Project Proposal: Learning Variable Mappings to Repair Programs

Pedro Orvalho[1]*, Jelle Piepenbrock[23], Mikoláš Janota[2], and Vasco Manquinho[1]

[1] INESC-ID, IST - U. Lisboa, Portugal
[2] CIIRC, Czech Technical University in Prague, Czechia
[3] Radboud University Nijmegen, The Netherlands

Abstract

The increasing demand for programming education has given rise to all kinds of online evaluations, such as Massive Open Online Courses (MOOCs) focused on introductory programming assignments (IPAs), especially over the last few years due to the coronavirus outbreak. As a consequence of a large number of enrolled students, one of the main challenges in these courses is to provide valuable and personalized feedback to students. This personalized feedback can be provided as a list of possible repairs to a student's program. Typically semantic program repair tools repair an incorrect program using a correct implementation for the same IPA. In order to compare both programs, a relation between both programs' sets of variables is required. Thus, in this work, we propose to learn how to map the set of variables between different small imperative programs based on both programs' abstract syntax trees (ASTs) using graph neural networks (GNNs).

Introduction. Program Synthesis, the task to automatically generate programs and mathematical objects that satisfy a given high-level specification [3, 12], is a well-studied problem in Theorem Proving [4, 5], and it has even been considered the Holy Grail of Computer Science [6, 9]. Program Repair can be seen as a special case of Program Synthesis, where a given program has a faulty region that needs to be repaired by synthesizing a correct patch or by reusing code snippets from other correct programs. Automated program repair [1, 7, 8, 13] has become crucial to provide feedback to each novice programmer by checking their introductory programming assignments (IPAs) submissions using a pre-defined test suite. Semantic program repair frameworks use a correct implementation, provided by the lecturer or submitted by a previously enrolled student, to repair a new incorrect student's submission. These tools need to compare both programs, i.e., the correct and the faulty implementation. In order to compare both programs, a relation between both programs' sets of variables is required. For example, consider both programs presented in Listings 1, where having a mapping between both programs' variables lets us reason about which repairs one should perform to fix the faulty program. In this position paper, we propose to take advantage of the structural information of the *abstract syntax trees (ASTs)* of small imperative programs to learn how to map the set of variables between a correct program and a faulty one using *graph neural networks (GNNs)*.

IPAs Dataset. We used the C-Pack-IPAs [10] benchmark to evaluate this work. This benchmark is composed by student programs developed during an introductory programming course in C language were collected at Instituto Superior Técnico. First, we selected only submissions that compiled without any error and satisfied a set of input-output test cases for each IPA. Afterwards, we used MultIPAs [11], a program transformation tool that can augment IPAs benchmarks by performing program mutations and introducing bugs to the programs.

*This work was done while this author was visiting CIIRC, CTU in Prague.

Listing 1: Function that finds and returns the maximum number among `n1, n2` and `n3`.

```
int max(int n1, int n2, int n3)
{
  int m = n1 > n2 ? n1 : n2;
  return n3 > m ? n3 : m;
}
```

Listing 2: Function that finds and returns the maximum number among `x, y` and `z`.

```
int max(int x, int y, int z){
  int m = 0;
  m = x > m ? x : m;
  m = y > m ? y : m;
  return z > m ? z : m;
}
```

Listings 1: Both functions find and return the maximum number among their parameters' values. However, the function in Listing 2 is only correct for positive numbers, if we consider negative numbers the function is incorrect since it assigns the variable `m` to 0. The mapping between these programs' sets of variables is {`m : m`; `n1 : x`; `n2 : y`; `n3 : z`}.

MultIPAs can perform simple mutations to each program (e.g. swapping comparison operators, swapping the if's then-block with the else-block and negating the test condition) to generate semantically equivalent programs with the same variables. Hence, we gathered a dataset of programs and the mappings between their sets of variables. For example, we have 94 correct submissions for the first IPA. By just swapping comparison operators (e.g. $\geq, \leq, ==, \neq$), we are able to compute a dataset of 27261 pairs of programs and the mappings between their sets of variables. We plan to perform more complex mutations to the set of IPAs.

Program Representations. We represent programs based on their abstract syntax trees (ASTs). An AST is described by a set of nodes that correspond to non-terminal symbols in the programming language's grammar and a set of tokens that correspond to terminal symbols. Then, we create a unique node in the graph for each distinct variable in the program and connect all the variable occurrences in the program to the same unique node. Regarding the edges of the program representation, we consider two types of edges in our representation: child and sibling edges. Child edges correspond to the typical edges in the AST representation that connect each parent node to its children. Child edges are bidirectional. Sibling edges connect each child to its sibling successor. These edges denote the order of the arguments for a given node [2]. Sibling edges allow the program representation to differentiate between different arguments when the order of the arguments is important (e.g. binary operation such as $\leq$). For example, consider the node that corresponds to the operation $\sigma(A_1, A_2, \ldots, A_m)$. The parent node σ is connected to each one of its children by a child edge e.g. $\sigma \leftrightarrow A_1, \sigma \leftrightarrow A_2, \ldots, \sigma \leftrightarrow A_m$. Additionally, each child its connected to its successor by a sibling edge e.g. $A_1 \rightarrow A_2, A_2 \rightarrow A_3, \ldots, A_{m-1} \rightarrow A_m$. The interested reader is referred to appendix A for a graphical representation of a small example.

GNNs. Graph Neural Networks are a subclass of neural networks designed to operate on graph-structured data, which may be citation networks, first-order logic or representations of computer code. Here, we use a pair of ASTs, representing two programs for which we want to match variables, as the input. The main operative mechanism is to perform *message passing* between the nodes, so that information about the global problem can be passed between the local constituents. The content of these messages and the final representation of the nodes is parameterized by neural network operations (matrix multiplications composed with a non-linear function). For the variable matching task, we do the following to train the parameters of the network. After several message passing rounds, through the edges defined by the program

representations above, we obtain numerical vectors corresponding to each variable node in the two programs. We compute scalar products between each possible combination of variable nodes in the two programs, followed by a softmax function. As the correct mapping of variables is known because the samples are obtained by program mutation, we can compute a cross-entropy loss and minimize it so that the network output corresponds to the labeled variable matching.

Acknowledgments. This research was supported by Fundação para a Ciência e Tecnologia (FCT) through grant SFRH/BD/07724/2020, and projects UIDB/50021/2020, PTDC/CCI-COM/32378/2017; by European funds through COST Action CA2011; and by the Ministry of Education, Youth and Sports within the program ERC CZ under the project POSTMAN no. LL1902.

References

[1] Umair Z. Ahmed, Zhiyu Fan, Jooyong Yi, Omar I. Al-Bataineh, and Abhik Roychoudhury. "Verifix: Verified Repair of Programming Assignments". In: *ACM Trans. Softw. Eng. Methodol.* (2022). ISSN: 1049-331X. DOI: 10.1145/3510418. URL: https://doi.org/10.1145/3510418.

[2] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. "Learning to Represent Programs with Graphs". In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. 2018.

[3] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. "DeepCoder: Learning to Write Programs". In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. 2017.

[4] Chad E. Brown and Thibault Gauthier. "Self-Learned Formula Synthesis in Set Theory". In: *CoRR* abs/1912.01525 (2019).

[5] Thibault Gauthier. "Synthesis of Recursive Functions from Sequences of Natural Numbers1". In: *6th Conference on Artificial Intelligence and Theorem Proving, AITP* (2021).

[6] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. "Program Synthesis". In: *Foundations and Trends in Programming Languages* 4.1-2 (2017), pp. 1–119.

[7] Sumit Gulwani, Ivan Radicek, and Florian Zuleger. "Automated clustering and program repair for introductory programming assignments". In: *PLDI 2018*. ACM, 2018, pp. 465–480.

[8] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish K. Shevade. "DeepFix: Fixing Common C Language Errors by Deep Learning". In: *AAAI 2017*. Ed. by Satinder P. Singh and Shaul Markovitch. AAAI Press, 2017, pp. 1345–1351.

[9] A Solar Lezama. "Program synthesis by sketching". PhD thesis. UC Berkeley, 2008.

[10] Pedro Orvalho, Mikoláš Janota, and Vasco Manquinho. *C-Pack of IPAs: A C90 Program Benchmark of Introductory Programming Assignments*. 2022. DOI: 10.48550/arXiv.2206.08768. URL: https://doi.org/10.48550/arXiv.2206.08768.

[11] Pedro Orvalho, Mikoláš Janota, and Vasco Manquinho. "MultIPAs: Applying Program Transformations to Introductory Programming Assignments for Data Augmentation". In: *ESEC/FSE '22: 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Singapore, November 14-18, 2022*. ACM, 2022. DOI: 10.1145/3540250.3558931. URL: https://doi.org/10.1145/3540250.3558931.

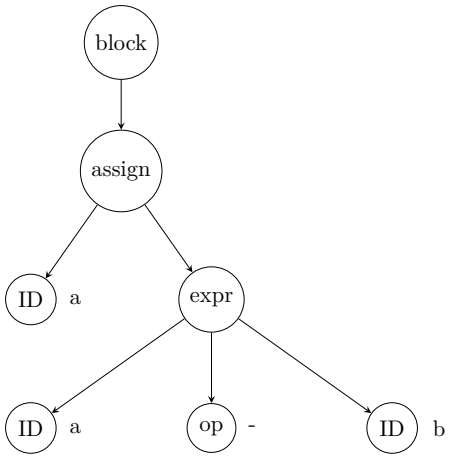
[12] Pedro Orvalho, Miguel Terra-Neves, Miguel Ventura, Ruben Martins, and Vasco M. Manquinho. "Encodings for Enumeration-Based Program Synthesis". In: *Principles and Practice of Constraint Programming - 25th International Conference, CP 2019, Stamford, CT, USA, September 30 - October 4, 2019, Proceedings*. 2019, pp. 583–599.

[13] Michihiro Yasunaga and Percy Liang. "Graph-based, Self-Supervised Program Repair from Diagnostic Feedback". In: *ICML 2020*. Vol. 119. Proceedings of Machine Learning Research. PMLR, 2020, pp. 10799–10808.

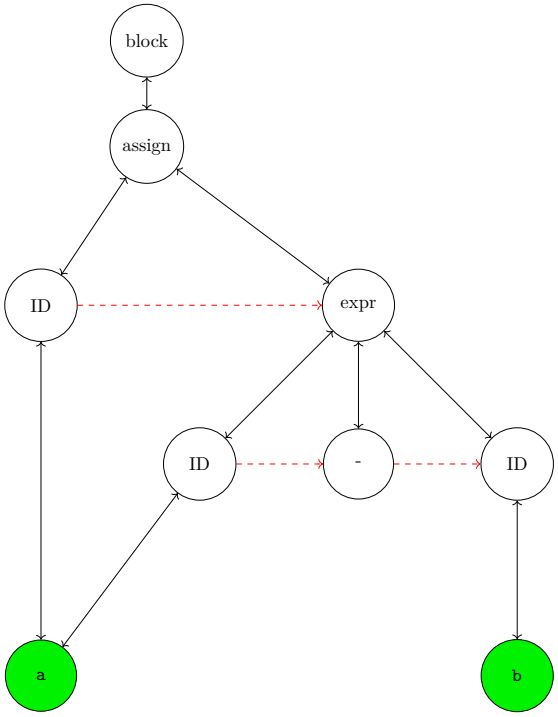
A Appendix

Listing 3: Small example of a C code block with an expression that uses int variables `a` and `b`, previously declared in the program.

```
{
  // a and b are ints
  a = a - b;
}
```



(a) Part of the AST representation of Listing 3.



(b) Our program representation for the program presented in Listing 3. We add additional variable nodes (green nodes), new sibiling edges (red dashed edges) and we also make the AST edges (black edges) bidirectional.

Figure 1: AST and our program representation for the small code snippet presented in Listing 3.