Special Issue of the Infocommunication Journal

Deep Learning-Based Refactoring with Formally Verified Training Data

Balázs Szalontai, Péter Bereczky and Dániel Horpácsi

***Abstract*—Refactoring source code has always been an active area of research. Since the uprising of various deep learning methods, there have been several attempts to perform source code transformation with the use of neural networks. More specifically, Encoder-Decoder architectures have been used to transform code similarly to a Neural Machine Translation task. In this paper, we present a deep learning-based method to refactor source code, which we have prototyped for Erlang. Our method has two major components: a localizer and a refactoring component. That is, we first localize the snippet to be refactored using a recurrent network, then we generate an alternative with a Sequence-to- Sequence architecture. Our method could be used as an extension for already existing AST-based approaches for refactoring since it is capable of transforming syntactically incomplete code. We train our models on automatically generated data sets, based on formally verified refactoring definitions and by using attribute grammar-based sampling.**

***Index Terms*—Deep learning, Formally verified training data, Neural Machine Translation, Sequence-to-Sequence.**

I. Introduction

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 observable behaviour [1].

Refactoring is commonly implemented as a transformation on a structured representation (such as a parse tree) of the source code. Admittedly, this approach works well in the typical scenarios with syntactically valid code; furthermore, when defined with syntactic rewriting, simple refactoring steps can be verified for correctness (semantics-preservation) by using formal methods. On the other hand, syntax-based approaches need to hardcode the logic of handling the various combinations of language constructs, and they cannot handle incomplete or ill-formed code fragments. In contrast, deep learning-based methods are inherently adaptive, and they can eliminate the need for hardcoding the vast amount of shapes and combinations syntactic constructs may take in a program. Due to its benefits, there has been an ever-growing interest in using deep neural networks for modifying source code recently. Some of these techniques, in addition to removing the burden of hand-crafting refactoring algorithms, come with the ability of transforming incomplete code fragments as the model is trained to transform code at the lexical level.

Balázs Szalontai, Department of Software Technology and Methodology, Péter Bereczky and Dániel Horpácsi, Department of Programming Languages and Compilers, ELTE Eötvös Loránd University, Budapest, Hungary (E-mail: {bukp00, berpeti}@inf.elte.hu, daniel-h@elte.hu)

Manuscript received April 10, 2023; revised August 21, 2023.

DOI: 10.36244/ICJ.2023.5.1

In this paper, we propose the combination of the above techniques: we apply deep learning for code refactoring and train on datasets generated with syntactic rewriting. We show that processing *Erlang* source code as a sequence of tokens and using deep learning methods to apply changes could serve as a great *extension* to the existing syntax-based methods, because our approach is capable of fixing incomplete or non-compilable code as well, supposing that the parts to be refactored are already complete. Moreover, we train our deep learning model on verified refactorings, that is, the code before and after the transformation are behaviourally indistinguishable. We present the following contributions:

- We formally define refactoring steps as conditional syntactic rewrite rules, and based on previous work [2], [3], [4], we verify the correctness of these steps by means of proving contextual equivalence (based on "CIU" equivalence [5]) between the matching and replacement patterns of the rewrite rules.
- Then we take the rewrite rules and instantiate the metavariables with randomly generated expressions, yielding semantically equivalent expression pairs. We also generate random context around these expressions to ultimately obtain the *formally verified training data*.
- Finally, we train a recurrent neural network to localize the code to be refactored and a Sequence-to-Sequence network with Attention Mechanism to carry out the refactoring steps autonomously. A very similar approach was presented in [6]. By using a similar architecture we show, that this approach is essentially language independent.

The paper is structured as follows. In Section II we discuss the related work, then in Sections III, IV, and V we show the above components of our approach. In Section VI we evaluate our approach, and finally Section VII concludes.

II. Related Work

There have been multiple attempts to transform source code with deep learning. The goal of such methods is generally to fix common errors (such as syntactic errors or semantic bugs) or to refactor code. Although in the current state of research, such techniques are not yet completely reliable, it is nevertheless a very active field.

Gupta et al. [7] aim to fix common C language errors with a Sequence-to-Sequence architecture. Their method is applied iteratively to fix errors one by one. Tufano et al. [8] train a recurrent Encoder-Decoder architecture on a dataset comprising data from pull requests. Their goal is to imitate human code fixing operations. Chen et al. [9] train a Sequence-to-Sequence architecture with Attention Mechanism to repair

programs. In their work, Copy Mechanism is also used to overcome the difficulties of the large number of possible identifiers occurring in code. Jiang et al. [10] first pretrain the model on a general task, then fine-tune an Encoder-Decoder architecture for the task of program repairing. Similarly to these works, we use a Sequence-to-Sequence (Encoder-Decoder) architecture with Attention Mechanism to transform nonidiomatic Erlang functions into their idiomatic alternative. We use formally verified data to train our model. Lutellier et al. [11] use a CNN-based Encoder-Decoder architecture and Ensemble learning techniques to automatically repair programs. Although our proposed Sequence-to-Sequence architecture is a recurrent one, we apply convolution on each chunk of the input source code when attempting to localize the nonidiomatically implemented ones. Chakraborty et al. [12] utilize a tree-based Encoder-Decoder architecture to modify code. First, structural modifications are applied, then the nodes of the modified tree are concretized. Li et al. [13] first train a tree-based recurrent model on the context of the code chunk to be transformed, then they use a tree-based Encoder-Decoder architecture for modifying the chunk. The downside of such tree-based methods is that they require the source code to be completed. A goal of ours is to experiment with incomplete code, which prevents the use of such tree-based approaches.

In previous work [6] we presented a method to localize and refactor nonidiomatic source code snippets in Python to improve code readability and program efficiency. The nonidiomatic snippet is localized using a model that performs sequence tagging: a recurrent model is trained to tag the source code lines with one of four tags (*START*, *IN*, *END*, *OUT*). This tagging indicates whether the line is part of a nonidiomatic snippet or not. The idiomatic alternative is generated based on the nonidiomatic snippet using a Sequence-to-Sequence architecture with attention. This approach of localizing and refactoring Python snippets is very similar to our proposed approach for refactoring Erlang source code.

III. Formally Verified Refactorings

In many cases, we encounter the problem of having a bad-quality dataset, which can decrease the performance of the neural network. Data collected from the web can contain errors, such as incorrect code transformations in the context of refactoring. In this work, we present a method that is trained on correct data. We achieve this by generating data according to formally verified equivalence rules. To the best of our knowledge, using such a verified dataset for training neural networks has never been proposed. Although this is not a common practice, being verified could be a highly desired aspect of training data.

In practice, refactorings are tested thoroughly, but are usually not proven-correct, thus in special circumstances (so-called edge cases) they could introduce errors. Let us motivate formal verification with a seemingly trivial example.[1] In Figure 1 we define the lazy version of conjunction in two ways, where one can be seen as a refactoring of the other. Without familiarity with Erlang, one may think that this two code fragments are equivalent definitions of the same functionality and transforming between them should not affect the program's behaviour.

```
lazy_and(X, Y) ->
  case X of
    true  -> Y;
    false -> false
  end.
```

(a) A correct implementation of lazy conjunction

```
lazy_and(X, Y) ->
  if X    -> Y;
     true -> false
  end.
```

(b) An incorrect implementation of lazy conjunction

Fig. 1: Motivational example for proving program equivalence

However, Erlang is dynamically typed, so `X` and `Y` can take any Erlang values, not just **`true`** or **`false`**—this introduces an extra level of complexity in the behaviour of these functions due to the dynamic type checking. In fact, it can be shown that the two variants of `lazy_and` do not behave the same way: in Figure 1a, if `X` was not a boolean value, we get an exception, while the function in Figure 1b will evaluate to **`false`**. Therefore, we can conclude that the transformation (read in either direction) in Figure 1 is *not* a refactoring.

To correct this inconsistency between the functions above (i.e., make them equivalent), we either need to change Figure 1a to use the wildcard pattern _ instead of **`false`** (in this case it would not be a correct implementation of conjunction any more, though), or we need to include a side condition requiring that `X` can only take boolean values. In either case, such simple, local refactorings are easiest carried out by defining and applying them as term rewriting rules [15], by extracting the irrelevant parts as metavariables denoting arbitrary expressions. This way, the concrete equivalent expression pairs can be obtained by instantiating the metavariables with expressions [16].

```
case e1 of true -> e2;          ⟶   if e1 -> e2;
        _ -> e3                      true -> e3
end                                  end
```

Fig. 2: Expression refactoring example

Figure 2 shows the previous expression refactoring as a rewrite rule with the wildcard pattern solution. As a matter of fact, this refactoring can be shown to be correct for any expressions e_1, e_2 and e_3.

```
f(x) when length(x) == 0 -> e
        ↓ when x ∉ vars(e1)
f([]) -> e
```

Fig. 3: Function clause refactoring example

Figure 3 presents another example, where the refactoring is applied to an Erlang function clause, and it shows how pattern matching can be used instead of a guard expression restricting the length of a list. The rewrite rule defining the refactoring has a side-condition, which needs to be met for the refactoring to be applicable: the parameter variable cannot appear in the

[1]The small refactoring examples we investigate here are inspired by Poór et al. [14].

function body. The variables f, x and e denote a function name, a variable, and an expression, respectively.

Formal semantics and program equivalence: To formally argue about the preservation of behaviour, we need a formal semantics of the language, and a suitable program equivalence definition. If two programs are proved to be equivalent, they are not distinguishable in any program context, that is, they can be exchanged.

Previously, we have defined several formal semantics for Core Erlang [2], [3], [4] which are capable of expressing program equivalence of Core Erlang expressions, and we also implemented these semantics in Coq. Core Erlang is an intermediate language of Erlang in the official implementation; we utilize this by reasoning about equivalence in Erlang via the trusted translation from Erlang to Core Erlang.

We use an (extended) version of the frame-stack semantics we defined previously [4], and the concept of CIU ("closed instances of use") equivalence [5]. The termination relation defined there is denoted by $\langle K, e\rangle \Downarrow$, meaning that expression e terminates in the frame stack K. The frame stacks describe continuations, i.e., K includes what should be evaluated next, once e has terminated. Here, we show a simplified version of the equivalence definition, and refer to [4] and the Coq formalisation [17] for further details.

Definition 1 (CIU equivalence)**.** $e_1 \equiv_{ciu} e_2 \stackrel{\text{def}}{=} (\forall K : \langle K, e_1\rangle \Downarrow \iff \langle K, e_2\rangle \Downarrow)$

We also showed [4] that reasoning about termination is sufficient to ensure the behavioural and contextual equivalence of the expressions (i.e., they evaluate to equivalent values, and equivalent expressions are interchangeable in arbitrary syntactic contexts). Based on the CIU equivalence, we can show the correctness of local refactoring steps.

Definition 2 (Correctness of refactorings)**.** *For all expressions e_1, e_2 and conditions P, $e_1 \to e_2$ when P is a correct refactoring step if P implies $e_1 \equiv_{ciu} e_2$.*

We have already proved the equivalence for the refactoring steps shown in Figure 2 and Figure 3. The proofs are extensive and their presentation is out of the scope of this paper, but we refer to the Coq formalisation [17] for more details.

IV. Generation of Training Data

To produce training data for the neural networks, we require two datasets: one dataset should contain *(nonidiomatic code, snippet location)* pairs to train the localizer network, while the other should consist of *(nonidiomatic snippet, idiomatic snippet)* pairs to train the refactoring network. We have set forth some general expectations for generating these datasets.

- The generated code must be syntactically correct and tokenizable.
- The generated code should share similarity with real-world code, i.e., it should apply functions of the standard library, apply other generated functions, use Erlang-specific values (e.g., `ok`, `false`, `true`), etc.
- The datasets must not only be diverse in terms of code size but also in the internal structure and meaning.
- Generating the idiomatic alternative must be deterministic, to allow the network to grasp the refactoring procedure.
- In the localizer's dataset, each nonidiomatic source code should contain at least one nonidiomatic snippet.
- Sufficient data should be available for training both networks: we believe that a good starting point would be to generate about 50.000 training examples in both datasets.

Based on the verified refactoring rules, we sample loads of concrete program modules including parts where the proven-correct refactoring steps can be applied. At the same time we synthesise the result of the refactoring too, by applying the rewrite rules.[2] With this dual synthesis, we create training data both for the localizing of refactoring candidates as well as for the application of the refactoring.

The program generation is based on a stochastic attribute grammar defining (a subset of) the Erlang programming language. In particular, we randomly generate elements of the language defined by the grammar, where the probabilities associated with the nonterminal symbols, along with some constrains carried in attributes, control the shapes and style of the generated programs; for instance, we can set the maximum number of functions and expressions within clauses, as well as we can fine-tune how deeply, and how likely, expressions get nested. For details about the attribute grammar notation, we refer to [18].

The data generation process consists of the following main steps:

- First a *module context* is generated by a modified variant of the above-mentioned attribute grammar, which produces modules that may contain so-called *holes* (holes mark the designated locations where code to be refactored will be emplaced). Holes encode information about their context (e.g. variable and function names in their scope) so that the refactoring candidates are generated context-sensitively.
- Then a refactoring rule is instantiated with the metavariables replaced by randomly generated names and subexpressions — each such instance will embody a correct rewrite step applied locally. Then the code snippets representing the target and the result of the refactoring are emplaced into the context that was generated in the previous step.
- Finally, generated code is dumped into a data file that annotates/labels each module with the location of the refactoring candidates and the type of refactoring applicable at each candidate.

This grammar-based method for generating the refactoring dataset has some clear advantages. First of all, the generated modules are syntactically and static semantically valid; thus, when the strings are tokenized, the token stream surely represents a well-formed program. Secondly, due to the controlled random sampling, the more cases we synthesise, the wider the variety in size and structure the generated programs expose.

[2]Since the equivalence relation is a congruence, the modules containing the refactored chunks are also equivalent.

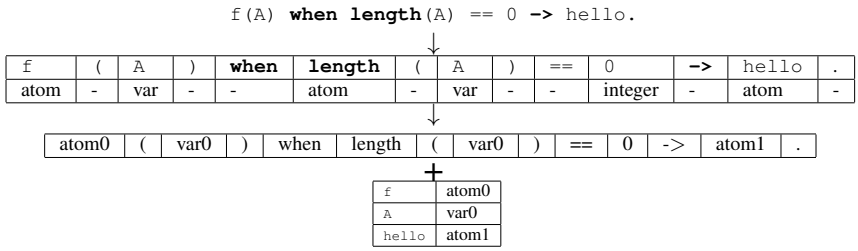

Fig. 4: The approach for tokenizing and creating the corresponding variable dictionary. The tokens `length` and `0` are kept in their original form, since they hold valuable information.

This guarantees that the contexts will teach various syntactic ways a refactoring candidate may be happen to be part of a program, and the refactoring instances themselves show a wide spectrum of syntactic variety. The generated programs are fully random, i.e., including programs that implement no useful behaviour. This is based on the fact that QuickCheck generators are implemented based on Erlang's pseudo-random seeder. We expect this not to cause an issue because the current rewriting approach is lexical, little to do with the actual semantics of the program under transformation.

V. Approach of Refactoring with Deep Learning

In this section, we offer a concise summary of the approach we took to localize and refactor function definitions in Erlang source code. First, we explain how we transformed a given piece of source code into a sequence of tokens that served as input to the models. Second, we introduce the neural network architecture that we trained to localize nonidiomatic code chunks. Finally, we describe the method for generating an alternative for the localized chunk using a Sequence-to-Sequence architecture with Attention Mechanism.

A. Preprocessing Source Code

To transform source code into a sequence of tokens, we undertake multiple steps. We utilize the Erlang module *tok* to tokenize the source code. Using this module, we obtain not only the tokens themselves, but also their types, such as *atom*, *integer*, *variable*, etc. To tackle the challenge of handling the unlimited number of valid identifiers, numbers, variables, etc., present in code, we replace these tokens with their type and a unique index. There are some exceptions to this process: the tokens that hold valuable information are not replaced, such as `length`, `0`, `true`, etc. We drew inspiration for this approach from the work of Chirkova et al. [19]. When generating an idiomatic alternative, we invert the changes made to these tokens that appear in the model's output. The process is visualized on Figures 4 and 5: Figure 4 shows how a function implementation is turned into a sequence of tokens and its corresponding dictionary. Figure 5 shows how the changes are inverted for an idiomatic alternative generated by the refactoring model.

B. Localizing Nonidiomatic Functions

Localizing nonidiomatic patterns (i.e., refactoring candidates) in source code is solved as a sequence tagging task.

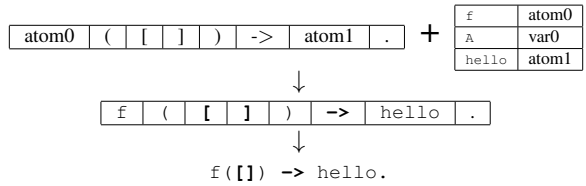

Fig. 5: The approach for turning the output of the refactoring model back to a code string.

The idea behind this approach is that we tag each chunk of the source code with one of two tags: *OUT* or *IN*. The used tag indicates whether a certain chunk of code is to be refactored (*IN*) or not (*OUT*). Using this approach, we gather candidates for refactoring, while the rest of the code remains unchanged. The source code gets splitted by '.' characters. This way, the function definitions get separated and are ready to be tagged.

The proposed neural network architecture (Figure 6) consists of convolutional, recurrent, and feedforward components. Firstly, the preprocessed (tokenized, splitted) code is provided as input to the network. Secondly, the tokens of each chunk are embedded into a 64-dimensional vector space. Thirdly, a one-dimensional convolution is applied to each code chunk using 128 filters and a kernel size of 5. Fourthly, two pooling operators are applied to the convolutional outputs: average and minmax. The average pooling calculates the element-wise average, while the minmax pooling is a unique pooling operator that calculates the element-wise maximum or minimum depending on which value is further away from the average. These pooling operations yield two intermediate representations for each chunk of the source code.

Having the intermediate representations provided by the two pooling operators, the following step is to obtain context-dependent aspects for the chunks as well. To achieve this, we feed the intermediate representations into two separate Bidirectional LSTM[3] (BiLSTM) layers with 32 units and two fully connected layers with 64 units. The BiLSTM outputs are fed into another set of two BiLSTM layers. The activation function used for both the BiLSTM and fully connected layers is *tanh*, and 20% dropout is applied after each. As a result of these steps, we obtain four vectors that represent each chunk of code. These four vectors are concatenated to create a final hidden representation, which is then passed

[3]The LSTM layers used in our method all return the whole sequence of outputs.

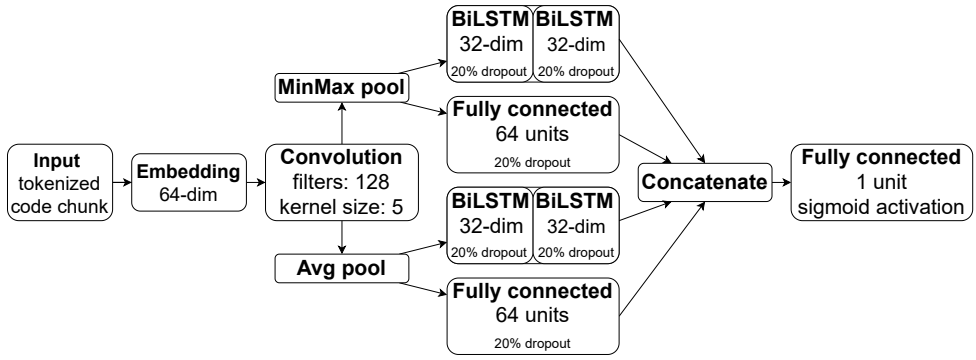

Fig. 6: Proposed neural network architecture to localize refactoring candidates

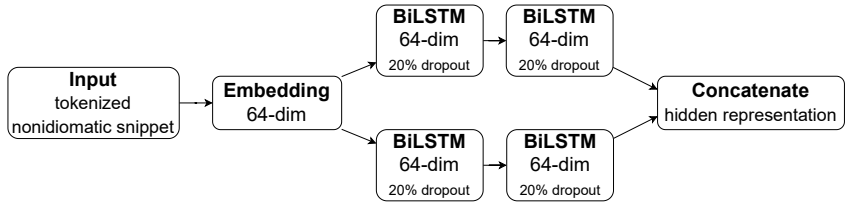

Fig. 7: Proposed architecture of the encoder of the refactoring model

to a fully connected layer to obtain the final output, where the fully connected layer has one unit and uses the *sigmoid* activation function. This result indicates which chunks are to be refactored (1 for *IN*) and which are not (0 for *OUT*). The Adam optimizer is used with a learning rate of 0.001, and binary cross-entropy is the loss function.

We based the selection of hyperparameters on previous work [6] and on intuition. This is the case for both the localizer and refactoring component. Since our current goal is to prove that using deep learning for refactoring nonidiomatic Erlang code is a viable option, we did not focus on optimizing the hyperparameters too much.

C. Generating Idiomatic Alternative

The idiomatic alternative is generated using a recurrent Sequence-to-Sequence architecture with Attention Mechanism. That is, we first feed the tokenized nonidiomatic code to the encoder, and then expect the decoder to generate the idiomatic alternative token-by-token.

The encoder component (Figure 7) aims to produce a hidden representation of the input sequence. This is achieved through the use of a recurrent neural network consisting of four BiLSTM layers with 64 units each. First, the tokens of the input sequence are embedded into a 64-dimensional vector space. Next, the sequence of tokens is fed into the four BiLSTM layers in the following way: the input sequence is fed into two BiLSTM layers, then the outputs of these layers are fed into another BiLSTM layer each. The resulting outputs of the two latter BiLSTM layers are concatenated to produce the final output of the encoder, which serves the hidden representation of the code chunk. Additionally, the BiLSTM layers are followed by 20% of dropout. This architecture is designed with the aims of (1) obtaining two independent representations of the same code chunk by using parallel BiLSTM layers, and (2) acquiring a higher-level representation by using such a stacked architecture.

The decoder uses a single LSTM layer with 256 units to generate an output sequence element-by-element. The LSTM layer takes as input the fixed-length vector representation produced by the encoder, along with the previously generated output token (which is of the idiomatic code). For the first element of the output sequence, a special $START$ token is used in place of the previous output. An attention layer is used to compute the attention weights between the encoder output and the decoder outputs. The attention weights are used to compute the context vector, which then gets concatenated with the decoder outputs. The concatenated vector is passed through a fully connected layer with *softmax* activation to obtain the final decoder outputs, which determines the next token of the idiomatic code. We use the Adam optimizer with the learning rate of 0.001 and categorical crossentropy as the loss function.

VI. Evaluation and Experiments

In this section, we first present the measured accuracy of the two main components of our approach: the localizer and refactoring models. Then we showcase experiments on our method's capability to perform refactoring steps on Erlang code. Finally, we make notes about the usability of our method on real-world codes.

Both the localizer and refactoring models were evaluated on a test set that was separated from the training data before the training process: 10% for the localizer and 4% for the refactoring model. The accuracy of the localizer model is the ratio of the correctly classified code chunks divided by the total number of chunks in the test set: this ratio turned out to be **99.09%**. For the refactoring component, we measured

TABLE I
EXAMPLE OF REFACTORED CODE SNIPPET

Original (nonidiomatic) code		Refactored code
`f(L) when length(L) == 0 -> error;` `f(L) -> lists:max(L).`	→	`f([]) -> error;` `f(L) -> lists:max(L).`
`f(L) when length(L) == 0 ->` `  case X of 0 -> true; 1 ->`	→	`f([]) ->` `  case X of 0 -> true; 1 ->`

TABLE II
EXAMPLE OF REFACTORED SOURCE CODE

Original source code	Refactored source code
`-module(mod).`	`-module(mod).`
`-compile(export_all).`	`-compile(export_all).`
`fact(0) -> 1;`	`fact(0) -> 1;`
`fact(N) -> N * fact(N - 1).`	`fact(N) -> N * fact(N - 1).`
`f(L) when length(L) == 0 -> error;`	`f([]) -> error;`
`f(L) -> double(lists:max(L)).`	`f(L) -> double(lists:max(L)).`
`double(N) -> N * 2.`	`double(N) -> N * 2.`

the ratio of error-free transformations against the total number of attempted transformations. The resulting accuracy of this evaluation was **99.46%**. These results indicate that our models have been trained succesfully and are capable to perform refactorings that are similar to the ones in the training datasets.

We now present some of our experiments with refactoring various kinds of nonidiomatic programs, including complete and incomplete programs. First, we focus only on the refactoring component by experimenting with refactoring nonidiomatic snippets without their surrounding context. After running our model on some nonidiomatic snippets, we compared the output of the model with the original code and checked whether the output is more idiomatic after the changes have been applied. An example of such a refactoring is shown in the first row of Table I: here the original version used a guard to handle empty lists, which was replaced by pattern matching syntax. The refactored code is an equivalent and more idiomatic compared to the original code chunk.

As mentioned earlier, we also experimented with refactoring incomplete code. Of course this only makes sense if the parts that make the code nonidiomatic are present. In such a scenario, our model attempts to refactor the nonidiomatic part(s) only and leave the rest unchanged to allow for completion. The second row of Table I shows such a refactoring. The transformation was successful - that is, the incomplete part was left unaltered - in spite of the fact that the model was not trained on incomplete code.

Next, we performed further experiments on the entire method, including the localizer component. That is, we applied our method on a full Erlang code that contained a nonidiomatic function implementation. The expected result of the method is of course the modified code that only varies in the originally nonidiomatic section. A transformation is correct if the generated alternative is the properly refactored version of the original nonidiomatic snippet, while the behavior of the entire program is preserved. An example of such a refactoring performed by our method is shown in Table II.

While our proposed approach shows promising results in refactoring nonidiomatic Erlang code, it represents an initial proof of concept. At the current stage of research and development, our method cannot be used on real-world codes, because it too often classifies code chunks as refactoring candidates even when they are not. Since incorrectly localized snippets cannot be refactored, our idiomatizer network obviously fails to refactor correctly, which leads to broken code.

We are planning to address this issue by considering multiple approaches. Firstly, we will investigate ways to generate training data that is more representative and closer to real-world code. Secondly, we will explore other approaches for finding the best way to split the original source code: it could be the case that too much unnecessary information is given to the network at once, caused by splitting code by '.' characters. This might make it harder to decide whether or not to refactor. Thirdly, we will experiment with some architectural modifications for the localizer component. Possible modifications include introducing different types of layers (such as GRU or Convolutional LSTM) and optimizing the hyperparameters. Lastly, it is possible to perform an extra pass on the output of our method to filter out some incorrect refactorings, for example if the code before the refactoring was compilable, it should be compilable after it too. We note that although it is also theoretically possible to filter out incorrect outputs by post-verifying each refactoring instance and proving equivalence, it would be very costly as the formal verification is manual.

VII. CONCLUSION

In this paper, we have presented a novel approach to refactor source code using deep learning techniques. We prototyped this approach for Erlang. Our method includes a localizer and a refactoring component, which enable the localization and refactoring of nonidiomatic code patterns into their idiomatic counterparts. Our method processes the source code as a sequence of tokens, making it capable of transforming even incomplete or non-compilable code.

To ensure that the neural networks learn how to correctly refactor, we used formally verified data, which we obtained by instantiating conditional term rewrite rules whose behaviour preservation is formally proven. We do not aim to change already existing AST-based approaches, but rather propose our deep learning-based approach as an *extension* to these.

Finally, we highlight some areas for possible future work:

- Proving the correctness of a larger number of local refactorings, and including them into our approach.

- Extending the generator component to emit code including more language constructs (e.g., strings), standard library functions (e.g., calls to higher-order functions).
- Applying different tagging approaches for the localizer component in order to be able to identify refactoring candidates more precisely and limiting number of false positive localizations.
- Investigating other Sequence-to-Sequence architectures for refactoring, such as CNN-based [20] or Transformer [21].
- Performing a comprehensive analysis to evaluate the performance of the presented method on real-world code.

Acknowledgements

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.

References

[1] M. Fowler, *Refactoring: improving the design of existing code*. USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN: 0201485672.

[2] P. Bereczky, D. Horpácsi, and S. Thompson, "A proof assistant based formalisation of a subset of sequential Core Erlang," in *Trends in Functional Programming*, A. Byrski and J. Hughes, Eds. Cham: Springer, 2020, pp. 139–158, DOI: 10.1007/978-3-030-57761-2_7.

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

[4] D. Horpácsi, P. Bereczky, and S. Thompson, "Program equivalence in an untyped, call-by-value functional language with uncurried functions," *Journal of Logical and Algebraic Methods in Programming*, vol. 132, p. 100 857, 2023, DOI: 10.1016/j.jlamp.2023.100857.

[5] I. Mason and C. Talcott, "Equivalence in functional languages with effects," *Journal of Functional Programming*, vol. 1, no. 3, pp. 287–327, 1991, DOI: 10.1017/S0956796800000125.

[6] B. Szalontai, Á. Kukucska, A. Vadász, B. Pintér, and T. Gregorics, "Localizing and idiomatizing nonidiomatic python code with deep learning," in *Proceedings of the Computing Conference 2023*, June 2023, to appear.

[7] 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.

[8] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, "On learning meaningful code changes via neural machine translation," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 25–36, DOI: 10.1109/ICSE.2019.00021.

[9] Z. Chen et al., "Sequencer: Sequence-to-sequence learning for end-to- end program repair," *IEEE*, vol. 47, no. 9, pp. 1943–1959, 2019, DOI: 10.1109/TSE.2019.2940179.

[10] N. Jiang, T. Lutellier, and L. Tan, "CURE: Code-aware neural machine translation for automatic program repair," in *2021 IEEE/ACM 43rd International Conferenceon Software Engineering (ICSE)*. IEEE, 2021, pp. 1161–1173, DOI: 10.48550/arXiv.2103.00073.

[11] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. 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*, 2020, pp. 101–114, DOI: 10.1145/3395363.3397369.

[12] S. Chakraborty, Y. Ding, M. Allamanis, and B. Ray, "CODIT: Code editing with tree-based neural models," *IEEE Transactions on Software Engineering*, vol. 48, no. 4, pp. 1385–1399, 2020, DOI: 10.48550/arXiv.1810.00314.

[13] Y. Li, S. Wang, and T. N. Nguyen, "Dlfix: Context-based code transformation learning for automated program repair," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 602–614, DOI: 10.1145/3377811.3380345.

[14] B. Poór, M. Toth, and I. Bozó, "Transformations towards clean functional code," in *Proceedings of the 19th ACM SIGPLAN International Workshop on Erlang*, ser. Erlang 2020. New York, NY, USA: Association for Computing Machinery, 2020, pp. 24–30, DOI: 10.1145/3406085.3409010.

[15] F. Baader and T. Nipkow, *Term rewriting and all that*. Cambridge University Press, 1998, DOI: 10.1017/CBO9781139172752.

[16] D. Horpácsi, J. Kőszegi, and S. Thompson, "Towards trustworthy refactoring in Erlang," Electronic Proceedings in *Theoretical Computer Science*, vol. 216, pp. 83–103, jul 2016, DOI: 10.4204/eptcs.216.5.

[17] H.-A. R. Project. (2023) Core Erlang formalization. Accessed on 3rd of August, 2023. [Online]. Available: https://github.com/harp-project/Core-Erlang-Formalization/releases/tag/v1.0.3

[18] D. Drienyovszky, D. Horpácsi, and S. Thompson, "Quickchecking refactoring tools," in *Proceedings of the 9th ACM SIGPLAN Workshop on Erlang*, ser. Erlang '10. New York, NY, USA: Association for Computing Machinery, 2010, pp. 75–80, DOI: 10.1145/1863509.1863521.

[19] N. Chirkova and S. Troshin, "A simple approach for handling out-of-vocabulary identifiers in deep learning for source code," 2020, DOI: 10.48550/arXiv.2010.12663.

[20] J. Gehring, M. Auli, D. Grangier, D. Yarats, and Y. N. Dauphin, "Convolutional sequence to sequence learning," in *International conference on machine learning*. PMLR, 2017, pp. 1243–1252, DOI: 10.48550/arXiv.1705.03122.

[21] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017, DOI: 10.48550/arXiv.1706.03762.

Balázs Szalontai has completed his bachelor's degree in computer science at Eötvös Loránd University in 2020, followed by a master's degree in the same field at the same university in 2022. He is currently a Ph.D. student. His research focuses on source code transformation with Deep Learning. He also gives classes for university students in the fields of classical Artificial Intelligence, Object Oriented Programming and Logic Programming.

Péter Bereczky is a Ph.D. student at the Faculty of Informatics, Eötvös Loránd University. He received his master's degree in computer science at the same university, in 2020. His research interest include formal verification, formal semantics of programming languages, formal logics, functional programming, and interactive theorem proving.

Dániel Horpácsi is an assistant professor at Eötvös Loránd University. He received his Ph.D. for developing methods for verification and application of program transformations in functional programming languages, especially refactoring in Erlang. He keeps exploring the pragmatics of using formal methods in practical software tools, making them more reliable whilst maintaining their flexibility and accessibility.