Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

AI For Better Software

Automatic bug fixing with reverse program analyser

Master's Thesis

Romane Castel

rcastel@enssat.fr

DeepCode
SRI Lab
ETH Zürich

Supervisors:
Dr. Veselin Raychev, Prof. Dr. Martin Vechev
Prof. Dr. Gwénolé Lecorvé

September 7, 2020

Abstract

Deep learning has seen an increased interest in the recent ten years as it allows to solve tasks that are revealed to be complex. Moreover, as the information technologies field is becoming more and more important in the society, there will be more developers to write code. As it is impossible for humans to write a code without bugs, and that code debugging is mainly a task done manually, the interest in developing a tool that is able to correct bugs automatically is obvious. The idea is to parse source codes into structures that we know how to manipulate and analyse such as trees and graphs and extract from them sequences of instructions to fix the bugs in those structures. At the end, after reversing the parsing, the source codes would have been fixed. Those sequences may be learnt by deep learning techniques.

L'apprentissage profond a suscité un intérêt grandissant ces dix dernières années étant donné que cette technique permet de résoudre des tâches complexes. De plus, l'industrie des technologies de l'information est en plein essor, c'est pourquoi le nombre de dévelopeurs en activité augmente. Comme il leur est impossible d'écrire du code sans erreur et que corriger ses erreurs est une tâche essentiellement effectuée manuellement, l'intérêt de développer un outil qui puisse corriger de manière automatique ces bugs est évident. L'idée de cet outil est de représenter le code source par des structures que nous savons manipuler et analyser comme les graphes et les arbres. Il s'agit d'extraire de ces structures des séquences d'instructions qui permettent de corriger les erreurs dans ces représentations. Après quoi, nous pouvons extraire des ces structures corrigées un code qui est lui même corrigé. Ces sequences de correction peuvent être apprises par apprentissage profond.

Contents

Chapter 1

Thesis' purposes

1.1 Introduction

The high level goal of this thesis is to automatically learn how to fix bugs in source codes.

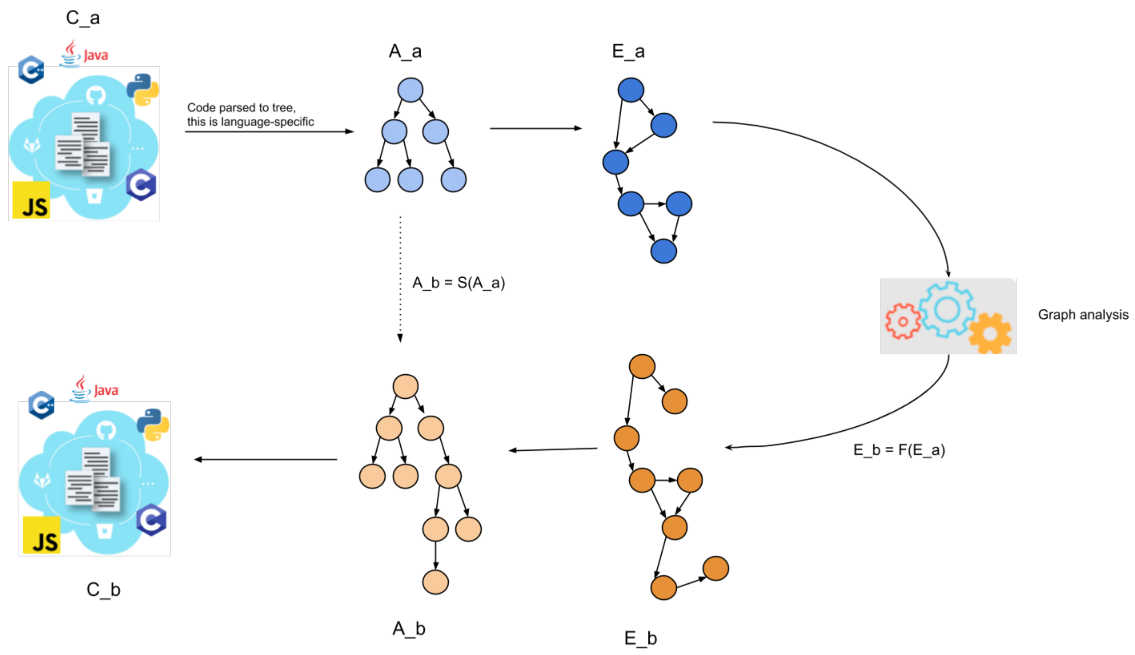
Note: the notation A_i represents abstract syntax trees (ASTs) and E_i represents event graphs (EGs).

The goal is, given a buggy code C_a (in Java, C/C++, Python or JavaScript), to generate all the fixed versions C_b for all the bugs detected in C_a. For that, from the buggy code C_a we generate the buggy AST A_a and from A_a we generate the buggy EG E_a. E_a is analysed and some bugs are found. For each bug found, we know exactly which nodes of E_a are concerned by the bug (so we know what parts of the code are concerned by the bug); and E_a is modified in order to get a fixed version E_b that does not contain the bug anymore. From this fixed E_b a fixed AST is extracted: A_b. And finally we get C_b the fixed version of the code C_a regarding the concerned bug (see figure 1.1). Note: all the nodes that are concerned by a specific bug in E_a compose what we call the matching area of the bug. **The main focus of this thesis is the step where we must generate a fixed AST from a fixed EG regarding one of the bugs detected.**

Note: those structures are used, because the work we use dispose of a static program analysis tool that is used on event graph and that event graphs are generated from abstract syntax trees.

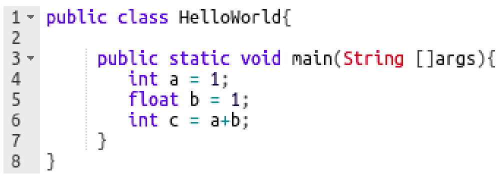
Let's consider an example containing only one bug. In figure 1.2, there are two pieces of code in Java. The programmer wants to add two int numbers, but in the buggy code (code(a) in figure 1.2) a float and an integer are added, so it brings an error such as: "error: incompatible types". This bug can be easily fixed by replacing `float b = 1;` by `int b = 1;`. The correction process is the following: we generate the AST for the buggy code (tree on the top in figure 1.3) and the EG for the buggy code (graph on the left in figure 1.4). The static analysis is done on the buggy EG, and in order to fix the bug

Figure 1.1: High level pipeline



"error: incompatible types" we must modify two nodes in the buggy EG (they compose the matching area of the bug detected); so that we get the fixed EG (graph on the right in figure 1.4). From this fixed EG, we extract the fixed tree (tree on the bottom in figure 1.3), and we see that in order to get this fixed AST we only need to modify one node in the buggy AST. And then finally, from the fixed AST we can extract the fixed code (code(b) in figure 1.2).

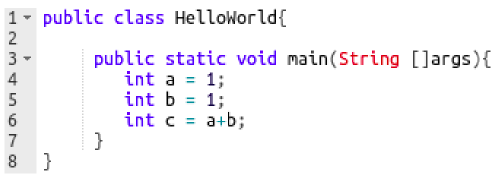
In this thesis there are two main stages: the training stage and the inference stage.
- During the training stage, we compare many pairs of codes C_a and C_b for which we generate A_a, A_b, E_a and E_b. The EG E_a is analysed and several bugs may be found in



```
public class HelloWorld{

    public static void main(String []args){
        int a = 1;
        float b = 1;
        int c = a+b;
    }
}
```

(a)



```
public class HelloWorld{

    public static void main(String []args){
        int a = 1;
        int b = 1;
        int c = a+b;
    }
}
```

(b)

Figure 1.2: (a) buggy and (b) fixed code

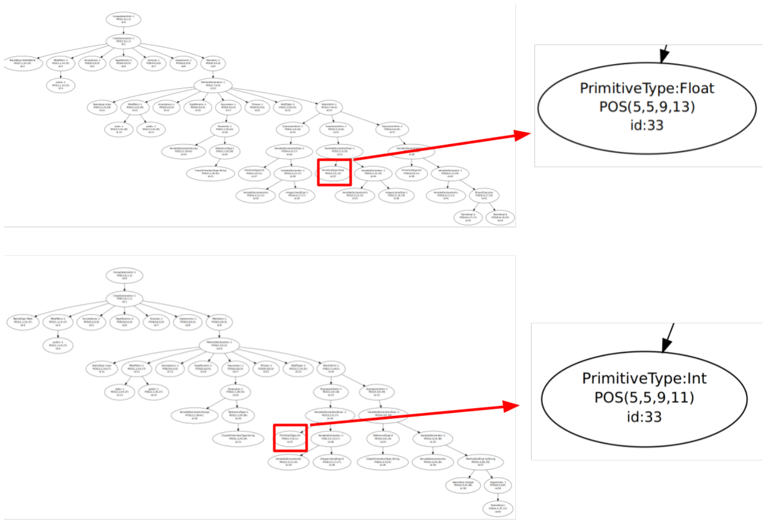


Figure 1.3: On the top the buggy tree and on the bottom the fixed tree

it. For each bug detected, the matching area of the bug is given (ie the nodes that are concerned by the bug). So for each bug, a sequence of instructions to apply on E_a is generated, it represents what are the differences between E_a and E_b only considering the matching area (see section 4.2.3 for graphs). Thus applying this sequence of instructions on E_a will modify the matching area so that will be equal to an equivalent of this area in E_b (E_b does not contain the bug) (see section 4.2.3 for graphs). Then the sequence of instructions to apply on A_a in order to get A_b (independently of the bugs detected in E_a) is generated. So if there are three bugs detected in E_a, we would generate three different sequences to apply on E_a, and three times the same sequence of instructions to apply on A_a. For each couple (a,b) we generate these sequences, then the sequences for EGs feed a neural network as input and the sequences for ASTs are the ground truth that the neural network should predict.
- During the inference stage, we only have many examples of code C_a from which we generate A_a from which we generate E_a. For each bug detected in E_a we generate a fixed E_b that does not contain this bug anymore. At this stage, the model is trained; **we generate the sequence of instructions between E_a and E_b and give it has input to the network; the network predicts a sequence of instructions to apply on A_a in order to get a fixed version A_b from which we can extract a fixed version of the code C_b that does not contain the bug anymore.**

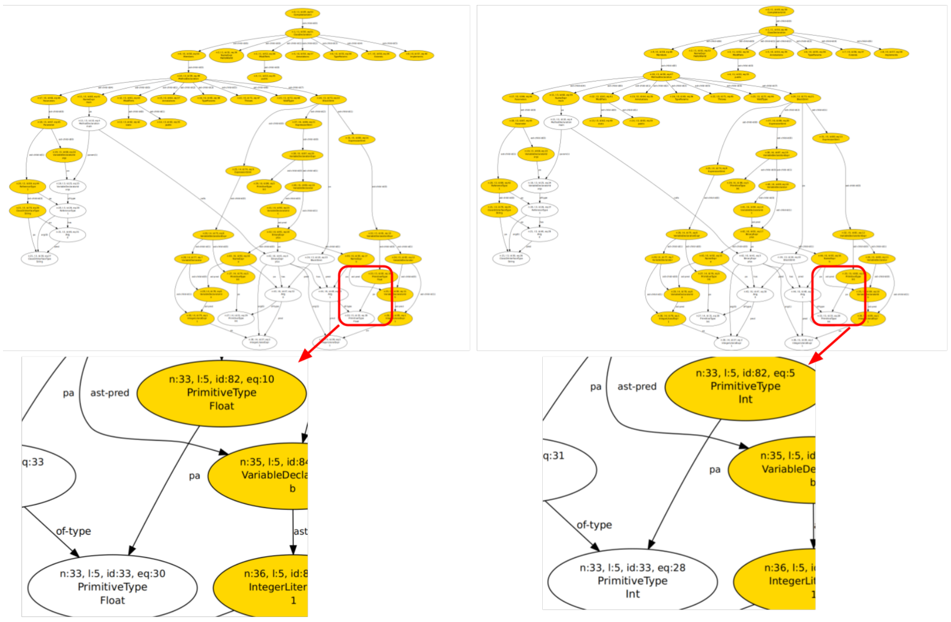


Figure 1.4: On the left the buggy graph and on the right the fixed graph

1.2 Objectives

The objective of this thesis is to predict a sequence of instructions to apply on A_a that fixes one bug, given the sequence of instructions to apply on E_a in order to get E_b that does not contain this bug anymore. Once this is done the pipeline would be complete and we could fix bugs in a source code C_a automatically. The sequences of instructions on graphs and the ones on trees are different, because the type of instructions we must apply on those two structures are different. This thesis is a specific application of translation problems.
In order to get the objective completed, we must:

- define a language for graph fixes, ie how the sequence of instructions to build E_b from E_a is defined,
- define a language for tree fixes, ie how the sequence of instructions to build A_b from A_a is defined,
- and run experiments with several training settings (train a neural network that learns sequences of instructions for ASTs, given sequences of instructions for EGs).

1.3 Problem statement

The purpose of this thesis is to be able to define a language for graph fixes that permits to generalize well the tree fixes. How can we learn fixing sequences of instructions for some bugs on ASTs given fixing sequences of instructions for those bugs on EGs?

At the end, the problem statement could be expressed as follow: **how to generate, for a piece of code C_a and for a bug detected in it, a sequence of instructions that fixes A_a and that could bring to a fixed version of C_a that does not contain this bug anymore?**

Chapter 2

The company

2.1 DeepCode

I have done my master's thesis at Deepcode in Zurich (Switzerland), which is an ETH Spinoff Company created in 2015.

The goal of DeepCode (Grammarly for code) is to provide developers with a tool that finds the bugs and critical vulnerabilities in their code. It gives live alerts to developers of critical bugs in their Integrated Development Environment (IDE) or upon every pull request. DeepCode is supported by many IDEs and text editors such as VsCode, IntelliJ, Atom, SublimeText... The programming languages that are supported by the tool for now are C, C++, Java, JavaScript and Python.

2.2 Corporate Social Responsibilities

"Corporate social responsibility (CSR) is a type of international private business self-regulation that aims to contribute to societal goals of a philanthropic, activist, or charitable nature by engaging in or supporting volunteering or ethically-oriented practices." [12]

Are listed bellow some of the actions taken by Deepcode.

- Deepcode supports local communities, it actively supports online groups from Zurich: Java development communities (https://github.com/analysis-tools-dev/static-analysis).

- Deepcode encourages its employees for continuous learning by taking classes (in JavaScript for some employees working in the front-end team), and it obtains subscriptions for other courses.

- Deepcode uses Google Cloud, which is told to make the use of a lot of renewable energy.

- Deepcode greatly enables and enforces all Swiss/Zurich provided recycling mechanisms: paper, aluminium, plastic, electronics, batteries, glass, compost...

- Deepcode is continuously growing by hiring new employees.

- Deepcode makes the employee safety the highest priority, more particularly during this covid19 situation.

- Deepcode makes all its employees taking part in important decisions for the company.

- Deepcode helps developers to save time and increases the performance of their code; which is helping with more efficient usage of resources.

Chapter 3

Related works

More and more work is done in the automatic bug fixing field. For instance in early January 2018, R. Gupta et al [17] focused on fixing programming language syntax bugs using reinforcement learning. In late 2019, J. Bader et al (Facebook) developed Getafix [5], an automatic bug fixing tool, that parses source codes into abstract syntax trees in order to learn fixes by identifying changes at this level. Our method also uses ASTs. And still in 2019, M. Tufano [18] tried to automatically fix bugs using neural machine translation; we use that kind of model in our experiments too, it uses AST parsing as well.

In this thesis we focus our efforts on representing the correction of bugs by sequences of instructions to apply on ASTs; so we focus more on works, such as neural machine translation [18]; because at the end the purpose of this thesis is to translate a sequence in " graph language " into a sequence in " tree language ". An extensive amount of work has been done in natural language processing and more particularly in neural machine translation field. The work done in [7] shows how well the recurrent neural networks (RNN) are performing on sequential problems; one of the models used in this thesis is based on RNNs. Moreover, using some attention mechanisms as presented in the work [1] significantly increases the precision obtained in those sequential purposes. This is what inspired the model in [9] (one of the model we use), based on the work [2], to include attention mechanisms in encoder-decoder models.

The work done in [4] makes us rethink our thesis not only as sequence translation but also as language generation task, using transformers. In the last few years, thanks to the A. Vaswani's work [1] the transformer architecture has helped the field of language processing making progresses. Transformers were able to generate meaningful text given a prompt; in our thesis, we would use this in order to generate a sequence of instructions for a tree given a sequence of instructions for a graph.

Chapter 4

Development

The main focus of this thesis is on creating a language that describes the instructions that must be applied on a graph E_a regarding a bug to get a fixed graph E_b that does not contain this bug anymore; and same from tree A_a to fixed tree A_b (independently from bugs). Once the language created, the goal is to learn sequences for trees from sequences for graphs.

4.1 Technologies

The main technologies that have been used for developments are C++ and Python (Pytorch). Some bash scripts have been implemented, and the protocol buffer method (method of serializing structured data, developed by Google) has been used. To build and test the code, bazel, developed by Google, was used.

4.2 Preliminaries

4.2.1 The pipeline

The bug autofixor is composed of several steps going from a source code C_a (that is buggy) to another source code C_b (with an instance of bugs fixed).

The goal of the thesis is to be able to build a fixed tree A_b from a buggy tree A_a given the sequence of instructions that allows correct one bug in E_a.

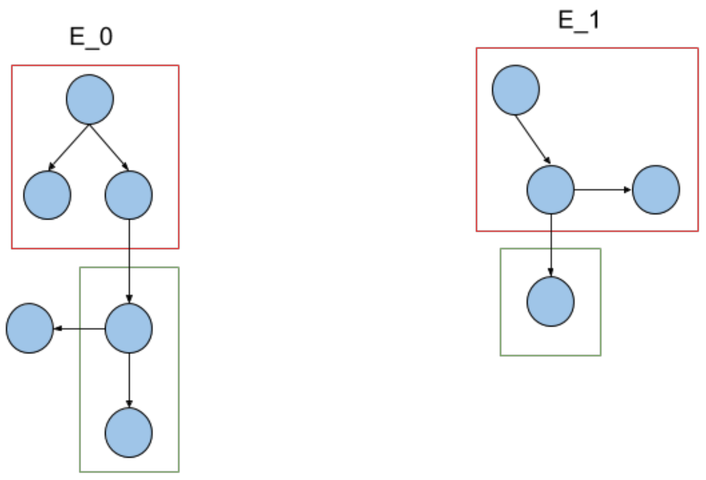
Any code written in C/C++, Java, Python or JavaScript C_a can be parsed into an AST (abstract syntax tree) A_a. Then, once that A_a is generated, an EG (event graph) E_a can be generated from it.
E_a is then analysed (well known patterns in the graph are scanned) so that instances of bug are detected in it. For each bug, fixes are suggested in order to remove the bug from E_a. Those fixes can be: removing a node or an edge from E_a, modifying a node in

E_a or adding a node or an edge in E_a. When E_a is analysed, an instance of bug must be detected with its "matching area". The matching area is the set of the nodes of the graphs that are concerned by the instance of bug. With this we can identify what node in E_a must be modified, removed ... in order to solve this particular bug.

At this point, from the code C_a has been generated A_a then E_a that for each bug has been modified into an E_b that does not contain the bug. So the rest of the pipeline should be about extracting a fixed tree A_b from the fixed graph E_b and then having from A_b a code C_b which would be the fixed version (regarding an instance of bug) of the code C_a. However, the part of the pipeline of the extraction of A_b from E_b is missing. This is the purpose of this thesis. **Note: at inference time, there are as many E_b that there are bugs in C_a, but at training time we only dispose of one E_b that has been generated from C_b; which is the code that is compared to C_a.**

What has been explained so far in this part, is at inference time. But, first we must train some neural networks in order to be able to predict the sequence to apply on A_a in order to get a fixed A_b for a given bug. At training time, we are looking at data that are repositories on github. From those repositories we isolate different source codes and for each version (commit) of this code, we compare it with the following version. For instance, we have a piece of code C_0, this code has 3 versions C_0, C_1 and C_2 because it has been committed three times, first we compare C_0 with C_1 then C_1 with C_2. At each comparison (for instance between C_0 and C_1) we generate the corresponding trees (A_0 and A_1) and graphs (E_0 and E_1), make the analysis on E_0 (we know what changes must be done in the matching area for each bug detected), and create the sequences between the two graphs only in the match area for each instances of bug detected in E_0. If we detected 2 bugs in E_0 then two different sequences must be generated from E_0 to E_1, each sequence fixing one bug. For instance in figure 4.1, two bugs are detected in E_0, so we have two matching area (the red one and the green one). In E_1 the areas are the ones corresponding to the matching area of E_0 (see section 4.2.3: Compute the differences: mapping function, to see how the corresponding matching areas in E_b are defined). We can see, that the sequence of instructions for the red bug must consider one node to remove, one edge to remove, one node to add and one edge to add. For the green bug, the sequence should consider one node to remove and so one edge to remove. Because at inference time, between E_0 and E_1 the changes would have been made only on the matching area of one of the instances of bug detected in E_0 at a time (static analysis on E_0 gives the changes to do on nodes concerned by the bug, in order to get E_1), at training time we must take it into account. So, when we generate the sequences of instructions at training time to feed the neural network with as inputs, we must consider changes (nodes to add, node to remove, edge to add ...) only in the matching area of the one bug detected (rule) in the graph source E_0 at a time. Back to training time, we also generate the sequence of instructions between A_0 and A_1. We do it for every comparison of every code collected on github. The neural network takes as input all the sequences created for graphs and the ground truths that should be outputted are sequence for trees.

Figure 4.1: Graph example



4.2.2 Event graphs and Abstract Syntax Trees

As explained in the previous section, the thesis has to be integrated in the pipeline such that it interacts with two data structures: the event graph (EG) and the abstract syntax tree (AST).

Abstract Syntax Tree

This is a tree structure that represents a source code.

Definition 4.1. Abstract Syntax Tree (AST): An AST is a set of nodes represented by couples of strings (t,v) with t the type of the node represented by the couple and v its value (the value of a node is optional, whereas the type is mandatory).

This tree does not represent every syntax details of the code (like grouping parenthesis, or coding formatting style), but mostly represents the structural or content-related details. Each node can represent a condition, a variable... but important, variable types must be preserved, as well as the location of each declaration in the source code (so if we know what node is concerned by a bug in the AST, we can know exactly where the bug is in the source code), besides, the order of the nodes is important.

Event Graph

The event graph is a graph structure that represents a source code. This structure is resulting from a program analyser.

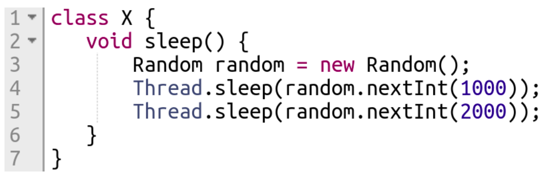
Definition 4.2. Event Graph (EG): An EG is a couple (N,E) where N is a sequence of nodes and E a sequence of edges. For each node n in N we store couple of strings (t_n,v_n) with t_n its type and v_n its value (the value of a node is optional, whereas the type is mandatory). And edge e in E is represented by a tuple (o_e,d_e,t_e) where o_e is the origin node of the edge, d_e is the data of the edge (a string as well) and t_e its target (ie the node to which the edge is pointing to).

EGs are extracted from ASTs an by construction, an AST from which an EG has been extracted is included in the EG. But more than containing the AST, the event graph contains all the functions that are called in the source code, what are the arguments of those functions and what elements call these functions. It is another level of description of the source code, more precise than the AST.

There are two types of nodes in a graph: the AST nodes and program analysis nodes. There are three types of edges: the edges between program analysis nodes, the edges between AST nodes and edges between the two types of nodes. And there are 17 different sub-types of edges. Between AST nodes, the edges are mainly describing relationships (node Y is the i-th child of X, node Y is the right sibling of X...). Between program analysis nodes, the edges describe the program itself (node Y is the i-th parameter of the function defined in X, the function in X may call the function in Y, if Y is an if statement we check the condition of the node X,...). Between an AST node and a program analysis node, there is just one kind of edge: "the AST node X corresponds to the program analysis node Y".

Let's see an example of an event graph in figure 4.3 corresponding to code in figure 4.2. The corresponding AST is included in the EG.

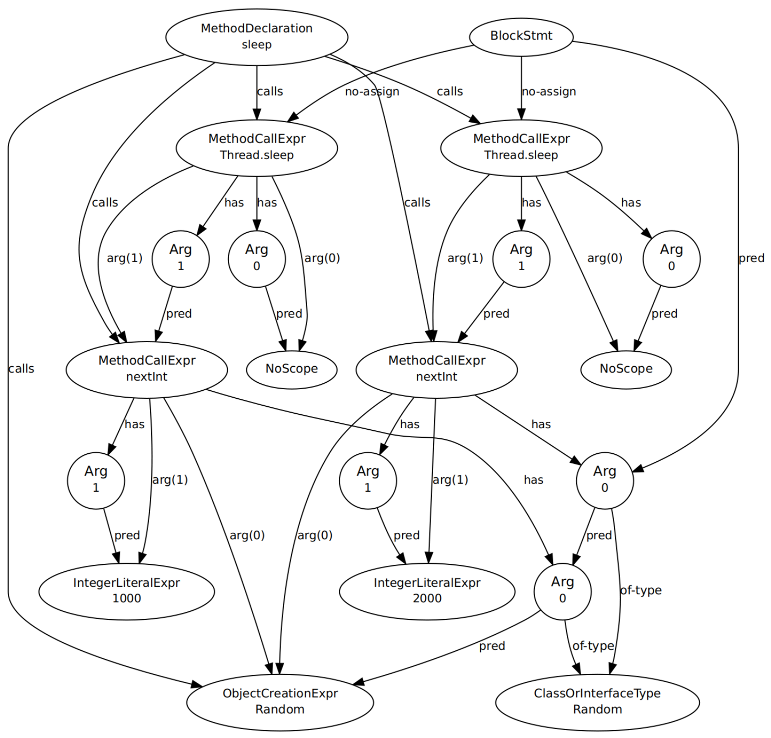
Figure 4.2: Java code example



```
class X {
    void sleep() {
        Random random = new Random();
        Thread.sleep(random.nextInt(1000));
        Thread.sleep(random.nextInt(2000));
    }
}
```

In figure 4.3, we can see in the graph that there are nodes that represents some variables, as for instance the node with type "IntegerLitteralExpr" and value 1000, representing the integer 1000; but there are also nodes representing methods declaration such as "MethodDeclaration sleep", or method calls such as "MethodCallExpr Thread.sleep". We can also see the different relationships that link those different elements. For example, the node "MethodDeclaration sleep" is linked to "MethodCallExpr Thread.sleep" by an edge "calls": it just means that in the method sleep, the method Thread.sleep is called. But we can see that this node "MethodCallExpr Thread.sleep" is linked to two

Figure 4.3: Graph example



other nodes by edges "arg", this means that the targets of those edges are the arguments that are given to the "Thread.sleep" call.

4.2.3 Compute the differences

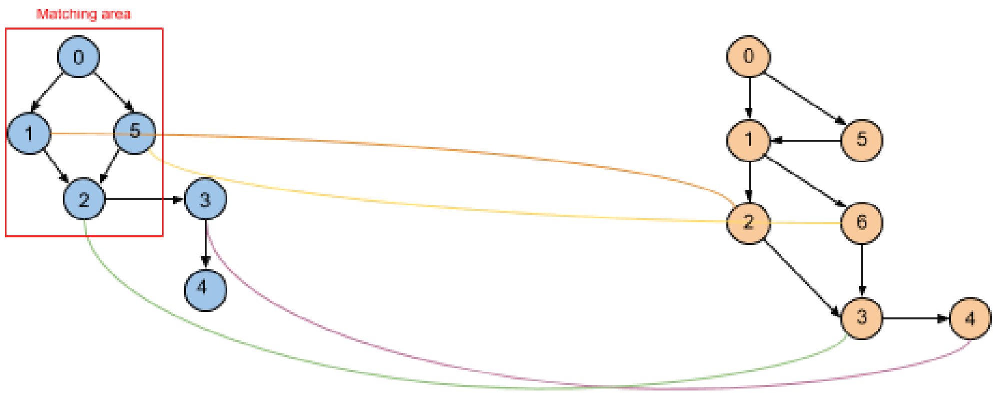
Mapping function

One main task to understand the instructions needed to compute a fix from a graph or a tree is the following: given two event graphs: E_a and E_b, we must compute the difference between those two graphs. The difference consists in finding which nodes are removed, modified or added from E_a to E_b, but also which edges are removed and which ones are added from E_a to E_b.

The graph analysis framework we use provides a mapping function that gives between two graphs the mapping between nodes. More precisely, we know that in the graph E_a the node X corresponds to the node X' in E_b (see figure 4.4).

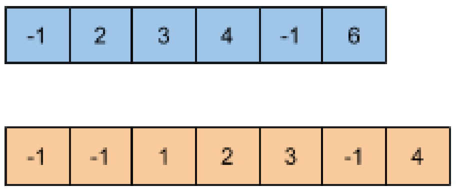
This mapping function works as following: for each node in E_a we compute a hash to designate it (the hash is a combination of the type, the value of the node, and the types

Figure 4.4: Linking of nodes between two graphs



and values of its parents and children), then it does the same in E_b; and it links uniquely the hashes from E_a and E_b so that we get a mapping of the nodes in the two graphs. By this way we obtain two mapping vectors, one from E_a to E_b ($m_{a,b}$) and another from E_b to E_a ($m_{b,a}$), stating that node X in one graph corresponds to node X' in the other graph. If the hash of a node n in E_a cannot be found in E_b, then n would be mapped to -1 in the mapping vector $m_{a,b}$ (-1 is chosen because the identifiers of the nodes are positive, so being mapped to -1 means that n isn't linked to any node in E_b). In the same way, if the hash of a node n' in E_b cannot be found in E_a, then n' would be mapped to -1 in the mapping vector $m_{b,a}$.
According to the example given in figure 4.4, we would have those two mapping vectors that are in figure 4.5. The first position in those mapping vectors is indexed by 0 and it corresponds to the node with identifier 0.

Figure 4.5: Mapping vectors



-1	2	3	4	-1	6

-1	-1	1	2	3	-1	4

The function that computes the mapping vectors mainly considers nodes having at least three neighbors, because it is highly probable that for nodes with only two neighbors, the computed hash is not unique in the graph. So it brings that some nodes are considered mapped to -1 (so either removed or added) whereas they are not, this is because of the non uniqueness of their hashes.

So given this function that computes the mappings, we are able to know which nodes are deleted from E_a to E_b, these are the ones that are mapped to -1 in $m_{a,b}$. We also know which nodes are added from E_a to E_b, these are the ones that are mapped to -1 in $m_{b,a}$. And we know which nodes are modified (nodes in E_a that are linked to nodes in E_b but have different types and/or values). Finally we know which edges are deleted from E_a to E_b (edges for which the target node or the origin node is removed) and added (edges for which the target node or the origin node is added).

So from the mapping vectors in figure 4.5, one can conclude that in the source graph (the blue one, E_a, in figure 4.4) the nodes with id 0 and 4 are removed (because $m_{a,b}[0] = m_{a,b}[4] =$ -1) and in the destination graph (the orange one, E_b, in figure 4.4) the nodes with id 0,1 and 5 are added (because $m_{b,a}[0] = m_{b,a}[1] = m_{b,a}[5] =$ -1). Moreover, as the node with id 0 is removed in the source graph, we know that the edges starting from 0 are removed as well. In the same way, because 0,1 and 5 are added nodes, every edge arriving in those nodes and starting from them is added. To know if other edges are added or removed, the two graphs must be scanned. In order to know what nodes have to be modified from the source to the destination, we must check that all the nodes that are linked (for example node 2 in source and node 3 in destination) have the same type and value; if not, the node has to be modified. However, the red area represents the matching area for a specific bug, and we must **only consider** changes that are in this area. So concerning removed nodes/edges and the modified nodes, we only considers the ones in the area: so we keep considering the node with id 0 and the edges starting from 0, but not the node with id 4. Regarding the added nodes and edges, we keep considering nodes added if at least one their parents is linked to a node in the matching area or if one of the parents of the node is kept considered, and for edge we consider them only if their origin and/or their target are considered.

We need to compute the differences between two trees as well, for which we don't dispose of any matching area. The main difference comparing to graphs is that in trees the edges are not considered. In graphs, edges have data; there are different types of edges, whereas in trees, edges have not data. Another difference between graphs and trees is that in graphs a node can have more than one parent (edges coming to the node may have different sources); in trees, a node has one unique parent. So it is clear that a tree can directly be converted into a graph without any effort, because all the conditions to be a tree (unique parent), can be satisfy by specific graphs (on the contrary, not all graphs can be represented by a tree; for instance graphs where some nodes have more than one parent). Moreover, in the used framework, there is only the above-described mapping function for graphs. So we just need to translate trees into graphs, so that we can get the mappings between trees. Then, exactly as for graphs, we can compute the difference between trees, without considering the matching area.

Note: having two graphs E_a and E_b at training time, we can compute the mappings of their nodes. For each bug detected in E_a, there is a matching area for this bug in E_a, it represents all the nodes that are concerned by the bug (it could be nodes to remove, nodes to modify or even nodes that remain unchanged). We can determine what is the corresponding area of the matching area in E_b. The corresponding area contains all the nodes in E_b that are linked ($m_{2,1}[node] > 0$) to nodes in E_a that are in the matching area, but also all the nodes that must be added in E_a ($m_{2,1}[node] < 0$) that have for ancestors at least one node that is linked ($m_{2,1}[ancestor] > 0$) to a node in the matching area.

Difference computer

From this mapping function, we can store all interesting information regarding the differences between two graphs or the differences between two trees.

For EGs, the interesting information is the removed nodes, the modified nodes, the added nodes, the removed edges and the added edges. From the mapping function, we have this; but only the nodes and edges contained in the matching area for the instance of bug detected have to be kept. For each node that is removed from E_a, if the node is in the matching area then we store it in the **removed_nodes** set. For each node that is modified, if it is in the matching area then we store it in the **modified_nodes** set. For each node that is added, if its parents is in the matching area or already in the **added_nodes** set, we must store it in the **added_nodes** set. Regarding the edges that must be removed or added, we need them to have their origin and/or their target in the matching area, in this case we add them in the **removed_edges** set or the **added_edges** set. Those sets are used to create the sequences of instructions between graphs.

For ASTs, the interesting information is the removed nodes, the modified nodes and the added nodes; but we have to consider all of the nodes that are in those categories, we must not limit ourselves to some matching area in the tree, a complete sequence of instructions is expected.

The difference computer returns **removed_nodes**, **modified_nodes**, **added_nodes**, **removed_edges** and **added_edges** when it is called for graphs. When it is called for trees that has just been translated into graphs, it returns **removed_nodes**, **modified_nodes** ans **added_nodes**.

Once the information has been extracted from the mapping vectors (deletions, additions and modifications), it is important to visualize those changes between graphs and trees in order to understand how to encode the changes. That is why a small visualization tool has been implemented. The visualization tool consists in displaying the source and destination graphs (same for trees), and highlighting in the source the nodes and edges that are removed and in the matching area, and same for the nodes that are modified. It also highlights in the destination the nodes and edges that are added and in the matching area. With this tool, it is simple to look at many use cases in order to apprehend the different instructions that might be contained in the sequences for tree (to build A_b from A_a) and for graphs (to build E_b from E_a).

4.2.4 Instructions

From the observations of use cases, there have been determined four different types of modification possible on nodes in order to go from a source graph to a destination graph. We can either add a node, delete a node, or modify a node, we can also move a node to another place, but this last instruction will not be considered in this thesis, as it is complex to handle. There are also two modifications on edges: we can either add or remove an edge from a source to a destination graph. Concerning the trees, only matter the four types of modification regarding the nodes, as the edges don't have to be considered.

However, according to the pipeline in section 4.2.1, at the end we need to build a fixed AST A_b from buggy A_a regarding a bug, that is why we need to define some other instructions. All the instructions that we can meet in sequences between graphs and sequences between trees are listed in the table 4.1. Some use cases of these instructions are illustrated in sections bellow; and there are also explanations about why do we have these instructions.

instruction code	instruction
0	GOTO
1	ADDNODE
2	REMOVENODE
3	MODIFYNODE
4	ADDEDGE
5	REMOVEEDGE
6	ADDDOWN
7	ADDRIGHT
8	MOVE

Table 4.1: Exhaustive list of possible instructions

As the goal is to apply a sequence of instruction on A_a in order to get A_b, all the instructions needed must be completely implemented for trees. Regarding graphs, we don't need to actually apply a sequence of instructions on E_a to get E_b, so we don't need to implement the instructions.

4.2.5 Implementation of instructions for ASTs

At inference time, there is a source code that contains some bugs; from this code an AST is generated, from which an event graph is generated. After graph analysis, for each bug detected, a fixed event graph is generated. Then a sequence between the "buggy" graph and the fixed graph is generated. From this sequence of instructions, a sequence from the "buggy" AST to a fixed AST must be predicted for the concerned bug. This sequence between trees might contain several of the instructions listed in Table 4.1 (sequences containing "move" instructions are skipped, for more simplicity). So, once the sequence is predicted, the fix tree should be built from the "buggy" tree on which the instructions are applied. So we must explain how those instructions modifies the buggy tree in order to get the fixed one.

The operations that are listed in Table 4.1 and that are concerned for tree sequences are GOTO, REMOVENODE, MODIFYNODE, ADDDOWN, and ADDRIGHT; they must be implemented; first to look at the correctness of the outputs of the network for good training (tree sequences), and then to be able to rebuild the fixed tree; because at the end, the goal is to build a fixed tree. Bellow are the explanations of the different instructions.

GOTO instruction

Depending on the semantic chosen; in a sequence between trees, the "goto" instruction must be furnished with an information X about the node to go to. X must be enough to identify with precision which node is designated. There are different semantics possible in order to identify a node with X: it could be the identifier of the node, or it could be a mapped identifier, a hash... This instruction only consists in accessing a node in the tree.

REMOVENODE instruction

This instruction removes a node from a tree. When removing a node from a tree, it removes all its children as well.

MODIFYNODE instruction

Modifying a node means that the type and/or the value of a node has to be modified. In the sequence of instructions, we should have access to the new type and the new value. Once this information is obtained, to modify the node we just have to access it with "goto" and rewrite its type and/or value.

ADDDOWN instruction

This instruction consists in adding a node as the first child of another node (the parent). The information provided about this new node from the sequence are its parent, its type and its value.
This instruction is composed of two steps, first we give to the node that we add a geographical context: we must know where this node is in the tree. Then we give it its characteristics, ie its type and its value.
Regarding the geographical context, we first must give to this new node a parent, that is given in the instruction.
Then we must give to the new node its child_index (this is the child position among all the children of its parent), as we call the " adddown " instruction, we add this new node as the first child of its parent. It may happen that the added node is the last child of its parent, in this case, the first child of the parent is also the last child.
But if before adding this new node thanks to " adddown " the parent already had children, then the previous first child is now the second child, so its child index has to be incremented by one, more generally, all the children that were already there must have their child_index incremented by one.
Then it is important that we give to the new added node a right sibling (not a left sibling as it is the first node), so the right sibling is the previous first child, and the left sibling of the previous first child is the new added node.

ADDRIGHT instruction

This instruction consists in adding a node as the right sibling of another node (the left sibling). This instruction is similar to " adddown ".
The difference is that from the instruction in the sequence, the information we have about the new node is its type, its value and its left sibling.
In the same way as previously, this instruction is composed of two steps; first the geographical context then give it its characteristics.
The added node and its left sibling have the same parent. Then we must give to the new node its child_index, but as we call the " addright " instruction, we have that the child_index of the new node is incremented by one compared to the child_index of its left sibling (from which we add the new node). It may happen that the added node is the last child of its parent. There is no case where the node added with " addright " could be the first node of its parent.
All the children of the parent of the new node that are coming after the left sibling from which we add the node must have their child_index incremented by one.
Then it is important that we give to the new added node a right sibling and a left sibling. The right sibling of the new node is the previous right sibling of the left sibling from which we are adding, and the left sibling of this right sibling is the new added node.
Concerning the left sibling of the new node, it is the left sibling from which we are adding, and the right sibling of this left sibling is the new added node.

The implementation details for these instructions can be found in the appendix A.

Applying the instructions

In order to know the precision in rebuilding trees of the predictions made by the model, we need a tool that is able to apply these sequences of instructions on trees. The implementation of this module highly depends on the semantic chosen to define the sequences of instructions.
The input of this module is a text sequence representing several instructions that must be applied on the tree, so this text must be parsed in order to get the different instructions (parsing depending on the semantic), so that all the information needed to apply the instructions is available.
This tool is also used when generating the sequences used for training in order to check the correctness of these sequences.

4.3 Sequence generation

This is done at training time, so we compare many pairs of codes C_a and C_b for which we generate A_a, A_b, E_a and E_b. The purpose is to generate sequences of instructions for EGs (what are the instructions to apply on E_a regarding a bug, to get the matching area equals to the cleaned area in E_b, that does not the bug anymore) in order to predict sequences of instructions on ASTs (what are the instructions to apply on A_a to get A_b). In order to train the network that should be able to do those predictions, we must describe how to define sequences for graphs and for trees.

4.3.1 Sequences of instructions for graphs

In the Table 4.1 listing the different instructions possible, the instructions that may be contained on sequences for graphs are: **addnode**, **removenode**, **modifynode**, **addedge**, **removeedge**.

The semantic chosen to compute the sequences for graphs is straightforward. The results from the Difference Computer between two EGs are used (see part 4.2.3 that explains how the differences are computed). The difference computer provides us with all the nodes that are removed, modified, added from E_a to E_b, and all the edges that are removed and added from E_a to E_b, all of that in the matching area for an instance of bug.

So, the sequence of instructions is just composed of all deletion instructions (first nodes then edges), then all the modifications, and finally all additions (first nodes then edges) (see algorithm 1: **reminder: the generation the of sets removed_nodes, added_nodes, modified_nodes, removed_edges and added_edges has been**

done according to section 4.2.3 concerning the graphs).

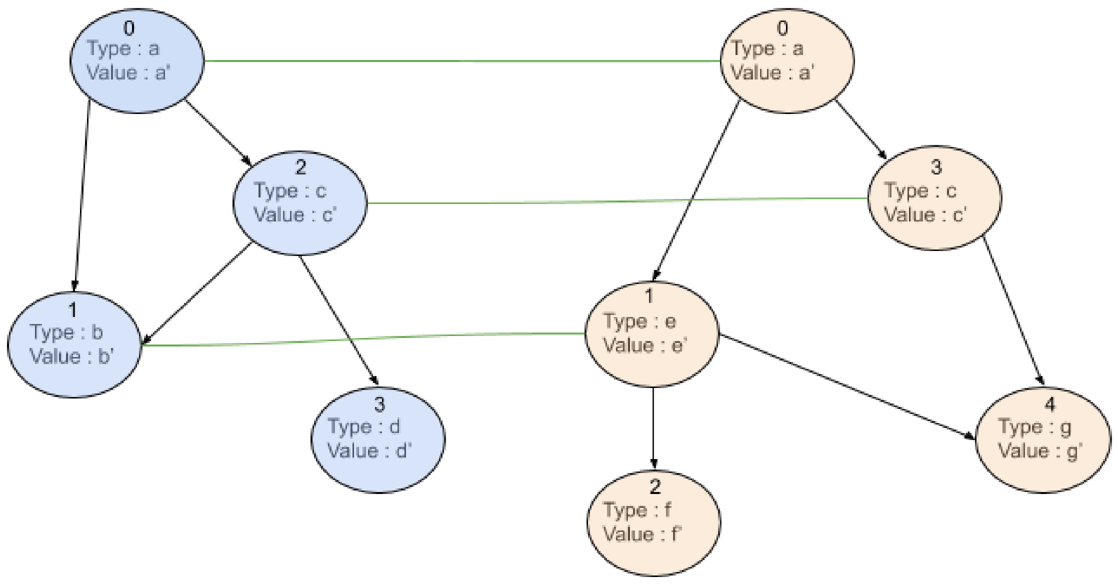
What is important to know in order to clean the matching area in E_a as it is E_b, is that for instance a node with that type and that value has to be removed, and an edge pointing to a node having that type and that value has to be added. As there is no need to apply this sequence on E_a (as we have it), we don't need to identify uniquely and precisely the concerned node, we just need to know that we must apply an instruction on a node with that type and that value. In the following, if a concerned node has the type X and the value Y, having a hash function h, the result of applying this function to X and Y is stored in $H_{x,y}$: $H_{x,y} = h(X, Y)$; so in the sequences of instructions for graphs we put the result of the application of the hash function on the type and the value. That said, the semantic of the instructions in sequences for graphs is the following:
- For a node deletion, if the node to delete has a type X and a value Y, in the sequence should be written `removenode` $H_{x,y}$.
- For a node addition we have `addnode` $H_{x,y}$.
- For node modification, if the node should have the new type X and/or the new value Y, the instruction is `modifynode` $H_{x,y}$.
- For the edge deletion, it has been decided to keep the information of the node that the edge was pointing to, and in the same way as previously, if this node has type X and value Y, then the syntax of edge deletion is `removeedge` $H_{x,y}$.
- And finally, in the same way as for the previous instruction, for the edge addition we have `addedge` $H_{x,y}$.
The algorithm to get a sequence between two graphs is really straightforward (see algorithm 1: Compute sequence graph). Note node.type is the type of the node and node.value its value, edge.target is the node the the edge is pointing to.

Algorithm 1 Compute sequence graph

Input: g1 : event graph, g2 : event graph
Output: sequence_instructions: sequence of instructions
Initialization: Difference computer (see section 4.2.3 for graphs) computed removed_nodes, added_nodes, modified_nodes, removed_edges, added_edges

```
1  for node in removed_ nodes do
2      sequence_instructions.append(REMOVENODE($H_{node.type,node.value}$))
3  for edge in removed_ edges do
4      sequence_instructions.append(REMOVEEDGE($H_{edge.target.type,edge.target.value}$))
5  for node in modified_ nodes do
6      sequence_instructions.append(MODIFYNODE($H_{node.type,node.value}$))
7  for node in added_ nodes do
8      sequence_instructions.append(ADDNODE($H_{node.type,node.value}$))
9  for edge in added_ edges do
10     sequence_instructions.append(ADDEDGE($H_{edge.target.type,edge.target.value}$))
```

Figure 4.6: Graph sequence example



Let's study a small example. After computing the mapping vector for the two graphs in the figure 4.6, the nodes of the blue graph E_a are mapped to the nodes in the orange

graph E_b according to the green lines. We consider that all the nodes are in the matching area.
From the figure 4.6, we can see that in the graph E_a the node 1 must be modified and have new type e and new value e', the edge from node 2 to node 1 must be removed and the node 3 must be removed. Also, we can see that in the graph E_b the edge from the node 1 to the node 4 must be added and the node 2 and 4 must be added. Let's say that $H_{d,d'} = 1$, $H_{b,b'} = 17$, $H_{e,e'} = 9$, $H_{f,f'} =$ -6 and $H_{g,g'} = 32$. The sequence is:
`removenode 1 removeedge 17 modifynode 9 addnode -6 addnode 32 addedge 32`
which means that in order to get the EG E_b from the EG E_a (because all the nodes are in the matching area) a node with type d and value d' (reminder: type and value are strings) must be removed, an edge pointing to a node with type b and value b' must be removed, a node must be modified with new type e and new value e', one node with type f and value f and another one with type g and value g' must be added and finally one edge pointing to a node with type g and value g' must be added.

4.3.2 Sequences of instructions for trees

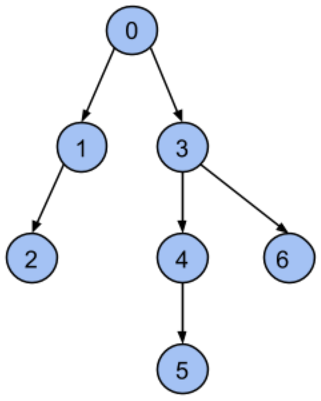
Regarding sequences of instructions for ASTs, this is different, because we actually need to build the A_bs from A_as by applying the sequences (do not consider any matching area here). Indeed, to remind the pipeline, the goal is to be able to build the fixed AST A_b in order to get the fixed code C_b regarding a bug. Concerning the graphs, we don't need to apply the sequences, as we have them at training and inference time, moreover, the way we described the sequences of graphs makes it impossible apply it on E_a. The reason why we cannot apply the sequences on source graphs, is because in graphs we might have a lot of nodes with same types and values (thus many nodes may have a same hash), so in the general case there are too many possibilities to try. Because, when in the sequence we read `removenode` $H_{x,y}$ we might have 5 different nodes having a type X and a value Y, so we should try those 5 possibilities; moreover when adding a new node `addnode` $H_{x,y}$ we must try to add this node from every other nodes in the graph. Thus, it makes it impossible to build E_b from E_a, but as we don't need it, we keep this semantic. On the contrary, we need to build the fixed AST from the buggy AST, that is why we cannot take the same semantic for trees.

This is why, for trees, when an instruction must be applied, we must know what is the node targeted by this instruction; the "goto" instruction has been introduced for this purpose. For instance, for graphs, when a node of type X and value Y has to be removed, the instruction is `removenode` $H_{x,y}$, with h an hash function and $H_{x,y} = h(X, Y)$; for trees, this is more precise, we know that we must remove the node that has the identifier i, so for that, we must go to ("goto") the parent of the node that has the identifier i and remove the node i; we will see bellow how this instruction should be written.

As the model will be trained having as input the sequence on graphs and as output sequences on trees, the sequences on trees must not be too "tree-specific" so that the model could generalize well. So, to apply an instruction on a tree, we must first go to the target node of the instruction then apply the instruction from it. In order to get better generalization, we will not call the "goto" instruction on the target node identifier, but we will use relative identification. The nodes in trees are arranged in pre order order, so the idea behind relative identification is to remember which was the last node concerned by an instruction, and then while calling the "goto" for the current instruction, if the previous node had identifier I_1 and this current node has identifier I_2, then to go to the current node we will call `goto R` where R $= I_2 - I_1$ is the distance between the current and the previous nodes.

However, as we use relative identification by taking the distance between the previous and the current nodes, we must define what we take to identify the first node (because there is no previous node). We could take the identifier of the first node concerned by a change, but in order to get better generalization, the first node concerned by a change is identified by its hash $H_{type,value} = h(type, value)$ (where h is the same function as previously given). This means that for the first node we go to should be identified by its hash, then we apply the instruction and for the following instruction, in the "goto" instruction we put the distance between the first node and the current node.

Figure 4.7: Goto instruction illustration



Here is an example of those relative identifications in the figure 4.7, let's say that we must remove the node 2 and remove the nodes 4 and 5. Let's say that the type of the node 1 is X and its value is Y and let's consider the hash function h. The sequence for this tree would be: `goto` $H_{x,y}$ `removenode 1 goto 2 removenode 1`.

Explanations: As the first instruction is about removing the node 2, we must go to its parent, which is 1, but, as this is the first instruction, there is not previous node visited, so we go to the node 1 by expressing its hash, so `goto` $H_{x,y}$. The "removenode" instruction will be explained bellow. As we went to node 1 with the "goto" instruction, the current node is node 1; so as we must go to node 3 for the second instruction (as we want to remove the node 4, we have to go to its parents which is node 3), because the previous node we went to was node 1, to go to 3, the distance between them is 2, so to reach 3 we must call `goto 2`. Note, as removing a node in a tree removes all the children of this node, we don't need an extra intruction to remove the node 5.

The notion of relative identification has be explained, now we must understand how to build the sequences of trees. What is really important while creating those sequences for trees is to keep in mind that the sequences created should be complete in order to be able to build A_b entirely from A_a; but it is crucial that the sequence should be applicable from A_a without having a single information on A_b. As a reminder, while training the model, the inputs are the sequences of instructions on the graphs (we are disposing of E_a and E_b) and the outputs are sequences of instructions on trees (we are disposing, at training time, of A_a and A_b); but at inference time, once the model is trained, the model will predict sequences on trees given sequences on graphs, and we must be able to build fix trees with these predicted sequences.

In the Table 4.1 listing the different instructions possible, the instructions that may be contained in the sequences for ASTs are: **goto**, **removenode**, **modifynode**, **adddown** and **addright**.

The semantic chosen for those instructions is the following:
- To remove a node with the "removenode" instruction, we first must go to the parent of the node that we must remove `goto relativeIdentificationOfParent` then we must remove the node, for that we use relative identification as well, if the parent has identifier I_1 and the node to remove has identifier I_2, then we have the instruction `removenode R` with R= I_2 - I_1.
- To modify a node with the "modifynode" instruction, we first go to the parent of the node that we must modify `goto relativeIdentificationOfParent`, then we must modify the node, if the parent has identifier I_1 and the node to modify has identifier I_2 and if the node should have a new type X and/or a new value Y, then we have the instruction `modify_node R X Y` with R= I_2 - I_1.
- To add a node from a parent with the instruction "adddown", we must go to this parent `goto relativeIdentificationOfParent` and then if the node to add has a type X and a value Y then we have the instruction `adddown X Y`.
- Finally, to add a node from a left sibling, we must go to this left sibling `goto relativeIdentificationOfLeftSibling` and then if the node to add has a type X and a value Y then we have the instruction `addright X Y`.

Now that we have the semantic, all the nodes that are concerned by a change from the abstract syntax tree A_a to the fix abstract syntax tree A_b must be determined. The nodes that are concerned by a change are the following:

- the parents of nodes that must be removed from A_a to A_b, those parents should not be removed themselves,
- the parents of nodes that must be modified from A_a to A_b, those parents should not be removed themselves,
- the nodes that must be added from A_a to A_b if their parents should not be added themselves.

Once the concerned nodes are stored, we must arrange those nodes and apply needed instructions on. To order the nodes, we apply instructions on nodes of smaller identifiers first (pre-order order). So we will apply instructions on all concerned nodes in increasing order according to their identifier. The thing is that, regarding how the set of the concerned nodes is filled, it seems that there are nodes concerned by a change that are not present in the set. This is the case of :

- nodes whose parents are removed themselves : this case has not to be handled, because they are necessarily descendants of nodes that must be removed but whose parents are not removed themselves. So, those antecedents would be removed. When a node is removed from a tree, all its descendants are removed too, so the nodes whose parents are removed themselves will be removed too.
- nodes whose parents are added, those nodes are added thanks to a depth first search like algorithm (see algorithm 2: DFSAddNodes) that is called from nodes that are added but whose parents don't have to be added. So the node that needs to be added but that has its parent already added, will be part of this depth-first-search addition.

So this is clear that we don't have to handle the cases of nodes whose parents are removed themselves; this case is automatically handled by construction. However for this second case of nodes whose parents are added, a depth first search like algorithm must be implemented in order to add all the nodes that need to be added from the tree A_a to A_b.
In a tree, if a node is added, so necessarily all its children have to be added too. However there are two different cases to handle (see figure 4.8 and 4.9). In the figure 4.8, we can see that from the node 0, we must add the node 1 (because it is represented in green); but the node 1 has no left-sibling, in this case, we must add the node 1 from its parent which is 0; that is why we invoke the instruction `adddown`. From the node 1, its children will be added in a pre-order order with a depth-first-search like algorithm(its children are represented in green, because they must be added). As long as a node being visited has children, then add the first child with the instruction "adddown", if the node added has children or right siblings itself, then we must go to this node and add from it. If the node being visited has no children but siblings, then add the direct right sibling by invoking "addright" and go to this node if it has children or a right siblings. In this example figure 4.8 , to add all the nodes, we will need to visit 1, 2, 3, 4, 5, 6 and 7. As we must keep

in mind that the relative identifications must be used, let's say that the node i has type i.type and value i.value, given a hash function h with $H_{type,value} = h(type, value)$, the sequence to add the nodes in figure 4.8 would be:
`goto` $H_{0.type,0.value}$ `adddown 1.type 1.value goto 1 adddown 2.type 2.value goto 1 addright 3.type 3.value goto 1 adddown 4.type 4.value goto 1 adddwon 5.value 5.type goto 1 addright 6.type 6.value goto -1 addright 7.type 7.value`
This sequences means, we go to node 0 then we add a first child from 0 which is the node 1, then go to the node 1 and add a first child from it which is 2, go to 2 and add a right sibling which is 3, go to 3 and add to it a first child which is 4, go to 4 and add to it a first child which is 5, go to 5 and add to it a right sibling which is 6, go to 4 and add to it a right sibling which is 7. This algorithm is described in algorithm 2: DFSAddNodes.

In the case of the figure 4.9, the main difference is that the origin of the subtree (composed of nodes 2, 3 , 4, 5 and 6) is not the first child of its parent (which is node 0), so the only difference is for the first instruction, instead of "goto" 0 and "adddown", we do " goto 1" and "addright". For the rest of the sequence regarding the subtree, the logic remains the same. This algorithm is described in algorithm 2: DFSAddNodes.

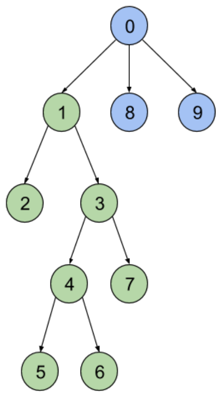


Figure 4.8: Add node as first child

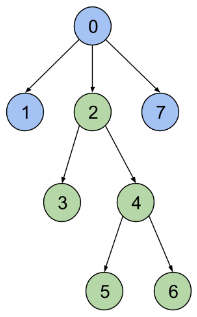


Figure 4.9: Add node not as first child

The algorithm to generate sequences for trees is in algorithm 3 : Tree sequence generation. **Reminder the sets removed_nodes, added_nodes and modified_nodes are generated as explained in section 4.2.3 concerning the trees**

Algorithm 2 DFSAddNodes

```
Input: t2 : abstract syntax tree, origin: int, from_parent: boolean
Output: sequence_instructions: sequence of instructions
Initialization: to_visit: set of int, stop_node, previous = origin
if from_parent is True then
    stop_node is the right sibling of the first child of origin
else
    stop_node is the right sibling of the right sibling of origin
while to_visit not empty do
    node = first element in to_visit
    remove node from to_visit
    if previous is parent of node then
        sequence_instructions.append(ADDDOWN)
        for child of node do
            to_visit.insert(child)
        if node has children or right siblings then
            sequence_instructions.append(GOTO node)
    else
        if previous and node have same parent then
            sequence_instructions.append(ADDRIGHT)
            for child of node do
                to_visit.insert(child)
            if node has children or right siblings then
                sequence_instructions.append(GOTO node)
        else
            sequence_instructions.append(GOTO left sibling of node)
            sequence_instructions.append(ADDRIGHT)
            for child of node do
                to_visit.insert(child)
            if node has children or right siblings then
                sequence_instructions.append(GOTO node)
```

Algorithm 3 Compute sequence tree

Input: t1 : abstract syntax tree, t2 : abstract syntax tree
Output: sequence_instructions: sequence of instructions
Initialization: Difference computer (see section 4.2.3 for trees) computed removed_nodes, added_nodes, modified_nodes, concerned_nodes are computed

```
for node in concerned_nodes do
    if node not in added_nodes then
        sequence_instructions.append(GOTO node)
        for child of node do
            if child in modified_node then
                sequence_instructions.append(MODIFY child)
            // In this case the child is in removed_nodes
            else
                sequence_instructions.append(REMOVENODE child)
    // In this case the node is in added_nodes
    else
        if node to add is first child then
            sequence_instructions.append(GOTO node)
            DFSAddNodes(t2, node.parent, true, sequence_instructions)
        else
            sequence_instructions.append(GOTO node)
            DFSAddNodes(t2, node.left_sibling, false, sequence_instructions)
```

4.3.3 Other tries for sequences generation

One of the most important other solution that was implemented before this solution was about using node identifiers for sequences on trees instead of using this relative identification. A second important solution implemented was using a mapping of the identifiers of the nodes: for instance if the nodes 3, 5 and 17 were concerned by a change, we were identifying 3 by 0, 5 by 1 and 17 by 2. But those solutions did not bring good results, as the models were not able to generalize well, because the identifiers are too tree-specific.

Other solutions have been implemented also, the differences were on the way we built the sequences of instructions for graphs. Instead of putting all the instructions without geographical context, we did as for the current tree sequences solution: we were going through the graph in pre-order order, identifying the nodes by their identifier or a mapping of their identifiers and calling the instructions on those nodes. But as well, for this we did not get good results, as it did not allow the model to generalize well.

4.4 Models used

So, now we have a method to generate sequences for graphs (inputs) and sequences for trees (outputs), and we are confident about the correctness of the outputs. To run experiments, we decided to work with two different deep learning models that are described in this section.

4.4.1 Background

This approach was inspired by [8].

Neural networks

"Neural networks are complicated functions that are composed of simpler component parts that each have parameters that control their behavior" [8]. One simple multi-layer neural network, can be represented in equation (4.1).

$$\begin{aligned} h =& \tanh\left(W_1 x + b_1\right) \\ y =& \mathrm{W}_2 h + b_2 \end{aligned} \tag{4.1}$$

Where W_1 and W_2 are parameter matrices and b_1 and b_2 bias vectors (or parameter vectors). h is called the hidden layer.
Those parameters W_1, W_2, b_1 and b_2 have to be learnt from training data. For that, we have an input x and we compute y' with the model (according to the equation (4.1)) and we try to minimize a function l (the loss function) that characterizes how far the prediction y' is from the ground truth y that the model should have predicted given the input x; $l(y', y) = 0$ if the model predicted made a correct prediction given x. So the goal is to modify the parameters in order to minimize the loss function; thus we take the derivative of the loss function with respect to the parameters; for instance

$$\frac{\partial l(y', y)}{\partial W_1}$$

and we move W_1 in the direction that minimizes the loss (α is the learning rate)

$$W_1 \leftarrow W_1 - \alpha \frac{\partial l(y', y)}{\partial W_1} \tag{4.2}$$

However, we can meet difficulties while computing the derivatives. This can be solved by back-propagation that computes derivatives of complicated functions by using the chain rule [11].

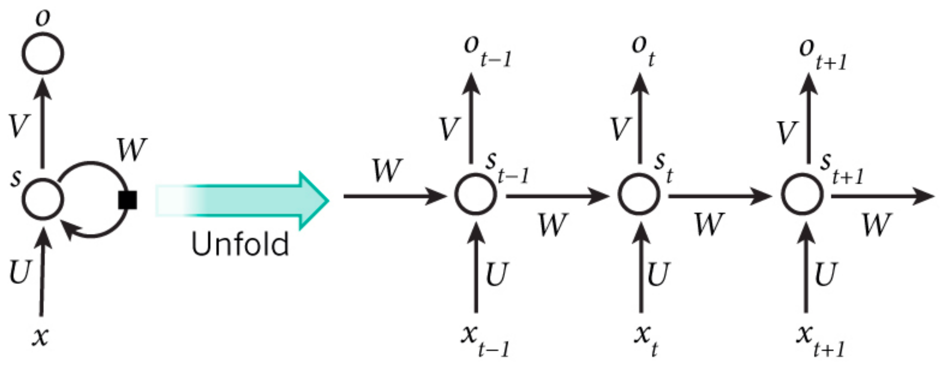
Recurrent neural networks (RNNs)

This is a kind of neural networks. The idea behind RNNs is to make use of sequence information. In traditional neural networks, the inputs are assumed to be independent of each other, which is not a good idea if the goal is to learn sequences, because the current word depends on the previous one. They are called recurrent, because they are performing the same task for every element of the sequence [3].

In the figure 4.10, the unrolled network is just a representation, to show how it works on the complete sequence x in input. So, if we care about a sequence of n words, the network would be unrolled into a n-layer neural network (one layer per word).
In the figure 4.10, the notations are:

- x_t is the input at step t, for instance the one-hot vector of the x_{t+1}-th word in the sequence input x,
- U, V, W some weight matrices that are shared between each layers (so it reduces the number of parameters to learn),
- s_t is the hidden state at step t, it is the memory of the network, it captures what happened in the previous steps. We get s_t by computing $s_t = f(Ux_t + Ws_{t-1})$, where f is a nonlinear function (eg, ReLU, tanh...). For initialization, s_{-1} is only 0s,
- o_t is the output at step t. If we want to predict the next word in a sentence, this would be a vector of probabilities across our vocabulary $o_t = softmax(Vs_t)$.

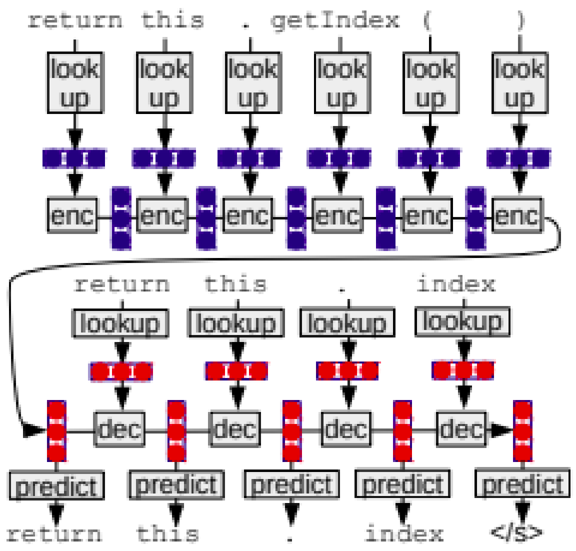
Figure 4.10: RNN with unfolded representation. Source: www.analyticsvidhya.com



Encoder-Decoder models

Also called "sequence to sequence" models, encoder-decoders models (see figure 4.11) work in two stage, first it encodes the input $x = x_1x_2...x_n$ in an hidden vector h_x

Figure 4.11: Encoder-Decoder example for translation task



using the encoding function $h_{x,|x|} = encode(x)$. For each word x_i in input x we have a numerical vector representation (embedding) and we compute the hidden layer using previous results. So if $x = x_1 x_2 ... x_n$, first we can have $h_{x,1} = \tanh(We_1 + b)$ where W and b are parameters to learn, and e_1 the embedding for x_1.Then for the second word x_2, we would have $h_{x,2} = tanh(W_{h_1} h_{x,1} + We_2 + b)$. Using results from previous time steps allows the network to keep a "memory" of what it met before.
Once the input is encoded, the encoded vector can be used to predict the first word of the output. Generally, we define the first hidden layer in decoder $h_{y,0}$ as being the last hidden layer of the encoder $h_{x,n}$. A score function for the first word to predict is given by $g_{y_1} = W_{pred} h_{y,0} + b_{pred}$, and so by using the softmax function, we know what is the probability for the word y_1 to be the first word of the output y: $P(Y_1 = y_1) = \frac{\exp g_{y_1}}{\sum_{possibilities for y_1} g_{possibilities}}$.
A new hidden layer is computed given the first word predicted for output and this is done until we meet a token that means that the end of the sequence has been reached.

Attention mechanism

This is a mechanism that allows the models in which it is contained to focus on some particular words in the input x of the model in order to generate y [8]. This corresponds in calculating an "attention vector" a_i given the input hidden layer h_x of the sequence to sequence model and the current output hidden vector $h_{y,i}$.This "attention vector" is composed of values between 0 and 1, one for each word in input $x = x_1 x_2 ... x_n$ stating how much we must focus on those words when generating a word y_i in output y.
Attention mechanisms are useful when there are not one-to-one correspondence between

language of input and language of output; which is the case in our thesis.

Neural Machine Translation

This kind of models is used for the complex task of translation [8]. In order to generate an output y given an input x (which are both sequences of words), the model must generate one word by one of $y = y_1 y_2 .... y_n$. This is done thanks to probabilistic tools; to generate y_1 first, we must find the word that is the most probable given that the input is x: $P(y_1|x)$; then we do it for the second word considering what is the most probable word for the first word $P(y_2|y_1, x)$... Finally, to predict y we must find the words y_1, y_2, ... and y_n that maximize $P(y|x) = P(y_1|x)P(y_2|y_1, x)...P(y_n|y_1, ..., y_{n-1}, x)$. To learn this probability, we use encoder-decoder models.

Transformers

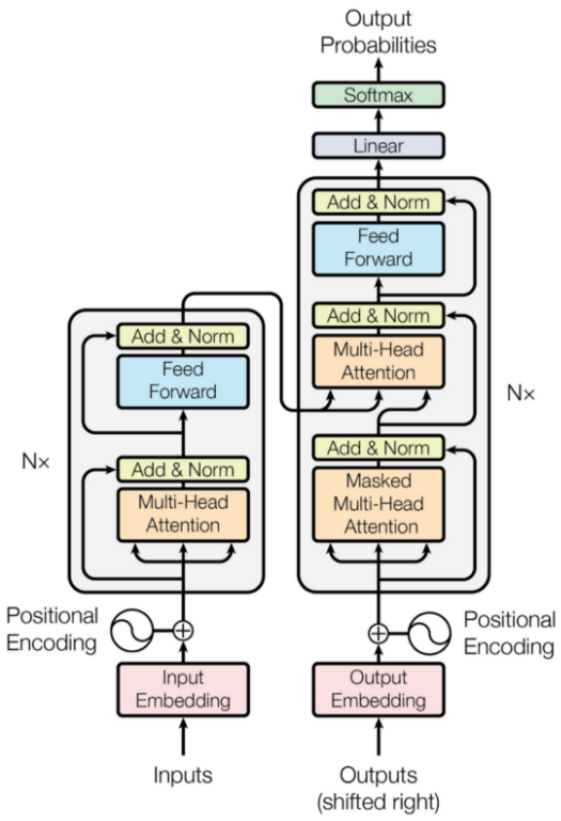
Transformers are the basic architectures behind the language models (models that predict the probability of a sequence of words; they predict the most probable following word given a sequence of words). A transformer is composed of encoders and decoders (see figure 4.12).
The encoder and the decoder have modules that can be stacked together as represented by Nx (see figure 4.12). Encoders and decoders have the feed-forward (simple neural network) and an attention components. The inputs and outputs are embedded into a n-dimensional space before passing them on to the components.
The encoder's inputs first go through a self-attention layer that helps the encoder to focus on some other words in the input as it encodes a specific word.
The outputs of the self-attention layer are fed to a feed-forward neural network (the same FF neural network is applied to each output).

The decoder has the same layers but between them there is an attention layer that helps the decoder to focus on relevant parts of the input.

One important step in the input and output components is the Positional Encoding wherein we provide information regarding the position of the word to the transformer. These encodings are added to the embeddings of each word and the resulting embeddings are passed to the transformer.

An encoder block has multiple encoder blocks and the decoder block has the same number of decoder blocks. The number of blocks is a hyper-parameter which can be tuned while training.

Figure 4.12: The transformer - model architecture



4.4.2 Neural Machine Translation by Jointly Learning to Align and Translate

Inspired by [9] and some quotations from it. This model found on [9] is based on the paper [2] published in May 2016. This model is a sequence-to-sequence model composed of an encoder, and attention layer and a decoder.

Encoder

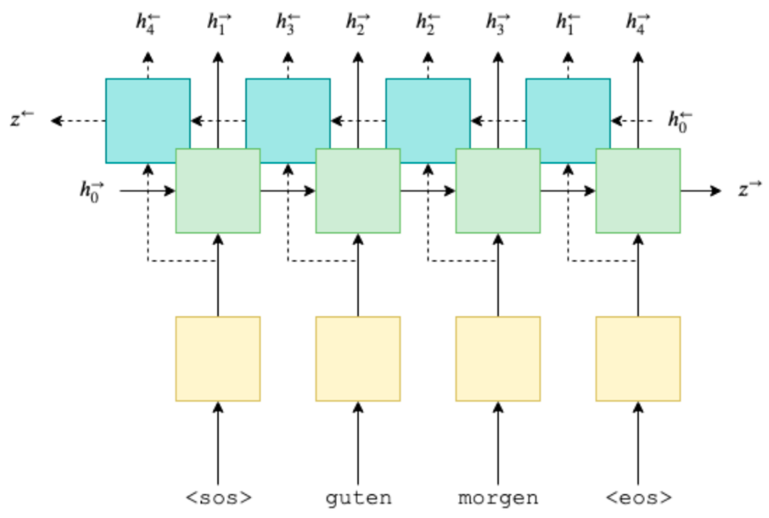
The encoder is a single-layer bidirectional RNN (two RNN at each layer, a forward going over the embedded sentence from left to right and a backward going from right to left) (see figure 4.13).

The hidden layers for forward and backward RNN of the t-th word of the input are computed and outputted as following:
$h_{\overrightarrow{t}} = EncoderGRU(e(x_{\overrightarrow{t}}, h_{\overrightarrow{i-1}})$ and $h_{\overleftarrow{t}} = EncoderGRU(e(x_{\overleftarrow{t}}, h_{\overleftarrow{t-1}})$, where $e(x_t)$ is the embedding of the t-th word in the input sequence.

$h_{\overrightarrow{n}} = z_{\overrightarrow{n}}$ is the last hidden state (the context vector) obtain from the last word x_n of the input x ($x = x_1 x_2 ... x_n$). And $h_{\overleftarrow{n}} = z_{\overleftarrow{n}}$ is the last hidden state obtain from the first word x_1 of the input x.

As the decoder is not bidirectional, it only needs one single context vector z to be used as the first hidden state s_0; but we have two context vectors ($h_{\overrightarrow{n}}$ and $h_{\overleftarrow{n}}$); so we concatenate them, then pass them through a linear layer g and apply the tanh activation function; $z = \tanh(g(h_{\overrightarrow{n}}, h_{\overleftarrow{n}})) = s_0$.

Figure 4.13: Encoder with bidirectional RNN



Attention layer

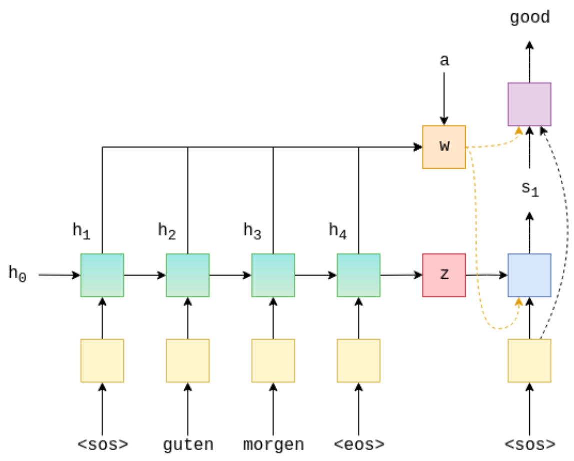
The attention layer takes the previous hidden state s_{t-1} (what we have decoded so far) and all the hidden states H (forward and backward) from the encoder. This layer will output an attention vector a_t (same length than the input sequence, each element between 0 to 1, sum of elements is equal to 1). The attention vector represents which words in the input sequence we should pay more attention on in order to correctly predict y_{t+1}. More detailed description of attention vector computation can be found in [9].

Decoder

The decoder contains the attention layer that takes the previous hidden state and all encoder hidden layers H in order to return the attention vector a_t. This attention vector is used to create a weighted vector w (which is a weighted sum of the encode hidden layers): $w_t = a_t H$. Given the embedding for the a word in output $e(y_t)$, w_t and the previous decoder hidden state s_{t-1} we get the hidden state at current step s_t. Then $e(y_t)$, w_t and s_t are passed to a linear function that gives the next word in the output y_{t+1}.

In figure 4.14 we can see the decoding of the first word of the input. The German "guten" has been decoded as "good" in English.

Figure 4.14: First word decoding for the example of translation from German to English



The legend for the figure 4.14: green blocks are the forward and backward encore RNNs, combined they output H. The red block is the context vector: $z = h_n = s_0$.The blue block is the decoder RNN which outputs s_t the hidden state at time step t. The purple block is the linear layer which outputs y_{t+1}. The orange block is the calculation of the weighted sum over H by a_t and outputs w_t.

In table 4.2, we have the settings used for this model.

encoder embedding dimension	32
decoder embedding dimension	32
encoder hidden layer dimension	64
decoder hidden layer dimension	64
attention dimension	8
encoder dropout	0.5
decoder dropout	0.5

Table 4.2: Characteristics of the sequence to sequence model

4.4.3 GPT-2 (Generative Pre-training)

"GPT-2 is a large transformer-based language model with 1.5 billion parameters trained on a data set of 8 mission web pages (massive 40GB data set)" [13] (see figure 4.12). The goal of GPT-2 is to predict the next word of a sentence given all the previous words in this sentence (predict one word at a time). It achieves state-of-the art scores on a variety domain-specific language modeling tasks; this is why we try to use it on our sequences.

GPT-2 is only using the transformer decoder stack and does not keep the encoder blocks. In the decoder blocks, there is a component called " masked self-attention ", it does like attention, but does not look at all the words of the input, but only previous tokens; ie the incomplete text given as inout of the decoder stack.

GPT-2 architecture uses 120-layer decoders with masked self-attention heads and trained for 100 epochs.

Chapter 5

Tests and results

To test the models we selected and evaluate the way we decided to generate the sequences for graphs and trees, we must first train the models on some data and compute statistics on the results.

A session of test is composed of several steps:

- Sequences generation
- Data set generation
- Training
- Testing
- Statistics

Sequences generation: Having those pieces of codes available (cloned from github), they are parsed into trees and graphs, then analysis are done on those structures as already explained (difference computation,...) and so we can generate the sequences for EGs and ASTs according to the method applied for each bug detected in EGs (see section 4.3 Sequence generation). However, we cannot keep all the sequences generated, because some are composed of too many instructions, and it is clear that those too long sequences may not be predicted well. So the maximal length of the instructions accepted for training and testing is 50-instructions-long. Moreover, we don't accept sequences containing "move" instructions, because this is a case difficult to handle. Note: for all the sequences generated, there are some sequences corresponding to the same source codes, because the static program analyser detected several bugs in those codes. For instance if in C_a we detected three bugs, we would have three sequences between graphs, one sequence correcting one bug. For the sequences between the trees, the three sequences would be the same, as we do not consider any matching area for trees.

Data set generation:
After generating the sequences for inputs and outputs of the neural network, the data

set is shuffled and must be split in two data sets, the training and the testing data set. Generally, on the entire amount of sequences available, 75% were dedicated to training and the rest was for testing. For some other experiments, we chose to dedicate 90% of the data for training and so 10% for testing.

Training:
The two models described in section 4.4 are used for the experiments. The number of epochs the models are trained on, the number of data used, and different type of hyper-parameters are tuned in order to improve the accuracy while inferring on non-met data.

Testing:
To test the accuracy on rebuilding trees from predicted sequence, we must test the models. For that, we feed the trained model with input sequences on graphs that were not met at training time, and we compare the predictions made with the ground truths that are sequences generated on trees. In order to have a global result regarding the model, we compute some statistics.

Statistics: Once the predictions have been made from the trained model, there is some information that is interesting to observe in order to evaluate the model but as well the two instruction-languages that have been defined. Below are the different indicators that are interesting.

- The proportion of well predicted sequences (entirely) in total and per rule (ie per instance of bugs detected): this is the ratio of sequences that are entirely well predicted over all the predictions made;
- the proportion of instructions well predicted in total and per rule: as sequences are composed of instructions, this is the ratio of well predicted sequences over the total number of instructions in the test data set;
- the average length of well predicted sequences: this is how much word (not instruction) the well predicted sequences are composed of;
- the proportion of well rebuilt trees in total and per rule: this is the ratio of well rebuilt trees from predicted sequences over all the rebuilt trees;
- the proportion of well predicted nodes in trees in total and per rule: this is the ratio of nodes that are well reconstructed after applying sequences predicted to rebuild trees over all the reconstructed nodes.

This is important to have those statistics on all the data in the testing data set in order to have an overall appreciation on the model trained. But this is more important to have those statistics per rule (meaning per instance of bugs), because we are more interested in understanding what instances of bugs have fixes that are more difficult or easier to be learnt by the neural network. This is important to know the strength of the model, so that we know that for some instance of bugs, the model would be efficient in detecting and correcting the bugs, and for some other categories, the model could be not useful.

As already explained, the data are collected from github repositories (public repositories: open source projects). For the experiments, we only took source codes written in Java. After that the parsing is done on those codes, for each piece of code collected, every versions (commits) are compared together so that we can observe what has been changed in a source code from a commit version to the following. With this is mind, we generated the sequences for each bug found in each pieces of code; but those sequence generated were filtered in order to get sequences from which we can learn how to generalize. As already mentioned, all cases where sequences for trees or sequences for graphs were containing more than 50 instructions were not kept to create the training and the testing data sets; moreover the sequences containing at least one "move" instruction were not kept as well.
This way, we were able to generate a maximum of 111,645 sequences of instructions from about 80,000 repositories cloned. Here are two histograms that gives the distribution of the length of the sequences generated for EGs and for ASTs (see figure 5.1 and 5.2).

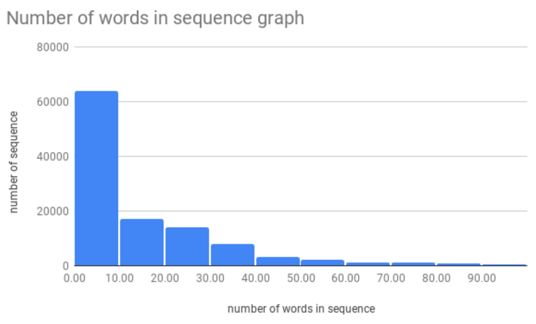


Figure 5.1: Sequences for graphs lengths

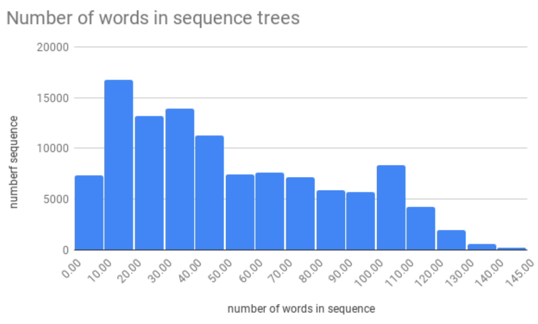


Figure 5.2: Sequences for trees lengths

And from those sequences generated, 447 different instances of bugs were the concern of the fixing sequences going to API (application programming interface) calls to manipulation of Null objects. Here are the most commons instances of bugs met and their occurrence (see Table 5.1). The bugs are identified by keys that express briefly what the bug is: CallOnNull means that a methods may be called from a Null Object, ApiMigration* concerns problems about imports...

Instance of bug	number of occurrences
ApiMigration java.lang.StringBuffer	8,962
dontUsePrintStackTrace	6,028
ReplaceBoxedConstructor java.lang.Integer	2,142
LogLevelCheck info	2,033
ApiMigration com.google.common.base.Objects.toStringHelper	1,846
dontCatch 1	1,596
CallOnNull	1,442
ApiMigration org.apache.commons.lang.StringUtils.isBlank	1,029
ApiMigration com.google.common.base.Optional.absent	956
ApiMigration org.apache.commons.lang.StringUtils.isEmpty	941

Table 5.1: Occurrence of more frequent bug instances

5.1 Results for model: Neural Machine Translation by Jointly Learning to Align and Translate

What we call a sample in this section is one sequence of instructions between two graphs or two trees.

The sequences have been generated as explained in the section 4.3; so we have four files: one with sequences for EGs (one sample per line) for training, another one with sequences for ASTs (one sample per line) for training, and two other similar files but for testing.
While training, the model receives the file with sequences of instructions for EGs as input and sequences of instructions for ASTs as output. Then to test if the model learnt well how to predict sequences of instructions for trees from sequences for graphs, we use the test files: the model predicts sequences from sequences for graphs and we compare those predictions with the sequences in the file of sequences for trees. At testing step, the file containing sequences of instructions for EGs is passed as input and for output we get a file with all corresponding sequences of instructions for ASTs (one per line).

There is an hyper-parameter that is interesting to tune here: the batch size. Augmenting the batch size makes the training going faster, and as the sequence to sequence model takes time to be trained it might be interesting to get a not-to-small batch size. On the other hand, having a smaller batch size makes the results more precise. So here there is a trade-off speed / precision.
To sum-up the results, for around 10,000 and around 30,000 samples, a training has been made on 50 epochs with a batch size of 15 and a batch size of 150 (we did not take a batch size greater than 150, because it triggered memory errors). We compared the

batch size choice by comparing the proportion of well entirely predicted AST sequences and the proportion of well predicted instructions among all the sequences. The results are listed in the table 5.2.

experiment no	1	2	3	4
#epochs	50	50	50	50
batch size	15	150	15	150
#training sequences	7534	7534	31469	31469
#testing sequences	2511	2511	4702	4702
training time (hours)	1.1	0.15	8.1	1.6
% well predicted sequences	0%	0%	3.89%	0.5%
% well predicted instructions	22.4%	16.13%	32.4%	21.4%

Table 5.2: Seq2Seq model, batch size comparison

We can see in table 5.2 that training with a batch size of 15 instead of 150 takes really more time, in the case of around 36,000 sequences, it takes five times more time; but we get significantly (see case of 36,000 sequences) better results for 15 as batch size than 150; in fact the time needed to get almost the same results with a batch size of 150 was 12 hours. So for the following experiment results, a batch size of 15 is used.

Here are the global results (not split by rules) of the several experiments in table 5.3.

experiment no	3
#epochs	50
#training sequences	31,469
#testing sequences	4,702
training time (hours)	8.1
% well predicted sequences	3.89%
mean length well predicted sequence (# instructions)	11.28
% well predicted instructions	32.39%
% well rebuilt trees	12.04%
% well rebuilt nodes	99.39%
# of rules	211

Table 5.3: Seq2Seq model, global results

We can see in the results of the table 5.3 that the proportion of well rebuilt trees is greater than the proportion of well predicted sequences; this is always the case. Indeed, when a sequence is well predicted then applying it to the buggy tree will always give the correct fixed tree, but in the case where the sequence is not well predicted, it still might give the correct fixed tree. For each buggy tree, there is not one unique sequence that fix it into the correct tree.

As already explained, it is interesting to have those global results to see if the trained model has the ability to correct bugs as expected. But it is also interesting to know what kind of bugs the model was more performing on (ie on which bugs the proportion of well rebuilt trees is higher). The results on the ten more well-learnt instances of bugs are given per experiment number.

Here are the results split by rules of the several experiments in table 5.4. Only some interesting rules from all are listed. Note: the different rule keys given are describing the bug itself, for instance CallOnNull means that some methods are called on Null objects, ApiMigration* rules concern problems about imports,...

Experiment 3			
Rule key	# occurrences	% well rebuilt trees	% correct nodes in trees
IncorrectConditionCheck isPresent	2	100%	100%
CallOnNull	175	26.3%	99.4%
IterateOver keySet	61	21.3%	99.4%
ReplaceBoxedConstructor java.lang.Long	83	18.1%	99.6%
ApiMigration java.lang.StringBuffer	968	17.6%	7 99.7%
dontUsePrintStackTrace	599	14%	99.3%
StringEquals	73	12.3%	99.2%
dontCatch 1	165	12.1%	99.2%
ReplaceBoxedConstructor java.lang.Integer	188	11.7%	99.6%
LogLevelCheck info	221	10.9%	99.4%

Table 5.4: Seq2Seq model, split by rules results

Only some interesting rules are listed in the table 5.4, but the model was able to correct bugs (rebuild correct tree entirely) for 81 instances of bugs out of 211, which is good.

The rules listed in the table 5.4 are interesting because there occur really often (not the first one), and so we can see that those rules are quite often entirely fixed. There are 14 rules for which the proportion of well rebuilt trees is 1, but they are not stored in the table because generally these rules occur just one or two times in the testing data set; but still, there are rules that we perfectly fix.

5.2 Results for model: GPT-2

What we call a sample in this section is the sequence of instructions between two graphs or two trees.

As for the previous model, sequences have been generated as explained in the section 4.3; so we got the same four files. However, for this model, the input file should be formatted, and there is no output file. The input file is created as following: at the i-th line of this created file, the line starts with the i-th sequence of instructions for graphs, followed by a character (that should be the same for each line, we decided to put #) followed by the i-th sequence of instructions for trees. For the testing file, at the i-th line, the line starts with the i-th sequence of instructions for graphs followed by the same special character #, but nothing after this #. Then the model should predict a file with each line containing a sequence of instructions for trees corresponding to the sequence of instructions for graphs in the same line of the testing file.

Here are the global results (not split by rules) of the several experiments in table 5.5.

experiment no	1	2
#epochs	2	4
#training sequences	100,480	100,480
#testing sequences	2,500	1,253
training time (hours)	8.3	17.5
% well predicted sequences	2.4%	2.8%
mean length well predicted sequence (# instructions)	3.61	4.37
% well predicted instructions	15.14	15.63%
% well rebuilt trees	5.78%	6%
% correct nodes in rebuilt trees	99.15%	98.96%
# of rules	211	164

Table 5.5: GPT2 model, global results

For this model also, we need results split by the rules.

Here are the results split by rules of the several experiments in table 5.6.

Experiment 1			
Rule key	# occurrences	% well rebuilt trees	% correct nodes in trees
ApiMigration java.lang.StringBuffer	438	25%	99.8%
dontCatch 1	59	10%	99.6%
LogLevelCheck	85	5.9%	99.1%
LogLevelCheck debug	77	5.2%	99%
ApiMigration java.util.LinkedList	29	3%	99.7%
NoStringConcatArg1 append	119	2.5%	97.7%
dontUsePrintStackTrace	337	1.5%	99%
Experiment 2			
Rule key	# occurrences	% well rebuilt trees	% correct nodes in trees
ApiMigration java.lang.StringBuffer	215	27.91%	99.87%
LogLevelCheck info	42	9.52%	99.34%
catchingInterruptedException WithoutInterrupt	14	7.14%	97.9%
dontCatch 1	28	7.14%	99.77%
LogLevelCheck debug	47	4.26%	99%
dontUsePrintStackTrace	177	1.13%	98.82%

Table 5.6: GPT2 model, split by rules results

Only some interesting rules are listed in the table 5.6 (for experiment 1), but the model was able to correct bugs (rebuild correct tree entirely) for only 9 instances of bugs out of 211. For experiment 2, the model learned to correct 10 bugs out of the 164 instances of bugs. In experiment 2, we can say that for sure the model was able to learn how to fix more than 10 bugs, but we would have need to do predictions on more samples; but it is dramatically time consuming.

According to table 5.5 and 5.6, predictions made with GPT2 are less precise than the ones made with the sequence to sequence model. Moreover the gpt2 model was able to learn to fix less bugs than the sequence to sequence model. We should probably run the training for this model for a longer time in order to get better results.

Chapter 6

Conclusion

I am really glad that I had the chance to do my master's thesis at DeepCode, helped by Dr. Veselin Raychev and Benjamin Mularczyk. It has been more difficult to work than expected regarding the Covid 19 situation, as the concepts were not that simple to exploit; it was sometimes difficult to explain the different ideas through the thesis, and I think it brought me to some misunderstanding and got the work done slower than I wanted it to.

This was the first time I worked on a more "researchy" project, I found it really interesting. I have learnt to think really differently and more deeply. Technically, it brought me to learn and apply C++ and work with deep learning models that I never worked with: reccurent neural networks, encoders-decoders ... Moreover I think that the covid19 situation made me work more autonomously than I should have, so it brought me some extra skills. This internship gave me passion for research.

Regarding Deepcode, I have been impressed by how fast the company reacted regarding the situation of the covid19, as the safety of its employees is at highest priority. All the measures were taken for employees that really needed and wanted to still work form the office (high hygiene measures). For the other employees that did not need or want to come work to the office, the working from home configuration was highly encouraged by managers and all the measures were as well taken in order to have the "best working from home experience" as possible: best working conditions, best emotional and physical health for employees. A lot of feedback was done from employees and managers (every two weeks) in order to understand how to make the situation easier and more comfortable for everyone. And many game evenings were organised remotely to keep the employees cohesion.
Moreover it has been really enjoyable to see that all the employees were actively taking part in the company decisions and discussions and the fact that their thoughts were considered. This is a point that is very important for DeepCode that everyone feels concerned and heard in the company.

Regarding the results, it has been difficult to get results first, because having totally correct sequences of instructions was not straightforward. Moreover many possibilities have been tried, and some of them seemed to give interesting results but they weren't. Finally we got models that definitely learn something, but there are maybe others models that we should have tried that give better results. I would have like to have the time to test other settings on the models I have too. There are also other ways to create the sequences that I would have liked to try, but I did not have the time to. This would have also been interesting to run experiments for other languages for which we have the static program analysis (JavaScript, Python and C/C++).

Bibliography

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser and I. Polsukhi, "Attention is all you need" *https://arxiv.org/abs/1706.03762*, June 2017.

[2] D. Bahdanau, K. Cho, Y. Bengio, "Neural Machine Translation by Jointly Learning to Align and Translate", *https://arxiv.org/abs/1409.0473*, May 2016.

[3] D. Britz, "Recurrent Neural Networks - Introduction" *http://www.wildml.com*, September 2015.

[4] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I.Sutskever, "Language Models are Unsupervised Multitask Learners" *https://cdn.openai.com/better-language-models/language_ models_ are_ unsupervised_ multitask_ learners.pdf*, 2019.

[5] J. Bader, A. Scott, M. Pradel, S. Chandra, "Getafix: Learning to Fix Bugs Automatically" *https://arxiv.org/pdf/1902.06111.pdf*, November 2019.

[6] D. Vrabac, N. Svensson, "Sequence to Sequence Machine Learning for Automatic Program Repair" *https://www.diva-portal.org/smash/get/diva2:1330082/FULLTEXT01.pdf*, 2019.

[7] I. Sutskever, O. Vinyals, Q. V. Le, "Sequence to sequence learning with neural network" *http://arxiv.org/abs/1409.3215*, Dec 2014.

[8] H. Hata, E. Shihab, and G. Neubig, "Learning to generate corrective patches using neural machine translation," *http://arxiv.org/abs/1812.07170*, Dec 2018.

[9] Pytorch, *https://pytorch.org/tutorials/beginner/torchtext_ translation_ tutorial.html.*

[10] I. Goodfellow, Y. Bengio, A. Courville, Deep learning. MIT press, 2016.

[11] D. E. Rumelhart, G. E. Hinton, R. J. Williams, "Learning representations by back-propagating errors," Cognitive modeling, vol. 5, no. 3, p. 1, 1988.

[12] https://en.wikipedia.org/wiki/Corporate_social_responsibility.

[13] https://openai.com/blog/better-language-models/.

[14] P. Dwivedi, Train a GPT-2 Transformer to write Harry Potter Books. Medium towards data science, March 2020.

[15] J. Alammar, *http://jalammar.github.io/illustrated-transformer/*, June 2018.

[16] J. Alammar, *http://jalammar.github.io/illustrated-transformer/*, August 2019.

[17] R. Gupta, A. Kanada, S. Shevade, "Deep Reinforcement Learning for Programming Languages Correction", *https://arxiv.org/pdf/1801.10467.pdf*, Jan 2018.

[18] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, D. Poshyvanyk, " An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation", *https://arxiv.org/pdf/1812.08693.pdf*, May 2019.

Appendix A

Implementation details

A.1 AST and EG: implementation details

Abstract Syntax Tree

In the given framework, an AST is represented by the class TreeStorage which is composed of many nodes (TreeNode in the class diagram) that are linked by some relationships; some nodes are parents of others (each node, apart from the root, has one unique parent: this is a property of a tree), and some nodes are siblings of others (one node have one right sibling and one left sibling; the sibling is -1 if this sibling does not exist). So the AST is composed of a vector of nodes (nodes_) which are identified by their position in the vector. In trees, nodes are arranged in pre-order order. In the vector of nodes, there can be some positions that are not allocated to a node, the number of those free positions is kept in num_deallocated_, and the first free position is kept in first_free_node. Each node (TreeNode) has a type and a value, as explained previously, an AST may contains some nodes that are variable for instance, so for those nodes, the type would be the type of the variable and value its value: eg, int i = 0. But in order to locate it in the space of nodes in the tree, a node has one parent (the parent is an integer, if parent = i, this means that the parent of the current node is the node at the i-th position of the vector of nodes nodes_), one left sibling and one right sibling (integers as well, same explanation than parent), it has also a child_index, this means that this node is the i-th child of its parent if child_index = i, and it has a first_child and a last_child (integers, same explanation as for parent) that represent the nodes that are respectively the first and the last child of the current node.

We also have a tree traversal class that provides methods that allows to go through trees by depth first search order or breadth first search order.

Concerning the trees, there are some instructions that are already implemented; we can remove a node from a tree or add a node, and go to a node.

Event Graph

In the given framework, an EG is represented by the class SourceGraphOp. A graph is composed of nodes and edges; a node of the graph has a type and a value (EventData), it is identified by its position in the vector of nodes nodes_ (if the node is at the i-th position of the vector nodes_, we say that this is the i-th node) and it is linked to other nodes by edges (Edge). An edge points to a target node (the node pointed is identified by its identifier) and the edge has data (EdgeData). A node has a list of edges arriving at (redges) and a list of edges starting from (edges).

What is noticeable is that ASTs are included in event graphs; in the way that the nodes that are in the corresponding AST are also contained in the EG.

A.2 Instructions: implementation details

Here are the implementation details of the instructions that we can apply on ASTs and that are described in section 4.2.6.

GOTO instruction

The "goto" instruction is just about accessing a node in the tree. For the tree *t*, if we need to access the node with identifier i (ie the node is at the i-th position of the vector nodes_), we just call `t.node(i)`.

REMOVENODE instruction

The instruction to remove nodes is already implemented. If we want to delete in the tree *t* the node with identifier i, we call `t.RemoveNode(i)`. This function removes the node i and all its children.

MODIFYNODE instruction

As said in section 4.2.6, to modify the node of identifier i, we just have to access it with `node_to_modify = t.node(i)` and then we change its type and or its value: `node_to_modify.SetType(new_type)`, `node_to_modify.SetValue(new_value)`.

ADDDOWN instruction

As explained in the section 4.2.6, adding a node is done in two steps; first we give the node its type and its value, and then we give the node its location context.

To give a node its type and value, we do as for node modification; we access it `node_to_modify = t.node(i)` and then we change its type and or its value: `node_to_modify.SetType(new_type)`, `node_to_modify.SetValue(new_value)`.
Giving a geographical context is a bit more complex. First we must give to this new node a parent, `new_node.parent = S(parent)`, where S is the semantic used to identify the parent (see GOTO instruction part).
Then we must give to the new node its child_index, as we call the " adddown " instruction, we add this new node as the first child of its parent, thus we have `new_node.child_index = 0`, and the new node is the first child of its parent: `tree.node(tree.node(new_id).parent).first_child = new_id`.
If it happens that the added node is the last child of its parent, in this case, the first child of the parent is also its last child: `parent_node.last_child = new_id`.
But if before adding this new node thanks to " adddown " the parent already had children, then the previous first child is now the second child, so its child index has to be incremented by one, more generally, all the children that were already there must have their child_index incremented by one.

```
for child of tree.node(new_id).parent do
  tree.node(child).child_index++.
```

Then it is important that we give to the new added node a right sibling (not a left sibling as it is the first node), so the right sibling is the previous first child, and the left sibling of the previous first child is the new added node : `tree.node(new_id).right_sib = previous first child` and `tree.node(previous first child).left_sib = new_id`.

ADDRIGHT instruction

The added node and its left sibling have the same parent:
`new_node.parent = tree.node(tree.node(new_id).left_sib).parent`.
Then we must give to the new node its child_index, as we call the " addright " instruction, we have that the child_index of the new node is incremented by one compared to the child_index of its left sibling (from which we add the new node):

```
new_node.child_index =
    tree.node(tree.node(new_id).left_sib).child_index++.
```

If it happens that the added node is the last child of its parent, we have:
`parent_node.last_child = new_id`.
There is no case where the node added with " addright " could be the first node of its parent.
All the children of the parent of the new node that are coming after the left sibling from which we add the node must have their child_index incremented by one.

```
for child of parent after the left sibling from which we addright
   tree.node(child).child_index++.
```

Then it is important that we give to the new added node a right sibling and a left sibling. The right sibling of the new node is the previous right sibling of the left sibling from which we are adding, and the left sibling of this right sibling is the new added node :

`tree.node(new_id).right_sib = tree.node(left sibling from which we are adding).right_sib` and
`tree.node(tree.node(left sibling from which we are adding).right_sib).left_sib = new_id`.
Concerning the left sibling of the new node, it is the left sibling from which we are adding, and the right sibling of this left sibling is the new added node: `tree.node(new_id).left_sib = left sibling from which we are adding` and `tree.node(left sibling from which we are adding).right_sib = new_id`.

This last instruction is similar to the previous one, the only difference is that the node added is not the first node of its parent.