InfoTech-2024

2024 International Scientific Conférence on Information Technologies
Section: **Information Technologies**

Methods and Algorithms for Cross-Language Search of Source Code Fragments

Artyom V. Gorchakov, **Liliya A. Demidova**

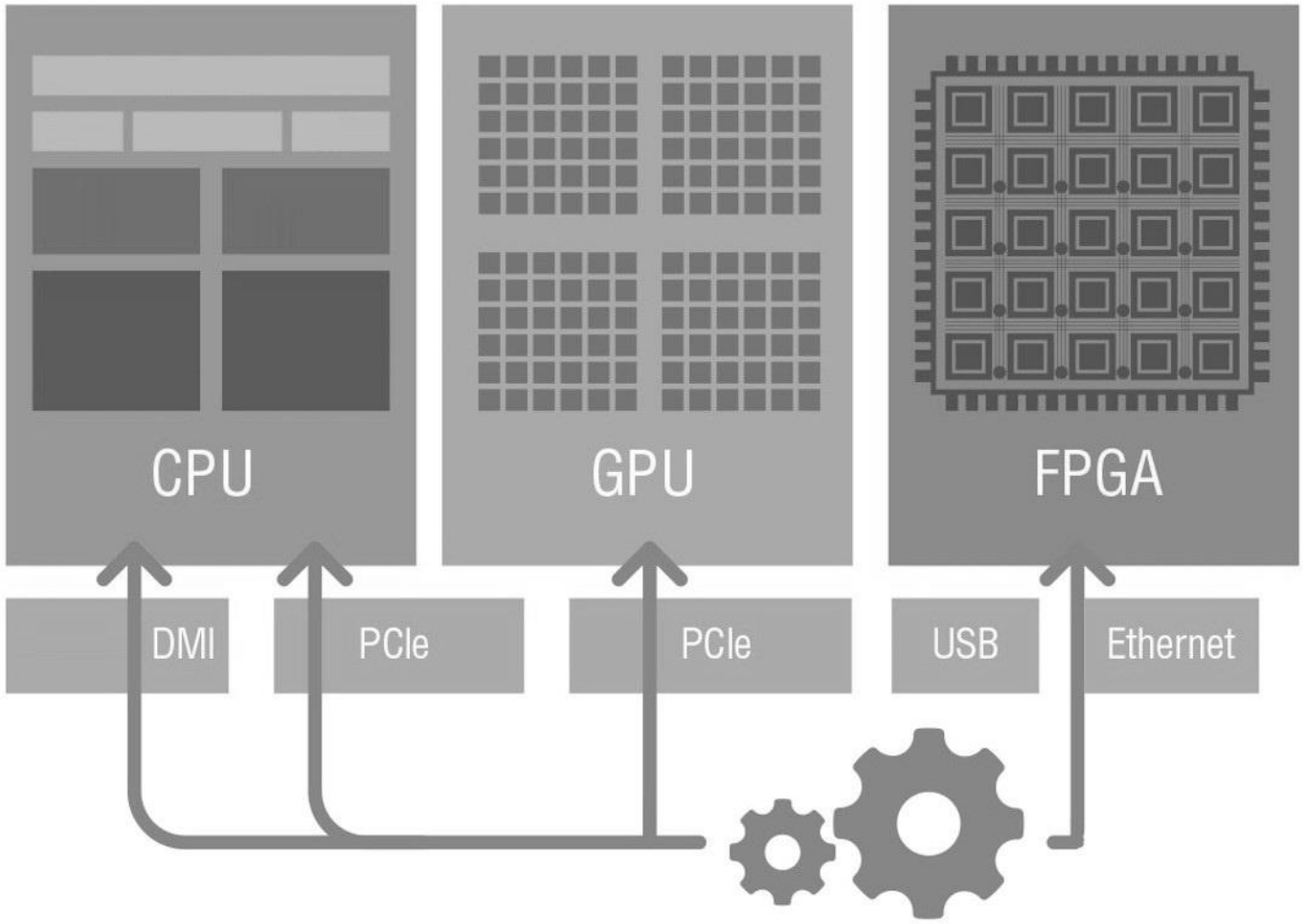
Corporate Information Systems Department, Institute of Information Technologies, Federal State Budget Educational Institution of Higher Education “MIREA—Russian Technological University”

Moscow, Russia

Relevance of the Study: Modern Software Systems for Heterogeneous Computing Platforms

Modern software systems are typically deployed on **heterogeneous computing platforms**, a heterogeneous computing platform consists of several computing units (CPU, GPU, FPGA, ASIC, …) that differ significantly in their characteristics or their types.

Components of modern software systems are typically implemented in **many different programming languages**, including: general-purpose languages such as Python, Java, C#, F#; system programming languages such as C, C++, and Rust; domain-specific languages (DSLs), such as structured query language (SQL) and embedded DSLs for high-level hardware synthesis.



Relevance of the Study: Static Analysis of Software Components

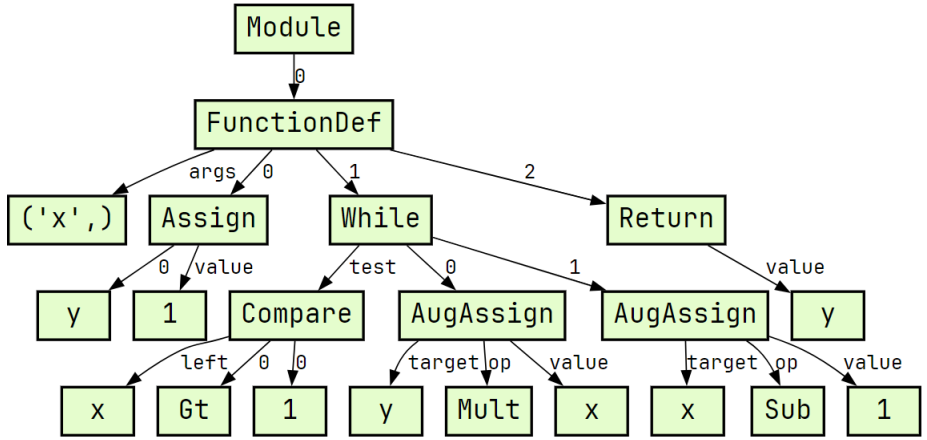
Static analyzers are widely used for scanning code for security issues, bugs, violations of programming standards, and other defects without executing the code.

Research in intelligent data analysis, machine learning, numerical modelling, and cryptography lead to the widespread incorporation of code fragments implementing **computationally intensive algorithms** into modern software systems.

Computationally expensive programs can be accelerated by moving their fragments to **most suitable coprocessors** among the computing units that are available on a given heterogeneous computing platform.

An example of a Python program and an Abstract Syntax Tree (AST)-based program representation which is commonly used for static code analysis:

```
def main(x):
    y = 1
    while x > 1:
        y *= x
        x -= 1
    return y
```



Most existing static source code analyzers are rule-based, designed for a given programming language. In this research we describe a **Unified Abstract Syntax Tree** (UAST) program model, and implement a decision support system which allows making recommendations for **code acceleration** on a given **heterogeneous computing platform** based on a code-to-code search algorithm and a database of code examples.

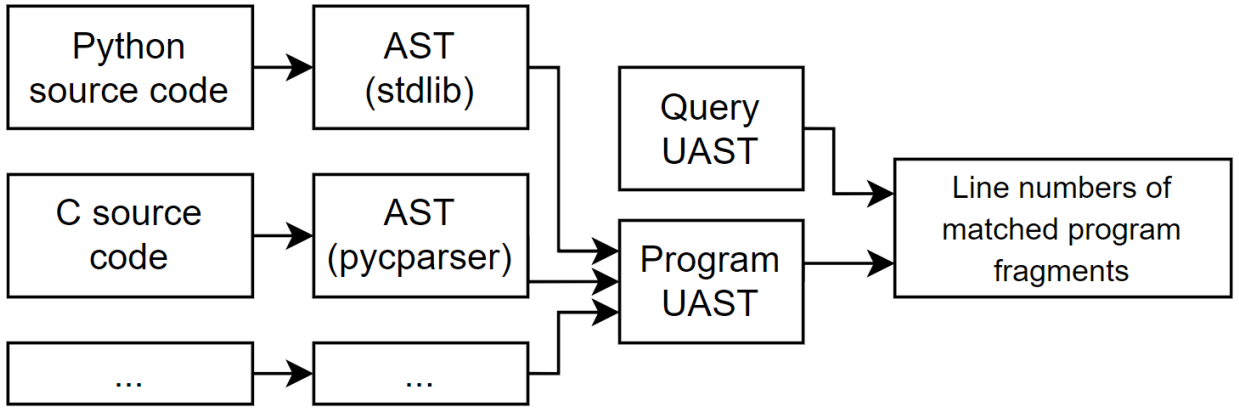
Overview of the Proposed Approach to Cross-Language Code Fragment Search: The General Approach

On the first step of the proposed approach the analyzed program is **transformed into UAST**.

The UAST can be built in different ways, for example, a language-dependent AST can be constructed, and then the AST can be **mapped to UAST**.

The supported syntactic structures in the proposed UAST model are **very limited**, it is expected that a language-specific AST is desugared before converting it into UAST.

If the converter encounters an AST node that can be **neither desugared nor mapped** into any of the known UAST nodes, then such AST node is ignored, in a way that is similar to **micro-grammars**.




```
1     <s> ::= " "
2    <loc> ::= <s> <number> <s> <number>
3    <bop> ::= "+" | "-" | "*" | "**" | "<" | ">" | "="
4   <body> ::= "(body" (<s> <stmt>)* <loc> ")"
5   <stmt> ::= "(func" <s> <string> <s> <params> <s> <body> <loc> ")"
6            | "(if" <s> <expr> <s> <body> <s> <body> <loc> ")"
7            | "(assign" <s> <expr> <s> <expr> <loc> ")"
8            | "(ret" <s> <expr> <loc> ")"
9            | "(loop" <s> <expr> <s> <body> <loc> ")"
10 <params> ::= "(params" ( <s> <expr> )* <loc> ")"
11   <args> ::= "(args" ( <s> <expr> )* <loc> ")"
12   <expr> ::= "(call" <s> <expr> <s> <args> <loc> ")"
13            | "(seq" ( <s> <expr> )* <loc> ")"
14            | "(get" <s> <expr> <s> <expr> <loc> ")"
15            | "(var" <s> <string> <loc> ")"
16            | "(const" <s> <string> <loc> ")"
17            | "(" <bop> <s> <expr> <s> <expr> <loc> ")"
```

Overview of the Proposed Approach to Cross-Language Code Fragment Search: UAST Example

(a)

```python
def factorial(n):
    fact = 1
    for i in range(n):
        fact = fact * (i + 1)
    return fact
```

(b)

```c
int factorial(int n) {
    unsigned long long fact = 1;
    for (int i = 0; i < n; i = i + 1) {
        fact = fact * (i + 1);
    }
    return fact;
}
```

(c)

```
(body
 (func factorial (params n 1 19) (body
  (assign (var fact 2 22) (const 1 2 29) 2 22)
  (assign (var i 3 12) (const 0 3 16) 3 12)
  (loop (< (var i 3 19) (var n 3 23) 3 19) (body
   (assign (var fact 4 5) (* (var fact 4 12) (+ (var i 4 20) (const 1 4 24) 4 20) 4 12) 4 5)
   (assign (var i 3 26) (+ (var i 3 30) (const 1 3 34) 3 30) 3 26) 3 3) 3 3)
  (ret (var fact 6 10) 6 3) 1 5) 1 5) 0 0)
```

In order to illustrate the proposed UAST construction process, the figure lists **2 programs computing factorial**, the programs are written in Python, denoted as (a), and in C, denoted as (b).

The UAST obtained from the C program is denoted as (c). The UAST obtained from the Python program is **mostly the same**, with a single exception for line and column numbers.

Algorithm 1 — Code-to-code search in a UAST.

Input: G_t ▷ UAST of the analyzed program.
G_q ▷ UAST of the search query.
d ▷ Distance function.
ϑ ▷ Number of statements in extracted tuples.
κ ▷ Number of most relevant fragments to return.

1. $R = \emptyset$.
2. **For each** body $v \in G_t$ **do**:
3. **Define** ξ_v ▷ Statements connected to v.
4. **For each** tuple $(v_1, \dots, v_\vartheta) \in \binom{\xi_v}{\vartheta}$ **do**:
5. **Build** UAST G containing $(v_1, \dots, v_\vartheta)$.
6. $\rho \leftarrow d(G, G_q)$. ▷ Compute the distance.
7. $R \leftarrow R \cup \{(\rho, G)\}$.
8. **Loop end**.
9. **Loop end**.
10. **Return** κ pairs with best ρ, belonging to R.

The UAST-based code models are scanned according to Algorithm 1, which **extracts fragments** from the analyzed UAST and compares the fragments to the UAST of the **query program**, returning top κ most similar fragments treated can be treated as **recommendations for acceleration**.

Overview of the Proposed Approach to Cross-Language Code Fragment Search: Program Representations

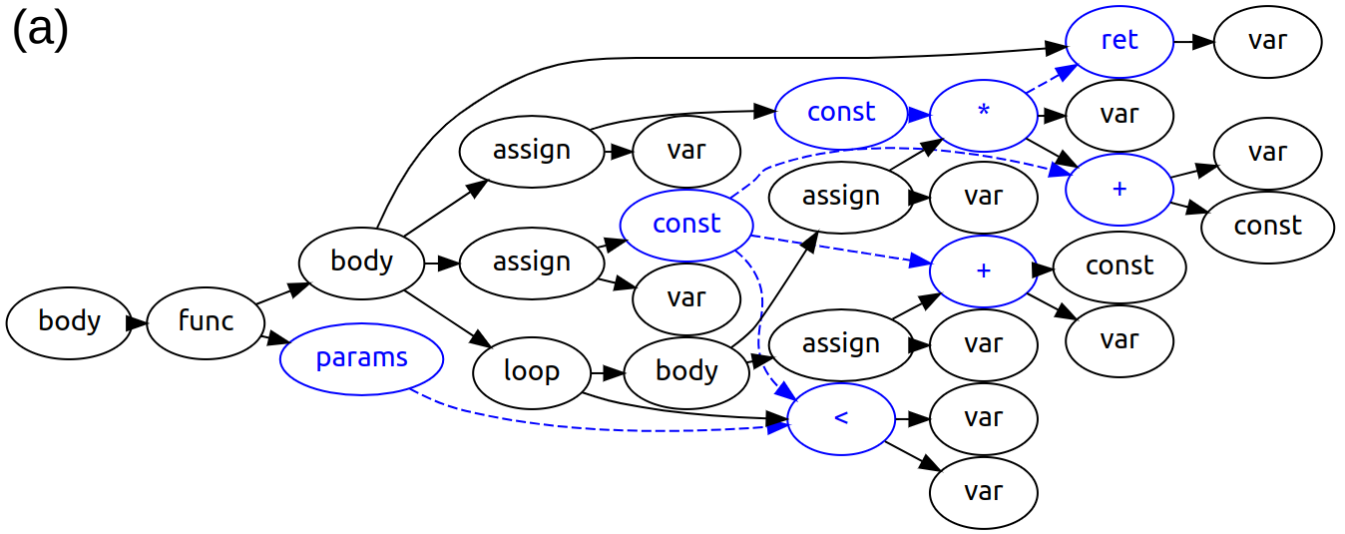
In this research, we use the Markov chain-based and Jensen-Shannon Divergence-based **distance function**.

The UASTs of the compared programs are augmented with edges from the "**definition-use**" graphs connecting UAST nodes that define variables with nodes that use the defined variables.

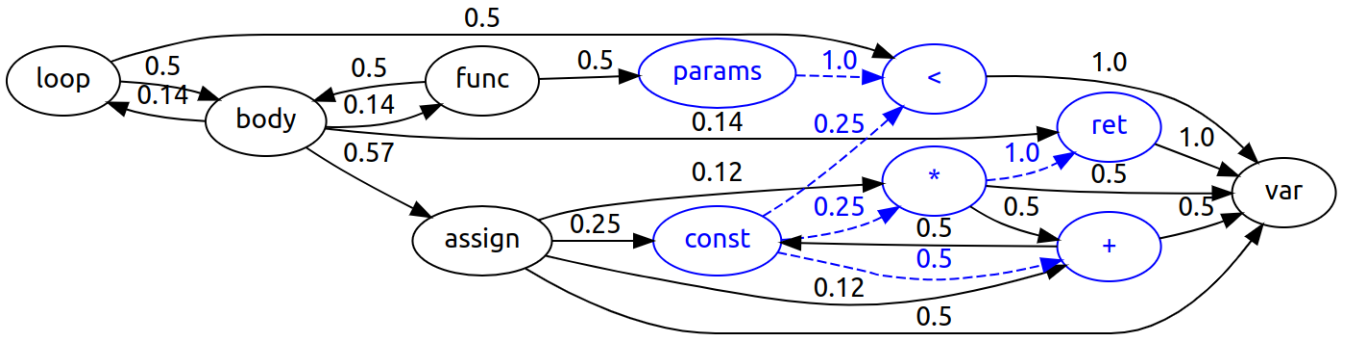
The **UAST code representation** augmented with DU-graph edges is denoted as (a). The **Markov chain** graph for the UAST is denoted as (b).

Weighted adjacency matrices of the Markov chains are converted into **vectors** by concatenating the rows.

(a)



(b)



Overview of the Proposed Approach to Cross-Language Code Fragment Search: Distance Function

The vector-based representations of 2 programs are then **compared to each other** according to:

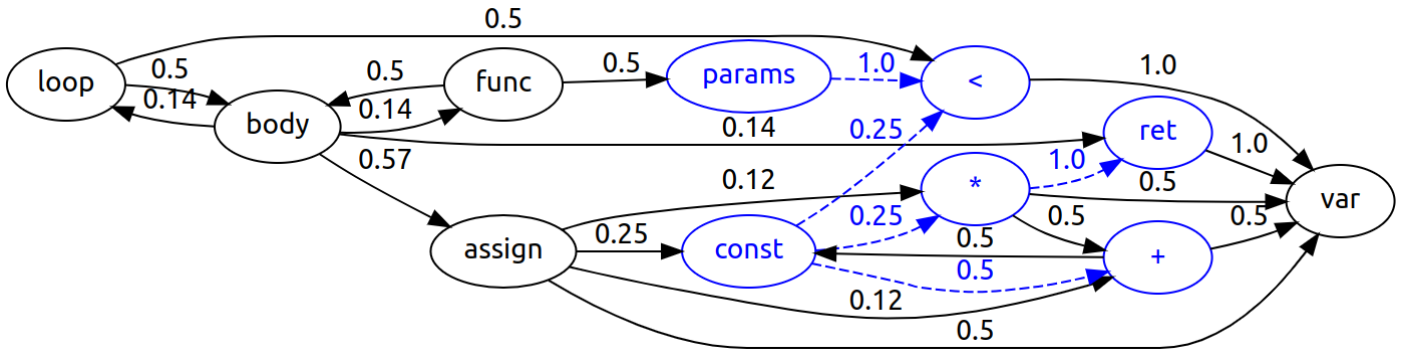
$$\rho = \omega_1\rho_1 + \omega_2\rho_2 + \omega_3\rho_3,$$

where

$$\rho_1 = \mathrm{JSD}\left(\vec{v}_1^{\mathrm{UAST}}, \vec{v}_2^{\mathrm{UAST}}\right),$$

$$\rho_2 = \mathrm{JSD}\left(\vec{v}_1^{\mathrm{DU}}, \vec{v}_2^{\mathrm{DU}}\right),$$

$$\rho_3 = 1 - \frac{\min(c_1, c_2)}{\max(c_1, c_2)},$$



$$\mathrm{JSD}(\vec{v}, \vec{v}') = \sum_{i=1}^{m} \frac{v_i}{2} \log_2 \frac{2v_i}{v_i + v_i'} + \sum_{i=1}^{m} \frac{v_i'}{2} \log_2 \frac{2v_i'}{v_i + v_i'},$$

$\omega_1, \omega_2, \omega_3$ are weights;
JSD denotes the **Jensen-Shannon divergence** for **program vectors** $\vec{v}$ and $\vec{v}'$;
m is the component count in program vectors $\vec{v}$ and $\vec{v}'$;
$\vec{v}_1^{\mathrm{UAST}}$ and $\vec{v}_2^{\mathrm{UAST}}$ denote Markov chain-based code vectors based built for UASTs (**black edges**);
$\vec{v}_1^{\mathrm{DU}}$ and $\vec{v}_2^{\mathrm{DU}}$ denote Markov chain-based code vectors built for DU-graphs (**blue edges**);
c_1 and c_2 denote **Cyclomatic Complexity** values of the programs.

Experiment: Comparison of Markov Chain-Based Distance with the Well-Known Tree Edit Distance

We compared the Markov chain-based and Jensen-Shannon Divergence-based **distance function** with **tree edit distance**, which is commonly used to determine how the 2 given trees differ from each other.

The **automatically obtained search results** were compared to the code fragments that were **found in code by an expert** using Precision and Recall metrics:

$$\text{Precision} = \frac{\text{TP}}{k} = \frac{\text{TP}}{\text{TP+FP}},$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP+FN}}.$$

The F_1-score of Precision and Recall was also computed. The obtained results are shown in Table 1.

TABLE 1. CODE-TO-CODE SEARCH QUALITY

Query	Metric	Tree Edit Distance	Eq. (1)
Q_1	Precision	0.50	**0.75**
	Recall	0.66	**1.00**
	F_1 SCORE	0.57	**0.86**
Q_2	Precision	0.50	0.50
	Recall	1.00	1.00
	F_1 SCORE	0.60	0.60
Q_3	Precision	0.00	**0.75**
	Recall	0.00	**1.00**
	F_1 SCORE	0.00	**0.86**

The improved quality of the results obtained with the Markov chain-based distance function are, on the one hand, justified by the fact, that this distance function takes **data dependencies** into account. On the other hand, Markov chain-based **code embeddings** are known to outperform word2vec, code2vec and histograms in **algorithm classification**.

Experiment: Recommendations for Software Acceleration

Using the described UAST-based approach to **cross-language code search**, we developed a plugin for the Visual Studio Code® IDE.

The results of cross-language code-to-code search based on the **database of algorithm implementations** are provided.

The database also contained **acceleration coefficients** for SIMD CPU and SIMD GPU-based code variants. The coefficients were obtained by **benchmarking algorithms from the database** on a heterogeneous computing platform that we used in our experiment.

Future research could cover additional comparison of different distance metrics using such metrics as performance and memory usage. Future research could also cover control flow graph-based code models.

```
1  double* train(double alpha, double** x, double* y, int n, int m, int iterations) {
2    int size = sizeof(double) * n;
3    int psize = sizeof(double*) * n;
       🚀 Preferred SIMD accelerator is CPU (1.41 times faster than GPU)
4    double** transpose = malloc(psize);
5    int rsize = sizeof(double) * m;
6    for (int i = 0; i < m; i = i + 1) {
7      transpose[i] = malloc(rsize);
8      for (int j = 0; j < n; j = j + 1) {
9        transpose[i][j] = x[j][i];
10     }
11   }
12   for (int iter = 0; iter < iterations; iter = iter + 1) {
         🚀 Preferred SIMD accelerator is GPU (25.13 times faster than CPU)
13     double* outputs = malloc(size);
14     for (int j = 0; j < n; j = j + 1) {
           🚀 Preferred SIMD accelerator is GPU (2.31 times faster than CPU)
15       double output = 0;
16       for (int i = 0; i < m; i = i + 1) {
17         output = output + x[j][i] * theta[i];
18       }
19       outputs[j] = output;
20     }
21     for (int i = 0; i < n; i = i + 1) {
22       outputs[i] = 1 / (1 + exp(1 - outputs[i]));
23     }
```

An example of C code fragment search results.

```
1  def predict(theta, x):
2    outputs = list()
3    for j in range(len(x)):
       🚀 Preferred SIMD accelerator is GPU (2.31 times faster than CPU)
4      output = 0
5      for i in range(len(x[j])):
6        output = output + x[j][i] * theta[i]
7      outputs.append(output)
     🚀 Preferred SIMD accelerator is GPU (25.13 times faster than CPU)
8    activations = list()
9    for i in range(len(outputs)):
10     activations.append(1 / (1 + math.exp(1 - outputs[i])))
11   return activations
```

An example of Python code fragment search results.