Learning Variable Mappings to Repair and Verify Programs

Pedro Orvalho

Instituto Superior Técnico
University of Lisbon
Portugal

Wednesday, 29th March 2023

Motivation

- The problem of program equivalence is undecidable;

Motivation

- The problem of program equivalence is undecidable;
- Thus, repairing an incorrect program based on a correct implementation is challenging.

Motivation

- The problem of program equivalence is undecidable;
- Thus, repairing an incorrect program based on a correct implementation is challenging.
- To compare both programs, program repair tools need to find a relation between these programs' variables.

Motivation

Function that finds and returns the maximum number among n1, n2 and n3.

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

Function that finds and returns the maximum number among x, y and z.

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

Motivation

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

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

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

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

Variable Mapping: {m : m; n1 : x; n2 : y; n3 : z}.

Motivation

Function that finds and returns the maximum number among n1, n2 and n3.

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

Function that finds and returns the maximum number among x, y and z.

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

Variable Mapping: {m : m; n1 : x; n2 : y; n3 : z}.

Motivation

Besides *program repair* [Ahmed et al., 2022], the task of mapping variables between programs is also important for:

- *program analysis*;
- *program equivalence*;
- *program verification*;
- *program clustering*;
- *clone detection*;
- *plagiarism detection*.

Goal

- Create a graph program representation that takes advantage of the structural information of the *abstract syntax trees (ASTs)* of programs;

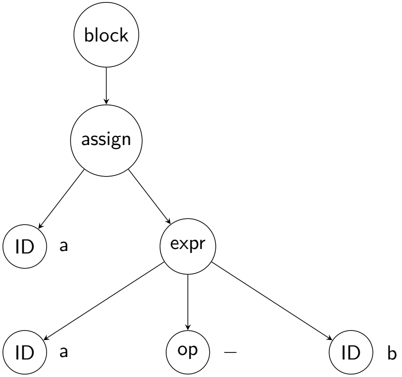
Goal

- Create a graph program representation that takes advantage of the structural information of the *abstract syntax trees (ASTs)* of programs;
- Use our program representation to learn how to map the set of variables between a correct program and a faulty one using *graph neural networks (GNNs)*.

Program Representation

An expression that uses int variables `a` and `b`, previously declared in the program.

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

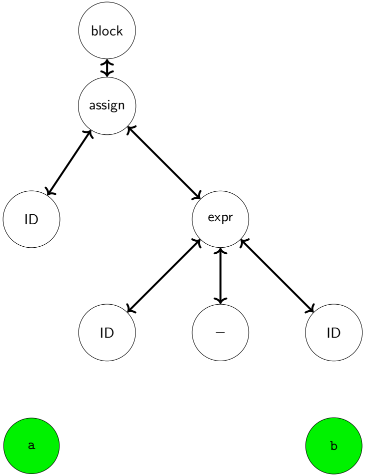


(a) Part of the AST representation.

Program Representation

An expression that uses int variables a and b, previously declared in the program.

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

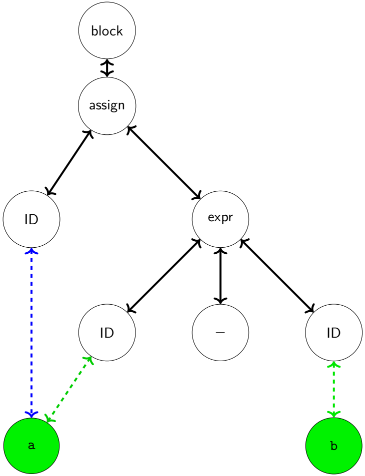


(b) Our program representation.

Program Representation

An expression that uses int variables a and b, previously declared in the program.

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

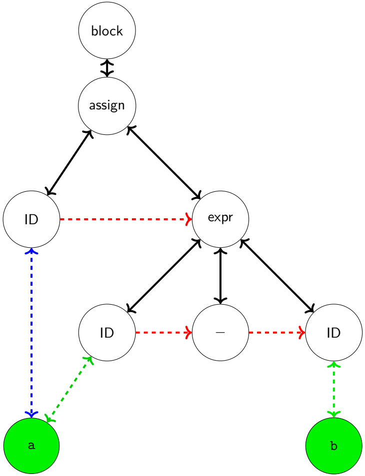


(c) Our program representation.

Program Representation

An expression that uses int variables `a` and `b`, previously declared in the program.

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

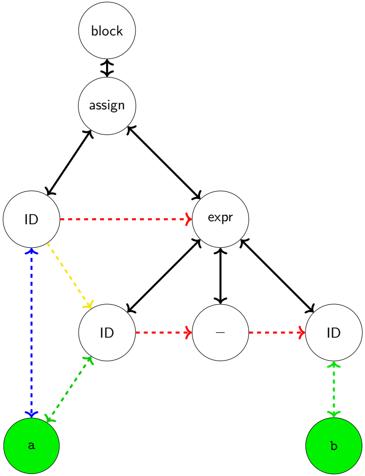


(d) Our program representation.

Program Representation

An expression that uses int variables a and b, previously declared in the program.

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



(e) Our program representation.

Graph Neural Networks (GNNs)

- We are using a Relational Graph Convolutional Neural Network (RGCN);

Graph Neural Networks (GNNs)

- We are using a Relational Graph Convolutional Neural Network (RGCN);
- We perform *message passing* between the nodes of our representations, so that information can be passed between the local constituents;

Graph Neural Networks (GNNs)

- We are using a Relational Graph Convolutional Neural Network (RGCN);
- We perform *message passing* between the nodes of our representations, so that information can be passed between the local constituents;
- After several message passing rounds, we obtain numerical vectors corresponding to each variable node in the two programs;

Graph Neural Networks (GNNs)

- We are using a Relational Graph Convolutional Neural Network (RGCN);
- We perform *message passing* between the nodes of our representations, so that information can be passed between the local constituents;
- After several message passing rounds, we obtain numerical vectors corresponding to each variable node in the two programs;
- We compute scalar products between each possible combination of variable nodes in the two programs, followed by a softmax function.

C-Pack-IPAs: Dataset of Introductory Programming Assignments (IPAs)

Table: Description of C-PACK-IPAS [Orvalho et al., 2022].

Academic Year	#IPAs	#Correct Submissions	#Incorrect Submissions
1st Year	10	238	107
2nd Year	10	78	60

Data Augmentation

- Since we need to know the real variable mappings between programs to evaluate our representation, we used MultIPAs [Orvalho et al., 2022] to generate a dataset of pairs of correct/incorrect programs.

Data Augmentation

- Since we need to know the real variable mappings between programs to evaluate our representation, we used MultIPAs [Orvalho et al., 2022] to generate a dataset of pairs of correct/incorrect programs.
- The second reason to use MultIPAs was that our dataset, C-Pack-IPAs, is too small, i.e., contains only a few hundred submissions.

Data Augmentation

The main goal of MultIPAs is to augment IPAs benchmarks with:

- more semantically correct implementations **(program mutations)**;

Data Augmentation

The main goal of MultIPAs is to augment IPAs benchmarks with:

- more semantically correct implementations **(program mutations)**;
- new semantically incorrect programs **(program mutilations/bugs)**;

Data Augmentation

The main goal of MultIPAs is to augment IPAs benchmarks with:

- more semantically correct implementations **(program mutations)**;
- new semantically incorrect programs **(program mutilations/bugs)**;
- **variable mappings** between the original program and the mutated and/or mutilated program;

Data Augmentation

The main goal of MultIPAs is to augment IPAs benchmarks with:

- more semantically correct implementations **(program mutations)**;
- new semantically incorrect programs **(program mutilations/bugs)**;
- **variable mappings** between the original program and the mutated and/or mutilated program;
- information about the **types and the number of bugs** present in each generated incorrect program.

MultIPAs

MultIPAs can perform six syntactic program mutations to change the programs' syntax but not their semantics, such as:

MultIPAs

MultIPAs can perform six syntactic program mutations to change the programs' syntax but not their semantics, such as:

- *Comparison Expression Mirroring*;
- *If-else-statements Swapping*;
- *Increment/Decrement Operators Mirroring*;
- *Variable Declarations Reordering*;
- *For-2-While Translation*;
- *Variable Addition*.

MultIPAs

- Next, MultIPAs can introduce three bugs:

MultIPAs

- Next, MultIPAs can introduce three bugs:
  - wrong comparison operator (WCO);

MultIPAs

- Next, MultIPAs can introduce three bugs:
  - wrong comparison operator (WCO);
  - variable misuse (VM);

MultIPAs

- Next, MultIPAs can introduce three bugs:
  - wrong comparison operator (WCO);
  - variable misuse (VM);
  - missing expression (ME).

Data Augmentation

Original program.

```c
int main(){
  int n;
  int i, s;
  scanf("%d", &n);
  s=0;
  for(i=1; i<=n; i++){
    s = s+i;
    printf("%d\n",s);
  }

  printf("%d\n",s);
  return 0;
}
```

Incorrect program.

```c
int main(){
  int n, s, i, y;
  scanf("%d", &n);
  s=0;
  i = 1;
  while(n>=i){

    printf("%d\n",s);
    ++i;
  }
  printf("%d\n",s);
  return 0;
}
```

C-Pack-IPAs: Augmented Dataset

Table: The description of the training, validation, and evaluation sets based on C-Pack-IPAs.

	Buggy Programs			
	WCO Bug	VM Bug	ME Bug	All Bugs
Training set (1st Year)	3372	5170	2908	11450
Validation set (1st Year)	1457	1457	1023	3937
Evaluation set (2nd Year)	1078	1936	1152	4166

Use Cases: Program Repair

- We use variable mappings to repair an incorrect program using a correct implementation for the same IPA without considering the programs' structures.

Use Cases: Program Repair

- We use variable mappings to repair an incorrect program using a correct implementation for the same IPA without considering the programs' structures.
- We claim that variable mappings are informative enough to repair these three realistic types of bugs.

Use Cases: Program Repair

General idea:

- Given a buggy program, we search for and try to repair all three types of bugs;

Use Cases: Program Repair

General idea:

- Given a buggy program, we search for and try to repair all three types of bugs;
- First, we rename all the variables in the incorrect program based on the variable mapping;

Use Cases: Program Repair

General idea:

- Given a buggy program, we search for and try to repair all three types of bugs;
- First, we rename all the variables in the incorrect program based on the variable mapping;
- Then, by comparing the expressions of both programs, we try to fix the incorrect one by replacing the expressions that do not appear in the correct program, with the correct program's expressions;

Use Cases: Program Repair

General idea:

- Given a buggy program, we search for and try to repair all three types of bugs;
- First, we rename all the variables in the incorrect program based on the variable mapping;
- Then, by comparing the expressions of both programs, we try to fix the incorrect one by replacing the expressions that do not appear in the correct program, with the correct program's expressions;
- Whenever we find a possible fix, we check if the program is correct using the test suite.

Results

Training

Table: Validation mappings fully correct after 20 training epochs.

	Buggy Programs			
	WCO Bug	VM Bug	ME Bug	All Bugs
Accuracy	93.7%	95.8%	93.4%	96.49%

Evaluation

Table: The number of correct variable mappings generated by our GNN on the evaluation dataset and the average overlap coefficients between the real mappings and our GNN's variable mappings.

Evaluation Metric	**Buggy Programs**			
	WCO Bug	VM Bug	ME Bug	All Bugs
# Correct Mappings	87.38%	81.87%	79.95%	82.77%
Avg Overlap Coefficient	96.99%	94.28%	94.51%	95.05%

Ablation Study

Table: Percentage of variable mappings fully correct on the validation set for different sets of edges used. Each type of edge is represented by an index using the mapping: $\{$0: AST; 1: sibling; 2: write; 3: read; 4: chronological$\}$.

Edges Used	All	(1,2,3,4)	(0,2,3,4)	(0,1,3,4)	(0,1,2,4)	(0,1,2,3)	(0,1)
Accuracy	**96.49%**	52.53%	73.76%	95.45%	94.87%	96.06%	94.74%

Repairing Programs

Table: The number of programs repaired by each different repair technique: Verifix, Clara, and our repair approach based on our GNN's variable mappings. The last row shows the results of repairing the programs using the real variable mappings (ground truth).

	Buggy Programs				Not Succeeded	
Repair Method	WCO Bug	VM Bug	ME Bug	All Bugs	**% Failed**	**% Timeouts (60s)**
Verifix	555 (51.48%)	1292 (66.74%)	741 (64.32%)	2588 (62.12%)	1471 (35.31%)	107 (2.57%)
Clara	722 (66.98%)	1517 (78.36%)	764 (66.32%)	3003 (72.08%)	1153 (27.68%)	**10 (0.24%)**
GNN	**942 (87.38%)**	**1537 (79.39%)**	**898 (77.95%)**	**3377 (81.06%)**	**711 (17.07%)**	78 (1.87%)
Ground Truth	1078 (100.0%)	1877 (96.95%)	1129 (98.0%)	4084 (98.03%)	0 (0.0%)	82 (1.97%)

Repairing Programs

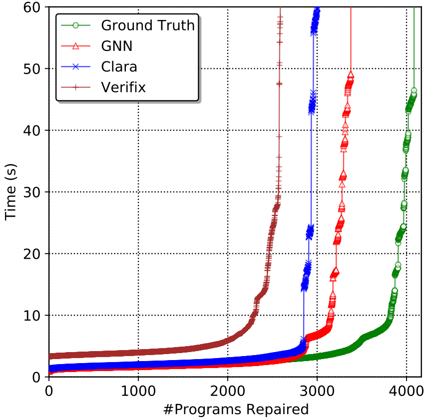


Figure: Cactus plot - The time spent by each method repairing each program of the evaluation dataset, using a timeout of 60 seconds.

Use-case: Program Verification

- We can also use the variable mappings to map assertions between different programs.

Use-case: Program Verification

- We can also use the variable mappings to map assertions between different programs.
- This way, we can automatically verify students' submissions, using CBMC [Clarke et al., 2004], based on similar previously submitted correct implementations for the same programming exercise.

Example: Program Verification

Variable Mapping: {n : l; i : j}.

A semantically correct student's implementation.

```
int main(){
    int n, i;
    scanf("%d", &n);
    for(i = 1; i <= n; i++){
        assert(1 <= i && i <= n);
        printf("%d\n", i);
    }
    return 0;
}
```

A semantically incorrect student's implementation since the variable j in the main function is not initialized.

```
void loop(int j, int l){
  while (l >= j){
    assert(1 <= j && j <= l);
    printf("%d\n", j);
    ++j;
  }
}
int main(){
  int j, l;
  scanf("%d", &l);
  loop(j, l);
  return 0;
}
```

Obrigado!

Děkuju!

Thank you!

References

Clarke, Edmund and Kroening, Daniel and Lerda, Flavio (2004)
A Tool for Checking ANSI-C Programs.
TACAS 04, 168–176.

Gulwani, Sumit and Radiček, Ivan and Zuleger, Florian (2018)
Automated clustering and program repair for introductory programming assignments.
PLDI 18 52(4), 465 – 480.

Ahmed, Umair Z and Fan, Zhiyu and Yi, Jooyong and Al-Bataineh, Omar I and Roychoudhury, Abhik (2022)
Verifix: Verified repair of programming assignments.
TOSEM 22 12(3), 45 – 678.

References

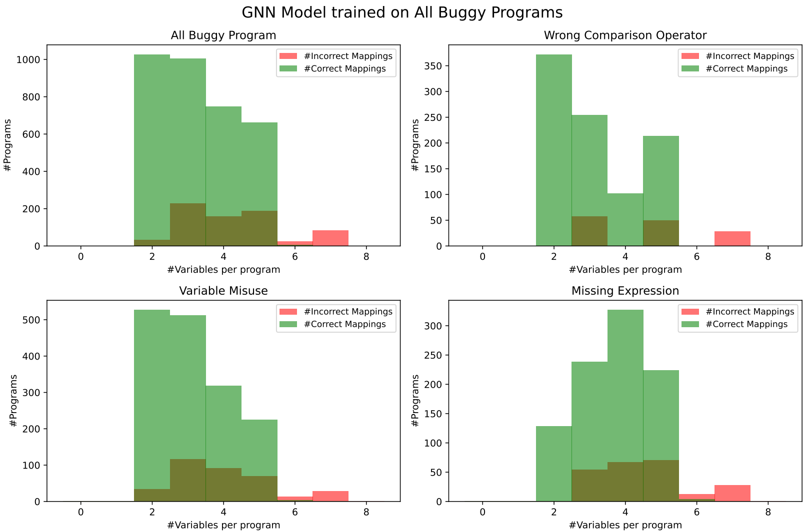
Orvalho, Pedro and Janota, Mikoláš and Manquinho, Vasco (2022)
C-Pack of IPAs: A C90 Program Benchmark of Introductory Programming Assignments.
arXiv:2206.08768.

Orvalho, Pedro and Piepenbrock, Jelle and Janota, Mikoláš and Manquinho, Vasco (2022)
Project Proposal: Learning Variable Mappings to Repair Programs.
AITP 2022.

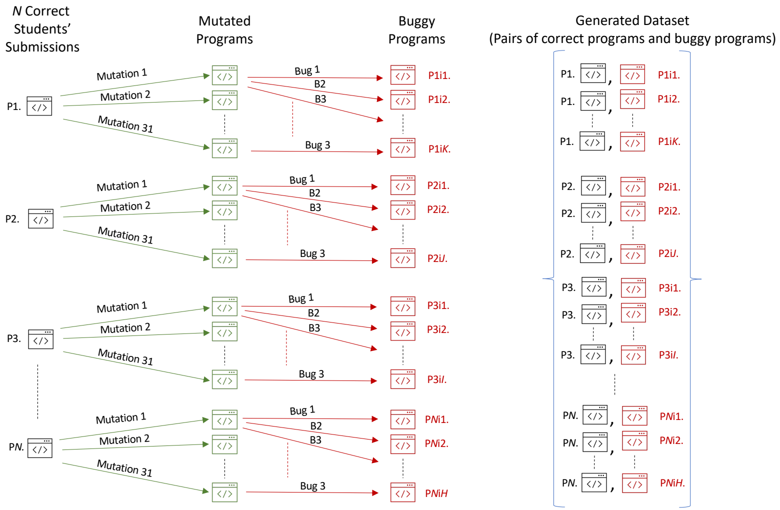
Orvalho, Pedro and Janota, Mikoláš and Manquinho, Vasco (2022)
MultIPAs: Applying Program Transformations to Introductory Programming Assignments for Data Augmentation.
ESEC/FSE 2022.

Appendix

#Correct/Incorrect Mappings vs #Variables



Dataset Generation using MultIPAs



Overlap coefficient

The overlap or Szymkiewicz–Simpson coefficient measures the overlap between two sets (e.g. mappings). This metric can be calculated by dividing the size of the intersection of two sets by the size of the smaller set, as follows:

$$overlap(A, B) = \frac{|A \cap B|}{min(|A|, |B|)} \tag{1}$$

An overlap of 100% means that both sets are equal or one of them is a subset of the other. The opposite, 0% overlap, means there is no intersection between both sets.