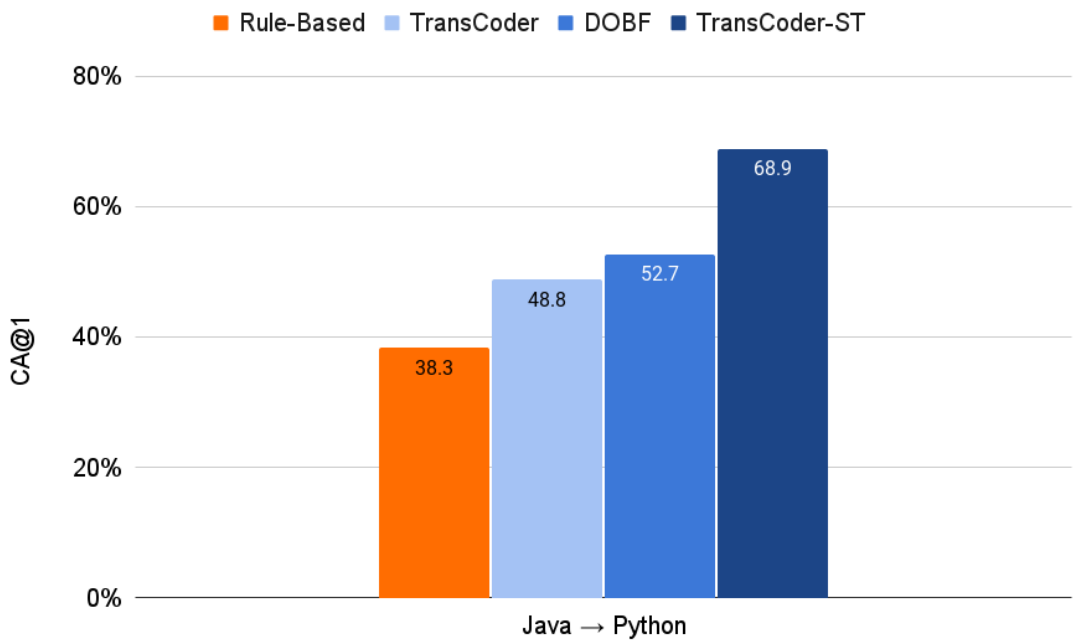

Figure 1.1: **Illustration of the improvements brought by our methods for Java to Python Translation.** In orange, a rule-based baseline called j2py. In shades of blue, the unsupervised methods detailed in this thesis. TransCoder, described in Chapter 3, is our initial method for unsupervised translation of programming languages. DOBF is a novel pre-training objective for programming languages detailed in Chapter 4. In Chapter 5, we present TransCoder-ST, which is trained on aligned data generated using automated unit tests. The y axis is the computational accuracy for a single generation. It measures the percentage of generations that pass a series of unit tests.

Latent space optimization for Generative Adversarial Networks. These works study the use of gradient-based and evolutionary methods to optimize the latent space of Generative Adversarial Networks (GAN).

- EvolGAN (Roziere et al., 2020b) optimize either the technical or artistic quality of images to improve the output of GANs generating many types of pictures.

- Tarsier (Roziere et al., 2021b) use a custom loss and an image quality assessment network to improve the output of GANs for super-resolution.

- In Inspirational adversarial image generation (Rozière et al., 2021), we allow the generation of images similar to input inspirational images or corresponding to human preferences.

Miscellaneous

- Garcelon et al. (2020) study theoretical adversarial attacks on linear contextual bandit algorithms, and validate the feasibility of these attacks on synthetic and real-world datasets. Evrard Garcelon and Laurent Meunier are equal contributors for this work.

Chapter 2

Related Work

2.1 Neural Machine Translation

2.1.1 Transformer architecture

The transformer architecture (Vaswani et al., 2017) leverages the self-attention mechanism to translate or perform other tasks without any recurrent cells. This architecture improves parallelization at training time and has a better capacity to learn long-term dependencies.

Parallelization. While recurrent layers are sequential in nature, transformer layers can compute representations of each token in parallel, making them more efficient on GPUs or dedicated hardware.

Long-term dependencies. Long-term dependencies are notoriously difficult to learn with recurrent neural networks. The length of the path that the forward and backward signals have to traverse to learn such dependencies can grow up to the length of the sequence, which makes them difficult to learn. In practice, these networks are generally trained with truncated backpropagation through time (Sutskever, 2013; Pascanu et al., 2013), which further hinders the learning of long term dependencies. In contrast, in attention layers, there is a path involving a constant number of operations between any two tokens, making learning long-term dependencies easier.

Encoder-decoder architecture. Vaswani et al. (2017) uses an encoder-decoder architecture inspired by those developed for recurrent neural networks (Bahdanau et al., 2015; Cho et al., 2014). In the context of machine translation, the encoder learns

high-level representations of each token in the source sentence, using information from tokens on the left and right. The decoder generates tokens in an auto-regressive manner, using the representation of the entire source sentence outputted by the encoder and the previously generated target tokens.

Encoder-only architectures. Alternative transformer architectures have been proposed. For instance, models in the BERT family (Devlin et al., 2018; Liu et al., 2019; Yang et al., 2019; Sanh et al., 2019) contain only an encoder and are used for natural language understanding. They are generally tested on the GLUE benchmark (Wang et al., 2018a), which contains tasks such as question to answer matching, paraphrase detection, and sentiment analysis. Such models only generate high-level representations of tokens in the source sentence. They are not trained to generate new tokens in an auto-regressive manner, and are generally not used to generate new sentences of arbitrary length in the context of machine translation.

Decoder-only architectures. Models such as GPT (Radford et al., 2018, 2019; Brown et al., 2020) train only a decoder in an auto-regressive manner. This type of architecture is especially suitable for text completion. Using carefully selected prompts, it is also applicable to a wide range of tasks. For instance, for translation, a well trained model can be expected to complete a prompt formatted as "Translate English to French: cheese => " with "fromage", which is the translation of the English word "cheese". Brown et al. (2020) show improved few-shot performance when providing several examples of translations in the prompt.

Scaled dot product attention. Vaswani et al. (2017) defines the scaled dot product attention by adding a scaling factor $\frac{1}{d}$ to dot product attention, with d the dimension of the keys:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d}})V \tag{2.1}$$

With Q, K, and V the matrices containing the queries, keys, and values, which are computed as linear transformations of token representations. In self-attention, the keys, queries, and values are computed from the same token representations. In the encoder, each position can attend to any other position in the previous layer. In the decoder, they use masking to ensure that no element can attend to positions after its own. It guarantees that the information flows only forward and that the

tokens are generated in an auto-regressive manner. Another type of attention is the cross-attention. It is present in the decoder and allows each generated token to attend to any token representation at the last layer of the encoder. It is done by computing the keys and values from the encoded input sequence, and the queries from the output tokens generated by the decoder.

Feed-forward network (FFN). Each transformer layer contains a feed-forward network (FFN), which is applied after the attention mechanism. It consists in two linear fully-connected layers and an activation function in between (generally ReLU (Agarap, 2018) or GELU (Hendrycks and Gimpel, 2016)). This network is applied to each position separately with the same weights.

Multi-head attention. Instead of performing a single attention operation at each layer, transformer models perform such operations with independently learned query, key, and value projection matrices. The outputs of the attention heads are then concatenated and passed through the FFN.

Positional encodings. The attention mechanism defined in Equation 2.1 processes tokens independently of their positions in the sequence. Hence, with only stacked self-attention layers computed on sequences of token embeddings, transformers would not be able to make use of the sequential nature of the input. Vaswani et al. (2017) use positional encodings (Gehring et al., 2017), which are added to the token embeddings and allow the model to learn representations of positions in a sentence. They obtain similar performance with sinusoidal and learned encodings. In our experiments, we use learned positional encodings as we observed that it improved our performance.

Linear attention. The time and space complexities of the self-attention are quadratic with respect to the sequence length. It makes vanilla transformers difficult to scale to large sequences. Several methods have been proposed to reduce the complexity of attention layers and make it linear. Beltagy et al. (2020) replace the full attention with a linear number of windowed, dilated, and global attention patterns. Zaheer et al. (2020) propose a similar approach but replace the dilated attention pattern with a sparse random attention. In both cases, the global attention tokens attend to all tokens and all tokens attend to them. They allow signal to flow between any pair of tokens after two attention layers. Instead, Wang et al. (2020a) demonstrate that the self-attention can be approximated by a low-rank matrix

and propose to project the keys and values matrices to a lower dimensional space. It reduces the complexity of self-attention from quadratic to linear and provides substantial speedups and memory usage improvements empirically.

Encoding tree structures. Programming languages are strictly structured, and are generally designed to be parsed into semantically rich structures such as Abstract Syntax Trees (AST). ASTs contains all the semantic information required to understand the code and compile it further. They represent concepts such as scopes and token types clearly, and are often used to perform static analysis. Several methods have been proposed to leverage the information contained in ASTs. Shiv and Quirk (2019) propose to encode the position of each node by its path from the root. Kim et al. (2021) develop relative tree positional encodings, where the number of up and down moves in the unique path to go from A to B in the AST is used to compute the attention matrix. Chirkova and Troshin (2021) compare the performance of sequential positional encodings to tree positional encodings, and graph neural networks from Hellendoorn et al. (2019) on four tasks. They observe that sequential relative attention (Shaw et al., 2018) performs well in average and that transformers are generally capable of understanding the structure of the code without tree positional encodings. Chen et al. (2021b) propose several alternative representations of code as graphs, and show that leveraging graph representations leads to significant gains for several tasks on source code.

2.1.2 Language modeling

A language model assigns a probability distribution over sequences of tokens. In deep learning, language models are usually trained by evaluating the probability of the next token in a sequence, given the previous tokens. Then, the probability of the sequence can be written as follows:

$$P(w_1 \dots w_m|\theta) = \prod_{1}^{m} P\left(w_i|w_{i-1} \dots w_1, \theta\right) \tag{2.2}$$

with m the size of the sequence, w_i the i-th token and θ the weights of the model. It is common to maximize the estimated probability of real sequences by minimizing the average negative log-likelihood of the sequence.

2.1.3 Unsupervised Machine Translation

The quality of NMT systems depends on the quality of the available parallel data. However, for the majority of languages, parallel resources are rare or nonexistent. Since creating parallel corpora for training is not realistic (creating a small parallel corpus for evaluation is already challenging (Guzmán et al., 2019)), some approaches have investigated the use of monolingual data to improve existing machine translation systems (Gulcehre et al., 2015; He et al., 2016; Sennrich et al., 2015a; Zheng et al., 2017).

Several methods were proposed to train a machine translation system exclusively from monolingual corpora, using either neural models (Lample et al., 2018a; Artetxe et al., 2018b) or statistical models (Lample et al., 2018c; Artetxe et al., 2018a). In Chapter 3, we describe how these methods can be instantiated in the setting of unsupervised transcompilation. More recently, Brown et al. (2020) showed that large language models are able to translate with few-shot prompts, and Han et al. (2021) proposed to bootstrap an unsupervised neural translation system using a pre-trained generative language model.

2.2 Program Synthesis and Translation

2.2.1 Code synthesis from natural language.

Early methods. Program synthesis generally refers to the generation of code from natural language prompts, and has been a longstanding dream of artificial intelligence (Backus et al., 1957; Shaw et al., 1975; Manna and Waldinger, 1971). Hindle et al. (2012) were the first to use a n-gram language model on source code, followed by Nguyen et al. (2013). Raychev et al. (2014) combined statistical models (i.e. n-gram models and recurrent neural networks) and code analysis tools for code completion. Later, recurrent neural networks were shown to outperform n-gram models and several studies trained neural networks on source code at character (Karpathy et al., 2015; Cummins et al., 2017) or token level (Lin et al., 2017; Ling et al., 2016). A common issue with standard seq2seq models, is that the generated functions are not guaranteed to compile, and even to be syntactically correct. To address this issue, several approaches proposed to use additional constraints on the decoder, to ensure that the generated functions respect the syntax of the target language (Alon et al., 2019a,b; Amodio et al., 2017; Rabinovich et al., 2017). Maddison and Tarlow

(2014); Allamanis et al. (2015b) and Yin and Neubig (2017) developed syntactic models generating ASTs. They generate syntactically correct code by construction but are more difficult to train efficiently.

Large language models. Recently, large language models for code have shown impressive capabilities for code synthesis from natural language prompts, such as docstrings and problem statements (Chen et al., 2021a; Austin et al., 2021; Li et al., 2022; Chowdhery et al., 2022). They also showed non-trivial performance for few-shot program translation with well-selected prompts.

Synthesis from docstrings. Docstrings are used to document a specific segment of code (e.g. a function). Clement et al. (2020) trained a model on a large python dataset, in which they separated function signatures, docstrings and bodies. Then, they trained a model to generate any of these elements from some of the others (e.g. function bodies from signatures and docstrings). Chen et al. (2021a) showed that a prompt, containing only the function signature and its docstring, is enough to generate functions that pass all the tests for close to 30% of the elements in their HumanEval dataset. In practice, docstrings vary in quality, and are often not enough to perfectly specify the behavior of complex functions.

Synthesis from competitive programming questions. Competitive programming questions often remove ambiguity by providing input ranges, example input/output pairs, and target time/space complexities. However, writing well-specified problem statements in a natural language is difficult in practice, and coding platforms also allow users to remove ambiguity by comparing their code to the ground truth on custom inputs. Moreover, such specific prompts would be difficult to obtain for much larger projects or non-isolated code snippets. While docstrings aim to give clear indications about the semantics of the code, problem statements often use non-standard wording to obfuscate the meaning and do not describe a particular solution. Hendrycks et al. (2021) and Li et al. (2022) used language models to generate solutions to competitive programming questions.

Synthesis from pseudo-code. Kulal et al. (2019) create a parallel pseudo-code-to-code datasets. They train a model, and use sampling to synthesize compilable source code from pseudo-code.

2.2.2 Program Translation

Program translation can be seen as a type of program synthesis, where the input prompt is also source code. Contrarily to natural languages, source code is unambiguous if the corresponding compiler or interpreter is known. Hence, code translation is generally a better-specified task than code synthesis.

The particularities of some languages allow the creation of very successful rule-based transcompilers for a few language pairs (e.g. Java → Scala, CoffeeScript → JavaScript). However, source-to-source translation between arbitrary Turing-complete languages remains an open problem. Several studies have investigated the use of machine learning for translating between programming languages. For instance, Nguyen et al. (2013) trained a Phrase-Based Statistical Machine Translation (PBSMT) model, Moses (Koehn et al., 2007), on a Java-C# parallel corpus. They created their dataset using the implementations of two open source projects, Lucene and db4o, developed in Java and ported to C#. Similarly, Karaivanov et al. (2014) developed a tool to mine parallel datasets from ported open source projects. Aggarwal et al. (2015) trained Moses on a Python 2 to Python 3 parallel corpus created with 2to3, a Python library [1] developed to port Python 2 code to Python 3. Chen et al. (2018) used the Java-C# dataset of Nguyen et al. (2013) to translate code with tree-to-tree neural networks. They also use a transcompiler to create a parallel dataset CoffeeScript-Javascript. Unfortunately, all these approaches are supervised, and rely either on the existence of open source projects available in multiple languages, or on existing transcompilers, to create parallel data. Moreover, they essentially rely on BLEU score (Papineni et al., 2002) to evaluate their translations (Aggarwal et al., 2015; Miceli-Barone and Sennrich, 2017; Karaivanov et al., 2014; Nguyen et al., 2013), which is not a reliable metric, as a generation can be a valid translation while being very different from the reference. Methods leveraging verified lifting (Kamil et al., 2016), which offer formal guarantees, can significantly speed up some pre-defined code fragments (Ahmad and Cheung, 2016; Ahmad et al., 2019).

2.2.3 Evaluation Metrics

The goal of program synthesis and translation is to help developers to perform specific tasks. However, getting production metrics about the usage of such models can be difficult in practice. Given two models, it is important to be able to compare

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

them offline, without having to expose users to the outputs of experimental models. Similarly to other machine learning methods, machine translation and program synthesis models can be evaluated on held-out test examples, which are not seen at training time. In this section, we briefly present the metrics that are generally used for that purpose.

Perplexity.

Sequence synthesis and translation tasks are often based on language models, which learn a probability distribution over sequences of words. These models can be evaluated by measuring how well they predict the probability of real unseen samples. The perplexity score is defined as the exponential of the average negative log-likelihood (or equivalently, of the cross-entropy) of the sequence, which is computed by the model.

$$\text{perplexity}(W) = 2^{-\frac{1}{N}\log(P(w_1 w_2 ... w_N))} \tag{2.3}$$

Exact match accuracy.

The exact match score, also called perfect match accuracy score, is a simple metric for evaluating code synthesis and code translation methods (Rabinovich et al., 2017; Chen et al., 2018). It computes the percentage of generated programs that are exactly the same as the ground truth. The main drawback of this metric is that it considers programs that are semantically equivalent to the ground truth but differ by one or several tokens as negatives. It is too strict, especially in the context of source code, where there are often many correct and idiomatic ways to implement a function. For instance, identifiers (e.g. variable names, function names) can be chosen arbitrarily, some instructions can be reordered or rewritten without impacting the semantics of the code.

BLEU score.

Machine translation methods for natural languages are generally evaluated using the BLEU score (Papineni et al., 2002; Bahdanau et al., 2015; Wu et al., 2016; Vaswani et al., 2017). In the context of programming languages, early works used the same metric to evaluate their code synthesis or translation outputs (Aggarwal et al., 2015; Miceli-Barone and Sennrich, 2017; Karaivanov et al., 2014; Nguyen et al., 2013). For $K \in \mathbb{N}$, the BLEU-K score computes the n-gram precision score $precision_n$ for each

$n \in [1, \ldots, n]$ as follows:

$$precision_n = \frac{\sum_{C \in Candidates} \sum_{n\text{-gram} \in C} Count_{clip}\,(n\text{-gram, ref})}{\sum_{C' \in Candidates} \sum_{n\text{-gram'} \in C'} Count\,(n\text{-gram', candidate})} \tag{2.4}$$

The count in the numerator, $Count_{clip}$, is clipped to ensure that any n-gram cannot be counted more times than it appears in the reference text. For instance, the unigram precision of a sequence repeating the word `def` n times, compared to a reference containing one python function and a single `def` token, is clipped to $\frac{1}{n}$ instead of 1 with the unclipped version. However, a candidate sequence containing a single `def` token would still have a perfect unigram precision, motivating the need for a brevity penalty (BP) penalizing short sentences:

$$BP = \min\left(1, e^{1-\frac{r}{c}}\right) \tag{2.5}$$

With r and c respectively the lengths of the reference and candidate sentences.

Then, the BLEU-K is defined as BP multiplied with the geometric mean of the n-grams precisions. In practice it is computed with the mathematically equivalent formula:

$$\text{BLEU-K} = BP \cdot \exp\left(\frac{1}{K}\sum_{n=1}^{K} \log(precision_n)\right) \tag{2.6}$$

When K is not given, the BLEU score generally refers to the BLEU-4 scores which is computed using the 1, 2, 3 and 4-gram overlaps between the source and the target. In this thesis, we also use the term BLEU to refer to the BLEU-4 score.

Criticisms of the BLEU score. Despite its pervasiveness in machine translation models evaluation, BLEU has been criticised in the NLP community (Callison-Burch et al., 2006; Kocmi et al., 2021). In the context of programming languages, it does not take the semantic correctness of the generations into account. Two programs with small syntactic discrepancies will have a high BLEU score while they could lead to very different compilation and computation outputs. Conversely, semantically equivalent programs with different implementations can have low BLEU scores. In practice, the BLEU score correlates poorly with the correctness of the generated code (Roziere et al., 2020a; Ren et al., 2020; Austin et al., 2021). Ren et al. (2020) proposed CodeBLEU, which is a weighted mean between the BLEU score, a weighted n-gram match score similar to BLEU, and structural matches based on the AST and

dataflow graphs of the program.

Computational accuracy.

Metrics based on token match fail to capture the semantics of the code and typically correlate poorly with the correctness of the generated function, prompting the use of new metrics checking if the generated solution passes series of test cases. The pass@k (Kulal et al., 2019) or computational accuracy (CA@k)(Roziere et al., 2020a) are defined based on unit tests. More precisely, a generated solution is considered correct if it passes a series of corresponding test cases. Hendrycks et al. (2021); Chen et al. (2021a); Drain et al. (2021) and Austin et al. (2021) adopted the same metric and, in the context of program synthesis, Hendrycks et al. (2021) and Austin et al. (2021) noticed that semantics-based metrics correlate poorly with BLEU score. This metric is presented in detail in Section 3.2.4 in the context of code translation.

2.3 Other Machine Learning Tasks for Programming Languages

This section surveys other tasks in machine learning for programming languages that are especially relevant to this thesis. Allamanis et al. (2018a) and the living literature website[2] provide information on more tasks.

2.3.1 Translating from source code

Other studies have investigated the use of machine translation from source code. For instance, Oda et al. (2015) trained a PBSMT model to generate pseudo-code. To create a training set, they hired programmers to write the pseudo-code of existing Python functions. Miceli-Barone and Sennrich (2017) built a corpus of Python functions with their docstrings from open source GitHub repositories. They showed that a neural machine translation model could be used to map functions to their associated docstrings, and vice versa. Similarly, Hu et al. (2018) proposed a neural approach, DeepCom, to automatically generate code comments for Java methods. Husain et al. (2019) introduced a dataset of aligned functions and docstrings in 6 languages. It was integrated in several tasks of the CodeXGLUE benchmark (Lu et al., 2021), including the code summarization task. Since then, many pre-trained

[2]https://ml4code.github.io/

models were evaluated on translating from source code to comments (Feng et al., 2020; Roziere et al., 2021a; Dong et al., 2019; Wang et al., 2021).

2.3.2 Bug Detection and Repair

Bug detection consists in finding bugs in software without human intervention. It is often associated with the repair task, which consists in automatically finding solutions to bugs (Monperrus, 2018b). Bug detection was traditionally tackled using static analysis tools such as FindBugs (Ayewah et al., 2007), ErrorProne (Aftandilian et al., 2012) and Infer (Calcagno et al., 2015). These tools typically rely on hard-coded rules to detect patterns commonly associated to bugs in the code, AST, or dataflow graph. However, they cannot generalize to new bug patterns, or propose automatic fixes to complex bugs, prompting the development of several machine learning methods for this task (Gupta et al., 2017; Wang et al., 2018b; Tufano et al., 2019; Chen et al., 2019; Allamanis et al., 2018b; Mesbah et al., 2019; Tarlow et al., 2020; Dinella et al., 2020; Yasunaga and Liang, 2020; Tufano et al., 2019; Drain et al., 2021; Jiang et al., 2021; Allamanis et al., 2021). Tufano et al. (2019) mined commits containing bug fixes from GitHub, using simple patterns, and framed the repair problem as a translation from buggy to fixed code. Other methods used either recurrent (Chen et al., 2019), convolutional (Lutellier et al., 2020; Jiang et al., 2021), or graph neural networks working on AST features (Dinella et al., 2020; Tarlow et al., 2020; Chen et al., 2021b) to generate the fix.

Monperrus (2018a) provides a comprehensive and regularly-updated survey on program repair.

2.3.3 Model pre-training.

Masked Language Modeling pre-training. Large pre-trained transformers such as BERT (Devlin et al., 2018) and RoBERTa (Liu et al., 2019) led to significant improvements in most natural language processing tasks. The quality of pre-training mainly comes from the Masked Language Modeling (MLM) objective (i.e. the cloze task Taylor (1953)), which allows the model to make predictions by leveraging left and right contexts, unlike causal language modeling (CLM), where the model predictions are only conditioned on previous words. In Masked Language Modeling (MLM), the model takes a sentence as input and uniformly selects 15% of its tokens. Of the selected tokens, 80% are replaced by a special symbol [MASK], 10% are

left unchanged, and the remaining 10% are replaced by random tokens from the vocabulary. The MLM objective consists in recovering the initial sentence given the corrupted one. Lample and Conneau (2019) noticed that the masked words are often easy to predict, and proposed to sample the 15% masked words according to their frequencies instead of uniformly. This way, rare words are sampled more often, making the pre-training task more difficult for the model, which results in an improved learning signal and a faster training. Sun et al. (2019) also noticed that recovering the tokens masked by MLM is too simple in some contexts (e.g. predicting the two tokens "Harry Potter" is much harder than predicting only "Harry" if you know the next word is "Potter"). To address this issue, they proposed to mask phrases and named entities instead of individual tokens. Joshi et al. (2020) and Song et al. (2019) made a similar observation and proposed to mask random spans of text. They showed that this simple modification improves the performance on many downstream NLP tasks.

Alternative objectives. Other pre-training objectives have been proposed in addition to MLM. For instance, Devlin et al. (2018) also use the next sentence prediction (NSP) objective, a binary classification task that consists in predicting whether two input sentences follow each other in the original corpus. The NSP objective was originally designed to improve the performance on downstream NLP tasks, but recent studies (Lample and Conneau, 2019; Liu et al., 2019) showed that training MLM on a stream of sentences to leverage longer context, and removing the NSP objective, improves the quality of pre-training. To improve the sample-efficiency of MLM (where only 15% of tokens are predicted), Electra (Clark et al., 2020) proposed to replace (and not mask) some tokens with plausible alternatives, and to train a network to detect the tokens that have been replaced. They showed that this new Replaced Token Detection (RTD) objective matches the performance of RoBERTa while using four times less computational resources. Dong et al. (2019) proposed a model that combines multiple pre-training tasks, including bidirectional, but also left-to-right and right-to-left language modeling objectives. Lewis et al. (2020) also proposed different pre-training objectives, for instance to detect whether input sentences have been permuted, or tokens have been deleted or inserted.

Code Generation Pre-training. Pre-training methods developed for natural languages are also effective for programming languages. For instance, Kanade et al. (2020); Feng et al. (2020) rely mostly on the MLM objective to pre-train models

similar to BERT (Devlin et al., 2018) on programming languages. It leads to significant improvement gains on several downstream tasks. Ahmad et al. (2021) propose to train an encoder-decoder model similarly to BART (Lewis et al., 2020) instead, and obtain good results on several tasks in the CodeXGLUE benchmark (Lu et al., 2021).

Other methods propose to leverage the structure of programming languages to pre-train models for source code. Jain et al. (2020) train a model with a contrastive loss, ensuring that the representations are robust to some semantic-preserving transformations. In GraphCodeBERT (Guo et al., 2020), the MLM objective is complemented by an edge-prediction objective, in which the model predicts edges in the data flow graph to make the model understand the structure of the code. DOBF (detailed in Chapter 4) leverages the structure of the programming languages to train a model to deobfuscate identifiers. CodeT5 (Wang et al., 2021) improves it by adding other tasks specific to programming languages, such as identifier tagging, docstring generation, and code generation from docstrings.

Chapter 3

Unsupervised Translation of Programming Languages with Multilingual Pre-Training

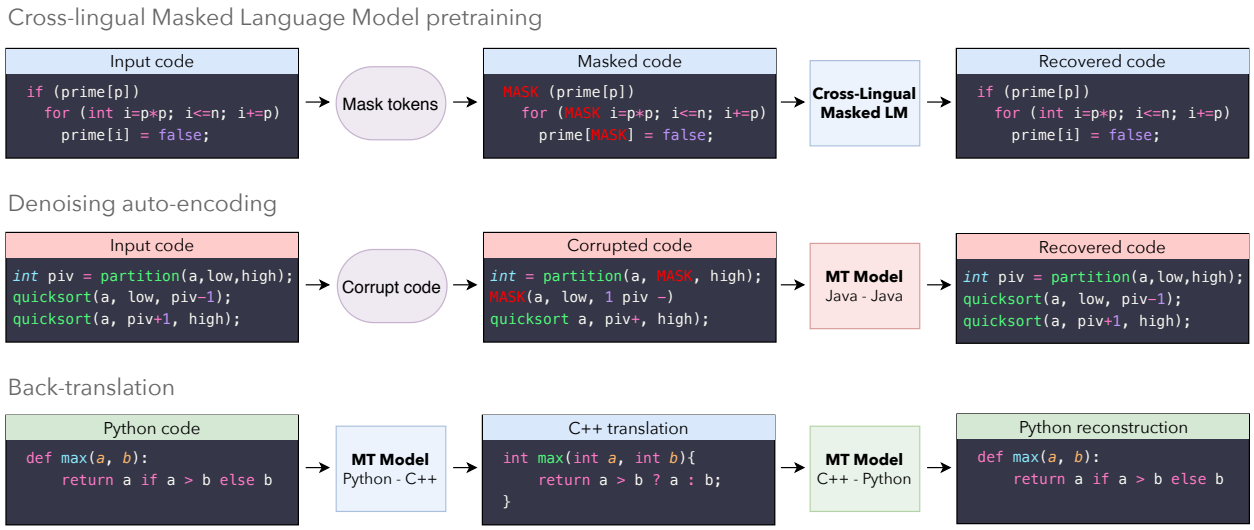

Figure 3.1: **Illustration of the three principles of unsupervised machine translation used by our approach.** The first principle initializes the model with cross-lingual masked language model pretraining. As a result, pieces of code that express the same instructions are mapped to the same representation, regardless of the programming language. Denoising auto-encoding, the second principle, trains the decoder to always generate valid sequences, even when fed with noisy data, and increases the encoder robustness to input noise. Back-translation, the last principle, allows the model to generate parallel data which can be used for training. Whenever the Python $\rightarrow$ C++ model becomes better, it generates more accurate data for the C++ $\rightarrow$ Python model, and vice versa. Figure 3.2 provides a representation of the cross-lingual embeddings we obtain after training.

In this chapter, we present TransCoder, an unsupervised model leveraging objectives developed for NLP to translate between programming languages.

3.1 Model

We consider a sequence-to-sequence (seq2seq) model with attention (Sutskever et al., 2014; Bahdanau et al., 2015), composed of an encoder and a decoder with a transformer architecture (Vaswani et al., 2017). We use a single shared model for all programming languages. We train it using the three principles of unsupervised machine translation identified in Lample et al. (2018c), namely initialization, language modeling, and back-translation. In this section, we summarize these principles and detail how we instantiate them to translate programming languages. An illustration of our approach is given in Figure 3.1.

3.1.1 Cross Programming Language Model pretraining

Pretraining is a key ingredient of unsupervised machine translation Lample et al. (2018c). It ensures that sequences with a similar meaning are mapped to the same latent representation, regardless of their languages. Originally, pretraining was done by initializing the model with cross-lingual word representations (Lample et al., 2018a; Artetxe et al., 2018b). In the context of unsupervised English-French translation, the embedding of the word "cat" will be close to the embedding of its French translation "chat". Cross-lingual word embeddings can be obtained by training monolingual word embeddings and aligning them in an unsupervised manner (Lample et al., 2018b; Artetxe et al., 2017).

Subsequent work showed that pretraining the entire model (and not only word representations) in a cross-lingual way could lead to significant improvements in unsupervised machine translation (Lample and Conneau, 2019; Lewis et al., 2020; Song et al., 2019). In particular, we follow the pretraining strategy of Lample and Conneau (2019), where a Cross-lingual Language Model (XLM) is pretrained with a masked language modeling objective (Devlin et al., 2018) on monolingual source code datasets.

The cross-lingual nature of the resulting model comes from the significant number of common tokens (anchor points) that exist across languages. In the context of English-French translation, the anchor points consist essentially of digits and city and people names. In programming languages, these anchor points come from common keywords (e.g. `for`, `while`, `if`, `try`), and also digits, mathematical operators, and English strings that appear in the source code. In practice, the "cross-linguality" of the model highly depends on the amount of anchor points across languages. As a result, an XLM model trained on English-French will provide better cross-lingual representations than a model trained on English-Chinese, because of the different alphabet, which reduces the number of anchor points. In programming languages, the majority of strings are composed of English words, which results in a fairly high number of anchor points, and the model *naturally* becomes cross-lingual. Figure 3.2 shows a t-SNE vizualization of the embeddings of a few C++, Java and Python tokens. Thanks to anchor words, tokens with similar semantics in different languages such as `map` and `dict` have similar embeddings.

For the masked language modeling (MLM) objective, at each iteration we consider an input stream of source code sequences, randomly mask out some of the tokens, and train TransCoder to predict the tokens that have been masked out based on

their contexts. We alternate between streams of batches of different languages. This allows the model to create high quality, cross-lingual sequence representations. An example of XLM pretraining is given on the top of Figure 3.1.

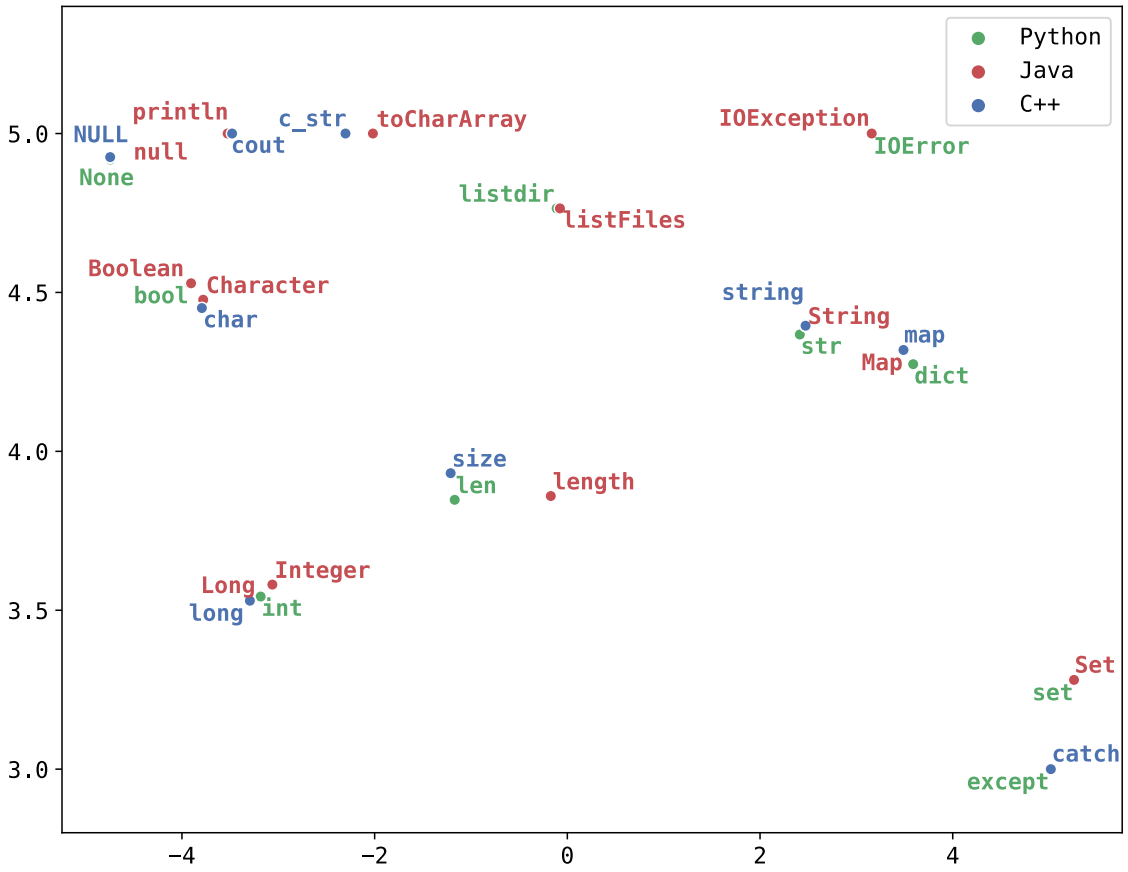

Figure 3.2: **Cross-lingual token embedding space.** We show a t-SNE visualization of our cross-lingual token embeddings. These embeddings are obtained by encoding programming language tokens into TransCoder's lookup table. We show the embeddings of C++, Java, and Python keywords. Keywords of different programming languages that are used in similar contexts are very close in the embedding space. For instance, `except` in Python and `catch` in Java and C++, which are both used to catch exceptions, are mapped to very similar embeddings. The same phenomenon is observed for implementations of maps (`Map`, `map` and `dict`), for `c_str` and `toCharArray` which are used to transform a string into a char array, and for similar primitive types (e.g. `Long`, `long`, `Integer`, and `int`).

3.1.2 Denoising auto-encoding

We initialize the encoder and decoder of the seq2seq model with the XLM model pretrained in Section 3.1.1. The initialization is straightforward for the encoder, as it has the same architecture as the XLM model. The transformer decoder, however, has extra parameters related to the source attention mechanism (Vaswani et al., 2017). Following Lample and Conneau (2019), we initialize these parameters randomly.

XLM pretraining allows the seq2seq model to generate high quality representations of input sequences. However, the decoder lacks the capacity to translate, as it has never been trained to decode a sequence based on a source representation. To address this issue, we train the model to encode and decode sequences with a Denoising Auto-Encoding (DAE) objective (Vincent et al., 2008). The DAE objective operates like a supervised machine translation algorithm, where the model is trained to predict a sequence of tokens given a corrupted version of that sequence. To corrupt a sequence, we use the same noise model as the one described in Lample et al. (2018a). Namely, we randomly mask, remove and shuffle input tokens. Masking spans of tokens instead of single tokens similarly to BART (Lewis et al., 2020) leads to similar performances.

The first symbol given as input to the decoder is a special token indicating the output programming language. At test time, a Python sequence can be encoded by the model, and decoded using the C++ start symbol to generate a C++ translation. The quality of the C++ translation will depend on the "cross-linguality" of the model: if the Python function and a valid C++ translation are mapped to the same latent representation by the encoder, the decoder will successfully generate this C++ translation.

The DAE objective also trains the "language modeling" aspect of the model, i.e. the decoder is always trained to generate a valid function, even when the encoder output is noisy. Moreover it also trains the encoder to be robust to input noise, which is helpful in the context of back-translation where the model is trained with noisy input sequences. DAE is illustrated in the middle of Figure 3.1.

3.1.3 Back-translation

In theory, XLM pretraining and denoising auto-encoding alone are enough to generate translations. However, the quality of these translations tends to be low, as the model is never trained to do what it is expected to do at test time, i.e. to translate functions from one language to another. To address this issue, we use back-translation, which is one of the most effective methods to leverage monolingual data in a weakly-supervised

scenario. Back-translation Lambert et al. (2011); Bojar and Tamchyna (2011) was initially applied to improve the performance of machine translation in the supervised setting (Sennrich et al., 2015a). It turned out to be an important component of unsupervised machine translation (Lample et al., 2018a,c; Artetxe et al., 2018b).

In the unsupervised setting, a source-to-target model is coupled with a backward target-to-source model trained in parallel. The target-to-source model is used to translate target sequences into the source language, producing noisy source sequences corresponding to the ground truth target sequences. The source-to-target model is then trained in a weakly supervised manner to reconstruct the target sequences from the noisy source sequences generated by the target-to-source model, and vice versa. The two models are trained in parallel until convergence. An example of back-translation is illustrated in Figure 3.1.

3.2 Experiments

We implement the model described above, and perform experiments to evaluate its performance for code translation.

3.2.1 Training details

We use a transformer with 6 layers, 8 attention heads, and set the dimensionality of the model to 1024. We use a single encoder and a single decoder for all programming languages. During XLM pretraining, we alternate between batches of C++, Java, and Python, composed of 32 sequences of source code of 512 tokens. At training time, we alternate between the denoising auto-encoding and back-translation objectives, and use batches of around 6000 tokens. We optimize TransCoder with the Adam optimizer (Kingma and Ba, 2015), a learning rate of 10^{-4}, and use the same learning rate scheduler as Vaswani et al. (2017). We implement our models in PyTorch (Paszke et al., 2017) and train them on 32 V100 GPUs. We use float16 operations to speed up training and to reduce the memory usage of our models.

3.2.2 Training data

We download the GitHub public dataset available on Google BigQuery.[1] It contains more than 2.8 million open source GitHub repositories. We filter projects whose

[1] `https://console.cloud.google.com/marketplace/details/github/github-repos`

license explicitly permits the re-distribution of parts of the project, and select the C++, Java, and Python files within those projects. Ideally, a transcompiler should be able to translate whole projects. In this work, we decide to translate at function level. Unlike files or classes, functions are short enough to fit into a single batch, and working at function level allows for a simpler evaluation of the model with unit tests (c.f. Section 3.2.4). We pretrain TransCoder on all source code available, and train the denoising auto-encoding and back-translation objectives on functions only. Please refer to Section 3.2.2 and Table 3.1 for more details on how the functions are extracted, and for statistics about our training set. We carry out an ablation study to determine whether it is better to keep or remove comments from source code. Keeping comments in the source code increases the number of anchor points across languages, which results in a better overall performance. Therefore, we keep them in our final datasets and experiments.

Function extraction

We train and evaluate our translation model on functions only. We differentiate class functions and standalone functions. By standalone functions, we refer to functions that can be used without instantiating a class. In C++ and Python, this corresponds to static methods of classes, and functions outside classes. In Java, it only corresponds to static methods. In GeeksforGeeks, solutions are implemented with standalone functions, and our evaluation protocol only involves these functions. In Table 3.1, the functions statistics are given for all kind of functions. In C++ and Python, 50% of functions are standalone functions. In Java, standalone functions only represent 15% of the dataset. We tried to train our model on standalone functions only, and observed better results than when training on all functions. Thus, all the results in this work are given for models pretrained on all available data and trained on standalone functions only.

3.2.3 Preprocessing

Recent approaches in multilingual natural language processing tend to use a common tokenizer (Kudo and Richardson, 2018), and a shared vocabulary for all languages. This reduces the overall vocabulary size, and maximizes the token overlap between languages, improving the cross-linguality of the model (Devlin et al., 2018; Lample and Conneau, 2019). In our case, a universal tokenizer without pre-tokenization would be suboptimal, as different languages use different patterns and keywords.

Table 3.1: **Statistics of our GitHub dataset.** We show the statistics for our entire GitHub dataset (All) and for the extracted functions. We give the size in GigaBytes, the number of files and functions, and the number of tokens.

	C++	Java	Python
All - Size	168 GB	352 GB	224 GB
All - Nb of files	15 M	56 M	18 M
All - Nb of tokens	38 B	75 B	50 B
Functions - Size	93 GB	185 GB	152 GB
Functions - Nb of functions	120 M	402 M	217 M

The logical operators && and || exist in C++ where they should be tokenized as a single token, but not in Python. The indentations are critical in Python as they define the code structure, but have no meaning in languages like C++ or Java. We use the `javalang`[2] tokenizer for Java, the tokenizer of the standard library for Python[3], and the `clang`[4] tokenizer for C++. These tokenizers ensure that meaningless modifications in the code (e.g. adding extra new lines or spaces) do not have any impact on the tokenized sequence. An example of tokenized code is given in Figure 3.3. We learn common BPE codes (Sennrich et al., 2015b) on extracted tokens, and split tokens into subword units. The BPE codes are learned with fastBPE[5] on the concatenation of tokenized C++, Java, and Python files. We also use a common vocabulary for all languages.

3.2.4 Evaluation

GeeksforGeeks is an online platform[6] with computer science and programming articles. It gathers many coding problems and presents solutions in several programming languages. From these solutions, we extract a set of parallel functions in C++, Java, and Python, to create our validation and test sets. These functions not only return the same output, but also compute the result with similar algorithm. In Figure 3.5, we show an example of C++-Java-Python parallel function that determines whether an integer represented by a string is divisible by 13.

The majority of studies in source code translation use the BLEU score to evaluate the quality of generated functions (Aggarwal et al., 2015; Miceli-Barone and Sennrich,

[2] `https://github.com/c2nes/javalang`
[3] `https://docs.python.org/3/library/tokenize.html`
[4] `https://pypi.org/project/clang`
[5] `https://github.com/glample/fastBPE`
[6] `https://practice.geeksforgeeks.org`

Python function v1

```
def rm_file(path):
    try:
        os.remove(path)
        print("Deleted")
    except:
        print("Error while deleting file", path)
```

Python function v2

```
def rm_file(path):

    try:
        os.remove( path )
        print( "Deleted" )
    except  :
        print("Error while deleting file", path)
```

```
def rm_file ( path ) : NEWLINE try : NEWLINE INDENT os . remove (path) NEWLINE print ( " Deleted " )
DEDENT except : NEWLINE INDENT print ( " Error _ while _ deleting _ file " , path ) DEDENT
```

Figure 3.3: **Example of function tokenization.** We show two versions of the same Python function and their common tokenization. These function versions differ by extra spaces and one extra new line. Our Python tokenizer is robust to extra spaces and extra new lines except in strings. In strings, spaces are tokenized as ▁ (U+2581). Indentation is meaningful in Python: indented blocks are surrounded by INDENT DEDENT tokens.

2017; Karaivanov et al., 2014; Nguyen et al., 2013), or other metrics based on the relative overlap between the tokens in the translation and in the reference. A simple metric is to compute the reference match, i.e. the percentage of translations that perfectly match the ground truth reference (Chen et al., 2018). A limitation of these metrics is that they do not take into account the syntactic correctness of the generations. Two programs with small syntactic discrepancies will have a high BLEU score while they could lead to very different compilation and computation outputs. Conversely, semantically equivalent programs with different implementations will have low BLEU scores. Instead, we introduce a new metric, the computational accuracy, that evaluates whether the hypothesis function generates the same outputs as the reference when given the same inputs. We consider that the hypothesis is correct if it gives the same output as the reference for every input. We run the generated function on 10 input examples, and compare its output to that of the ground truth.

Unit test generation. We generate some unit tests to check that the functions are semantically correct and to compute the computational accuracy. These unit tests are contained in a script, which contains a reference function — named `f_gold` — from the parallel dataset, a commented `TOFILL` marker which is to be replaced with a generated function, and a main which runs both functions on a series of inputs and compares the behaviors of the two functions. We have one script per function and

per programming language.

In order to generate these scripts, we extract the parameters and their types from the Java implementation of the solution. Then, we generate 10 random inputs for these types, which are hardcoded in the test script and used to test the function. We test the generated scripts by injecting the reference function a second time with the name `f_filled` instead of the `TOFILL` comment and running it. We keep only the scripts that return a perfect score in less than 10 seconds. As Python is dynamically typed, we need to infer the Python parameters types from the Java types, and to assume that the order and types of the parameters is the same in Java and Python. When this assumption happens to be wrong, the generated script fails the tests and is discarded. As this approach is quite effective, we generated the C++ scripts in a similar manner and barely use the C++ parameter types which can be extracted from the function definition.

Equality tests. We adapt the tests checking that the reference and gold function behave in the same way based on the output type of the function (extracted from its Java implementation). For instance, we test the equality of `int` outputs with `==`, while we use `equals` for `String` outputs and relative tests for `double` outputs. If the function is inplace (the output type is `void`), we check the side effects on all its mutable arguments instead.

Special cases for random input generation. The goal of our scripts is to decide whether a function is semantically equivalent to from the reference function, and the way we generate the random inputs is critical to how discriminative the script will be. For instance, if the input of the reference function is a string, a naive solution may be to generate strings of random length and with characters sampled randomly from the set of all characters. However, our dataset contains several functions such as `checkDivisibility` in Figure 3.5 which considers the string to be a representation of a long integer. This type of function could always return the same result (e.g. `False`) on inputs strings that do not contain only digits. As many functions in our dataset assume the input strings or characters to be representations of long integers or representations of integers in base 2, we alternate between sampling the characters from (i) the set of all lowercase and uppercase letters plus the space character, (ii) the set of all digits, and (iii) the set containing 0 and 1. For similar reasons, when there is an integer array in the function parameters, we alternate between the sets $\{0 \dots 100\}$, $\{-100 \dots 100\}$ and $\{0, 1\}$ to sample the integers inside the array. When

Table 3.2: **Number of functions with unit tests for our validation and test sets.** We report the number of function with unit tests for C++, Java, and Python, for the validation and test sets. We also show the average number of tokens per function. A unit test checks whether a generated function is semantically equivalent to its reference. For each function, we have 10 unit tests, each testing it on a different input. As a result, the number of functions with unit tests per language gives the size of the validation and test sets of each pair of languages. For instance, we have 231 C++ functions with unit tests for the validation set, which means that we have a validation set of 231 functions for Java → C++ and Python → C++.

	C++	Java	Python
Nb of functions with unit tests - valid set	231	234	237
Nb of functions with unit tests - test set	466	481	463
Average #tokens per function	105.8	112.0	103.1

the function takes no argument, we do not generate any input for it and only check that the output is the same for the reference function and the generated function.

Manual verifications. In order to ensure that our unit tests are appropriate, we manually check and modify the scripts when the output of the function is the same on all 10 inputs, when the function is inplace, or when the function contains prints. As we only check the side effects affecting the mutable arguments, we remove all the functions which mainly print or write to a file. Table 3.2 shows some statistics on the final validation and test sets that we create and open-source.

At inference time, TransCoder can generate multiple translations using beam search decoding (Koehn, 2004). In machine translation, the considered hypotheses are typically the ones with the highest log-probabilities in the beam. In our case, we have access to unit tests to verify the correctness of the generated hypotheses, so we report two sets of results for our computational accuracy metric: **CA@N**, the percentage of functions with at least one correct translation in the beam, and **N Beams - CA@1** the percentage of functions where the hypothesis in the beam with the highest log-probability is a correct translation. We select our best model using greedy decoding (CA@1) for speed efficiency.

3.2.5 Results

We report the results on our test set in Table 3.3, using greedy decoding (beam size 1), for the three metrics presented in Section 3.2.4. In Table 3.4, we report our

Python input

```python
def SumOfKsubArray(arr, n, k):
  Sum = 0
  S = deque()
  G = deque()
  for i in range(k):
    while (len(S) > 0 and arr[S[-1]] >= arr[i]):
      S.pop()
    while (len(G) > 0 and arr[G[-1]] <= arr[i]):
      G.pop()
    G.append(i)
    S.append(i)
  for i in range(k, n):
    Sum += arr[S[0]] + arr[G[0]]
    while (len(S) > 0 and S[0] <= i - k):
      S.popleft()
    while (len(G) > 0 and G[0] <= i - k):
      G.popleft()
    while (len(S) > 0 and arr[S[-1]] >= arr[i]):
      S.pop()
    while (len(G) > 0 and arr[G[-1]] <= arr[i]):
      G.pop()
    G.append(i)
    S.append(i)
  Sum += arr[S[0]] + arr[G[0]]
  return Sum
```

Unsupervised C++ translation

```cpp
int SumOfKsubArray(int arr[], int n, int k){
  int Sum = 0;
  deque <int> S;
  deque <int> G;
  for(int i = 0; i < k; i ++){
    while((int) S.size() > 0 && arr[S.back()] >= arr[i])
      S.pop_back();
    while((int) G.size() > 0 && arr[G.back()] <= arr[i])
      G.pop_back();
    G.push_back(i);
    S.push_back(i);
  }
  for(int i = k; i < n; i ++){
    Sum += arr[S.front()] + arr[G.front()];
    while((int) S.size() > 0 && S.front() <= i - k)
      S.pop_front();
    while((int) G.size() > 0 && G.front() <= i - k)
      G.pop_front();
    while((int) S.size() > 0 && arr[S.back()] >= arr[i])
      S.pop_back();
    while((int) G.size() > 0 && arr[G.back()] <= arr[i])
      G.pop_back();
    G.push_back(i);
    S.push_back(i);
  }
  Sum += arr[S.front()] + arr[G.front()];
  return Sum;
}
```

Figure 3.4: **Example of unsupervised Python to C++ translation**. TransCoder successfully translates the Python input function `SumOfKsubArray` into C++. TransCoder infers the types of the arguments, of the variables, and the return type of the function. The model maps the Python `deque()` container, to the C++ implementation `deque<>`, and uses the associated `front`, `back`, `pop_back` and `push_back` methods to retrieve and insert elements into the `deque`, instead of the Python square brackets `[]`, `pop` and `append` methods. Moreover, it converts the Python `for` loop and `range` function properly.

results with beam search decoding, and compare TransCoder to existing baselines. We give an example of unsupervised translation from Python to C++ in Figure 3.4.

Evaluation metric differences. In Table 3.3, we observe that a very large fraction of translations differ from the reference, and are considered as invalid by the reference match metric although they successfully pass the unit tests. For instance, when translating from C++ to Java, only 3.1% of the generations are strictly identical to the ground truth reference, although 60.9% of them return the expected outputs. Moreover, the performance in terms of BLEU is relatively flat and does not correlate well with the computational accuracy. These results highlight the issues with the traditional reference match and BLEU metrics commonly used in the field.

Table 3.3: **Results of TransCoder on our test set with greedy decoding.** We evaluate TransCoder with different metrics: reference match, BLEU score, and computational accuracy. Only 3.1% of C++ to Java translations match the ground truth reference, although 60.9% of them successfully pass the unit tests, suggesting that reference match is not an accurate metric to evaluate the quality of translations. Similarly, the BLEU score does not correlate well with the computational accuracy.

	C++ → Java	C++ → Python	Java → C++	Java → Python	Python → C++	Python → Java
Reference Match	3.1	6.7	24.7	3.7	4.9	0.8
BLEU	85.4	70.1	97.0	68.1	65.4	64.6
Computational Accuracy	60.9	44.5	80.9	35.0	32.2	24.7

Beam search decoding. In Table 3.4, we study the impact of beam search, either by considering all hypotheses in the beam that pass the unit tests (CA@N) or by only considering the ones with the highest log-probabilities (N Beams - CA@1). Compared to greedy decoding (CA@1), beam search significantly improves the computational accuracy, by up to 24.6% in Python → C++ with 25 beams (CA@25). When the model only returns the hypothesis with the highest log-probability, the performance drops, indicating that TransCoder often finds a valid translation, although it sometimes gives a higher log-probability to incorrect hypotheses. More generally, beam search allows minor variations of the translations which can make the unit tests succeed, such as changing the return or variable types in Java and C++, or fixing small errors such as the use of `/` instead of the `//` operator in Python. More examples of errors corrected by beam search are presented in Figure 3.11.

In a real use-case, checking whether the generated functions are syntactically correct and compile, or creating unit tests from the input function would be better approaches than comparing log-probabilities in order to select an hypothesis from the beam. Table 3.5 shows that many failures come from compilation errors when the target language is Java or C++. It suggests that the "N Beams - CA@1" metric could easily be improved. We explored that in Chapter 5: the beam reordering line in Table 5.6 shows that reordering the elements based on automatically generated unit tests slightly improves the computational accuracy.

Comparison to existing baselines. We compare TransCoder with two existing approaches: j2py[7], a framework that translates from Java to Python, and a commercial solution from Tangible Software Solutions[8], that translates from C++ to Java. Both systems rely on rewrite rules manually built using expert knowledge. The latter

[7] `https://github.com/natural/java2python`

[8] `https://www.tangiblesoftwaresolutions.com/`

Table 3.4: **Computational accuracy with beam search decoding and comparison to baselines.** Increasing the beam size improves the performance by up to 24.6% in Python → C++. When the model only returns the hypothesis with the highest log-probability (10 Beams - CA@1), the performance drops, indicating that the model often finds a correct translation, although it does not necessarily assign it with the highest probability. TransCoder significantly outperforms the Java → Python baseline (+29.5%) and the commercial C++ → Java baseline (+13.8%), although it is trained in a fully unsupervised manner and does not leverage human knowledge.

	C++ → Java	C++ → Python	Java → C++	Java → Python	Python → C++	Python → Java
Baselines	61.0	-	-	38.3	-	-
TransCoder CA@1	63.0	42.3	80.0	46.9	31.6	32.6
TransCoder 10 Beams - CA@1	64.9	43.4	78.8	48.8	33.7	35.6
TransCoder CA@5	70.9	57.5	86.1	60.7	43.6	42.4
TransCoder CA@10	73.4	62.0	88.8	64.6	49.4	47.6
TransCoder CA@20	74.8	65.4	91.0	67.8	56.2	51.6

handles the conversion of many elements, including core types, arrays, some collections (Vectors and Maps), and lambdas. In Table 3.4, we observe that TransCoder significantly outperforms both baselines in terms of computational accuracy, with 74.8% and 67.8% in the C++ → Java and Java → Python directions, compared to 61% and 38.3% for the baselines. TransCoder particularly shines when translating functions from the standard library. In rule-based transcompilers, rewrite rules need to be manually encoded for each standard library function, while TransCoder learns them in an unsupervised way. In Figure 3.12, we present several examples where TransCoder succeeds, while the baselines fail to generate correct translations.

3.2.6 Discussion - Analysis

In Figure 3.4, we give an example of TransCoder unsupervised translation from C++ to Java. Additional examples can be found in Figure 3.6 and Figure 3.8, 3.7. We observe that TransCoder successfully understands the syntax specific to each language, learns data structures and their methods, and correctly aligns libraries across programming languages. For instance, it learns to translate the ternary operator "`X ? A : B`" in C++ or Java to "`if X then A else B`" in Python, in an unsupervised way. In Figure 3.2, we present a t-SNE (Maaten and Hinton, 2008) visualization of cross-lingual token embeddings learned by the model. TransCoder successfully map tokens with similar meaning to the same latent representation, regardless of their languages. Figure 3.10 shows that TransCoder can adapt to small modifications. For instance, renaming a variable in the input may result in different translated types, still with valid translations. In Figure 3.13, we present

some typical failure cases where TransCoder fails to account for the variable type during generation. For instance, it copies the C++ NOT operator `!` applied to an integer in Java, while it should be translated to ~. It also translates the Python `min` function on lists to `Math.min` in Java, which is incorrect when applied to Java arrays. Table 3.5 gives detailed results on failure cases. It shows that a large proportion of errors happen at compilation time and could be caught automatically. There are no compilation errors when translating to Python since it is an interpreted language, but many of the runtime errors could be caught by static analysis tools. Finally, Table 3.6 gives the model accuracy for different function lengths. It shows that the accuracy of the model decreases with the length of the generated sequences.

Table 3.5: **Detailed results for greedy decoding.** Many failures come from compilation errors when the target language is Java or C++. It suggests that our method could be improved by constraining the decoder to generate compilable code. Runtime errors mainly occur when translating from Java or C++ into Python. Since Python code is interpreted and not compiled, this category also includes syntax errors in Python. The majority of remaining errors are due to the program returning the wrong output on one or several of the unit tests. Timeout errors are generally caused by infinite loops and mainly occur in the Java ↔ Python pair.

	#tests	success	Compilation	Runtime	Wrong output	Timeout
C++ → Java	481	63.0%	27.5%	4.2%	5.2%	0.2%
C++ → Python	463	42.3%	0.0%	38.0%	19.0%	0.7%
Java → C++	466	80.0%	12.5%	1.1%	6.2%	0.2%
Java → Python	463	46.9%	0.0%	32.0%	20.3%	0.9%
Python → C++	466	31.6%	38.6%	3.9%	25.3%	0.6%
Python → Java	481	32.6%	31.2%	10.4%	24.5%	1.3%

Table 3.6: **Performance v.s. function lengths.** CA@1 metric for various function lengths (in number of tokens) for C++ → Python with 10 beams.

Number of Tokens	CA@1
$0 \leq n < 30$	93.0%
$30 \leq n < 50$	91.7%
$50 \leq n < 61$	93.5%
$61 \leq n < 72$	71.7%
$72 \leq n < 85$	70.2%
$85 \leq n < 99$	56.3%
$99 \leq n < 121$	37.0%
$121 \leq n < 147$	42.2%
$147 \leq n < 179$	39.6%
$179 \leq n < 326$	28.3%

3.3 Translation examples

C++

```cpp
bool checkDivisibility(string num){
  int length = num.size();
  if(length == 1 && num[0] == '0')
    return true;
  if(length % 3 == 1){
    num += "00";
    length += 2;
  }
  else if(length % 3 == 2){
    num += '0';
    length += 1;
  }

  int sum = 0, p = 1;
  for(int i = length - 1;
              i >= 0; i-){
    int group = 0;
    group += num[i-] - '0';
    group += (num[i-] - '0') * 10;
    group += (num[i] - '0') * 100;
    sum = sum + group * p;
    p *= (-1);
  }

  sum = abs(sum);
  return (sum % 13 == 0);
}
```

Java

```java
static boolean checkDivisibility(
                    String num){
  int length = num.length();
  if(length == 1 && num.charAt(0) == '0')
    return true;
  if(length % 3 == 1){
    num += "00";
    length += 2;
  }
  else if(length % 3 == 2){
    num += "0";
    length += 1;
  }

  int sum = 0, p = 1;
  for(int i = length - 1; i >= 0; i-){
    int group = 0;
    group += num.charAt(i-) - '0';
    group += (num.charAt(i-) - '0') * 10;
    group += (num.charAt(i) - '0') * 100;
    sum = sum + group * p;
    p *= (-1);
  }

  sum = Math.abs(sum);
  return (sum % 13 == 0);
}
```

Python

```python
def checkDivisibility(num):
  length = len(num)
  if(length == 1 and num[0] == '0'):
    return True
  if(length % 3 == 1):
    num = str(num) + "00"
    length += 2
  elif(length % 3 == 2):
    num = str(num) + "0"
    length += 1

  sum = 0
  p = 1
  for i in range(length - 1, -1, -1):
    group = 0
    group += ord(num[i]) - ord('0')
    i -= 1
    group += (ord(num[i]) - ord('0')) * 10
    i -= 1
    group += (ord(num[i]) - ord('0')) * 100
    sum = sum + group * p
    p *= (-1)

  sum = abs(sum)
  return (sum % 13 == 0)
```

Figure 3.5: **Example of parallel function from our test set.** We extracted parallel functions from GeeksforGeeks to create validation and test sets. Here, we have the parallel implementations in C++, Java, and Python of the checkDivisibility function, which determines whether a long integer represented as a string is divisible by 13.

C++ Input

```cpp
int maxLen(int arr[], int n){
  unordered_map<int, int> hM;
  int sum = 0;
  int max_len = 0;
  int ending_index = - 1;
  for(int i = 0; i < n; i++)
    arr[i] = (arr[i] == 0) ? -1 : 1;
  for(int i = 0; i < n; i++){
    sum += arr[i];
    if(sum == 0){
      max_len = i + 1;
      ending_index = i;
    }
    if(hM.find(sum + n) != hM.end()){
      if(max_len < i - hM[sum + n]){
        max_len = i - hM[sum + n];
        ending_index = i;
      }
    }
    else hM[sum + n] = i;
  }
  for(int i = 0; i < n; i++)
    arr[i] = (arr[i] == -1) ? 0 : 1;
  printf("%d to %d\n",
          ending_index - max_len + 1,
          ending_index);
  return max_len;
}
```

```cpp
std::unordered_set<int> uniqueElements(
                    const std::vector<int>& l){
  return std::unordered_set<int>(l.begin(), l.end());
 }
```

```cpp
std::vector<int> squareList(const std::vector<int>& l){
  std::vector<int> result = {};
  for(int elmt : l){
    result.push_back(elmt * elmt);
  }
  return result;
 }
```

Unsupervised Java Translation

```java
public static int maxLen(int arr[], int n){
  HashMap<Integer, Integer> hM
        = new HashMap<Integer, Integer>();
  int sum = 0;
  int maxLen = 0;
  int endingIndex = - 1;
  for(int i = 0; i < n; i++){
    arr[i] = (arr[i] == 0) ? - 1 : 1;
  }
  for(int i = 0; i < n; i++){
    sum += arr[i];
    if(sum == 0){
      maxLen = i + 1;
      endingIndex = i;
    }
    if(hM.containsKey(sum + n)){
      if(maxLen < i - hM.get(sum + n)){
        maxLen = i - hM.get(sum + n);
        endingIndex = i;
      }
    }else{
      hM.put(sum + n, i);
    }
  }
  for(int i = 0; i < n; i ++){
    arr[i] = (arr[i] == -1) ? 0 : 1;
  }
  System.out.println(endingIndex - maxLen + 1
                     + " to " + endingIndex);
  return maxLen;
}
```

```java
public static Set<Integer> uniqueElements(
                    Collection<Integer> l){
  return new HashSet<Integer>(l);
 }
```

```java
public static List<Integer> squareList(
                    List<Integer> l){
  List<Integer> result = new ArrayList<>();
  for(Integer elmt : l) result.add(elmt*elmt);
  return result;
}
```

Figure 3.6: **Examples of correct translations using TransCoder**. In all these examples, TransCoder converts properly the primitive types, find the equivalent data structures and the corresponding methods, and libraries between languages. For instance, in the `maxLen` function, TransCoder converts the C++ `unordered_map` container into the Java implementation of the `Map` interface, `HashMap`, and properly uses the associated `get` and `put` methods to retrieve and insert keys and values into the map, instead of the C++ square brackets `[]`.

Input

C++

```
void worstFit(int blockSize[],int m,int processSize[],int n){
  int allocation[n];
  memset(allocation, -1, sizeof(allocation));
  for(int i = 0; i < n; i ++){
    int wstIdx = - 1;
    for(int j = 0; j < m; j++){
      if(blockSize[j] >= processSize[i]){
        if(wstIdx == - 1)
          wstIdx = j;
        else if(blockSize[wstIdx] < blockSize[j])
          wstIdx = j;
      }
    }
    if(wstIdx != -1){
      allocation[i] = wstIdx;
      blockSize[wstIdx] -= processSize[i];
    }
  }
  cout « "\nProcess No.\tProcess Size\tBlock no.\n";
  for(int i = 0; i < n; i++){
    cout « "    " « i+1 « "\t\t" « processSize[i] « "\t\t";
    if(allocation[i] != -1) cout « allocation[i] + 1;
    else
      cout « "Not Allocated";
  cout « endl;
  }
}
```

TransCoder Translation

Python

```
def worstFit(blockSize, m, processSize, n):
  allocation = [-1] * n
  for i in range(n):
    wstIdx = -1
    for j in range(m):
      if blockSize[j] >= processSize[i]:
        if wstIdx == -1:
          wstIdx = j
        elif blockSize[wstIdx] < blockSize[j]:
          wstIdx = j
    if wstIdx != -1:
      allocation[i] = wstIdx
      blockSize[wstIdx] -= processSize[i]
  print("\nProcess No.\tProcess Size\tBlock no.\n")
  for i in range(n):
    print("    " + str(i+1) + "\t\t" +
          str(processSize[i]) + "\t\t")
    if allocation[i] != -1:
        print(allocation[i] + 1)
    else:
        print("Not Allocated")
    print()
```

Java

```
public static int max(int a, int b){
  return a > b ? a : b;
}

public static void createDirectory(Path path)
            throws IOException{
  if(!Files.exists(path)){
    Files.createDirectories(path);
  }
}
```

Python

```
def max(a, b):
  return a if a > b else b


def create_directory(path):
  if not os.path.exists(path):
    os.makedirs(path)
```

Figure 3.7: **Examples of correct translations to Python using TransCoder**. TransCoder is able to translate C++ and Java functions and idioms to python correctly. For instance, to initialize a list with n `-1` integers in the first example, or for inline if conditions and safely creating a directory in the second and third examples.

Input	TransCoder Translation
Python	C++

`def sum_elements(values):`
`  return sum(values)`
	`template <typename T> T`
	`sum_elements(const std::vector<T> &values){`
	`  return std::accumulate(values.begin(), values.end(),`
	`  0);`
	`}`

`def no_letters(s):`
`  return s.lower() == s.upper()`
	`static bool noLetters(const QString& str){`
	`  return str.toLower() == str.toUpper();`
	`}`

`def get_env_variable(name):`
`  return os.environ[name]`
	`const char* GetEnvVariable(const char* name){`
	`  return getenv(name);`
	`}`

Python	Java

`def calcMaxValue(str):`
`  res = ord(str[0]) - 48`
`  for i in range(1, len(str)):`
`  if(str[i] == '0' or str[i] == '1' or res < 2):`
`  res += ord(str[i]) - 48`
`  else:`
`  res *= ord(str[i]) - 48`
`  return res`
	`public static int calcMaxValue(String str){`
	`  int res = (int)(str.charAt(0) - 48);`
	`  for(int i = 1; i < str.length(); i++){`
	`  if(str.charAt(i) == '0'`
	`  || str.charAt(i) == '1'`
	`  || res < 2){`
	`  res += (int)(str.charAt(i) - 48);`
	`  }`
	`  else{`
	`  res *= (int)(str.charAt(i) - 48);`
	`  }`
	`  }`
	`  return res;`
	`}`

`def foo(x):`
`  return bar(x) + 1`
	`public static int foo(int x){`
	`  return bar(x) + 1;`
	`}`

`def area(r):`
`  return 2 * PI * r ** 2`
	`public static double area(double r){`
	`  return 2 * PI * r * r;`
	`}`

Figure 3.8: **Examples of correct translations from Python using TransCoder.** When translating from Python, TransCoder successfully infers types. Here, TransCoder infers the Python `list` type and translates it into its C++ equivalent `std::vector`. The last two examples show that TransCoder does not modify the call to the non-standard function `bar` or the global variable `PI`.

C++ input

```
int minPalPartion(char* str){
  int n = strlen(str);
  int C[n][n];
  bool P[n][n];
  int i, j, k, L;
  for(i = 0; i < n; i ++){
    P[i][i] = true;
    C[i][i] = 0;}
  for(L = 2; L <= n; L ++){
    for(i = 0; i < n - L + 1; i ++){
      j = i + L - 1;
      if(L == 2) P[i][j] = (str[i] == str[j]);
      else P[i][j] = (str[i] == str[j])
                  && P[i+1][j-1];
      if(P[i][j] == true) C[i][j] = 0;
      else{
        C[i][j] = INT_MAX;
        for(k = i; k <= j - 1; k ++)
            C[i][j] = min(C[i][j],
                        C[i][k]+C[k+1][j]+1);
      }
    }
  }
  return C[0][n - 1];
}
```

TransCoder Java translation

```
public static int minPalPartion(String str){
  int n = str.length();
  int[][] C = new int[n][n];
  boolean[][] P = new boolean[n][n];
  int i , j , k , L;
  for(i = 0; i < n; i ++){
    P[i][i] = true;
    C[i][i] = 0;}
  for(L = 2; L <= n; L ++){
    for(i = 0; i < n - L + 1;  i ++){
      j = i + L - 1;
      if(L == 2){
        P[i][j] = (str.charAt(i) == str.charAt(j));}
      else{
        P[i][j] = (str.charAt(i) == str.charAt(j))
                    && P[i + 1][j - 1];}
      if(P[i][j] == true){
        C[i][j] = 0;}
      else{
        C[i][j] = Integer.MAX_VALUE;
        for(k = i; k <= j - 1; k ++){
          C[i][j] = Math.min(C[i][j],
                    C[i][k] + C[k + 1][j] + 1);}}}}
  return C[0][n - 1];
}
```

Figure 3.9: **TransCoder robustness to variable names.** We consider the C++ function `minPalPartion`, with a parameter `char*` parameter named `str` indicating that it could be translated to a str parameter. In Figure 3.10, we study the influence of the parameter name on the output of TransCoder.

C++ input

```
int minPalPartion(char* arr){
  int n = strlen(arr);
  int C[n][n];
  bool P[n][n];
  int i, j, k, L;
  for(i = 0; i < n; i ++){
    P[i][i] = true;
    C[i][i] = 0;}
  for(L = 2; L <= n; L ++){
    for(i = 0; i < n - L + 1; i ++){
      j = i + L - 1;
      if(L == 2) P[i][j] = (arr[i] == arr[j]);
      else P[i][j]= (arr[i] == arr[j])
                    && P[i+1][j-1];
      if(P[i][j] == true) C[i][j] = 0;
      else{
        C[i][j] = INT_MAX;
        for(k = i; k <= j - 1; k ++)
            C[i][j] = min(C[i][j],
                          C[i][k]+C[k+1][j]+1);
      }
    }
  }
  return C[0][n - 1];
}
```

TransCoder Java translation

```
public static int minPalPartion(char[] arr){
  int n = arr.length;
  int[][] C = new int[n][n];
  boolean[][] P = new boolean[n][n];
  int i , j , k , L;
  for(i = 0; i < n; i ++){
    P[i][i] = true;
    C[i][i] = 0;}
  for(L = 2; L <= n; L ++){
    for(i = 0; i < n - L + 1;  i ++){
      j = i + L - 1;
      if(L == 2){
        P[i][j] = (arr[i] == arr[j]);}
      else{
        P[i][j] = (arr[i] == arr[j])
                    && P[i + 1][j - 1];}
      if(P[i][j] == true){
        C[i][j] = 0;}
      else{
        C[i][j] = Integer.MAX_VALUE;
        for(k = i; k <= j - 1; k ++){
          C[i][j] = Math.min(C[i][j],
                    C[i][k] + C[k + 1][j] + 1);}}}}
  return C[0][n - 1];
}
```

```
int minPalPartion(char* input){
  int n = strlen(input);
  int C[n][n];
  bool P[n][n];
  int i, j, k, L;
  for(i = 0; i < n; i ++){
    P[i][i] = true;
    C[i][i] = 0;}
  for(L = 2; L <= n; L ++){
    for(i = 0; i < n - L + 1; i ++){
      j = i + L - 1;
      if(L == 2) P[i][j] = (input[i] == input[j]);
      else P[i][j] = (input[i] == input[j])
                    && P[i+1][j-1];
      if(P[i][j] == true) C[i][j] = 0;
      else{
        C[i][j] = INT_MAX;
        for(k = i; k <= j - 1; k ++)
            C[i][j] = min(C[i][j],
                          C[i][k]+C[k+1][j]+1);
      }
    }
  }
  return C[0][n - 1];
}
```

```
public static int minPalPartion(String input){
  int n = input.length();
  int[][] C = new int[n][n];
  boolean[][] P = new boolean[n][n];
  int i , j , k , L;
  for(i = 0; i < n; i ++){
    P[i][i] = true;
    C[i][i] = 0;}
  for(L = 2; L <= n; L ++){
    for(i = 0; i < n - L + 1;  i ++){
      j = i + L - 1;
      if(L == 2){
        P[i][j] = (input.charAt(i) == input.charAt(j));
        }
      else{
        P[i][j] = (input.charAt(i) == input.charAt(j))
                    && P[i + 1][j - 1];}
      if(P[i][j] == true){
        C[i][j] = 0;}
      else{
        C[i][j] = Integer.MAX_VALUE;
        for(k = i; k <= j - 1; k ++){
          C[i][j] = Math.min(C[i][j],
                    C[i][k] + C[k + 1][j] + 1);}}}}
  return C[0][n - 1];
}
```

Figure 3.10: **TransCoder robustness to variable names.** We take the C++ function `minPalPartion` from Figure 3.9, change the parameter name from `str` to `arr` and `input` and show the three corresponding TransCoder Java translations. All translations are correct. In the Figure 3.9 and second cases, TransCoder translates `char* str` and `char* input` into Java `String` and uses the `charAt` method to retrieve elements. This shows that TransCoder is robust to variable name changes and that it remembers variable types along the whole translation to apply the appropriate methods. In the first example, TransCoder translates `char* arr` into Java `char[]` and uses `[]` to retrieve elements, showing that TransCoder can adjust its translations to parameter names while remaining accurate.

C++ input	TransCoder Python translation	
	Greedy decoding	Beam search

`int compute_average(int a, int b){`
`  return (a + b) / 2;`
`}`
	`def compute_average(a, b):`
	`  return (a + b) / 2`
		`def compute_average(a, b):`
		`  return (a + b) // 2`

`bool isPalindrome(string str){`
`  int l = 0;`
`  int h = str.length() - 1;`
`  while(h > l)`
`  if(str[l++] != str[h-])`
`  return false;`
`  return true;`
`}`
	`def isPalindrome(str):`
	`  l = 0`
	`  h = len(str) - 1`
	`  while h > l:`
	`  if str[l] != str[h-]:`
	`  return False`
	`  return True`
		`def isPalindrome(str):`
		`  l = 0`
		`  h = len(str) - 1`
		`  while h > l:`
		`  if str[l] != str[h]:`
		`  return False`
		`  l += 1`
		`  h -= 1`
		`  return True`

Figure 3.11: **Examples of incorrect greedy decoding translations versus correct beam translations.** We take C++ inputs and translate them into Python with TransCoder. In the second column, we use greedy decoding and the translations are incorrect. In the third column, we use beam search and obtain accurate translations. A common error corrected by beam search for C++ $\rightarrow$ Python is the usage of the double instead of the integer division operator (first example). Beam search is also able to correct errors such as the ++ and $--$ operators that do not exist in Python (second example).

Java

```java
static String getEnvVariable(
        String name){
  return System.getenv(name);
 }
```

Baseline translation — Python

```python
def getEnvVariable(name):
  return System.getenv(name)
```

TransCoder translation — Python

```python
def getenvh(name):
  return os.environ.get(name, '')
```

Java

```java
static List<Integer> truncate(
    List<Integer> l,
    Integer e){
  if(l.contains(e)){
    return l.subList(0,
            l.indexOf(e));
  }
  else{
    return l;
  }
 }
```

Baseline translation — Python

```python
def truncate(l, e):
  if l.contains(e):
    return l.subList(0,
             l.indexOf(e))
  else:
    return l
```

TransCoder translation — Python

```python
def truncate(l, e):
  if l.count(e) > 0:
    return l[ : l.index(e)]
  else:
    return l
```

Java

```java
static void deleteFile(Path path){
  try{
    Files.delete(path);
  }
  catch(Exception e){
    System.err.println(
    "Error deleting " + path);
  }
}
```

Baseline translation — Python

```python
def deleteFile(path):
  try:
    Files.delete(path)
  except Exception as e:
    System.err.println(
      "Error deleting " + path)
```

TransCoder translation — Python

```python
def delete_file(path):
  try:
    os.remove(path)
  except:
    sys.stderr.write(
        "Error deleting %s\n"
        % path
    )
```

C++

```cpp
memset(prime, 0, sizeof(prime));

sort(a, a + n);

for(char ch : str)
```

Baseline translation — Java

```java
memset(prime, 0,
  (Integer.SIZE/Byte.SIZE));

sort(a, a + n);

for(char ch : str)
```

TransCoder translation — Java

```java
Arrays.fill(prime, 0);

Arrays.sort(a);

for(char ch : str.toCharArray())
```

Figure 3.12: **Examples of incorrect baseline translations versus correct TransCoder translations.** When translating from Java to Python, the baseline fails to translate the `System.getenv`, `System.err.println`, and `Files.delete` functions from the standard library, and the `contains`, `subList`, and `IndexOf` methods of the Java `List` interface. Instead, it simply copies them, showing the limitations of a rule-based system. On the other hand, TransCoder converts properly all of these functions into their Python equivalents. In the C++ $\rightarrow$ Java direction, baseline translations are made at token-level, and are incorrect. For instance, the first example shows that the baseline tries to translate the `sizeof` function, and leaves `memset` unchanged although it does not exist in Java. Instead, TransCoder correctly uses `Arrays.fill` to fill the array `prime` with zeros.

Input	Java failed translations	Description
`bool isEven (int n){ return (!(n & 1)); }`	`static boolean isEven(int n){ return (!(n & 1)); }`	The `!` operator works on boolean and integers in C++ (it returns `true` if the integer is positive) but it only works on boolean in Java.
`int summingSeries(long n){ return pow(n, 2); }`	`static int summingSeries(long n){ return Math.pow(n, 2); }`	In Java, `Math.pow(n, 2)` returns a `double` which should be cast to `int` to match the function return type.
`def minSum(A): min_val = min(A) return min_val * (len(A) - 1)`	`static double minSum(double[] A){ double minVal = Math.min(A); return minVal*(A.length - 1); }`	`Math.min` is a Java function but does not take as input a `double[]` array but a pair of `double`.

Figure 3.13: **Examples of failed TransCoder translations.** TransCoder fails to translate these C++ and Python functions into Java, showing its limitations. In these examples, it fails to account for the variable types when using a method or an operator. In particular, the NOT operator `!` in C++ should have been translated to ~ in Java, because it is applied to an integer. Similarly, the `Math.min` function in Java cannot be applied to arrays.

3.4 Conclusion

In this chapter, we show that approaches of unsupervised machine translation can be applied to source code to create a transcompiler in a fully unsupervised way. TransCoder can easily be generalized to any programming language, does not require any expert knowledge, and outperforms commercial solutions by a large margin. Our results suggest that a lot of mistakes made by the model could easily be fixed by adding simple constraints to the decoder to ensure that the generated functions are syntactically correct, or by using dedicated architectures (Chen et al., 2018). Leveraging the compiler output or other approaches such as iterative error correction (Fu et al., 2019) could also boost the performance.

Moreover, our training objectives were all designed for natural language processing. We could wonder if they are well suited to programming languages, or if there could be a way to design objectives leveraging the strict syntax of source code.

Chapter 4

DOBF: A Deobfuscation Pre-Training Objective for Programming Languages

Model pre-training with self-supervised methods such as BERT (Devlin et al., 2018), RoBERTa (Liu et al., 2019), XLM (Lample and Conneau, 2019) or XLNet (Yang et al., 2019), has become ubiquitous in Natural Language Processing (NLP), and led to significant improvements in many tasks. These approaches are based on the Masked Language Modeling (MLM) objective, which is presented in Section 2.3.3.

In this chapter, we are interested in pre-training deep learning models for programming languages. We argue that MLM is actually sub-optimal in the context of programming languages, and propose a new objective based on deobfuscation of identifier names in source code.

Code obfuscation consists in modifying source code in order to make it harder for humans to understand, or smaller while keeping its behaviour unchanged. In some ancient interpreted languages, name minimization could also reduce the memory usage of the program. Today, it is used to protect intellectual property by preventing people from understanding and modifying the code, to prevent malware detection, and to compress programs (e.g. JavaScript code) to reduce network payload sizes. Moreover, C compilers discard variable names, and current rule-based and neural-based decompilers generate obfuscated C code with uninformative variable names (Fu et al., 2019). Obfuscators typically apply several transformations to the code. While some operations can be reversed (e.g. dead code injection), the obfuscation of identifier names—renaming every variable, method and class with uninformative names—is

irreversible and has a substantial impact on code comprehension (Gellenbeck and Cook, 1991; Takang et al., 1996; Lawrie et al., 2006).

By analyzing the overall structure of an obfuscated file, an experienced programmer can always, with time, understand the meaning of the obfuscated code. For instance, in the obfuscated example in Figure 4.1, one can recognize the function and guess that it implements a breadth-first search algorithm. We also expect neural networks, which excel in pattern recognition, to perform well on this task. We propose to pre-train a model to revert the obfuscation function, by training a sequence-to-sequence (seq2seq) model to convert obfuscated functions, where names of functions and variables have been replaced by uninformative names, back to their original forms. Suggesting proper variable and function names is a difficult task that requires to understand what the program does. In the context of source code, it is a more sensible, but also a more difficult task than MLM. Indeed, we observe (c.f. Figure 4.1) that predicting the content of randomly masked tokens is usually quite simple, as it often boils down to making syntax-related predictions (e.g. predicting that was has been masked out is a parenthesis, a semi-column, etc.). These simple predictions actually provide little training signal to the model. In practice, MLM also masks out variable names, but if a given variable appears multiple times in a function, it will be easy for the model to simply copy its name from one of the other occurrences. Our model does not have this issue, as all occurrences of masked variables are replaced by the same `VAR_i` special tokens. In this chapter, we make the following contributions:

- We present DOBF, a new pre-training objective based on deobfuscation, and show its effectiveness on multiple programming languages.

- We show that DOBF significantly outperforms MLM (e.g. BERT) on multiple tasks such as code search, code summarization and unsupervised code translation. We show that pre-training methods based on DOBF outperform all existing pre-training methods on all the considered tasks.

- We show that, by design, models pre-trained with DOBF have interesting applications and can be used to understand functions with uninformative identifier names. Besides, the model is able to successfully deobfuscate fully obfuscated source files.

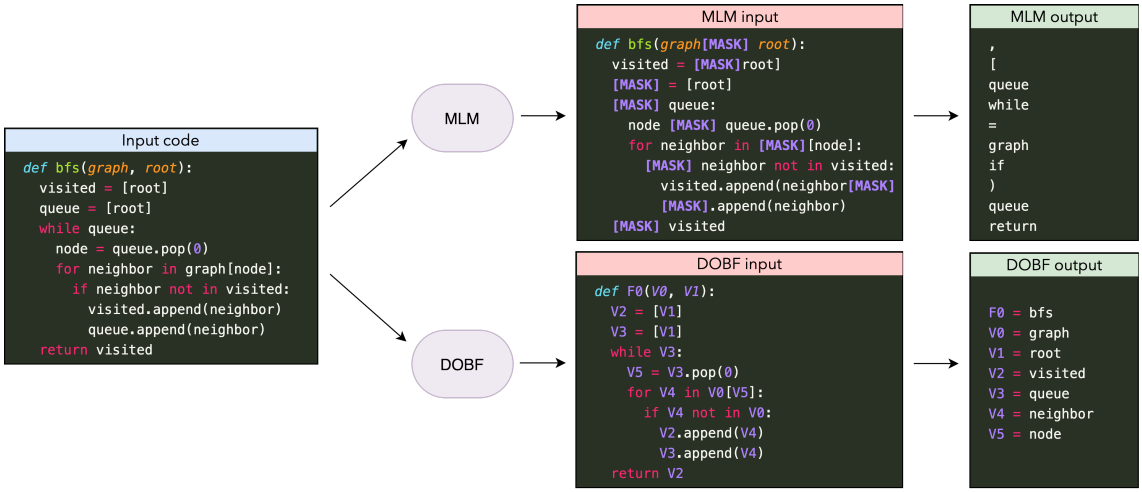

Figure 4.1: **Illustration of the MLM and DOBF objectives.** Given an input function, the masked language modeling (MLM) task randomly samples tokens to mask out. With source code, a large fraction of these tokens are related to the language syntax (e.g. commas, parentheses, etc.) that are trivial for the model to predict, and provide a poor training signal. Instead, we propose to obfuscate the code by masking the name of functions and variables, and to train the model to recover the original function by deobfuscating the code (DOBF). When a variable is masked out, we mask all occurrences of this variable with the same mask symbol (e.g. all occurrences of "visited" are replaced by "V0") to prevent the model from copying names. The DOBF objective is more difficult and provides a better learning signal.

4.1 Context

Code Generation Pre-training. Recent studies showed that pre-training methods developed for natural language processing are also effective for programming languages. For instance, Feng et al. (2020) proposed CodeBERT, a RoBERTa-based model trained on source code using the MLM and RTD objectives. With GraphCodeBERT (Guo et al., 2020), the MLM objective is complemented by an edge-prediction objective, in which the model predicts edges in the data flow graph to make the model understand the structure of the code. In Jain et al. (2020), a model is trained on JavaScript code using a contrastive loss ensuring that the representations are robust to some semantic-preserving transformations. They showed that their model performs well on downstream code generation tasks and outperforms previous pre-training approaches. Kanade et al. (2020) applied MLM and the next sentence prediction objectives to pre-train models on Python code. In chapter 3, we trained a model on monolingual source code from GitHub using MLM, denoising auto-encoding and back-translation. We showed that the resulting model, TransCoder, was able to

translate source code between Python, Java, and C++, in a fully unsupervised way. In this chapter, we propose to use a code-specific objective to better pre-train models designed to be fine-tuned on code generation tasks: code deobfuscation. Machine learning is frequently used on tasks involving programming languages, including code completion (Li et al., 2018; Liu et al., 2020; Kim et al., 2021; Svyatkovskiy et al., 2021), bug detection and code repair (Allamanis et al., 2018b; Wang et al., 2018b; Chen et al., 2019; Murali et al., 2021; Tufano et al., 2019; Tarlow et al., 2020), code summarization (Alon et al., 2019a; Hu et al., 2018; Xie et al., 2021), clone detection (Wei and Li, 2017; Ain et al., 2019; Wang et al., 2020b), code search (Gu et al., 2018; Cambronero et al., 2019) and code translation (see Chapter 3). Most of these tasks can benefit from pre-trained models that capture code semantics.

Code deobfuscation. Empirical studies show that naming conventions and the use of informative identifier names make code more understandable, easier to maintain and lead to fewer bugs (Takang et al., 1996; Liblit et al., 2006; Butler et al., 2009). It motivated other works studying deobfuscation of identifier names and identifier name proposal using n-grams (Allamanis et al., 2014, 2015a), probabilistic models (Raychev et al., 2015; Bichsel et al., 2016; Vasilescu et al., 2017; Alon et al., 2018), and recurrent neural networks (Bavishi et al., 2018; Lacomis et al., 2019). Alon et al. (2018) extract features from Abstract Syntax Tree (AST) paths and train a Conditional Random Field to predict variable and method names, and infer types for several languages. DIRE (Lacomis et al., 2019) uses a commercial decompiler to obtain C code with uninformative identifier names from binaries. They also use AST features, which go through a Graph Neural Network trained jointly with a LSTM model on the sequence of C tokens to retrieve relevant identifier names. More recently, David et al. (2020) used a transformer, together with augmented representations obtained from static analysis, to infer procedure names in stripped binary files. These models are already used to understand obfuscated and compiled source code. However, none of these studies investigated the use of deobfuscation for model pre-training.

4.2 Model

4.2.1 MLM and denoising for Programming Languages

A countless number of pre-training objectives have been introduced in the literature (Devlin et al., 2018; Clark et al., 2020; Lewis et al., 2020; Liu et al., 2019; Dong et al.,

2019). Most of them rely on hyper-parameters and seemingly arbitrary decisions (Should we mask individual tokens or spans? Which fraction of them? What do we do with masked out tokens? etc.). These choices are typically based on intuition and validated empirically on natural language processing tasks. However, source code is much more structured than natural language, which makes predicting masked tokens much easier for programming languages.

The first row in Figure 4.1 shows an example of input / output for the MLM objective. We can see that the majority of tokens are composed of Python keywords or symbols related to syntax: `, [ while = if ) return`. These symbols are easy to recover, and a model will quickly learn to predict them with perfect accuracy. This effect is accentuated by the verbosity of the language. For instance, we would see significantly more of these tokens in Java. Retrieving the obfuscated `graph` token is also relatively simple: the model only needs to retrieve the most relevant variable in the scope. More generally, retrieving an identifier name is often easy when given its full context, including its definition and usages. The denoising-auto-encoding (DAE) objective (Vincent et al., 2008), which trains an encoder-decoder model to retrieve masked token and recover randomly modified input sentences, is quite similar to MLM and the model can also retrieve identifier names easily by finding their definition or usages. Overall, we suspect that the MLM objective is too simple in programming languages and we introduce a new objective, DOBF, which encourages the model to learn a deeper understanding of code semantics.

4.2.2 Deobfuscation Objective

Instead of MLM, we propose a new pre-training objective, DOBF, that leverages the particular structure of programming languages. We obfuscate code snippets by replacing class, function and variable names with special tokens, and train a model to recover the original names. When an identifier is selected, all of its instances in the code are replaced by the same special token. This differs from MLM where the name of a variable can appear multiple times while being masked a single time. For instance, in Figure 4.1, DOBF will replace the two occurrences of `node` by the same symbol `V5`, while MLM will only mask one of these occurrences. As a result, the fraction of meaningful tokens masked by the objective is language independent: for more verbose languages (e.g. Java), the less informative syntax-related tokens will not be masked out by the DOBF objective.

Each identifier is replaced with probability $p_{obf} \in [0, 1]$. We ensure that the

original input is modified: if no identifier is replaced, we draw a random one to obfuscate. When $p_{obf} = 0$, we always obfuscate exactly one random identifier in the input. When $p_{obf} = 1$, we obfuscate all the identifiers defined in the file. We ensure that the obfuscated code has the same behavior as the original. The second row in Figure 4.1 shows an example of obfuscated code with $p_{obf} = 1$, where we obfuscate a function `bfs` which implements a breadth-first search. The function `append` is not obfuscated as it is a standard Python function not defined in the file. The model is given the obfuscated code as input and has to restore the original name of each special token `CLASS_i`, `FUNC_i` and `VAR_i`. In other words, the model needs to output a dictionary mapping special tokens to their initial values.

Finding informative names for obfuscated identifiers requires the model to learn a deep understanding of code semantics, which is desirable for a pre-training task. MLM will mask only some of the occurrences of the identifiers and leave the other ones unchanged so that the model can simply copy identifier names. In Figure 4.1, with MLM masking, the model can simply notice that a variable named `queue` is called on the fourth line. Since the variable is not defined, the model can easily guess that `queue` has to be defined on the third line, and infer the value of the corresponding `[MASK]` token. With the deobfuscation objective, the model needs to analyze code patterns and understand the semantics of the variable to infer that, since its elements are popped with `.pop(0)`, the variable `V3` implements a queue. If its elements were popped with `.pop()`, our model would name it `stack` instead of `queue` (c.f. Figure 4.8).

4.2.3 Implementation

Overall, the deobfuscation objective operates like a supervised machine translation objective, where a seq2seq model is trained to map an obfuscated code into a dictionary represented as a sequence of tokens. At inference time, the model is able to suggest meaningful class, function and variable names for a piece of code with an arbitrary number of obfuscated identifiers. Obfuscated classes, functions, and variables, are replaced with associated special tokens: `CLASS_0` ... `CLASS_N`, `FUNC_0` ... `FUNC_N` and `VAR_0` ... `VAR_N`. We serialize the output dictionary as a sequence of tokens where the entries are separated by a delimiter symbol `|`.[1]

[1] In the obfuscated example given in Figure 4.1, the model is trained to generate: `FUNC_0 bfs | VAR_0 graph | VAR_1 root | VAR_2 visited | VAR_3 queue | VAR_4 neighbor | VAR_5 node`.

4.3 Experiments

We train DOBF with the deobfuscation objective. First, we evaluate our model on two straightforward deobfuscation applications. Then, we show its performance on multiple downstream tasks.

4.3.1 Deobfuscation

We evaluate our model on two applications of the deobfuscation task: when $p_{obf} = 0$ (the model has to retrieve a single identifier name), and $p_{obf} = 1$ (the model has to retrieve all the identifier names).

Deobfuscating a single identifier When $p_{obf} = 0$, only one identifier is obfuscated. In that case, the model has to propose a relevant name for that identifier using the rest of the non-obfuscated file as context. It can be used as a tool that suggests relevant variable names. Integrated development environments (e.g. PyCharm, VSCode) already perform this task, often using handcrafted rules.

Deobfuscating all identifiers Obfuscators are commonly used to make code smaller and more efficient or to protect it by making it more difficult to understand and reuse. They typically apply several transformations, one of them being to replace every identifier name with short and uninformative names (e.g. a, b, c). In our work, such a transformation corresponds to obfuscating a file with $p_{obf} = 1$. To measure our model's ability to revert the obfuscation operation, we evaluate its accuracy when obfuscating all identifier names. Another application would be to help understand source code written with uninformative variable names.

Evaluation metric We evaluate the ability of our model to retrieve identifier names from the original non-obfuscated code. We report the accuracy, which is the percentage of recovered tokens that exactly match the ground truth. Following previous works (Allamanis et al., 2015a, 2016; Alon et al., 2018, 2019c), we also report the *subtoken score*, a more flexible metric which computes the precision, recall, and F1 scores for retrieving the original case-insensitive subtokens. Each token is broken into subtokens using uppercase letters for CamlCase and underscores for snake_case. For instance, `decoderAttention` would be considered to be a perfect match for `decoder_attention` or `attentionDecoder`. `attention` would have a perfect precision but a recall of 0.5, so a F1 score of 66.7. `crossAttentionDecoder`

would have a perfect recall but a precision of $\frac{2}{3}$, corresponding to a F1 score of 80.0. We compute the overall subtoken precision, recall and F1 scores averaged over each file in our validation and test datasets.

4.3.2 Fine-tuning on downstream tasks

In order to evaluate DOBF as a pre-training model, we fine-tune DOBF on TransCoder and on three tasks from CodeXGLUE (Lu et al., 2021), a benchmark for programming languages. The data, code and models from CodeXGLUE and TransCoder are available respectively under the MIT and the Creative Commons license. We only consider the Java and Python tasks with an encoder in the model architecture for which the training, validation, and test sets are publicly available.

CodeXGLUE Clone Detection This task is a binary classification problem where the model has to predict whether two code snippets are semantically equivalent. It is evaluated using the macro F1 score. The model is composed of a single encoder and a classification layer. An input consists in two snippets of code, which are concatenated before being fed to the model. This task is available in Java.

CodeXGLUE Code Summarization Given a code snippet, the model is trained to generate the corresponding documentation in natural language. The architecture is a sequence-to-sequence transformer model evaluated using the BLEU score (Papineni et al., 2002). The dataset includes Java and Python source code.

CodeXGLUE NL Code Search Given a code search query in natural language the model has to retrieve the most semantically related code within a collection of code snippets. This is a ranking problem evaluated using the Mean Reciprocal Rank (MRR) metric. The model is composed of two encoders. The natural language query and the code are encoded separately, and we compute the dot product between the first hidden states of the encoders' last layers. This task is available in Python.

TransCoder TransCoder is the model described in Chapter 3. It is pre-trained with MLM, and trained with denoising auto-encoding and back-translation. TransCoder is evaluated using the Computational Accuracy metric, which computes the percentage of correct solutions according to series of unit tests. In this chapter, we only consider a single model output (CA@1), with beam sizes of 1 and 10.

4.3.3 Experimental details

Model Architecture We consider a seq2seq model with attention, composed of an encoder and a decoder using a transformer architecture (Vaswani et al., 2017). We train models with the same architecture and tokenizer as CodeBERT (Feng et al., 2020) and GraphCodeBERT (Guo et al., 2020) in order to provide fair comparisons: 12 layers, 12 attention heads and a hidden dimension of 768. We also train a model with the same parameters as TransCoder (see Figure 4.4).

Training dataset As in Chapter 3, we use the GitHub public dataset available on Google BigQuery and select all Python and Java files within the projects with licenses authorizing use for research purposes. Following Lopes et al. (2017) and Allamanis (2019), we remove duplicate files. We also ensure that each fork belongs to the same split as its source repository. We obfuscate each file and create the corresponding dictionary of masked identifier names, resulting in a parallel (obfuscated file - dictionary) dataset of 19 GB for Python and 26 GB for Java. We show some statistics about this dataset in Table 4.1. For comparison purposes, we apply either the BPE codes used in Chapter 3 or in Feng et al. (2020). In practice, we train only on files containing less than 2000 tokens, which corresponds to more than 90% and 80% of the Java and Python files respectively.

Table 4.1: **Dataset statistics.**

	Java	Python
All - Size	26 GB	19 GB
All - Nb files	7.9M	3.6M
Av. nb of tokens / file	718	1245
Av. nb of identifiers / file	25.9	41.8

Training details We train DOBF to translate obfuscated files into lists of identifier names. During DOBF training, we alternate between batches of Java and Python composed of 3000 tokens per GPU. We optimize DOBF with the Adam optimizer (Kingma and Ba, 2015) and an inverse square-root learning rate scheduler (Vaswani et al., 2017). We implement our models in PyTorch (Paszke et al., 2019) and train them on 32 V100 GPUs for eight days. We use float16 operations to speed up training and to reduce the memory usage of our models. We try different initialization schemes: training from scratch and with a Python-Java MLM model like in Chapter 3. We train DOBF with three different obfuscation probability

```
def FUNC_0(VAR_0, VAR_1):
    VAR_2 = [VAR_1]
    VAR_3 = [VAR_1]
    while VAR_3:
        VAR_4 = VAR_3.pop(0)
        for VAR_5 in VAR_0[VAR_4]:
            if (VAR_5 not in VAR_2):
                VAR_2.add(VAR_5)
                VAR_3.append(VAR_5)
    return VAR_2
```

```
def bfs(graph, start):
    visited = [start]
    queue = [start]
    while queue:
        node = queue.pop(0)
        for neighbor in graph[node]:
            if (neighbor not in visited):
                visited.add(neighbor)
                queue.append(neighbor)
    return visited
```

Figure 4.2: **Full deobfuscation of a breadth-first-search function by DOBF.** The code on the left has been fully obfuscated. The code on the right was recovered using DOBF by replacing the function name and every variable name using the generated dictionary. DOBF is able to suggest relevant function and variable names. It makes the code much more readable and easier to understand.

parameters: $p_{obf} \in \{0, 0.5, 1\}$. For each p_{obf} value, we train models with multiple initial learning rates ranging from 10^{-4} to 3.10^{-4} and select the best one using the average subtoken F1 score computed on the validation dataset.

Fine-tuning details Depending on the fine-tuning tasks, we consider different model architectures: seq2seq models with encoder and decoder, architectures with two encoders or a single encoder. In all cases, we initialize the encoders of these models with the encoder of DOBF and fine-tune all parameters. For fair comparison, we rerun all baselines, and train models with the same architectures, number of GPUs, batch sizes and optimizers. For CodeXGLUE, we noticed that the tasks are quite sensitive to the learning rate parameter used during fine-tuning. We perform a grid search on five learning rate parameters ranging from 5.10^{-6} to 10^{-4} and we select the best parameter on the validation dataset. For TransCoder, we use a learning rate of 10^{-4} as in Chapter 3 and we train the models for 2 day on 32 Tesla V100 GPUs.

4.4 Results

4.4.1 Deobfuscation

In Table 4.2, we evaluate the ability of our model to recover identifier names, either when only one identifier is obfuscated ($p_{obf} = 0$) or when all identifiers are obfuscated ($p_{obf} = 1$), for models trained with $p_{obf} \in \{0, 0.5, 1\}$. Even when evaluating with $p_{obf} = 0$, training with $p_{obf} = 0$ is less efficient than $p_{obf} = 0.5$ since the model is only trained to generate a single variable for each input sequence. Training with $p_{obf} = 0.5$ is a more difficult task that requires the model to learn and understand more about code semantics. Forcing the model to understand the structure of the

Table 4.2: **Results on partial and full deobfuscation.** Token accuracy and subtoken F1 score of DOBF evaluated with $p_{obf} = 0$ (i.e. name proposal, where a single token is obfuscated) and $p_{obf} = 1$ (i.e. full deobfuscation, where all tokens are obfuscated). We consider models trained with different obfuscation probabilities p_{obf}. $\text{DOBF}_{0.5}$ performs well for both tasks, and it even performs better than DOBF_0 for Identifier Name Proposal. DOBF_0 and DOBF_1 perform poorly when evaluated on other p_{obf} parameters. Pre-training DOBF with MLM further improves the performance.

	Eval $p_{\text{obf}} = 0$		Eval $p_{\text{obf}} = 1$	
	Acc	F1	Acc	F1
DOBF_0	56.3	68.0	0.4	0.9
$\text{DOBF}_{0.5}$	61.1	71.2	41.8	54.8
DOBF_1	18.1	27.0	45.6	58.1
$\text{DOBF}_{0.5}$ init MLM	**67.6**	**76.3**	45.7	58.0
DOBF_1 init MLM	20.0	28.3	**49.7**	**61.1**

code may be useful even when testing with $p_{obf} = 0$, as some identifier names cannot be guessed only from the names of other identifiers. When DOBF has to recover a fully obfuscated function, it obtains the best accuracy when trained with $p_{obf} = 1$. It manages to recover 45.6% of the initial identifier names. We also observe that, for every configuration, initializing DOBF with MLM improves the performance.

Figure 4.2 shows an example of a fully obfuscated function recovered by our model. DOBF successfully manages to understand the purpose of the function and to predict appropriate variable names. Figure 4.4 shows examples of function name proposal by DOBF for functions implementing matrix operations in Python. We observe that DOBF manages to identify the key tokens and to properly infer the purpose of similar but very different functions. Figures 4.5, 4.6, and 4.7 show additional examples of function name proposals by DOBF in Java and Python. Figure 4.8 shows additional examples where we show that DOBF also leverages non-obfuscated identifier names to understand the meaning of input functions. Figures 4.9 and 4.3 show examples of deobfuscation of fully obfuscated Python code snippets using DOBF. It is able to understand the semantics and purposes of a variety of obfuscated classes and functions, including a LSTM cell.

Table 4.3: **Results on downstream tasks for different pre-training configurations.** Models pre-trained with DOBF initialized with MLM significantly outperform both CodeBERT and models trained with MLM only. DOBF+DAE outperforms other models on every task but clone detection, on which CodeBERT scores much higher than our MLM. It outperforms GraphCodeBERT by 0.02 MRR (+5.3%) on natural language code search (NLCS), and by 4.6% in Java → Python computational accuracy with beam size 10 (+12.2% correct translations). The tasks where MLM provides large improvements over the transformer baseline (first row, no pre-training) are also the tasks where DOBF provides the largest gains (clone detection, NL code search, unsupervised translation). The DAE baseline (initialized with MLM) already provides substantial improvements over MLM on most tasks and yields the best results for Python to Java translation while its results are poor for Java to Python.

	Clone Det (F1 score)	Code Sum Java (BLEU)	Code Sum Python (BLEU)	NLCS (MRR)	Python→Java (CA@1)		Java→Python (CA@1)	
					k=1	k=10	k=1	k=10
Transformer	88.1	16.6	16.4	0.025	24.0	28.4	29.0	29.7
MLM	91.9	18.6	18.0	0.308	44.8	45.4	34.5	35.6
DAE	96.3	19.2	18.3	0.380	**48.3**	**49.2**	32.1	32.8
CodeBERT	96.5	18.3	18.2	0.315	40.8	45.6	36.5	36.7
GraphCodeBERT	96.4	18.8	18.5	0.377	44.3	44.1	35.6	37.8
DOBF init scratch	**96.5**	18.2	17.5	0.272	43.9	44.1	35.2	34.7
DOBF	95.9	19.1	18.2	0.383	43.5	44.1	38.7	40.0
DOBF+DAE	95.8	**19.4**	**18.6**	**0.397**	46.6	47.3	**40.6**	**42.4**

4.4.2 Downstream tasks

Our results on downstream tasks using the same architecture as CodeBERT and GraphCodeBERT are shown in Table 4.3 and discussed below. Our results using the architecture of TransCoder are shown on Table 4.4. For fine-tuning, we considered models pre-trained with $p_{obf} = 0.5$ and $p_{obf} = 1$. Since they gave very similar results on downstream tasks, we only use models pre-trained with $p_{obf} = 0.5$ in the rest of this thesis. We initialize DOBF with MLM as it leads to better performance on our deobfuscation metrics. We still consider DOBF initialized randomly as a baseline in Table 4.3. We also consider a version where DOBF is trained together with a denoising auto-encoding (DAE) objective (Vincent et al., 2008), which was shown to be effective at learning code representations in Chapter 3. With DAE, the model is trained to recover the original version of a sequence which has been corrupted (by removing and shuffling tokens). As baselines, we consider a randomly initialized model and a model pre-trained with MLM only, and a model pre-trained with denoising and initialized with MLM. For CodeXGLUE tasks, we also consider

CodeBERT as a baseline. We compare results for DOBF trained from scratch and DOBF initialized with MLM, and report results in Table 4.3. The randomly initialized model is useful to measure the importance of pre-training on a given task. Pre-training is particularly important for the NLCS task: without pre-training, the model achieves a performance of 0.025 MRR while it goes up to 0.308 with MLM pre-training. The main differences between our MLM baseline and CodeBERT, are that 1) CodeBERT was trained on a different dataset which contains functions with their documentation, 2) it uses an additional RTD objective, and 3) is initialized from a RoBERTa model. Although code summarization and NL code search involve natural language and may benefit from CodeBERT's dataset that contains code documentation, we obtained very similar results on this task using a simpler dataset. However, our MLM baseline did not match their performance on clone detection. We also tried to initialize our MLM model with RoBERTa, but did not observe any substantial impact on the performance on downstream tasks.

The models based on DOBF obtain state-of-the-art results on all downstream tasks, outperforming GraphCodeBERT, CodeBERT and MLM. The deobfuscation objective is already effective as a pre-training task. Even when initialized randomly, it leads to results comparable to MLM on most tasks and is much more effective on clone detection. The DOBF+DAE model outperforms MLM on all downstream tasks, the major improvement being for NL code search, which is also the task that benefited the most from MLM pre-training For unsupervised translation, DOBF+DAE increases the computational accuracy by 1.9% when translating from Python to Java, and by 6.8% when translating from Java to Python with beam size 10. Also, DOBF beats CodeBERT by a wide margin on NL code search and code summarization, showing that programming language data aligned with natural language is not necessary to train an effective model on those tasks. DOBF initialized with MLM and combined with DAE yields higher scores than both DOBF alone and MLM, on most tasks. It shows that objectives such as MLM and DAE that provide unstructured noise are complementary to DOBF.

Table 4.4: **Results on downstream tasks with the architecture of TransCoder.** This architecture has less layers (6 instead of 12), a higher embedding dimension (1024 instead of 768) and less activation heads (8 instead of 12) resulting in a slightly larger model (143M parameters instead of 126M). It also uses ReLU activations instead of GELU. Models pre-trained with MLM and DOBF significantly outperform both CodeBERT and models trained with MLM only. MLM+DOBF outperforms CodeBERT by 7% on natural language code search (NLCS), and MLM by 6% in Java → Python computational accuracy. It also beats CodeBERT on every task except Clone Detection, on which CodeBERT scores much higher than our MLM. GraphCodeBERT only beats our model on python summarization and Python to Java translation by a shallow margin and is below on other tasks. The tasks where MLM provides large improvements over the transformer baseline (first row) are also those where DOBF provides the largest gains (i.e. clone detection, natural language code search, and unsupervised translation).

	Clone Det (F1 score)	Sum Java (BLEU)	Sum Py (BLEU)	NLCS (MRR)	Py→Ja (CA@1)		Ja→Py (CA@1)	
					k=1	k=10	k=1	k=10
Transformer	88.1	16.6	16.4	0.025	37.6	38.9	31.8	42.1
CodeBERT	96.5	18.3	18.2	0.315	-	-	-	-
GraphCodeBERT	96.4	18.8	**18.5**	0.377	-	-	-	-
MLM	91.9	18.6	18.0	0.308	40.3	42.2	44.7	46.6
DOBF	**96.5**	18.2	17.5	0.272	38.9	**45.7**	44.7	46.4
MLM+DOBF	95.9	**19.1**	18.2	**0.383**	**43.5**	44.9	**49.2**	**52.5**

4.5 Deobfuscation examples

Input Code	Deobfuscated Identifiers	
`def FUNC_0(VAR_0, VAR_1):` `  return sum(map(operator.mul, VAR_0, VAR_1))`	FUNC_0 VAR_0 VAR_1	dotProduct list1 list2

`def FUNC_0(VAR_0):`
`  VAR_1 = urllib2.urlopen(VAR_0)`
`  VAR_2 = VAR_1.read()`
`  return VAR_2`
	FUNC_0
	VAR_0
	VAR_1
	VAR_2
		get_html
		url
		response
		html

`def FUNC_0(VAR_0):`
`  VAR_1 = set(VAR_0)`
`  return (len(VAR_1) == len(VAR_0))`
	FUNC_0
	VAR_0
	VAR_1
		all_unique
		iterable
		s

`def FUNC_0(VAR_0, VAR_1):` `  return list(collections.deque(VAR_0, maxlen=VAR_1))`	FUNC_0 VAR_0 VAR_1	tail s n
`def FUNC_0(VAR_0):` `  return sum((VAR_1 for VAR_1 in VAR_0 if ((VAR_1 % 2) == 0)))`	FUNC_0 VAR_0 VAR_1	even_sum nums n

Figure 4.3: **Examples of full deobfuscations of Python functions.** Even when every identifier is obfuscated, DOBF is able to propose relevant names. The proposed function name is informative and relevant in all examples since the first function computes a dot product, the second downloads a HTML page and returns its content, the third evaluates whether the input contains only unique elements, the fourth computes the tail of an iterable, and the fifth computes the sum of the even elements of an iterable.

Input Code	Function Name Proposals	

`def FUNC_0(m1, m2):`
`  assert m1.shape == m2.shape`
`  n, m = m1.shape`
`  res = [[0 for _ in range(m)]`
`  for _ in range(n)]`
`  for i in range(n):`
`  for j in range(m):`
`  res[i][j] = m1[i][j] + m2[i][j]`
`  return res`
	matrix_add
	matrixAdd
	matrixadd
	matrix_sum
	matrix_addition
		25.9%
		22.5%
		18.8%
		16.7%
		16.1%

`def FUNC_0(m1, m2):`
`  assert m1.shape == m2.shape`
`  n, m = m1.shape`
`  res = [[0 for _ in range(m)]`
`  for _ in range(n)]`
`  for i in range(n):`
`  for j in range(m):`
`  res[i][j] = m1[i][j] - m2[i][j]`
`  return res`
	matrix_sub
	matrix_subtract
	matrix_subtraction
	sub
	sub_matrix
		26.1%
		21.5%
		19.7%
		17.6%
		15.0%

`def FUNC_0(matrix):`
`  n, _ = matrix.shape`
`  for i in range(n):`
`  for j in range(i,n):`
`  matrix[i][j], matrix[j][i] = \`
`  matrix[j][i], matrix[i][j]`
	transpose
	rotate
	rotate_matrix
	symmetric
	rotate_matrix_by_row
		36.7%
		29.5%
		17.1%
		8.9%
		7.7%

`def FUNC_0(m1, m2):`
`  n1, m1 = m1.shape`
`  n2, m2 = m2.shape`
`  assert n2 == m1`
`  res = [[0 for _ in range(m2)]`
`  for _ in range(n1)]`
`  for i in range(n1):`
`  for j in range(m2):`
`  res[i][j] = sum([m1[i][k] * m2[k][j]`
`  for k in range(n2)])`
`  return res`
	matrix_product
	mat_mult
	matmul_mat
	matprod
	matrixProduct
		28.8%
		23.8%
		17.0%
		16.0%
		14.4%

Figure 4.4: **Examples of function name proposals for matrix operations in Python.** DOBF is able to find the right name for each matrix operation, showing that it learned to attend to the most important parts of the code. Even when the functions are similar, DOBF successfully and confidently (c.f. scores) understands the semantics of the function and its purpose.

Input Code	Proposed Function Name	

`public static void FUNC_0(String path){`
`  try {`
`  Files.delete(path);`
`  }`
`  catch (Exception e) {`
`  System.err.println("Error deleting file " + path);`
`  }`
`}`
	deleteFile
	remove
	DeleteFile
	removeFile
	deleteFileQuietly
		48.3%
		16.9%
		13.2%
		13.1%
		8.4%

`public static void FUNC_0(String path){`
`  if (!Files.exists(path)) {`
`  Files.createDirectories(path);`
`  }`
`}`
	createDir
	createDirectory
	createDirIfNotExists
	ensureDirectoryExists
	createDirectoryIfNotExists
		23.5%
		20.9%
		20.8%
		18.5%
		16.3%

`public static List<Pair<String, Double>> FUNC_0(List<String> list1,`
`  List<Double> list2)`
`{`
`  return IntStream.range(0, Math.min(list1.size(), list2.size()))`
`  .mapToObj(i -> new Pair<>(list1.get(i), list2.get(i)))`
`  .collect(Collectors.toList());`
`}`
	zip
	intersect
	combine
	merge
	intersection
		28.6%
		20.0%
		17.9%
		17.5%
		16.0%

`public static int FUNC_0(int n){`
`  int a = 0, b = 1;`
`  int tmp;`
`  for (int i = 0; i < n; i ++){`
`  tmp = a + b;`
`  a = b;`
`  b = tmp;`
`  }`
`  return a;`
`}`
	fib
	fibonacci
	fibon
	fibo
	fibonacci_series
		41.5%
		36.6%
		9.1%
		8.8%
		4.0%

`public static float FUNC_0(List<Float> vec1,`
`  List<Float> vec2) {`
`  float size = vec1.size();`
`  assert size == vec2.size();`
`  float result = 0.0f;`
`  for (int i = 0; i < size; i++) {`
`  result += vec1.get(i) * vec2.get(i);`
`  }`
`  return result;`
`}`
	dotProduct
	dot
	dot_product
	dotproduct
	inner
		40.9%
		23.9%
		16.5%
		10.5%
		8.3%

Figure 4.5: **Examples of name proposal in Java.** DOBF is able to suggest relevant function names for a variety of Java methods and demonstrates its ability to understand the semantics of the code. In the first two examples, the first element in the beam shows that it is able to select relevant names in the context to find a function name: it uses `Files.delete` and `Files.createDirectories` to suggest the tokens `deleteFile` and `createDir`. DOBF finds relevant names for Java methods without copying any part of the other tokens. For example for the third method combining two lists as in the python `zip` function, for the fourth method which computes the n-th element of the Fibonacci series and for the last method which computes the dot product between two vectors.

Input Code	Proposals for Highlighted Identifiers	
`def FUNC_0(name):` `  return os.environ[name]`	get_env get_envvar env getenv get_env_variable	25.3% 19.3% 19.2% 18.5% 17.7%
`def FUNC_0(l):` `  return list(set(l))`	unique remove_duplicates removeDuplicates uniquify unique_items	24.8% 23.8% 18.8% 18.7% 13.8%

`def FUNC_0(path):`
`  with gzip.open(path, 'rb') as f:`
`  content = f.read()`
`  return content`
	read_gzip_file
	read_gzip
	ungzip
	gzip_content
	gzip_read
		22.9%
		22.1%
		20.8%
		18.2%
		16.0%

`def FUNC_0(n):`
`  v = [True for i in range(n + 1)]`
`  p = 2`
`  while (p * p <= n):`
`  if (v[p] == True):`
`  for i in range(p * 2, n + 1, p):`
`  v[i] = False`
`  p += 1`
`  v[0]= False`
`  v[1]= False`
`  return [p for p in range(n+1) if v[p]]`
	sieve
	prime_sieve
	sieve_of_eratosthenes
	primes
	eratosthenes
		36.1%
		18.5%
		15.5%
		15.3%
		14.5%

`def f(n):`
`  VAR_0 = [True for i in range(n + 1)]`
`  p = 2`
`  while (p * p <= n):`
`  if (VAR_0[p] == True):`
`  for i in range(p * 2, n + 1, p):`
`  VAR_0[i] = False`
`  p += 1`
`  VAR_0[0]= False`
`  VAR_0[1]= False`
`  return [p for p in range(n+1) if VAR_0[p]]`
	prime
	l
	isPrime
	a
	primes
		30.6%
		20.5%
		18.0%
		16.4%
		14.6%

Figure 4.6: **Examples of name proposal in Python.** Our model trained with DOBF goes well beyond copying tokens from the context. For instance, in the first example, it understands that this function is used to get environment variables. In the second example, it proposes names related to what this function actually does (removing duplicates in a list) instead of the individual operations it uses (converting to set and then to list). The last two rows show proposals for two different identifiers in a function computing the list of prime numbers below n using the sieve of Eratosthenes. The proposals for the function name are all relevant, and the third one names exactly the algorithm which is used. The variable `v` is a list of booleans. At the end of the algorithm, `v[i]` is true if and only if `i` is prime. The proposed names `prime` and `isPrime` are very relevant as they describe what the list contains. Although `l` and `a` are not very informative, they indicate that the variable is a list or an array.

Input Code	Proposed Function Name	

`def FUNC_0(v1, v2):`
`  assert len(v1) == len(v2)`
`  return [a * b for a, b in zip(v1, v2)]`
	multiply_lists
	multiply_list
	multiply
	multiply_vectors
	mul
		28.7%
		23.5%
		18.1%
		14.9%
		14.8%

`def FUNC_0(v1, v2):`
`  assert len(v1) == len(v2)`
`  return sum([a * b for a, b in zip(v1, v2)])`
	dotproduct
	dot_product
	dotProduct
	dot
	multiply_by_addition
		34.8%
		19.2%
		18.1%
		15.6%
		12.3%

`def FUNC_0(v1, v2):`
`  assert len(v1) == len(v2)`
`  return [a ^ b for a, b in zip(v1, v2)]`
	xor
	XOR
	vector_xor
	xors
	xor_lists
		62.9%
		12.8%
		10.8%
		7.4%
		6.1%

`def FUNC_0(v1, v2):`
`  assert len(v1) == len(v2)`
`  return [a ** b for a, b in zip(v1, v2)]`
	power
	list_power
	lcm
	power_list
	powersum
		29.8%
		20.9%
		19.9%
		15.1%
		14.3%

`def FUNC_0(v1, v2):`
`  assert len(v1) == len(v2)`
`  return [a + b for a, b in zip(v1, v2)]`
	add_lists
	add
	sum_lists
	list_concat
	list_add
		27.0%
		22.9%
		17.9%
		17.7%
		14.5%

`def FUNC_0(v1, v2):`
`  assert len(v1) == len(v2)`
`  return [a - b for a, b in zip(v1, v2)]`
	minus
	subtract
	difference
	subtract_lists
	substract
		30.4%
		29.8%
		14.1%
		13.3%
		12.4%

Figure 4.7: **Examples of function name proposal in Python using DOBF.** DOBF is able to identify the key tokens in each function, to properly infer its purpose, and to suggest appropriate names along with a confidence score. In particular, even though the first two code snippets are very similar in terms of edit distance, they implement very different functions and DOBF is able to name them appropriately.

BFS Implementation

```
def FUNC_0(graph, node):
  visited = [node]
  VAR_0 = [node]
  while VAR_0:
    s = VAR_0.pop(0)
    for neighbour in graph[s]:
      if neighbour not in visited:
        visited.add(neighbour)
        VAR_0.append(neighbour)
  return visited
```

```
FUNC_0 bfs | VAR_0 queue
```

DFS Implementation

```
def FUNC_0(graph, node):
  visited = [node]
  VAR_0 = [node]
  while VAR_0:
    s = VAR_0.pop()
    for neighbour in graph[s]:
      if neighbour not in visited:
        visited.add(neighbour)
        VAR_0.append(neighbour)
  return visited
```

```
FUNC_0 dfs | VAR_0 stack
```

DFS with Erroneous Variable Name

```
def FUNC_0 (graph, node):
  visited = [node]
  queue = [node]
  while queue:
    s = queue.pop()
    for neighbour in graph[s]:
      if neighbour not in visited:
        visited.append(neighbour)
        queue.append(neighbour)
  return visited
```

```
FUNC_0 bfs
```

Figure 4.8: **Deobfuscation on graph traversal functions.** These three functions perform graph traversals. The only difference between the first and the second function is that the first uses a queue to select the next element (`.pop(0)`) while the second uses a stack (`.pop()`). The first function implements a breadth-first search (bfs) in the graph and the second implements a depth-first search (dfs). DOBF is able to find the right function and variable names in each case. In the last function, we replaced the anonymized `VAR_0` variable with `queue` in the implementation of depth-first search. This erroneous information leads DOBF to believe that this function performs breadth-first search. It shows that, just like human programmers, DOBF uses the names of the other variables to understand programs and choose relevant identifier names. When working on code with misleading identifier names, it is often preferable to obfuscate several identifiers.

Obfuscated Code

```
class CLASS_0(nn.Module):

    def __init__(VAR_0, VAR_1, VAR_2, VAR_3):
        super(CLASS_0, VAR_0).__init__()
        VAR_0.VAR_1 = VAR_1
        VAR_0.VAR_2 = VAR_2
        VAR_0.VAR_4 = nn.Linear(VAR_1, (4 * VAR_2), bias=VAR_3)
        VAR_0.VAR_5 = nn.Linear(VAR_2, (4 * VAR_2), bias=VAR_3)
        VAR_0.FUNC_0()

    def FUNC_0(VAR_6):
        VAR_7 = (1.0 / math.sqrt(VAR_6.VAR_8))
        for VAR_9 in VAR_6.VAR_10():
            VAR_9.data.uniform_((- VAR_7), VAR_7)

    def FUNC_1(VAR_11, VAR_12, VAR_13):
        (VAR_14, VAR_15) = VAR_13
        VAR_14 = VAR_14.view(VAR_14.size(1), (- 1))
        VAR_15 = VAR_15.view(VAR_15.size(1), (- 1))
        VAR_12 = VAR_12.view(VAR_12.size(1), (- 1))
        VAR_16 = (VAR_11.VAR_4(VAR_12) + VAR_11.VAR_5(VAR_14))
        VAR_17 = VAR_16[:, :(3 * VAR_11.VAR_8)].sigmoid()
        VAR_18 = VAR_16[:, (3 * VAR_11.VAR_8):].tanh()
        VAR_19 = VAR_17[:, :VAR_11.VAR_8]
        VAR_20 = VAR_17[:, VAR_11.VAR_8:(2 * VAR_11.VAR_8)]
        VAR_21 = VAR_17[:, (- VAR_11.VAR_8):]
        VAR_22 = (th.mul(VAR_15, VAR_20) + th.mul(VAR_19, VAR_18))
        VAR_23 = th.mul(VAR_21, VAR_22.tanh())
        VAR_23 = VAR_23.view(1, VAR_23.size(0), (- 1))
        VAR_22 = VAR_22.view(1, VAR_22.size(0), (- 1))
        return (VAR_23, (VAR_23, VAR_22))
```

Code Deobfuscated using DOBF

```
class LSTM(nn.Module):

    def __init__(self, input_size, hidden_size, bias):
        super(LSTM, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.h1 = nn.Linear(input_size, (4 * hidden_size), bias=bias)
        self.h2 = nn.Linear(hidden_size, (4 * hidden_size), bias=bias)
        self.init_weights()

    def init_weights(self):
        stdv = (1.0 / math.sqrt(self.hidden_size))
        for m in self.modules():
            m.data.uniform_((- stdv), stdv)

    def forward(self, x, prev_state):
        (prev_h, prev_c) = prev_state
        prev_h = prev_h.view(prev_h.size(1), (- 1))
        prev_c = prev_c.view(prev_c.size(1), (- 1))
        x = x.view(x.size(1), (- 1))
        h = (self.h1(x) + self.h2(prev_h))
        s = h[:, :(3 * self.hidden_size)].sigmoid()
        c = h[:, (3 * self.hidden_size):].tanh()
        r = s[:, :self.hidden_size]
        g = s[:, self.hidden_size:(2 * self.hidden_size)]
        o = s[:, (- self.hidden_size):]
        c = (th.mul(prev_c, g) + th.mul(r, c))
        h = th.mul(o, c.tanh())
        h = h.view(1, h.size(0), (- 1))
        c = c.view(1, c.size(0), (- 1))
        return (h, (h, c))
```

Figure 4.9: **Deobfuscation of an LSTM cell.** DOBF is able to recover several of the original tokens, including the class name (`LSTM`) and the full signature of the `__init__` method. The table of ground truth and recovered tokens is provided in Table 4.5. Even though DOBF does not always recover the original token, it generally proposes very relevant tokens which improves code readability. In particular, for some tokens the accuracy and subtoken scores would be zero but the recovered tokens are still very relevant. For instance, `reset_parameters` (`FUNC_0`) was renamed to `init_weights`, `std` (`VAR_7`) was renamed to `stdv`, and `hidden` (`VAR_13`) was renamed to `prev_state`. In those instances, the original and recovered tokens share no subtoken despite having very similar semantics.

ID	Ground Truth	DOBF
CLASS_0	LSTM	LSTM
FUNC_0	reset_parameters	init_weights
FUNC_1	forward	forward
VAR_0	self	self
VAR_1	input_size	input_size
VAR_2	hidden_size	hidden_size
VAR_3	bias	bias
VAR_4	i2h	h1
VAR_5	h2h	h2
VAR_6	self	self
VAR_7	std	stdv
VAR_8	hidden_size	hidden_size
VAR_9	w	m
VAR_10	parameters	modules
VAR_11	self	self
VAR_12	x	x
VAR_13	hidden	prev_state
VAR_14	h	prev_h
VAR_15	c	prev_c
VAR_16	preact	h
VAR_17	gates	s
VAR_18	g_t	c
VAR_19	i_t	r
VAR_20	f_t	g
VAR_21	o_t	o
VAR_22	c_t	c
VAR_23	h_t	h

Table 4.5: Table of ground truth and recovered tokens for the obfuscated LSTM cell shown in Figure 4.9. While the accuracy of the model is far from perfect, the tokens it retrieves generally make sense and facilitate the comprehension of the code.

4.6 Conclusion

In this chapter, we introduce a new deobfuscation objective and show that it can be used for three purposes: recover fully obfuscated code, suggest relevant identifier names, and pre-train transformer models for programming language related tasks. Although it does not require any parallel corpora of source code aligned to natural language, methods based on DOBF outperform GraphCodeBERT, CodeBERT and MLM pre-training on multiple downstream tasks, including clone detection, code summarization, natural language code search, and unsupervised code translation. These results show that DOBF leverages the particular structure of source code to add noise to the input sequence in a particularly effective way. Other noise functions or surrogate objectives adapted to source code may improve the performance further. For instance, by training a model to find the type of given variables, the signature of a method, or to repair a piece of code which has been corrupted.

Since models pre-trained on source code benefit from structured noise, it would be interesting to see whether these findings can be applied to natural languages as well, where identifiers could be seen as analogous to named entities. More generally, natural languages also have an underlying structure. Leveraging the constituency or dependency parse trees of sentences (as opposed to abstract syntax trees in programming languages) may help designing better pre-training objectives for natural languages.

This method still relies only on anchor words to align the representations and learn to translate without any parallel data. However, source code can also be compiled and executed, and software developers use this type of signals in their daily work. For instance, using unit tests to verify the semantics of a function on a series of carefully-chosen examples is a standard procedure in software engineering. In the next chapter, we show how to leverage this type of signal to improve unsupervised translation methods for source code.

Chapter 5

Leveraging Automated Unit Tests for Unsupervised Code Translation

TransCoder (see Chapter 3) showed that unsupervised methods can be used to translate source code. However, it is trained without any supervised signal and only learns the semantics of tokens from their contexts. As shown in Figure 5.1, it can confuse tokens that have different semantics in different languages, for instance the float division in Python and integer division in C++ and Java which use the token `/` or more subtle operator priority differences (e.g. Java prioritizes `==` over `&`, unlike Python). While small inaccuracies often merely hinder comprehension in natural languages, they often make the entire translation erroneous in the context of programming languages.

TransCoder leverages back-translation (Sennrich et al., 2015a), an effective data-augmentation scheme where the model translates source sequences to generate training data for the target-to-source direction, and vice versa. Although being highly effective in low-resource translation, back-translation also has issues, as the model is trained on potentially invalid input-output pairs. Neural machine translation models being highly sensitive to input noise (Belinkov and Bisk, 2018; Khayrallah and Koehn, 2018), this can severely deteriorate the performance. Fortunately, many programming languages come with relatively mature tools and technologies for automated test data generation. In this chapter, we propose to leverage these tools to guide the translation process, weeding out unsuccessful translations, thereby increasing the overall confidence in the machine translation process.

The topic of automated test data generation has been active for over three decades in the software engineering research community (Myers, 1979; Miller and Spooner,

Input function

```
def is_odd(x):
  return x & 1 == 1

static void printb(int x){
  while (x > 0){
    System.out.println(x % 2);
    x /= 2;
  }
}
static String reverse(char[] str){
  Stack <Character> st = new Stack<>();
  for(int i = 0; i<str.length; i++)
    st.push(str[i]);
  for(int i = 0; i<str.length; i++){
    str[i] = st.peek();
    st.pop();
  }
  return String.valueOf(str);
}
```

TransCoder

```
static boolean isOdd(int x) {
  return x & 1 == 1;
}
def printb(x):
  while x > 0:
    print(x % 2)
    x /= 2

def reverse(str):
  st = Stack()
  for i in range(len(str)):
    st.push(str[i])
  for i in range(len(str)):
    str[i] = st.pop()
    st.push(str[i])
  return str
```

TransCoder-ST

```
static boolean isOdd(int x) {
  return ( x & 1 ) == 1;
}
def printb(x):
  while x > 0:
    print(x % 2)
    x //= 2

def reverse(data):
  st = []
  for c in data :
    st.append(c)
  for i in range(len(data)):
    data[i] = st[-1]
    st.pop()
  return ''.join(data)
```

Figure 5.1: **Improvements over TransCoder.** The first function returns whether an input integer is odd and is translated from Python to Java. The translation of TransCoder does not compile because the `==` operator has precedence over `&` in Java, and parentheses are required unlike in Python. The second example is a function that prints an integer in base two, which is translated from Java to Python. TransCoder translates does not modify the expression `x/=2`, even though it corresponds to the integer division in Java and to the float division in Python. In the third example, a function reversing a char array, TransCoder does not manage to translate the Java Stack object into the right Python object and uses the unsafe `str` parameter name. In all three cases, TransCoder-ST (described in this chapter) manages to leverage the semantics contained in unit tests to translate the function correctly.

1976). There are now many existing mature tools for test data generation, both open source research tools (Fraser and Arcuri, 2011; Lakhotia et al., 2013; Cadar et al., 2008), and production testing systems (Alshahwan et al., 2018; Tillmann et al., 2014). Because of its pivotal impact on practical software engineering, automated testing remains a highly active research area (Anand et al., 2013), with the result that future automated testing advances will lead to ongoing improvement in automated translation.

We use one such open source automated test generation tool, EvoSuite (Fraser and Arcuri, 2011), in this thesis. EvoSuite is a well-established test generation tool for Java which uses coverage metrics (Chekam et al., 2017) and mutation scores (Jia and Harman, 2011) to generate high-quality tests. It has been widely used in the Software Testing research literature for test data generation although it has not, hitherto, been used as part of an automated code translation approach, the topic of the present chapter.

More generally, software testing tools have been largely ignored by the machine

learning community (Zhang et al., 2020). In this chapter, we propose to use automatically created unit tests to guide unsupervised translation models for programming languages. More precisely, we create unit tests automatically for a large number of functions from the source dataset. Since the unit tests are composed of simple inputs and asserts, they can easily be translated to semantically equivalent tests in the target languages using simple scripts. Using our unit-tests and a pre-trained unsupervised translation model, we create parallel datasets by translating functions and selecting the translations that have the same semantics as the original function for the tested inputs. In this chapter, we make the following contributions:

- We introduce a novel approach, TransCoder-ST (for Self-Trained), that leverages an automated unit test generation pipeline to filter out invalid translations and reduce the noise coming from the back-translation process in unsupervised machine translation.
- We present two implementations of this approach (online and offline), and show that it significantly outperforms the previous state of the art in code translation on all the language pairs we considered. In particular, we improve the state of the art for translating between Java, Python and C++ by an average of 12.6% Computational Accuracy (CA@1), corresponding to an average relative improvement of 25.5%. For Python $\rightarrow$ C++, we improve the CA@1 by 24%, reducing the error rate by 35.7% compared to previous models.
- We generate multilingual unit tests for hundreds of thousands of Java functions and create a large parallel dataset of 135,000 parallel functions between Java, Python, and C++.
- Our method is completely unsupervised and could easily be generalized to other programming languages and unit test creation tools.

5.1 Context

Unit Test Generation. Software testing is challenging due to the large number of possibilities to be tested, and the inherent cost of covering reasonable representative sample (Myers, 1979). When test design is performed by humans, the cost can be prohibitive. To reduce such cost, much research over the last three decades has focused on automating the process of test generation (Anand et al., 2013). Although

automated test generation has been studied since the mid-1970s (Miller and Spooner, 1976), it was only in the last decade that industrial-strength tools have become widely available. There are now several test data generation tools for languages, including C (Cadar et al., 2008; Lakhotia et al., 2013) and Java (Fraser and Arcuri, 2011). Popular test data generation techniques include symbolic execution of the code (Cadar and Sen, 2013), dynamic execution guided by a fitness function (Harman et al., 2015), and hybrids of these two techniques (Baars et al., 2011). Recently, neural networks have also been used successfully to generate unit tests (Tufano et al., 2020).

One of the most well-established and widely-used open source tools for test data generation is the EvoSuite system (Fraser and Arcuri, 2011). EvoSuite uses search based software engineering (SBSE) (Harman et al., 2012) to generate test cases. Like all SBSE techniques, EvoSuite is guided by fitness functions, in this case aimed at capturing the test suite's coverage and mutation score of the code being tested. We use EvoSuite in our work for three reasons: it is publicly available in open source (thereby facilitating replication), it is under current active development (thereby supporting future work), and it is widely used by other researchers (thereby enabling interoperability). The test framework can be considered as a parameter in our overall approach and could be substituted with another.

In order to assess the effectiveness of the test suites generated, we use mutation testing, a topic also widely-studied since the 1970s (DeMillo et al., 1978). A mutant is a version of the program into which a fault is deliberately inserted, thereby assessing the test suite's fault detection ability (Jia and Harman, 2011; Papadakis et al., 2019). For a given set of mutants and a test suite, the mutation score is defined to be the proportion of mutants for which the test suite distinguishes the behavior of the mutant from that of the original program. The mutation score is thus a proxy for the fault-revealing power of the test suite on a set of simulated faults (the mutants). Mutation scores have been empirically demonstrated to be correlated to real fault revelation (Chekam et al., 2017), motivating our adoption of this approach.

Translation of Programming Languages Several studies used statistical methods to translate between programming languages. Early methods extracted parallel datasets and trained phrase-based models to translate between C# and Java (Nguyen et al., 2013; Karaivanov et al., 2014) or from Python 2 to Python 3 (Aggarwal et al., 2015). Later, Chen et al. (2018) proposed a tree-to-tree neural network to translate between CoffeeScript and JavaScript and between C# and Java using the dataset cre-

ated by Nguyen et al. (2013). However, these approaches are limited to a few language pairs for which small parallel datasets were created manually (e.g. C#-Java) or can be created with rule-based tools (e.g. Python 2-Python 3 and CoffeeScript-JavaScript).

Instead, in Chapter 3, we proposed TransCoder, an unsupervised model that leverages the principles of unsupervised machine translation (Lample et al., 2018c), to translate between Python, Java and C++. We showed that our method outperforms well-established rule-based baselines, does not require any parallel data or expert knowledge, and can easily be generalized to other languages. We pre-trained our model with the Masked Language Modeling (MLM) objective of (Devlin et al., 2018), and trained it with the denoising auto-encoding (DAE) (Vincent et al., 2008) and the back-translation (BT) (Sennrich et al., 2015a) objectives. In Chapter 4, we showed that augmenting MLM with a deobfuscation objective (dubbed DOBF) can substantially improve the performance of TransCoder. In the rest of the chapter, we will refer to the transpiler pre-trained with DOBF as simply DOBF.

Even though unsupervised methods can be trained on large amounts of data, they sometimes lack the signal needed to differentiate between semantically different tokens that often occur in similar contexts (see Figure 5.1). There is a need for a method providing supervised signal directly related to the semantics of the code without manually crafted parallel datasets.

5.2 Method

In this paper, we present the methods we used to automatically generate parallel data and improve code translation models.

5.2.1 Mutation score

In mutation testing, mutants are programs transformed from the original programs based on a series of syntactic transformation rules called mutation operators. Mutation testing consists in introducing minor syntactic faults on the code and running the tests against the mutated code. A strong test suite is expected to detect the code changes by having at least one test failing. Table 5.1 shows the examples of mutation operators adopted in EvoSuite when generating mutants (Fraser and Arcuri, 2015).

A mutant is said to be killed by a test case if the output of this test case on the mutant is different from its output on the original program (i.e., the test fails the mutant). Otherwise, the mutant is said to have survived. Figure 5.5 shows

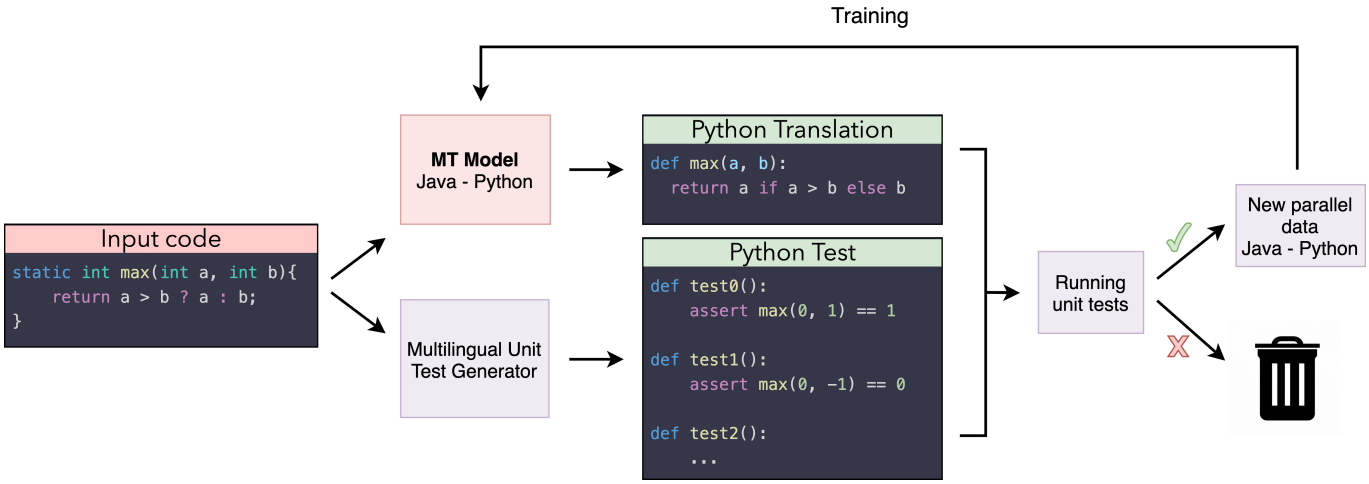

Figure 5.2: **Our iterative self-training method.** Using EvoSuite, we generate unit tests in Java, Python and C++ corresponding to several input Java functions. With a machine translation model (e.g. TransCoder), we generate several candidate translations of the Java function in Python and C++. Generated translations that pass the unit tests are used to create a parallel dataset on which we fine-tune the model. Discarding translations that fail the unit tests reduces the noise of data coming from the back-translation process, and significantly improves the overall performance of the model.

Table 5.1: **Examples of mutation operators in EvoSuite.**

Mutation operator	Explanation
Delete call operator	Remove a method invocation
Delete field operator	Remove a field access and replaces it with a default value (0 / null)
Insert Unary Operator	Add 1 to, subtract 1 from, or negate a numerical value after it was loaded on the stack
Replace arithmetic operator	Replace an arithmetic operator in an expression with other operators. E.g., $+ \rightarrow -$, $* \rightarrow /$
Replace constant operator	Replace constants with the special values -1, 0, +1
Replace variable operator	Replace variables with other variables of the same type

an example of a mutant generated by changing the $<$ in the return statement into $>$. The test with input (-800, -800, -1), as shown by Figure 5.7, does not kill this generated mutant, because its outputs on the original program and the mutant are the same. The mutation score is defined as the percentage of mutants that are killed by the test suite.

Mutation score is considered as the most effective criteria in accessing the fault-revealing ability of test suites. Other criteria, such as code coverage, are weak: they check only whether the test executes the code, but do not check whether the execution result is correct. A test suite without any assertions can achieve 100% code coverage, but could not detect any faults.

Java function

```java
public static int pow (int b, int e) {
  int r = 1;
  while (e > 0) {
    if ((e & 1) == 1)
      r = r * b;
    b = b * b;
    e = e >> 1;
  }
  return r;
}
```

Generated test suite

```java
public void test0()  throws Throwable  {
  int int0 = Example.pow((-1), (-1));
  assertEquals(1, int0);
}

public void test1()  throws Throwable  {
  int int0 = Example.pow(0, 1);
  assertEquals(0, int0);
}

public void test2()  throws Throwable  {
  int int0 = Example.pow((-13133), 2743);
  assertEquals((-1787379173), int0);
}

public void test3()  throws Throwable  {
  int int0 = Example.pow(1, 1);
  assertEquals(1, int0);
}
```

Figure 5.3: **A generated unit test suite with high mutation score.** The mutation score of this test suite is 95% and we selected it in our dataset for pseudo-labelling. The third test case (i.e. `test2`) may be too strict as it would make translations using the python int type fail the unit tests.

Java function

```java
public static int sizeBits_cmd() {
  return 8;
}
```

Generated test suite

```java
public void test0()  throws Throwable  {
  assertEquals(8, Example.sizeBits_cmd());
}
```

Figure 5.4: **A test suite with a good mutation score but only one assert.** Even though it contains only one test and one assert, this test suite tests the semantics of the function on the left properly since it only returns a constant and its mutation score is 100%. We found that test suites with good mutation scores and only one assert generally correspond to uninteresting input functions. Removing these functions and tests from our dataset for self labelling improves the performance of our model.

5.2.2 Parallel data creation

Parallel unit test generation: We use EvoSuite to automatically generate unit tests for Java functions. EvoSuite is a well-established open source tool for automated test generation in Java, which is still under active development and frequently used. It is designed for Java programs but its search-based technique is general and could be used for any programming language. Unit tests can be thought of as lists of inputs and asserts testing the semantics of a program (e.g., the output of the function, the side effects on its arguments such as sorting the input list). EvoSuite uses evolutionary methods to derive tests that maximize criteria such as code coverage or mutation score. During its search, each candidate solution in EvoSuite is a test input.

Original Java function

```
public class CLAMP_CLASS{
    public static double clamp(
    double a, double min, double max){
        return a<min?min:(a>max?max:a);
    }
}
```

Mutant

```
public class CLAMP_CLASS{
    public static double clamp(
    double a, double min, double max){
        return a>min?min:(a>max?max:a);
    }
}
```

Figure 5.5: **A mutant generated by the "Replace arithmetic operator" mutation in EvoSuite.** The $<$ operator in the return statement is replaced with $>$.

The candidate inputs are evolved using crossover and mutation, and filtered by a fitness function (e.g., mutation score). With each generation the fitness improves until it reaches a plateau or the budget is exhausted. The final test inputs are wrapped up as test cases. Each program is associated to a test suite containing a series of test cases. Figure 5.7 shows an example of a test case generated by EvoSuite.

Parallel test suites selection: Some test suites created by EvoSuite only cover a few parts of the semantics of functions. We only trust the translations verified by test suites which examine the function semantics thoroughly. We use the mutation score, which is the most effective test assessment metric in the literature (Jia and Harman, 2011), to pick out these test suites. The mutation score is computed through mutation testing, in which mutants (i.e., program variants with syntactic changes) are generated from the original program based on a set of transformation rules (more details in Section 5.2.1). A mutant is said to be killed if at least one test from the test suite has different results on the mutant and the original program. Otherwise, the mutant is said to survive. The mutation score is the ratio of killed mutants. A test suite with a higher mutation score checks the code semantics more thoroughly. We adopt a strict strategy in test suite selection: we keep only the Unit test suites with a mutation score larger than 90% for building the parallel dataset. In practice, we observe that more than half of the mutation scores are either above 0.9 or below 0.1 for tests generated on our dataset (see Figure 5.6). In practice, we did not observe significant differences on translation performances for mutation score thresholds varying between 0.3 and 0.95.

Parallel dataset building: The generated test suites can be used to test the semantics of programs written in any programming language as long as there is a clear mapping between the types of the output and parameters in the original language and the language of the translated unit tests. We transform the generated

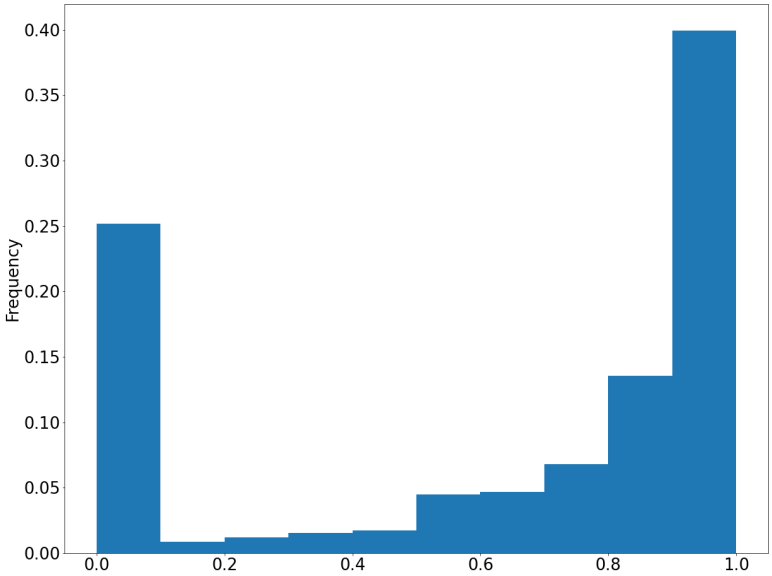

Figure 5.6: **Histogram of mutation scores for our generated unit tests.** We select about 40% of the unit tests with our threshold at 0.9. Many of the remaining unit tests have a mutation score of 0.

Java tests into C++ and Python tests with identical inputs and expected outputs and side effects (i.e., assertions). In practice, we selected the Java functions which can be compiled and run in isolation and with simple output and parameter types. These types are the Java primitive types (e.g. `int`, `long`, `bool`, `float`...), standard data types (e.g. `Integer`, `Double`, `String`...), array and `List` or `ArrayList` types of elements of supported types (e.g. `double[]`, `List<Integer>`...). It makes it easy to map parameter and return types in Java to Python or C++ types in the translated unit tests. While most of the unit tests are translated correctly, the translation sometimes fails due to EvoSuite generating test cases expecting exceptions. Our analysis shows that it happens for about 5.6% of all tests and less than 2% of the tests with high mutation scores. In that case, the candidate translations cannot pass the translated tests and no parallel examples are created.

We use the best unsupervised translation models available for Java to Python and Java to C++ translation, namely TransCoder (see Chapter 3) for Java to C++ and DOBF (see Chapter 4) for Java to Python. For each Java function, we generate 20 Python and C++ translations with beam search and select the first element in

Java function

```java
public class CLAMP_CLASS{
    public static double clamp(
    double a, double min, double max){
        return a<min?min:(a>max?max:a);
    }
}
```

A generated unit test

```java
@Test(timeout = 4000)
  public void test0()  throws Throwable  {
      double double0 = Example.clamp(
      742.0, 0.0, 0.0);
      assertEquals(0.0, double0, 0.01);
  }
```

Figure 5.7: **A unit test generated by EvoSuite.** The Java function clamps the given value a between the given min and max. This test case is not sufficient to test the semantics of the function thoroughly but could be part of a suitable test suite. See Figure 5.3 for a generated test suite with a high mutation score.

the beam that passes the unit tests. The created tests are executed against the translated functions. If all the tests pass, the Python and C++ functions have the same semantics assessed by the generated tests. Our method is illustrated in Figure 5.2.

5.2.3 Training method

Our parallel data generation method relies on a pre-existing model to translate from Java to Python and C++. There is little parallel data for these tasks and the best performing published models are unsupervised. TransCoder (see Chapter 3) is trained using the MLM, denoising and back-translation objectives and is able to translate between Java, C++ and Python. DOBF (see Chapter 4) provides clear improvements over TransCoder for translating between Java and Python but was not trained on C++. Therefore, we use DOBF to translate from Java to Python and TransCoder to translate from Java to C++. When fine-tuning, we also reload these models. For DOBF, we initialize the C++ language embeddings with those of Java.

The parallel examples we generate can be used to improve the performance of pre-existing translation models. Since the number of examples we generate also depends on the performance of the translation model, it creates a positive feedback loop where improving the model allows to improve the parallel dataset which in turn can be used to improve the model again. We propose offline and online approaches to use our method to maximize the unsupervised translation performance.

Offline training. With the offline training method, we use the method described in Section 5.2.2 to create parallel Java $\leftrightarrow$ Python, Java $\leftrightarrow$ C++ and Python $\leftrightarrow$ C++ datasets using every input Java function we selected. For the first iteration, we fine-tune the model on these parallel examples until convergence. We can iterate this

Table 5.2: **Size of the parallel datasets generated offline at each iteration.**

Languages	First iteration	Second iteration	Third iteration	Fourth iteration
Java ↔ C++	27,875	37,769	47,729	60,495
Java ↔ Python	33,496	43,194	43,956	45,311
C++ ↔ Python	14,935	21,026	27,080	32,869

process by selecting the best checkpoints for Java → Python and Java → C++ using the validation dataset and using them to generate new parallel datasets, which can in turn be used to train a better model. We iterate this process until convergence, i.e. when we see no significant improvements on the validation set.

Online training. With the online method, we create parallel examples on the fly while training the model. Compared to the offline method, it allows to always use the latest model to generate new examples and it is much more convenient to automate. However, this process can be unstable if done naively. For instance, the model can start over-fitting only a few examples and stop generating anything that passes the unit tests for any other example. In order to stabilize the training, we follow Likhomanenko et al. (2020) and implement a cache mechanism storing the previous examples that passed the unit tests. At each step, the model can either train on parallel functions sampled from the cache or create new parallel functions to add to the cache. When an example is sampled, we remove it from the cache with a given probability. The online training allows the model to always benefit from the performance of the latest model and the cache mechanism ensures that the model does not forget the correct examples that it was able to generate at previous time steps.

5.2.4 Evaluation

In the context of natural languages, machine translation models are generally benchmarked against a reference solution using the BLEU score (Koehn, 2009; Bahdanau et al., 2015; Vaswani et al., 2017). Early studies on source code translation used the same metric to evaluate the quality of the generated functions (Nguyen et al., 2013; Karaivanov et al., 2014; Aggarwal et al., 2015; Miceli-Barone and Sennrich, 2017), or the exact match score which requires the translation to be exactly equal to the ground truth (Chen et al., 2018). However, these metrics fail to capture the semantics of the code and typically correlate poorly with the correctness of

the generated function, prompting the use of new metrics checking if the generated solution passes series of test cases (see Chapter 3, Kulal et al. (2019); Hendrycks et al. (2021); Chen et al. (2021a); Drain et al. (2021)).

We evaluate our models on the full validation and test sets of TransCoder. It contains a few hundreds of parallel functions extracted from GeeksforGeeks along with associated unit tests. As our TransCoder and DOBF baselines, we evaluate our models with the CA@N metric, which checks if any of the top-N solutions proposed by the model passes all the corresponding unit tests. This metric can be computed independently of the beam size (as long as the beam size is greater or equal to N).

5.3 Experiments

5.3.1 Training details

Model architecture. We use a sequence-to-sequence model with attention composed of an encoder and a decoder model with a transformer architecture (Vaswani et al., 2017). In order to provide fair comparisons, we use the exact same architecture as TransCoder: an encoder and a decoder of 6 layers each, a hidden dimension of 1024 and 8 attention heads. We limit the size of the input to 512 tokens. In Chapter 4, we train models with two different architectures. For Java $\leftrightarrow$ Python, we compare ourselves to the version of DOBF using the same architecture as TransCoder. We initialize our models with either the best TransCoder checkpoint for Java $\rightarrow$ C++ or the best DOBF checkpoint for Java $\rightarrow$ Python with C++ language embeddings initialized with those of Java.

Datasets. As TransCoder and DOBF, we use the GitHub public dataset available on Google BigQuery filtered to keep only projects with open-source licenses[1]. As our unit test creation tool can only be used on Java code, we only use the Java files and we select only the functions that can be compiled in isolation. We obtain a dataset containing 333,542 Java functions. We run EvoSuite with a budget of 20 seconds and a criterion including the line, branch, cbranch and output coverages, as well as the weak and strong mutation scores. We set the maximum absolute value of integers that can be generated as an input to $\sqrt{2^{31}-1}$ to limit the number of overflows. We manage to obtain high-quality (mutation score > 0.9 and at least two asserts) test

[1]We select the open-source licenses: 'apache-2.0', 'mit', 'gpl-2.0', 'gpl-3.0', 'bsd-2-clause', 'bsd-3-clause'

cases for 103,488 functions. See Figures 5.7 and 5.3, 5.4 for examples of selected and filtered out test suites.

Training details. During the training, we alternate between batches for every source and target language so that language pairs for which we managed to create more parallel examples are not over-represented in our training batches. For the online version, we set a cache warm-up parameter to ensure that we always generate new parallel examples if there are less than 500 examples in the cache for any language pair. Otherwise, we sample from the cache with probability 0.5, or generate new examples, train on them once and put them in the cache also with probability 0.5. The sampled elements are removed from the cache with probability 0.3, so that each element we create is trained on about 4 times in average before being removed from the cache. We initialize the cache with parallel examples created offline.

During beam decoding, we compute the score of generated sequences by dividing the sum of token log-probabilities by l^{α} where l is the sequence length. We found that taking $\alpha = 0.5$ (and penalizing long generations) leads to the best performance on the validation set.

Reproducibility. We made sure to use the same architecture and framework as previous works in source code translation so that our results are comparable (see Section 5.3.1). We submit our code with this submission, along with a ReadMe file detailing clear steps to reproduce our results, including a script to set-up a suitable environment. We will open-source our code and release our trained models. Our models were trained using standard hardware (Tesla V100 GPUs) and libraries (e.g. PyTorch, Cuda) for machine-learning research.

5.3.2 Results and discussion

Results. In Tables 5.3 and 5.4, we compare the results of our offline and online training methods with those of TransCoder and DOBF. DOBF outperforms TransCoder for the Java $\leftrightarrow$ Python pair. We compare our models against the best baseline for each language pair and direction.

Training on the generated parallel examples brings substantial improvements for every language pair, direction, and metric. Offline training already provides clear improvements over the baseline after one iteration. The computational accuracy (CA@1) computed with beam size 10 is higher for every direction and it is substantially

Table 5.3: **Computational accuracy scores for our methods and baselines.** We show the CA@1 metric computed with beam size 10. Both the offline and online self-training methods lead to significant improvements over our baselines for every language pair and direction. Online self-training outperforms offline self-training, even after several iterations.

	C++ → Ja	C++ → Py	Ja → C++	Ja → Py	Py → C++	Py → Ja	AVG
TransCoder	65.1	47.1	79.8	49.0	32.6	36.6	51.7
DOBF	-	-	-	52.7	-	45.7	-
Offline ST 1	65.5	56.2	81.6	61.8	46.8	55.1	61.1
Offline ST 2	65.5	58.3	83.7	63.3	46.4	52.2	61.6
Offline ST 3	66.5	56.2	**85.2**	66.3	48.1	56.6	63.1
Offline ST 4	65.3	48.2	81.1	58.1	48.9	54.7	59.4
Online ST	**68.0**	**61.3**	84.6	**68.9**	**56.7**	**58.2**	**66.3**

Table 5.4: **CA@N metric for several beam sizes averaged on all language pairs.** The value k corresponds to the beam size. For instance, CA@1 k=10 means that we use beam decoding to generate 10 translations, and select the one with the highest score. The best baseline corresponds to taking the best model between TransCoder and DOBF for every language pair and direction. The error rate reduction of the offline and online self-training methods over the best baseline are high (> 20%) across all CA@N metrics and beam sizes.

	CA@1 k=1	CA@1 k=10	CA@1 k=20	CA@10 k=10	CA@20 k=20
Best baseline	52.2	53.7	53.4	67.3	70.5
Offline ST 1	60.8	61.1	61.1	72.9	75.3
Offline ST 2	61.4	61.6	61.4	73.3	75.8
Offline ST 3	61.7	63.1	63.0	73.3	75.8
Offline ST 4	58.5	59.4	59.2	70.8	73.6
Online ST	**64.7**	**66.3**	**66.3**	**75.4**	**77.2**

higher for the language pairs involving Python. It allows to reduce the error rate of the best baseline by 25.5% for Java → Python. In average, it increases the CA@1 by 7.4% over the best previous models, and reduces the error rate by 16.6%. In the two next iterations, the model is trained on significantly more examples (see Table 5.2). It results in average improvements of 2% points between the first and third iteration. Although the model for the fourth iteration is trained on more parallel samples, its performance on the test set of TransCoder is actually worse than after the third iteration. After three iterations, the model learned to generate more samples that pass the unit tests but some of them are actually incompatible with the types of translations expected by TransCoder (e.g. example with overflows in Figure 5.8), causing the computational accuracy score to go down.

Table 5.5: **Ablation study.** We show the CA@1 metric computed with greedy decoding at evaluation time except for the last line where the beam size is set to 10. We evaluate models trained with no cache system, without initializing the cache (with or without selecting the tests with a minimum mutation score of 0.9), and a beam size of 1 when generating examples. We also compare the CA@1 score of our full model when evaluating with greedy decoding and with beam size 10. Using a pre-filled cache and selecting only the tests with a high mutation score lead to substantially better performance, although these steps are not necessary to outperform our baseline. The online method already performs well with greedy decoding at generation time, but generating with beam size 20 further improves the results.

	C++ → Ja	C++ → Py	Ja → C++	Ja → Py	Py → C++	Py → Ja	AVG
No cache	66.5	52.7	83.7	60.3	41.2	51.8	59.4
Cache not initialized	64.9	51.6	82.4	62.4	46.6	52.6	60.1
+ No min mut. score	64.0	50.1	82.6	60.9	47.4	47.0	58.7
ST greedy decoding	65.9	54.2	82.2	60.9	56.2	56.6	62.7
Full model (ST beam 20)	66.7	61.1	84.1	67.8	52.2	56.7	64.7
+ Eval beam 10	**68.0**	**61.3**	**84.6**	**68.9**	**56.7**	**58.2**	**66.3**

The online self-training method provides further improvements over training on the pseudo-labeled examples offline. It outperforms every other method in every case except the third iteration of offline training for Java → C++. In average, this model outperforms the baseline by 12.6% points, corresponding to an error rate reduction of 25.5%. For Python → C++, it improves previous performance by more than 24% points, which corresponds to reducing the error rate by 35.7%. Examples of avoided errors can be found in Figure 5.1 and 5.4. Overall, all our models significantly improve previous results. As shown in Table 5.4, these improvements are stable across several beam sizes and CA@N metrics. The CA@20 metric shows that the number of examples for which none of the 20 elements in the beam are correct is reduced by more than 22% with online self-training. It indicates that, even though we train only on the output of the model, our method does much more than reordering the elements in the beam and allows the model to find correct solutions that were not assigned a high probability by the baseline model. See Table 5.6 for more results.

Beam reordering We also evaluate a simpler method where we create unit tests for the Java functions in the test dataset and use them to reorder the elements of the beam at test time. We compute the results of the tests for every proposed C++ or Python translation and prioritize the elements that pass the unit tests.

As shown on Table 5.6, reordering the elements of the beam at test time when translating from Java leads only to small improvements compared to the best baseline (up to 1.7% CA@1 for Java → Python) and the scores of this method are far from

those obtained when requiring any of the 10 element of the beam to be correct (i.e. CA@10). It can be explained by the fact that the tests generated by EvoSuite on these functions can have low mutation scores and be insufficient to thoroughly test the semantics of the functions. Moreover, the tests we create are sometimes incompatible with those of our test set (see Figure 5.8 for an example).

Table 5.6: **Extra results table.** We show the CA@1 metric computed with beam size 10 for our baselines, and our offline and online methods, the beam reordering, and a model trained from scratch with our dataset. Beam reordering leads only to small improvements compared to our offline and online self-training methods. Training on our generated parallel dataset from scratch leads to decent performances, but that are still below those of TransCoder and TransCoder-ST.

	C++ → Ja	C++ → Py	Ja → C++	Ja → Py	Py → C++	Py → Ja	AVG
TransCoder	65.1	47.1	79.8	49.0	32.6	36.6	51.7
DOBF	-	-	-	52.7	-	45.7	-
Beam reordering	-	-	80.3	54.4	-	-	-
Offline ST scratch	43.0	41.3	54.3	43.2	31.1	39.7	42.1
Offline ST 1	65.5	56.2	81.6	61.8	46.8	55.1	61.1
Offline ST 2	65.5	58.3	83.7	63.3	46.4	52.2	61.6
Offline ST 3	66.5	56.2	**85.2**	66.3	48.1	56.6	63.1
Offline ST 4	65.3	48.2	81.1	58.1	48.9	54.7	59.4
Online ST	**68.0**	**61.3**	84.6	**68.9**	**56.7**	**58.2**	**66.3**

Ablation study. The results of our ablation study are shown in Table 5.5. Training online with no cache makes the training much less stable. The model improves at the beginning of training and we can select a few checkpoints where it performs well, but it ends up over-fitting a few examples it generated and the performance drops after a few epochs. Starting with an empty cache slows down the training and hinders generalization, leading to a clear drop in performance. We also try removing the minimum mutation score requirement for the model with no initial cache, which leads to even lower scores as the model is trained partly on lower-quality parallel data.

All these models were trained using a self-training beam size of 20 when generating new examples. Training with greedy decoding is much faster since computing the results for all the 20 elements of the beam is costly. However, generating new examples with greedy decoding leads to a loss of about two percentage points in average compared to our full model using beams of size 20. It shows that initializing the cache of the model with beam size 20 is not sufficient and creating new examples with beam search is necessary to reach our best performance. Our full model provides

some improvements over the ablated versions for every language pair and direction, except over the model trained with greedy decoding for Python → C++ translation. Evaluating with beam size 10 (still returning only the first element) leads to some improvements for every language pair.

Limitations. We found that the unit tests we create with this method are sometimes incompatible with those of the test set of TransCoder, and that the capacity of a model to generate functions that pass these unit tests is not perfectly correlated to its score on the test set. It raises the deeper issue of defining what constitutes a correct translation. For instance, most programmers would translate a factorial function implemented with `long` integers into a factorial function implemented with Python's integer type. However, these functions are not semantically equivalent since the Java implementation would return a negative number for the input `21` due to integer overflow while the Python implementation would return 21! correctly. The human developers who wrote the parallel functions in the test set of TransCoder often assumed that these functions would only be used on a limited domain where no overflow occurs (see Figure 5.8). However, the test cases of EvoSuite and TransCoder are not limited to this domain and they sometimes assert different semantics. By using the test suites from EvoSuite as source of truth, we sometimes train the model to generate translations that are more rigorous but also less natural.

Input Java function

```java
static int factorial(int n){
    if (n < 2) return 1;
    return n * factorial(n - 1);
}
```

Gold translation

```python
def factorial(n):
    if n < 2:
        return 1
    return n * factorial(n-1)
```

Translation passing multilingual tests

```python
def factorial(n):
    n = np.int32(n)
    if n < 2:
        return np.int32(1)
    return n * factorial(n - 1)
```

Figure 5.8: **Example of disagreement between our multilingual tests and the test set of TransCoder.** The gold translation is only equivalent to the input Java function on a small domain where there is no integer overflow and does not pass our unit tests. The version that passes the unit tests casts uses the `np.int32` type, reproducing the behaviour of the original Java code but causing it to fail some of the unit tests of TransCoder.

5.4 Translation examples

Input Python function

```python
def rangeGCD(n, m):
    return n if (n==m) else 1
```

```python
def countSetBits(n):
  if(n == 0):
    return 0
  else:
    return(n & 1) \
      + countSetBits(n >> 1)
```

TransCoder C++ translation

```cpp
int rangeGCD(int n, int m){
    return n == (n==m) ? 1: -1;
}
```

```cpp
int countSetBits(int n) {
  if((n == 0) ||(n == 1))
    return 0;
  else
    return(n & 1)
    + countSetBits(n >> 1);
}
```

TransCoder-ST C++ translation

```cpp
int rangeGCD(int n, int m){
    return (n==m) ? n:1;
}
```

```cpp
int countSetBits(int n) {
  if((n == 0))
    return 0;
  else
    return(n & 1)
    + countSetBits(n >> 1);
}
```

Figure 5.9: **Python to C++ translation examples.** TransCoder sometimes fails to capture the semantics of the incoming code and translates them to other expressions that could occur in similar contexts. Self-training helps the model to avoid such mistakes.

Input Python function

```python
def cassini(n):
    return - 1 if(n & 1) else 1
```

```python
def turnOffK(n, k):
  if(k <= 0):
    return n
  return(n & ~(1 <<(k - 1)))
```

```python
def isNumber(s):
  for i in range(len(s)):
    if s[i].isdigit() != True:
      return False
    return True
```

TransCoder Java translation

```java
public static int cassini(int n) {
  return - 1 ==(n & 1) ? 1: - 1;
}
```

```java
static int turnOffK(int n, int k){
  if((k <= 0) (k > n))
    return n;
  return(n & ~(1<<(k - 1)));
}
```

```java
static boolean isNumber(String s) {
  for(int i = 0; i<s.length(); i++){
    if(s.charAt(i) != ' ') {
      return false;
    }
  }
  return true;
}
```

TransCoder-ST Java translation

```java
public static int cassini(int n) {
  return(n & 1) != 0 ? - 1: 1;
}
```

```java
static int turnOffK(int n, int k){
  if((k <= 0))
    return n;
  return(n & ~(1<<(k - 1)));
}
```

```java
static boolean isNumber(String s){
  for(int i = 0; i<s.length(); i++){
    if(Character.isDigit(s.charAt(i))
       != true
    ){
      return false;
    }
  }
  return true;
}
```

Figure 5.10: **Python to Java translation examples.** Similarly to Python to C++, TransCoder often fails to get the right semantics, especially for conditions where it can hallucinate extra clauses or write incorrect comparisons. TransCoder-ST often solves these issues.

Input Python function

```
int cassini(int n) {
  return(n & 1) ? - 1: 1;
}
int power(int n) {
  if(n == 1) return 2;
  return 2 * power(n - 1);
}
```

TransCoder Java translation

```
def cassini(n):
  return(n & 1)

def power(n):
    if n == 1:
        return 2
    return 2 ** power(n - 1)
```

TransCoder-ST Java translation

```
def cassini(n):
  return - 1 if(n & 1) != 0 else 1

def power(n):
    if n == 1:
        return 2
    return 2 * power(n - 1)
```

Figure 5.11: **C++ to Python translation examples.** For C++ to Python translation, many of the errors of TransCoder come from incorrectly translated conditions, wrong operators and badly translated functions. TransCoder-ST better comprehends the semantics of the code and is generally able to solve these issues.

Input C++ function

```
int minXOR(int arr[], int n){
  int min_xor = INT_MAX;
  for(int i = 0; i < n; i++)
    for(int j = i + 1; j < n; j++)
      min_xor = min(min_xor,
                    arr[i]^arr[j]);
  return min_xor;
}
```

```
int findLongestConseqSubseq(
    int arr[], int n
  ){
  unordered_set<int> S;

  for(int i = 0; i < n; i++)
    S.insert(arr[i]);
  int ans = 0;
  for(int i = 0; i < n; i++) {
    if(S.find(arr[i]-1)==S.end()){
      int j = arr[i];
      while(S.find(j) != S.end())
        j++;
      ans = max(ans, j - arr[i]);
    }
  }
  return ans;
}
```

TransCoder Java translation

```
static int minXOR(int arr[], int n){
  int min_xor = Integer.MIN_VALUE;
  for(int i = 0; i < n; i++)
    for(int j = i + 1; j < n; j++)
      min_xor = Math.min(min_xor,
                         arr[i]^arr[j]);
  return min_xor;
}
```

```
public static
int findLongestConseqSubseq(
    int arr[], int n
  ){
  HashSet<Integer> S =
                  new HashSet<Integer>();
  for(int i = 0; i < n; i++) {
    S.add(arr[i]);
  }
  int ans = 0;
  for(int i = 0; i < n; i++) {
    if(S.contains(arr[i] - 1)) {
      int j = arr[i];
      while(S.contains(j)) {
        j++;
      }
      ans = Math.max(ans, j - arr[i]);
    }
  }
  return ans;
}
```

TransCoder-ST Java translation

```
static int minXOR(int arr[], int n){
  int minXor = Integer.MAX_VALUE;
  for(int i = 0; i < n; i++)
    for(int j = i + 1; j < n; j++)
      minXor = Math.min(minXor,
                        arr[i]^arr[j]);
  return minXor;
}
```

```
public static
int findLongestConseqSubseq(
    int arr[], int n
  ){
  HashSet<Integer> S = new HashSet<>();

  for(int i = 0; i < n; i++) {
    S.add(arr[i]);
  }
  int ans = 0;
  for(int i = 0; i < n; i++) {
    if(S.contains(arr[i] - 1)==false){
      int j = arr[i];
      while(S.contains(j)) {
        j++;
      }
      ans = Math.max(ans, j - arr[i]);
    }
  }
  return ans;
}
```

Figure 5.12: **C++ to Java translation examples.** The first example computes the minimum XOR between two elements of an array, TransCoder erroneously translates `INT_MAX` into `Integer.MIN_VALUE`. This value is used in similar contexts (i.e. to compute a maximum instead of a minimum) but is inappropriate here. TransCoder-ST manages to correct this and outputs a function with the right semantics. In the second example, where the function computes the size of the largest subset of elements of the list that could form a sequence of consecutive integers, TransCoder-ST manages to translate the semantics of `S.find(arr[i]-1)==S.end()` appropriately while TransCoder translates it into its negation.

Input Java function

```
static boolean isCornerPresent(
                String str,
                String corner
  ) {
  int n = str.length();
  int cl = corner.length();
  if(n < cl) return false;
  return(str.substring(0,cl)
           .equals(corner)
      && str.substring(n - cl, n)
        .equals(corner));
}
...
PriorityQueue <Integer> Q =
    new PriorityQueue <Integer>();
...
```

TransCoder C++ translation

```
bool isCornerPresent(
     string str,
     string corner
  ) {
  int n = str.length();
  int cl = corner.length();
  if(n < cl) return false;
  return(str[0] == corner[0]
      && str[n - cl] == corner[n]);
}

...
priority_queue < int > Q;
...
```

TransCoder-ST C++ translation

```
bool isCornerPresent(
     string str,
     string corner
  ) {
  size_t n = str.length();
  size_t cl = corner.length();
  if(n < cl) return false;
  return(str.substr(0, cl)==corner
      && str.substr(n-cl, n)==corner);
}

...
priority_queue<int,
               vector<int>,
               greater<int>
              > Q;
...
```

Figure 5.13: **Java to C++ translation examples.** In the first example, which returns whether a given string `corner` is present at the beginning and at the end of a string `str`, TransCoder completely fails to translate the last logical expression correctly while TransCoder-ST manages to translate the logic to get the right substrings and to return the right output. The second example is a line defining a priority queue extracted from the kthLargestSum function in the test set of TransCoder. The `PriorityQueue` object in Java returns the smallest elements first by default, while `priority_queue` in C++ returns the largest. TransCoder, which was not trained on any semantic signal, manages to instantiate a priority queue object but instantiates a max queue instead of a min queue. TransCoder-ST, which was trained with some supervised signal directly linked to the semantics of the code, manages to instantiate the right type of priority queue.

Input Java function

```java
static int divisorSum(int n) {
    int sum = 0;
     for(int i = 1; i <= n; ++i)
        sum +=(n / i) * i;
     return sum;
}
```

```java
static
boolean check(int degree[], int n){
  int deg_sum = 0;
  for(int i = 0; i < n; i ++) {
    deg_sum += degree[i];
  }
  return(2*(n-1)==deg_sum);
}
```

```java
static int decimalToBinary(int N){
  int B_Number = 0;
  int cnt = 0;
  while(N != 0) {
    int rem = N % 2;
    double c = Math.pow(10, cnt);
    B_Number += rem * c;
    N /= 2;
    cnt ++;
  }
  return B_Number;
}
```

TransCoder Python Translation

```python
def divisor_sum(n):
  sum = 0
  for i in range(1 , n + 1):
    sum +=(n / i) ** i
  return sum
```

```python
def check(degree, n):
  deg_sum = 0
  for i in range(n):
    deg_sum += degree[i]
  return(2**(n-1)==deg_sum)
```

```python
def decimal_to_binary(N):
  B_Number = 0
  cnt = 0
  while N != 0:
    rem = N % 2
    c = pow(10, cnt)
    B_Number += rem * c
    N /= 2
    cnt += 1
  return B_Number
```

TransCoder-ST translation

```python
def divisor_sum(n):
  sum = 0
  for i in range(1 , n + 1):
    sum +=(n // i) * i
  return sum
```

```python
def check(degree, n):
  deg_sum = 0
  for i in range(n):
    deg_sum += degree[i]
  return(2*(n-1)==deg_sum)
```

```python
def decimal_to_binary(N):
  B_number = 0
  cnt = 0
  while N != 0:
    rem = N % 2
    c = pow(10, cnt)
    B_number += rem * c
    N //= 2
    cnt += 1
  return B_number
```

Figure 5.14: **Translation examples for Java to Python.** When translating to Python, TransCoder was often failing to distinguish between float division (`/`) and integer division (`//`). It also often confuses other operations, for instance `*` and `**`. Training on self-created labels often solves this issue.

Input Python function

```python
def transpose(A, B):
    for i in range(N):
        for j in range(N):
            B[i][j]=A[j][i]
```

```python
def exponential(n, x):
    sum=1.0
    for i in range (n, 0, -1):
        sum=1+x*sum/i
    print("e^x =", sum)
```

TransCoder C++ translation

```cpp
void transpose (
    const double* A ,
    double* B[N][N]
    ) {
  for(int i=0; i<N; i++)
    for(int j=0; j < N; j++)
        B[i][j]=A[j][i];
}
```

```cpp
void exponential(int n, double x){
  double sum=1.0;
  for(int i=n; i>0; i--)
    sum=1+x*sum/i ;
  cout << "e^x =" << sum << endl ;
}
```

TransCoder-ST C++ translation

```cpp
template<typename T>
void transpose (T* A, T* B){
  for(int i=0; i<N ; i++){
    for(int j=0; j<N; j++){
      B[i][j] = A[j][i];
    }
  }
}
```

```cpp
template <typename T>
void exponential(int n, T x) {
  T sum=1.0;
  for(int i=n; i>0; i--)
    sum=1+x*sum/i;
  cout << "e^x =" << sum << endl;
}
```

Figure 5.15: **Our parallel unit tests lead to the generation of more general solutions using templates.** Solutions using templates can pass the unit tests for several parameter types, while guessing the wrong parameter type can lead to some errors. Solutions using templates succeed more often, are more likely to appear in the parallel data we generate and, as a result, in our model's generations. It leads to our model generating more templates (three times more often for our online model trained the longest).

5.5 Conclusion

In this chapter, we introduced a novel method to grow a parallel corpus for automated code translation, from completely monolingual data. We leverage multilingual unit tests to filter good pseudo-labels, improving the model, and in turn the candidate translations. We show that both offline and online methods substantially improve the state of the art in unsupervised code translation, with an average improvement of 12.6% points in computational accuracy, and up to 24% points for Python $\rightarrow$ C++, corresponding to translation error rate reductions of 25.5% and 35.7% respectively, without using any unit test generation tool for Python and C++ (exclusively for Java).

Our method would automatically gain from improvements of automatic unit test generation tools. We could also increase the size of the dataset we generate by using test creation tools written for other languages in addition to Java, or by generating tests with EvoSuite on translated examples. Similarly, we could also extract the semantics of human-written unit tests found in open-source projects to obtain larger, and possibly higher-quality datasets. In this chapter, we focused on translation correctness and our parallel example validation criterion was only based on semantics. It could be supplemented with other requirements, such as a specific code formatting or the output of linters to generate code verifying arbitrary criteria.

Chapter 6

Conclusion and perspectives

In this thesis, we developed unsupervised methods for translating between high-level programming languages. Transformer networks provided us with a general and versatile architecture, to learn multilingual embeddings of source code, and translate between C++, Java and Python. We exhibited the shortcomings of previous metrics, such as the BLEU and exact match scores, and created a new test set and metrics evaluating the semantics of our translations with unit tests. We showed significant improvements compared to existing rule-based translators.

Then, we identified that pre-training methods designed for natural languages, such as MLM, are sub-optimal in the context of programming languages. Hence, we designed a novel pre-training method for programming languages based on identifier deobfuscation. Identifier names contain rich semantic information, and our objective leads the model to understand the meaning of the code. When used together with random masking schemes, it leads to significant gains on several programming languages tasks, including code translation.

After improving the model pre-training for programming languages, we decided to use another one of their distinct properties. Contrarily to natural languages, source code can be compiled and run. We used an automated unit test creation tool, to generate datasets of tens of thousands of translation examples. We proposed an online training method, and a cache mechanism, to use this signal and significantly improve our unsupervised transcompiler.

Weisz et al. (2022) conducted an experiment in which 32 software engineers translated code with and without the assistance of our unsupervised translator described in Chapter 3. They showed that our tool improves the translation quality, and that the participants overwhelmingly felt like it was useful. Such generative

models provide a starting point to work with, and can teach new constructs or standard functions in the target language. However, the programmers also reported wasting time and feeling frustrated when having to fix bugs in the imperfect outputs of the program, showing improving neural transcompilers would be needed and impactful. We identify the following areas for future works:

Constrained generations An important difference between natural and programming languages is that the later have a stricter syntax and better-defined semantics. Popular languages come with several compilers, linters, and other static analysis tools, which can find errors in generated code. This information, as well as runtime data, can be used to filter the outputs of the model. However, these methods significantly increase the computational resources and latency at generation time. Other methods using these tools to retrain the model, or constraining the generations of auto-regressive models could also be explored.

Parallel data In this thesis, we described only unsupervised methods for source code translation. Parallel data, composed of aligned code snippets in two programming languages, could also be used to either align the representations at the beginning of training or fine-tune our models. Such data would be expensive to create manually, since translating functions requires expertise in both the source and the target languages. The value of existing parallel datasets, such as the validation and test set that we released and solutions to competitive programming problems, has still not been estimated for training parallel models. This type of data leads to improved performance for translating functions from the same domain, but it is unclear whether they would help translate real-world functions. Noisy alignments with simple rules could also be explored.

Larger contexts Most machine learning methods for source code—including those presented in this thesis—consider either files or functions independently. However, real-world software engineering requires to dive into large codebases. Changes need to account for the way the codebase is organized, and can often be simplified by using classes and functions defined in other files. Even with linear attention transformers, there is a limit to the number of tokens that can be included in the context. Hence, methods adapting machine learning systems to the context of specific codebases would require to find another way to encode large contexts. Methods such as hierarchical models or retrieval could be explored.

Code translation as a well-specified code synthesis task. Code translation aims to synthesize a program in the target language, given a program written in the source language. At the level of the entire program, assuming that the input domain and the interpreter for the source language are known, it provides a perfect specification of the problem. The source can even be compiled and run on custom input, in order to compare its output and complexity to those of the translation. Hence, systems translating real-world codebases may be significantly easier to evaluate than those synthesizing code from natural language prompts. The specificity of programming language prompts also makes code translation systems suitable to test the capacity of machine learning models to generate programs, independently of their capacity to disambiguate prompts. Insights from programming languages translation research could provide directions for more general code synthesis tasks.

Bibliography

Edward Aftandilian, Raluca Sauciuc, Siddharth Priya, and Sundaresan Krishnan. Building useful program analysis tools using an extensible java compiler. In *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, pages 14–23. IEEE, 2012.

Abien Fred Agarap. Deep learning using rectified linear units (relu). *arXiv preprint arXiv:1803.08375*, 2018.

Karan Aggarwal, Mohammad Salameh, and Abram Hindle. Using machine translation for converting python 2 to python 3 code. Technical report, PeerJ PrePrints, 2015.

Maaz Bin Safeer Ahmad and Alvin Cheung. Leveraging parallel data processing frameworks with verified lifting. *SYNT@CAV*, 2016.

Maaz Bin Safeer Ahmad, Jonathan Ragan-Kelley, Alvin Cheung, and Shoaib Kamil. Automatically translating image processing libraries to halide. *ACM Transactions on Graphics (TOG)*, 38(6):1–13, 2019.

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668, 2021.

Qurat Ul Ain, Wasi Haider Butt, Muhammad Waseem Anwar, Farooque Azam, and Bilal Maqbool. A systematic review on code clone detection. *IEEE Access*, 7: 86121–86144, 2019.

Miltiadis Allamanis. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 143–153, 2019.

Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 281–293, 2014.

Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 38–49, 2015a.

Miltiadis Allamanis, Hao Peng, and Charles Sutton. A convolutional attention network for extreme summarization of source code. In *International conference on machine learning*, pages 2091–2100, 2016.

Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37, 2018a.

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *International Conference on Learning Representations*, 2018b.

Miltiadis Allamanis, Henry Jackson-Flux, and Marc Brockschmidt. Self-supervised bug detection and repair. *Advances in Neural Information Processing Systems*, 34, 2021.

Miltos Allamanis, Daniel Tarlow, Andrew Gordon, and Yi Wei. Bimodal modelling of source code and natural language. In *International conference on machine learning*, pages 2123–2132. PMLR, 2015b.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. A general path-based representation for predicting program properties. *ACM SIGPLAN Notices*, 53(4): 404–419, 2018.

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. *ICLR*, 2019a.

Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. Structural language models for any-code generation. *arXiv preprint arXiv:1910.00577*, 2019b.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019c.

Nadia Alshahwan, Xinbo Gao, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, Taijin Tei, and Ilya Zorin. Deploying search based software engineering with Sapienz at Facebook (keynote paper). In 10^{th} *International Symposium on Search Based Software Engineering (SSBSE 2018)*, pages 3–45, Montpellier, France, September 8th-10th 2018. Springer LNCS 11036.

Matthew Amodio, Swarat Chaudhuri, and Thomas Reps. Neural attribute machines for program generation. *arXiv preprint arXiv:1705.09231*, 2017.

Saswat Anand, Antonia Bertolino, Edmund Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Jenny Li, Phil McMinn, and Hong Zhu. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8): 1978–2001, August 2013.

Mikel Artetxe, Gorka Labaka, and Eneko Agirre. Learning bilingual word embeddings with (almost) no bilingual data. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 451–462, 2017.

Mikel Artetxe, Gorka Labaka, and Eneko Agirre. Unsupervised statistical machine translation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, 2018a.

Mikel Artetxe, Gorka Labaka, Eneko Agirre, and Kyunghyun Cho. Unsupervised neural machine translation. In *International Conference on Learning Representations (ICLR)*, 2018b.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Nathaniel Ayewah, William Pugh, J David Morgenthaler, John Penix, and YuQian Zhou. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 1–8, 2007.

Arthur Baars, Mark Harman, Youssef Hassoun, Kiran Lakhotia, Phil McMinn, Paolo Tonella, and Tanja Vos. Symbolic search-based testing. In 26^{th} *IEEE/ACM*

International Conference on Automated Software Engineering (ASE 2011), pages 53 – 62, Lawrence, Kansas, USA, 6th - 10th November 2011.

John W Backus, Robert J Beeber, Sheldon Best, Richard Goldberg, Lois M Haibt, Harlan L Herrick, Robert A Nelson, David Sayre, Peter B Sheridan, Harold Stern, et al. The fortran automatic coding system. In *Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability*, pages 188–198, 1957.

Dzmitry Bahdanau, Kyung Hyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *3rd International Conference on Learning Representations*, 2015.

Rohan Bavishi, Michael Pradel, and Koushik Sen. Context2name: A deep learning-based approach to infer natural variable names from usage contexts. *arXiv preprint arXiv:1809.05193*, 2018.

Yonatan Belinkov and Yonatan Bisk. Synthetic and natural noise both break neural machine translation. In *International Conference on Learning Representations*, 2018.

Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. Statistical deobfuscation of android applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 343–355, 2016.

Ondřej Bojar and Aleš Tamchyna. Improving translation model by monolingual data. In *Proceedings of the sixth workshop on statistical machine translation*, pages 330–336, 2011.

Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. Relating identifier naming flaws and code quality: An empirical study. In *2009 16th Working Conference on Reverse Engineering*, pages 31–35. IEEE, 2009.

Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Communications of the ACM*, 56(2):82–90, February 2013. ISSN 0001-0782.

Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter OHearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In *NASA Formal Methods Symposium*, pages 3–11. Springer, 2015.

Chris Callison-Burch, Miles Osborne, and Philipp Koehn. Re-evaluating the role of bleu in machine translation research. In *11th conference of the european chapter of the association for computational linguistics*, pages 249–256, 2006.

Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. When deep learning met code search. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 964–974, 2019.

Thierry Titcheu Chekam, Mike Papadakis, Yves Le Traon, and Mark Harman. An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 597–608, 2017.

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

Xinyun Chen, Chang Liu, and Dawn Song. Tree-to-tree neural networks for program translation. In *Advances in neural information processing systems*, pages 2547–2557, 2018.

Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, 2019.

Zimin Chen, Vincent Hellendoorn, Pascal Lamblin, Petros Maniatis, Pierre-Antoine Manzagol, Daniel Tarlow, and Subhodeep Moitra. Plur: A unifying, graph-based view of program learning, understanding, and repair. *Advances in Neural Information Processing Systems*, 34, 2021b.

Nadezhda Chirkova and Sergey Troshin. Empirical study of transformers for source code. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 703–715, 2021.

Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. In *EMNLP*, 2014.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. Electra: Pre-training text encoders as discriminators rather than generators. *arXiv preprint arXiv:2003.10555*, 2020.

Colin B Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. Pymt5: multi-mode translation of natural language and python code with transformers. *arXiv preprint arXiv:2010.03150*, 2020.

Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. Synthesizing benchmarks for predictive modeling. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 86–99. IEEE, 2017.

Yaniv David, Uri Alon, and Eran Yahav. Neural reverse engineering of stripped binaries using augmented control flow graphs. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–28, 2020.

Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on test data selection: Help for the practical programmer. *IEEE Computer*, 11:31–41, 1978.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. Hoppity: Learning graph transformations to detect and fix bugs in programs. In *International Conference on Learning Representations (ICLR)*, 2020.

Li Dong, Nan Yang, Wenhui Wang, Furu Wei, Xiaodong Liu, Yu Wang, Jianfeng Gao, Ming Zhou, and Hsiao-Wuen Hon. Unified language model pre-training for natural language understanding and generation. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, pages 13063–13075, 2019.

Dawn Drain, Colin B Clement, Guillermo Serrato, and Neel Sundaresan. Deepdebug: Fixing python bugs using stack traces, backtranslation, and code skeletons. *arXiv preprint arXiv:2105.09352*, 2021.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings*, pages 1536–1547, 2020.

Gordon Fraser and Andrea Arcuri. EvoSuite: automatic test suite generation for object-oriented software. In 8^{th} *European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '11)*, pages 416–419. ACM, September 5th - 9th 2011. ISBN 978-1-4503-0443-6.

Gordon Fraser and Andrea Arcuri. Achieving scalable mutation-based generation of whole test suites. *Empirical Software Engineering*, 20(3):783–812, 2015.

Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. Coda: An end-to-end neural program decompiler. In *Advances in Neural Information Processing Systems*, pages 3703–3714, 2019.

Evrard Garcelon, Baptiste Roziere, Laurent Meunier, Jean Tarbouriech, Olivier Teytaud, Alessandro Lazaric, and Matteo Pirotta. Adversarial attacks on linear contextual bandits. *Advances in Neural Information Processing Systems*, 33: 14362–14373, 2020.

Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N Dauphin. Convolutional sequence to sequence learning. In *International Conference on Machine Learning*, pages 1243–1252. PMLR, 2017.

Edward M Gellenbeck and Curtis R Cook. An investigation of procedure and variable names as beacons during program comprehension. In *Empirical studies of programmers: Fourth workshop*, pages 65–81. Ablex Publishing, Norwood, NJ, 1991.

Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 933–944. IEEE, 2018.

Caglar Gulcehre, Orhan Firat, Kelvin Xu, Kyunghyun Cho, Loic Barrault, Huei-Chi Lin, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. On using monolingual corpora in neural machine translation. *arXiv preprint arXiv:1503.03535*, 2015.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, LIU Shujie, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. In *International Conference on Learning Representations*, 2020.

Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common c language errors by deep learning. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

Francisco Guzmán, Peng-Jen Chen, Myle Ott, Juan Pino, Guillaume Lample, Philipp Koehn, Vishrav Chaudhary, and Marc'Aurelio Ranzato. Two new evaluation datasets for low-resource machine translation: Nepali-english and sinhala-english. *arXiv preprint arXiv:1902.01382*, 2019.

Jesse Michael Han, Igor Babuschkin, Harrison Edwards, Arvind Neelakantan, Tao Xu, Stanislas Polu, Alex Ray, Pranav Shyam, Aditya Ramesh, Alec Radford, et al. Unsupervised neural machine translation with generative language models only. *arXiv preprint arXiv:2110.05448*, 2021.

Mark Harman, Afshin Mansouri, and Yuanyuan Zhang. Search based software engineering: Trends, techniques and applications. *ACM Computing Surveys*, 45 (1):11:1–11:61, November 2012.

Mark Harman, Yue Jia, and Yuanyuan Zhang. Achievements, open problems and challenges for search based software testing (keynote paper). In 8^{th} *IEEE International Conference on Software Testing, Verification and Validation (ICST 2015)*, Graz, Austria, April 2015.

Di He, Yingce Xia, Tao Qin, Liwei Wang, Nenghai Yu, Tie-Yan Liu, and Wei-Ying Ma. Dual learning for machine translation. In *Advances in neural information processing systems*, pages 820–828, 2016.

Vincent J Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. Global relational models of source code. In *International conference on learning representations*, 2019.

Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*, 2021.

Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 837–847. IEEE, 2012.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*, pages 200–210, 2018.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph E Gonzalez, and Ion Stoica. Contrastive code representation learning. *arXiv preprint arXiv:2007.04973*, 2020.

Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649 – 678, September–October 2011.

Nan Jiang, Thibaud Lutellier, and Lin Tan. Cure: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1161–1173. IEEE, 2021.

Mandar Joshi, Danqi Chen, Yinhan Liu, Daniel S Weld, Luke Zettlemoyer, and Omer Levy. Spanbert: Improving pre-training by representing and predicting spans. *Transactions of the Association for Computational Linguistics*, 8:64–77, 2020.

Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. Verified lifting of stencil computations. *ACM SIGPLAN Notices*, 51(6):711–726, 2016.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. In *International Conference on Machine Learning*, pages 5110–5121. PMLR, 2020.

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*, pages 173–184, 2014.

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

Huda Khayrallah and Philipp Koehn. On the impact of various types of noise on neural machine translation. In *Proceedings of the 2nd Workshop on Neural Machine Translation and Generation*, pages 74–83, 2018.

Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. Code prediction by feeding trees to transformers. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 150–162. IEEE, 2021.

Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *ICLR*, 2015.

Tom Kocmi, Christian Federmann, Roman Grundkiewicz, Marcin Junczys-Dowmunt, Hitokazu Matsushita, and Arul Menezes. To ship or not to ship: An extensive evaluation of automatic metrics for machine translation. *arXiv preprint arXiv:2107.10821*, 2021.

Philipp Koehn. Pharaoh: a beam search decoder for phrase-based statistical machine translation models. In *Conference of the Association for Machine Translation in the Americas*, pages 115–124. Springer, 2004.

Philipp Koehn. *Statistical machine translation*. Cambridge University Press, 2009.

Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, Ondrej Bojar Chris Dyer, Alexandra Constantin, and Evan Herbst. Moses: Open source toolkit for statistical machine translation. In *Annual Meeting of the Association for Computational Linguistics (ACL), demo session*, 2007.

Taku Kudo and John Richardson. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 66–71, 2018.

Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. Spoc: Search-based pseudocode to code. *Advances in Neural Information Processing Systems*, 32:11906–11917, 2019.

Jeremy Lacomis, Pengcheng Yin, Edward Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. Dire: A neural approach to decompiled identifier naming. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 628–639. IEEE, 2019.

Kiran Lakhotia, Mark Harman, and Hamilton Gross. AUSTIN: An open source tool for search based software testing of C programs. *Journal of Information and Software Technology*, 55(1):112–125, January 2013.

Patrik Lambert, Holger Schwenk, Christophe Servan, and Sadaf Abdul-Rauf. Investigations on translation model adaptation using monolingual data. In *Sixth Workshop on Statistical Machine Translation*, pages 284–293, 2011.

Guillaume Lample and Alexis Conneau. Cross-lingual language model pretraining. *Advances in Neural Information Processing Systems*, 32:7059–7069, 2019.

Guillaume Lample, Alexis Conneau, Ludovic Denoyer, and Marc'Aurelio Ranzato. Unsupervised machine translation using monolingual corpora only. *ICLR*, 2018a.

Guillaume Lample, Alexis Conneau, Marc'Aurelio Ranzato, Ludovic Denoyer, and Hervé Jégou. Word translation without parallel data. In *ICLR*, 2018b.

Guillaume Lample, Myle Ott, Alexis Conneau, Ludovic Denoyer, and Marc'Aurelio Ranzato. Phrase-based & neural unsupervised machine translation. In *EMNLP*, 2018c.

Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. What's in a name? a study of identifiers. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 3–12. IEEE, 2006.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7871–7880, 2020.

Jian Li, Yue Wang, Michael R Lyu, and Irwin King. Code completion with neural attention and pointer networks. *IJCAI*, 2018.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*, 2022.

Ben Liblit, Andrew Begel, and Eve Sweetser. Cognitive perspectives on the role of naming in computer programs. In *PPIG*, page 11, 2006.

Tatiana Likhomanenko, Qiantong Xu, Jacob Kahn, Gabriel Synnaeve, and Ronan Collobert. slimipl: Language-model-free iterative pseudo-labeling. *arXiv preprint arXiv:2010.11524*, 2020.

Xi Victoria Lin, Chenglong Wang, Deric Pang, Kevin Vu, and Michael D Ernst. Program synthesis from natural language using recurrent neural networks. *University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, Tech. Rep. UW-CSE-17-03-01*, 2017.

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

generation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 599–609, 2016.

Fang Liu, Ge Li, Bolin Wei, Xin Xia, Zhiyi Fu, and Zhi Jin. A self-attentional neural architecture for code completion with multi-task learning. In *Proceedings of the 28th International Conference on Program Comprehension*, pages 37–47, 2020.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

Cristina V Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. Déjàvu: a map of code duplicates on github. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–28, 2017.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. Coconut: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, pages 101–114, 2020.

Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.

Chris Maddison and Daniel Tarlow. Structured generative models of natural source code. In *International Conference on Machine Learning*, pages 649–657. PMLR, 2014.

Zohar Manna and Richard J Waldinger. Toward automatic program synthesis. *Communications of the ACM*, 14(3):151–165, 1971.

Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Edward Aftandilian. Deepdelta: learning to repair compilation errors. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 925–936, 2019.

Antonio Valerio Miceli-Barone and Rico Sennrich. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. In *Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pages 314–319, 2017.

W. Miller and D. Spooner. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, 2(3):223–226, 1976.

Martin Monperrus. The living review on automated program repair. Technical Report hal-01956501, HAL/archives-ouvertes.fr, 2018a.

Martin Monperrus. Automatic software repair: a bibliography. *ACM Computing Surveys (CSUR)*, 51(1):1–24, 2018b.

Vijayaraghavan Murali, Lee Gross, Rebecca Qian, and Satish Chandra. Industry-scale ir-based bug localization: A perspective from facebook. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 188–197. IEEE, 2021.

Glenford J. Myers. *The Art of Software Testing*. Wiley - Interscience, New York, 1979. ISBN 0-471-04328-1.

Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. Lexical statistical machine translation for language migration. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 651–654, 2013.

Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. Learning to generate pseudo-code from source code using statistical machine translation (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 574–584. IEEE, 2015.

Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Mutation testing advances: an analysis and survey. In *Advances in Computers*, volume 112, pages 275–378. Elsevier, 2019.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics, 2002.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, pages 1310–1318. PMLR, 2013.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. *NIPS 2017 Autodiff Workshop*, 2017.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pages 8026–8037, 2019.

Maxim Rabinovich, Mitchell Stern, and Dan Klein. Abstract syntax networks for code generation and semantic parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1139–1149, 2017.

Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. 2018.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 419–428, 2014.

Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from" big code". *ACM SIGPLAN Notices*, 50(1):111–124, 2015.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020.

Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. *Advances in Neural Information Processing Systems (NeurIPS)*, 33, 2020a.

Baptiste Roziere, Fabien Teytaud, Vlad Hosu, Hanhe Lin, Jeremy Rapin, Mariia Zameshina, and Olivier Teytaud. Evolgan: Evolutionary generative adversarial networks. In *Proceedings of the Asian Conference on Computer Vision (ACCV)*, 2020b.

Baptiste Roziere, Marie-Anne Lachaux, Marc Szafraniec, and Guillaume Lample. DOBF: A deobfuscation pre-training objective for programming languages. *Advances in Neural Information Processing Systems (NeurIPS)*, 2021a.

Baptiste Roziere, Nathanaël Carraz Rakotonirina, Vlad Hosu, Andry Rasoanaivo, Hanhe Lin, Camille Couprie, and Olivier Teytaud. Tarsier: Evolving noise injection in super-resolution gans. In *2020 25th International Conference on Pattern Recognition (ICPR)*, pages 7028–7035. IEEE, 2021b.

Baptiste Rozière, Morgane Riviere, Olivier Teytaud, Jérémy Rapin, Yann LeCun, and Camille Couprie. Inspirational adversarial image generation. *IEEE Transactions on Image Processing*, 30:4036–4045, 2021.

Baptiste Roziere, Jie M Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. Leveraging automated unit tests for unsupervised code translation. *ICLR*, 2022.

Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.

Rico Sennrich, Barry Haddow, and Alexandra Birch. Improving neural machine translation models with monolingual data. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, pages 86–96, 2015a.

Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, pages 1715–1725, 2015b.

David E Shaw, William R Swartout, and C Cordell Green. Inferring lisp programs from examples. In *IJCAI*, volume 75, pages 260–267, 1975.

Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations. In *Proceedings of the 2018 Conference of the North*

American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers), pages 464–468, 2018.

Vighnesh Shiv and Chris Quirk. Novel positional encodings to enable tree-based transformers. *Advances in Neural Information Processing Systems*, 32, 2019.

Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. Mass: Masked sequence to sequence pre-training for language generation. In *International Conference on Machine Learning*, pages 5926–5936, 2019.

Yu Sun, Shuohuan Wang, Yukun Li, Shikun Feng, Xuyi Chen, Han Zhang, Xin Tian, Danxiang Zhu, Hao Tian, and Hua Wu. Ernie: Enhanced representation through knowledge integration. *arXiv preprint arXiv:1904.09223*, 2019.

Ilya Sutskever. *Training recurrent neural networks*. University of Toronto Toronto, ON, Canada, 2013.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

Alexey Svyatkovskiy, Sebastian Lee, Anna Hadjitofi, Maik Riechert, Juliana Vicente Franco, and Miltiadis Allamanis. Fast and memory-efficient neural code completion. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 329–340. IEEE, 2021.

Armstrong A Takang, Penny A Grubb, and Robert D Macredie. The effects of comments and identifier names on program comprehensibility: an experimental investigation. *J. Prog. Lang.*, 4(3):143–167, 1996.

Daniel Tarlow, Subhodeep Moitra, Andrew Rice, Zimin Chen, Pierre-Antoine Manzagol, Charles Sutton, and Edward Aftandilian. Learning to fix build errors with graph2diff neural networks. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pages 19–20, 2020.

Wilson L Taylor. cloze procedure: A new tool for measuring readability. *Journalism quarterly*, 30(4):415–433, 1953.

Nikolai Tillmann, Jonathan de Halleux, and Tao Xie. Transferring an automated test generation tool to practice: From Pex to Fakes and Code Digger. In *29th*

ACM/IEEE International Conference on Automated Software Engineering (ASE), pages 385–396, 2014.

Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(4):1–29, 2019.

Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. Unit test case generation with transformers. *arXiv preprint arXiv:2009.05617*, 2020.

Bogdan Vasilescu, Casey Casalnuovo, and Premkumar Devanbu. Recovering clear, natural identifiers from obfuscated js names. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 683–693, 2017.

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

Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103, 2008.

Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*, 2018a.

Ke Wang, Rishabh Singh, and Zhendong Su. Dynamic neural program embeddings for program repair. In *International Conference on Learning Representations*, 2018b.

Sinong Wang, Belinda Z Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768*, 2020a.

Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 261–271. IEEE, 2020b.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, 2021.

Huihui Wei and Ming Li. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In *IJCAI*, pages 3034–3040, 2017.

Justin D Weisz, Michael Muller, Steven I Ross, Fernando Martinez, Stephanie Houde, Mayank Agarwal, Kartik Talamadupula, and John T Richards. Better together? an evaluation of ai-supported code translation. *arXiv preprint arXiv:2202.07682*, 2022.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

Rui Xie, Wei Ye, Jinan Sun, and Shikun Zhang. Exploiting method names to improve code summarization: A deliberation multi-task learning approach. *arXiv preprint arXiv:2103.11448*, 2021.

Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. In *Advances in neural information processing systems*, pages 5753–5763, 2019.

Michihiro Yasunaga and Percy Liang. Graph-based, self-supervised program repair from diagnostic feedback. In *International Conference on Machine Learning*, pages 10799–10808. PMLR, 2020.

Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 440–450, 2017.

Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. Big bird: Transformers for longer sequences. *Advances in Neural Information Processing Systems*, 33:17283–17297, 2020.

Jie M Zhang, Mark Harman, Lei Ma, and Yang Liu. Machine learning testing: Survey, landscapes and horizons. *IEEE Transactions on Software Engineering*, 2020.

Hao Zheng, Yong Cheng, and Yang Liu. Maximum expected likelihood estimation for zero-resource neural machine translation. In *IJCAI*, 2017.

RÉSUMÉ

Un transcompilateur est un système qui convertit le code source d'un langage de programmation de haut niveau (tel que C++ ou Python) vers un autre. Les transcompilateurs sont principalement utilisés pour l'interopérabilité et pour transférer des bases de code écrites dans un langage obsolète (par exemple COBOL ou Python 2) vers un langage plus moderne. Ils reposent généralement sur des règles de réécriture manuelles, appliquées à l'arbre de syntaxe abstraite du code source. Malheureusement, les traductions qui en résultent manquent souvent de lisibilité, ne respectent pas les conventions du langage cible et nécessitent des modifications manuelles pour fonctionner correctement. Le processus global de traduction prend du temps et nécessite une expertise à la fois dans les langages source et cible, ce qui rend les projets de traduction de code coûteux. Bien que les modèles neuronaux surpassent considérablement leurs homologues basés sur des règles dans le cadre de la traduction en langues naturelles, leurs applications à la transcompilation ont été limitées en raison de la rareté des données parallèles dans ce domaine. Nous proposons des méthodes pour entraîner des transcompilateurs neuronaux efficaces sans données supervisées.
Les traducteurs de langues naturelles sont évalués avec des métriques basées sur la cooccurrence de tokens entre la traduction et la référence. Nous remarquons que ces métriques ne capturent pas la sémantique des langages de programmation. Nous construisons et publions donc une base de données de tests composée de 852 fonctions parallèles, ainsi que de tests unitaires pour vérifier l'exactitude sémantique des traductions. Nous exploitons d'abord les objectifs conçus pour les langues naturelles afin d'apprendre des représentations multilingues du code source, et entraînons un modèle à traduire, en utilisant seulement le code monolingue de projets open source GitHub. Ce modèle surpasse les méthodes basées sur des règles pour la traduction de fonctions entre C++, Java et Python. Ensuite, nous développons une méthode de pré-entraînement, amenant le modèle à apprendre des représentations sémantiques du code. Cela conduit à des performances améliorées sur plusieurs tâches, y compris la traduction de code non supervisée. Enfin, nous utilisons des tests unitaires automatisés pour créer des exemples de traductions de programmes. Entraîner un modèle sur ces exemples conduit à des améliorations significatives des performances de nos transcompilateurs neuronaux. Nos méthodes reposent exclusivement sur du code source monolingue, ne nécessitent aucune expertise dans les langues source ou cible, et peuvent facilement être généralisées à d'autres langages.

MOTS CLÉS

transcompilation, langages de programmation, synthèse de code, traduction, réseaux de neurones, apprentissage profond

ABSTRACT

A transcompiler, also known as source-to-source translator, is a system that converts source code from a high-level programming language (such as C++ or Python) to another. Transcompilers are primarily used for interoperability, and to port codebases written in an obsolete or deprecated language (e.g. COBOL, Python 2) to a modern one. They typically rely on handcrafted rewrite rules, applied to the source code abstract syntax tree. Unfortunately, the resulting translations often lack readability, fail to respect the target language conventions, and require manual modifications in order to work properly. The overall translation process is time-consuming and requires expertise in both the source and target languages, making code-translation projects expensive. Although neural models significantly outperform their rule-based counterparts in the context of natural language translation, their applications to transcompilation have been limited due to the scarcity of parallel data in this domain. In this thesis, we propose methods to train effective and fully unsupervised neural transcompilers.
Natural language translators are evaluated with metrics based on token co-occurences between the translation and the reference. e identify that they do not capture the semantics of programming languages. Hence, we build and release a test set composed of 852 parallel functions, along with unit tests to check the semantic correctness of translations. We first leverage objectives designed for natural languages to learn multilingual representations of source code, and train a model to translate, using source code from open source GitHub projects. This model outperforms rule-based methods for translating functions between C++, Java, and Python. Then, we develop an improved pre-training method, which leads the model to learn deeper semantic representations of source code. It results in enhanced performances on several tasks including unsupervised code translation. Finally, we use automated unit tests to automatically create examples of program translations. Training on these examples leads to significant improvements in the performance of our neural transcompilers. Our methods rely exclusively on monolingual source code, require no expertise in the source or target languages, and can easily be generalized to other programming languages.

KEYWORDS

transcompilation, programming languages, program synthesis, translation, neural networks, deep learning, transformer