ICAI 2023
12th International Conference on Applied Informatics
Eger, Hungary, 2023

Deep learning based refactoring with formally verified training data

Balázs Szalontai[a*], Péter Bereczky[a*], Dániel Horpácsi[a†]

[a]ELTE Eötvös Loránd University
bukp00@inf.elte.hu, berpeti@inf.elte.hu, daniel-h@elte.hu

Abstract

Behaviour-preserving program rephrasing (known as *refactoring*) is an inevitable step in any software development process. The goal of refactoring is to improve the quality of software source code without altering its behaviour.

In current practice, refactoring is realized as a hand-coded transformation of a structured representation (such as a parse tree) of the source code, defined as syntactic rewriting. This makes the transformations only applicable to syntactically valid code, and only in cases that were explicitly defined. Our goal is to generalise this, such that we employ deep learning to extend the syntactically defined local refactoring steps to be applicable to 1) syntactically incomplete code and 2) constructs that are not explicitly mentioned in the definition but carry similar features. Our approach provides a flexible yet trustworthy[1] refactoring implementation[2] for refactoring based on deep learning, consisting of the following main steps:

1. We formally define refactoring steps as conditional syntactic rewrite rules. We verify the correctness of these steps by means of proving contextual equivalence between the matching and replacement patterns of the rewrite rules.

*Supported by the ÚNKP-22-3 New National Excellence Program of the Ministry for Culture and Innovation from the source of the National Research, Development and Innovation Fund.

†"Application Domain Specific Highly Reliable IT Solutions" project has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the Thematic Excellence Programme TKP2020-NKA-06 (National Challenges Subprogramme) funding scheme.

[1]Implemented on the basis of formal equivalence proofs.

[2]The method is implemented for Erlang, but we expect it to be adaptable to other languages. Similar techniques were formerly applied by the authors to Python in [6].

2. Then we take the rewrite rules and instantiate the metavariables with randomly generated expressions, yielding (formally proven) semantically equivalent expression pairs. We also generate random context around these expressions to ultimately obtain the *formally verified training data*.

3. We train a sequence-to-sequence network with attention mechanism to carry out the refactoring steps autonomously, possibly even on incomplete code fragments.

To our knowledge, deep learning techniques have not yet been applied to implement Erlang refactoring. Admittedly, such methods cannot always outperform classical approaches (such as tree transformations), it is nevertheless an active field of research. Thus, our goal is not to try to outperform existing approaches for refactoring Erlang source code, but to prove that deep learning techniques offer a nice way of generalising existing syntax-based techniques.

Formally verified refactoring steps

Refactoring verification boils down to the correctness of the individual refactoring steps, expressed as conditional syntactic rewrite rules. To prove the correctness of these, we need to show the contextual equivalence between the expressions on the left- and right-hand sides, with the conditions as assumptions.

Here is a simple example for a conditional rewrite rule in Erlang:

$$F(Xs)\texttt{ when length(}Xs\texttt{)==0 -> }B \;\rightarrow\; F(\texttt{[]})\texttt{ -> }B \;\textit{if}\; Xs \notin vars(B)$$

With this rewrite rule, the trivial guard of a function clause can be replaced by a (more readable and more effective) pattern matching. Note that F, Xs, and B are metavariables, where F can be instantiated with an arbitrary function name, Xs can be any variable name, while B can be replaced for any expression sequence. To verify this refactoring step, we need to prove the following contextual equivalence:

$$Xs \notin vars(B) \implies F(Xs)\texttt{ when length(}Xs\texttt{)==0 -> }B \equiv_{ctx} F(\texttt{[]})\texttt{ -> }B$$

In previous work, we have defined formal semantics for Core Erlang in the Coq proof assistant [1]. Core Erlang is a sublanguage of Erlang, and Erlang can be translated to it with the standard compiler (we consider this translation a trusted component in the verification process). In our current approach, we reason about equivalence in Erlang by proving the corresponding Core Erlang expressions equivalent, which are then used in the training data generation phase.

Generation of training data

We create training data by randomly generating programs implemented with the so-called data generators employed in property-based testing. In particular, we use a specialized variant of a grammar-based random program generator previously developed for testing refactoring steps and language semantics [2, 4].

Training data is synthesised by taking the proven-equivalent expression pairs and instantiating their metavariables with randomly generated expressions. The random synthesis is driven by an attribute grammar that describes the syntax and static semantics of the language. Each generated (concrete) expression instance represents a correct, successful application of the proven-correct refactoring step. We surround these expression pairs with randomised program context; in fact, the context is generated first, and the random expressions are generated such that they can refer to the variable and function environment synthesised into the context.

The implemented training data generation method allows us to easily synthesise a large and diverse collection of correct refactoring instances, which provides the necessary amount of information for the training process.

Approach of refactoring with deep learning

Having the data generated, we use deep learning to train a network capable of refactoring Erlang programs. In particular, we train a sequence-to-sequence architecture with attention mechanism to perform the code modifications on Erlang source code. Sequence to sequence networks have been one of the prominent approaches to refactor non-idiomatic source code into an idiomatic variant (such as [3, 5, 7]). In this setting, the program is not being parsed but only scanned, effectively representing the code as its token stream. As stated already, this can be advantageous as parse trees are difficult to construct from grammatically invalid or incomplete code, and syntactic validity is sometimes an unnecessary and impractical prerequisite.

First, we embed the tokens of the nonidiomatic code, which we feed into a recurrent encoder to capture an adequate hidden representation. The refactored code is then constructed token-by-token using a decoder, which consists of recurrent and feedforward layers. The decoding process is aided by attention mechanism.

References

[1] P. Bereczky et al.: *Machine-checked natural semantics for Core Erlang: exceptions and side effects*, in: Erlang'20, ACM, 2020, pp. 1–13, DOI: 10.1145/3406085.3409008.

[2] P. Bereczky et al.: *Validating Formal Semantics by Property-Based Cross-Testing*, in: IFL 2020, ACM, 2021, pp. 150–161, ISBN: 9781450389631, DOI: 10.1145/3462172.3462200.

[3] Z. Chen et al.: *Sequencer: Sequence-to-sequence learning for end-to-end program repair*, IEEE 47.9 (2019), pp. 1943–1959, DOI: 10.1109/TSE.2019.2940179.

[4] D. Drienyovszky, D. Horpácsi, S. Thompson: *Quickchecking Refactoring Tools*, in: Erlang '10, ACM, 2010, pp. 75–80, ISBN: 9781450302531, DOI: 10.1145/1863509.1863521.

[5] R. Gupta et al.: *DeepFix: Fixing common C language errors by deep learning*, in: Thirty-First AAAI Conference on Artificial Intelligence, 2017, DOI: 10.1609/aaai.v31i1.10742.

[6] B. Szalontai et al.: *Detecting and Fixing Nonidiomatic Snippets in Python Source Code with Deep Learning*, in: DOI: 10.1007/978-3-030-82193-7_9.

[7] M. Tufano et al.: *An empirical investigation into learning bug-fixing patches in the wild via neural machine translation*, in: ASE'18, 2018, pp. 832–837, DOI: 10.1145/3238147.3240732.