Algorithmization of Objects in EO Programs

Anonymous Author(s)

Abstract

On one hand, there are pure object-oriented programming languages, such as Ruby or Self, which obey the principle “everything is an object” (West, 2004, p.121). In such languages even primitive data are objects, such as integers or Boolean values. On the other hand, there are also “hybrid” programming languages, such as C++ or Java. They distinguish objects and primitive data types such as numbers or arrays. Obviously, hybrid languages may demonstrate higher performance due to elimination of object allocation and disposal overhead, especially in data-intensive algorithms.

There is an opportunity to boost performance of programs written in a pure object-oriented language, if the language has a foreign function interface (FFI) to a hybrid language. For example, before execution, parts of a Ruby program that most actively manipulate with data, may automatically be re-written in C and then compiled into binaries. At runtime, the binaries will be called from Ruby via FFI. They will receive data from Ruby, process it, and communicate back to Ruby by a) returning new data and b) continuously exchanging data via some protocol. In Section 2 we demonstrate by example how this may work in Ruby.

Keywords: Control flow graph, Data flow diagram, Devirtualization, Foreign function interfaces, Object-Oriented Programming, Static analysis

1 Introduction

An optimization technique in which fragments of source code are replaced with inserts of code in the FFI language is called “algorithmization”. Algorithmization is present to some extent in some languages and their compilers. For example, Java has unboxing mechanism which at compile time turns objects into data primitives. In Section 3 we analyze existing solutions and evaluate their advantages and flaws.

In this paper we suggest a new method of algorithmization of pure object-oriented programs. We use EO programming language for implementation of the method and evaluation of its effect. In Section 4 we use φ-calculus, which is the foundation of EO, to explain the method. It Section 4 we also explain how our method may be applied to other programming languages. The contribution we make is two fold: 1) the method and 2) the software tool for algorithmization of EO programs.

2 Background

Consider this Ruby program:

```
class F
  def eof; ...; end # TRUE if end of file
  def next; ...; end # Reads next line
end
f = F.new
a = 0
loop do
  t = f.next.to_i
  next if t % 3 != 0
  a += t * t
  break if f.eof
end
puts "a = #{a}"
```

It reads lines from a text file. Each line presumably contains a textual form of an integer. If the result of the division of the integer by three is not equal to zero, the algorithm moves to the next line in the file. Otherwise, the number is multiplied by itself and then added to the accumulator. At the end of the file the algorithm stops and prints the value of the accumulator.

The simplest algorithmization of the code may look like the following (everything outside of the `loop` is skipped for the sake of brevity):

```
loop do
  t = f.next.to_i
  next if t % 3 != 0
  a = ffi("return x + y * y;")
    .with("x", a)
    .with("y", t)
    .exec()
  break if f.eof
end
```

Here, the builder function `ffi` takes a string with a simple C function. It must be compiled to a binary code before Ruby script is executed (this feature must be embedded into Ruby interpreter). At runtime, `x` and `y`

are replaced with the values of Ruby variables `a` and `b` and then the C function is evaluated.

This optimization moves two arithmetic operations to a lower-level language: multiplication and addition. They definitely are faster in C, taking into account that in Ruby they are not operators but methods of class `Integer`. However, the cost of calling `ffi` may be larger than the effect of optimization. A more complex algorithmization with a wider scope may give bigger effect:

```
a = ffi("""
  #include <stdlib.h>
  int t;
  int a = 0;
  while (true) {
    t = atoi(f.next());
    if (t % 3 != 0) {
      next;
    }
    a += t * t;
    if (f.eof()) {
      break;
    }
  }
  return a;
""").with("f", f).eval()
```

Here, the entire `loop` cycle is inside the C function. A reverse access to Ruby objects is implemented in `f` parameter passed into it. The cost of such a reverse call to Ruby may be higher than the effect of optimization, which is gained due to `atoi` C function instead or Ruby method `String.to_i`. Keeping a positive balance between effect and cost is what is expected from algorithmization method.

3 Related Work

The most similar algorithm to the one being developed is the packaging and unpacking mechanism implemented in Java. In Java, there are special wrapper classes for primitive data types such as *int* or *double*. It is possible to convert a primitive data type into a wrapper class object and vice (these processes are called packing and unpacking). In some cases, Java can optimize the code, convert wrapper classes into primitive data types and perform calculations in them to reduce the expected execution time of the program (the process is called autoboxing). Despite the similarity with the developed algorithm at first glance, this mechanism is very different from the developed algorithm. With autoboxing, data is simply transformed, while fragments of the program code are not replaced with fragments of code in another language and the source code of the program is almost not affected. The performance gain thus obtained during the operation of the program remains quite small.

4 Method

EO[1] is an object-oriented programming language based on φ-calculus introduced by Bugayenko (2021) and later formalized by Kudasov et al. (2022). EO is a "more" pure language than Ruby, meaning that it has more objects and less things that are not objects. For example, `to_i` method of `String` class in Ruby is not an object. It is a function implemented in C. EO also has atoms, which essentially are C or Assembly blocks of code, but the percentage of them in the entire standard library is much smaller than of similar entities in Ruby.

The file-reading program discussed above may be written in EO as such:

```
[] > file
  [] > eof /bool
  [] > next /string
file > f
memory 0 > a
goto
  [g]
    seq > @
      at.
        QQ.txt.sscanf
          "%d"
          f.next
        0
      if.
        (t.mod 3).eq 0
        a.write
          a.plus
            t.mul t
        g.backward
      if.
        f.eof
        g.forward a
        g.backward
QQ.io.stdout
  QQ.txt.sprintf
    "a = %d"
    a
```

[1] https://www.eolang.org

Our algorithm replaces EO objects with Rust functions. This code may be optimized to the following:

```
goto
  [g]
    rust
      """
      Here goes the code for Rust function:
      pub fn f(&mut uni: Universe, v: u32);
      """
```

Here, the code of Rust function body may look like this:

```
let t = uni.da("Φ.f.next.Δ").to_utf8()?;
if t % 3 == 0 {
  let mut a = uni.da("Φ.a.Δ").to_i64()?;
  a = a + t;
  let write = u.copy("Φ.a.write");
  let a0 = u.add();
  uni.bind(write, a0, "α0");
  uni.put(a0, Hex::from_i64(a));
  uni.da(format!("v{write}"));
} else {
  uni.da(format!("v{v}.α0.backward"));
}
let eof = uni.da("Φ.f.eof.Δ").to_bool()?;
if eof {
  let f = u.copy("v{v}.α0.forward");
  uni.bind(f, 0, "α0/Φ.a");
  uni.da(format!("v{f}"));
} else {
  uni.da(format!("v{v}.α0.backward"));
}
```

Rust code blocks encapsulated inside `rust` object communicate with other EO objects through Surging Object DiGraph (SODG) notation.

5 Problem statement and formalization

This section formalizes the task of selecting code fragments that must be converted to the FFI language. Section 5.1 describes the necessary definitions from graph theory. Sections 5.2-5.3 describe the control flow graph. Sections 5.4-5.5 describe the objective function - the developed algorithm will have to minimize it. Section 5.6 describes the definition of a valid code fragment - a part of the code that can be converted to the FFI language. Section 5.7 describes how to reduce the task of minimizing the objective function described in sections 5.4-5.5 to the maximal clique task.

5.1 Graph definitions

In this subsection, it is described definitions in graph theory, which will be needed for algorithmization task.

Definition 5.1 (Graph). A *graph* $G \in \mathbb{G}$ is a pair (V_G, E_G), where V_g is a set whose elements are called *vertices*, and E_g is a set of pairs of vertices, whose elements are called *edges*.

Definition 5.2 (Cycle). A *cycle* in a graph G is a sequence of vertices $v_0, v_1, \ldots, v_n$ where $v_i \in V_G$, and $v_0 = v_n$, and $n \geq 2$, and for any v_i where $0 > i < n$ exists an edge $e \in E_G$ equal to (v_{i-1}, v_i).

Example 5.3. Consider $V_G = \{v_1, v_2, v_3, v_4, v_5, v_6\}, E_G = \{(v_1, v_2), (v_1, v_3), (v_3, v_5), (v_4, v_2), (v_5, v_6), (v_6, v_1)\}$. Then sequence v_1, v_3, v_5, v_6, v_1 is cycle. This graph can be visualized.

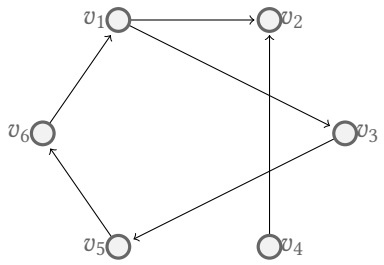

Image 1. Example of cycle in graph

Definition 5.4 (Simple path). Sequence $v_0, v_1, \ldots, v_n$, v_i where $v_i \in V_G$, and $n >= 1$ is *simple path*, if $(v_i, v_{i+1}) \in E_G$ for $i \geq 0 \wedge i < n$ and $v_i \neq v_j$ for $i \neq j$ and $0 \leq i \leq n$ and $0 \leq j \leq n$.

Definition 5.5 (The reachability predicate $\mathrm{P}_G(v_0, v_n)$). For $G \in \mathbb{G}$ denote $P_G(v_0, v_n)$, which equal true, if $\exists v_1, v_2, \ldots, v_{n-1} \in V_G$: sequence $v_0, v_1, \ldots, v_n$ is simple path and $(v_0, v_1) \in E_G$ and $(v_{n-1}, v_n) \in E_G$, or if $v_0 = v_n$.

Definition 5.6 ($\mathrm{CH}_G(v)$). For $v \in V_G$ denote $\mathrm{CH}_G(v) = \{x | x \in V_G \wedge (v, x) \in E_G\}$.

Definition 5.7 (Source). Vertex $v \in V_G$ is *source*, if $\nexists x \in V : (x, v) \in E_G$.

Definition 5.8 (Sink). Vertex $v \in V$ is *sink*, if $\nexists x \in V : (v, x) \in E_G$.

Definition 5.9 (Ancestor). For $r, v \in V_G$ denote $\mathrm{AN}_{G,r}(v) = \{x | x \in V_G \wedge \mathrm{P}_G(r, v) \wedge \mathrm{P}_G(v, x)\}$.

Definition 5.10 (Descendants). For vertex $v \in V_G$ $\mathrm{DE}_G(v) = \{x | x \in V_G \wedge \mathrm{P}_G(v, x)\}$.

Definition 5.11 (Back-edge). Edge $e = (v_a, v_b) \in E_G$ is *back-edge* if $\exists x \in \text{DE}_G(v_b) : (x, v_a) \in E_G$.

Definition 5.12 (Predicate of back-edge $\text{BE}_G(e)$). Denote $\text{BE}_G(e) : e = (v_a, v_b) \wedge v_a, v_b \in V_G$, equal true if e is back-edge.

5.2 CFG and task of algorithmization

Definition 5.13. *Programming object (PO)* is a pair (id, weight), where id is number and weight is number.

Definition 5.14. *Control flow graph (CFG) Control flow graph* c is $(G, \text{OBJ}, \text{RO})$, where G is graph and OBJ is set of PO and $\text{RO} \in V_G$ and RO is source.

Definition 5.15 (Laboriousness). For $G = G_c$, c is CFG, denote $\text{LA}_c(v, l)$ is number, where $v \in V_G$, $l \in \{\mathcal{A}, \mathcal{B}\}$.

Definition 5.16 (objects to read in vertex). For $v \in V_G$, $G = G_c$, c is CFG, denote $\text{VR}_c(v)$ - set of PO.

Definition 5.17 (objects to write in vertex). For $v \in V_G$, $G = G_c$, c is CFG, denote $\text{VW}_c(v)$ - set of PO.

Definition 5.18 (colored control flow graph). Denote *colored control flow graph (CCFG)*. This is a common CFG, but each vertex $v \in V_G, G = G_c, c$ is CFG, additionally has a CO attribute. $\text{CO}_c(v) \in \{\mathcal{A}, \mathcal{B}\}$.

Definition 5.19 (set of colored control flow graphs). For $G = G_c$, c is CFG, denote set of colored control flow graphs CCFG(c). All entities of CCFG(c) differing only by color of vertices $v \in V_G, G = G_c, c \in \text{CCFG}(G_x), x$ is CFG.

Theorem 5.20 (Size of CCFG(c)). *For given control flow graph* $G = G_c, c$ *is CFG:*

$$|CCFG(c)| = 2^{|V_G|}$$

Proof. Denote $u \subseteq V_G, \forall u : \text{CO}_c(u) = \mathcal{A}$. Then all vertices of $\overline{u}$ has color $\mathcal{B}$. Then the number of ways to choose u is equal $|\text{CCFG}(c)|$. $0 \le |u| \le |V_G|$, so find sum of ways to choose u by $|u|$, it's $\sum_{i=0}^{|V_G|} \binom{|V_G|}{i} = 2^{|V_G|}$ (according to Newton's binomial). □

Definition 5.21 (Root color). For any x is CCFG: $\text{CO}_x(\text{RO}_x) = \mathcal{A}$

Definition 5.22 (Sink color). For any c is CCFG, $G = G_c, V = V_G$: $(|\text{CH}_G(v)| = 0) \implies (\text{CO}_c(v) = \mathcal{A})$.

Definition 5.23 (Component). For vertex $v \in V_G$, $G = G_c$, c is CCFG, *component* $\text{COM}_c(v) = \{y \mid \text{CO}_c(v) = \text{CO}_c(y) \wedge (\exists x_1, \dots x_n : ((v, x_1) \in E_G \wedge (x_1, x_2) \in E_G \wedge \dots (x_{n-1}, x_n) \in E_G \wedge (x_n, y) \in E_G) \wedge (\text{CO}_c(x_1) = \text{CO}_c(v) \wedge \dots \wedge \text{CO}_c(x_n) = \text{CO}_c(v)))\}$.

Definition 5.24 (component estimator). For $G = G_c$, c is CCFG, q is $\text{CE}_c(v)$ if $(v \in \text{COM}_c(q)) \wedge (\nexists e \neq q : \text{CO}_c(e) = \text{CO}_c(q) \wedge q \in \text{COM}_c(e))$

Definition 5.25 (Independent component). For specified c is CCFG and $v \in V_G, G = G_c$, $\text{IC}_c(v) = \{\text{COM}_c(v) | v \in V_G : \nexists y \in V_G : v \neq y \wedge \text{COM}_c(v) \subseteq \text{COM}_c(y)\}$

5.3 Relationship CCFG with real programs

As mentioned earlier, CFG is $(G, \text{OBJ}, \text{RO})$. But also CFG is a representation of all possible ways to execute a program in the form of a graph. For any program in any programming language, it can be built control flow graph.

Example 5.26. Let's look at such a simple C++ program which sorts array and at control flow graph (CFG) for this program.

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
    int n;
    cin >> n;
    vector<int> a(n);

    for (int i = 0; i < n; ++i) {
        cin >> a[i];
    }

    for (int i = 0; i < n - 1; ++i) {
        for (int j = 0; j < n - i - 1; ++j) {
            if (a[j] > a[j + 1]) {
                int x = a[j];
                a[j] = a[j + 1];
                a[j + 1] = x;
            }
        }
    }

    for (int i = 0; i < n; ++i) {
```

```
        cout << a[i] << ' ';
    }

    return 0;
}
```

Code Listing 1. Simple c++ program.

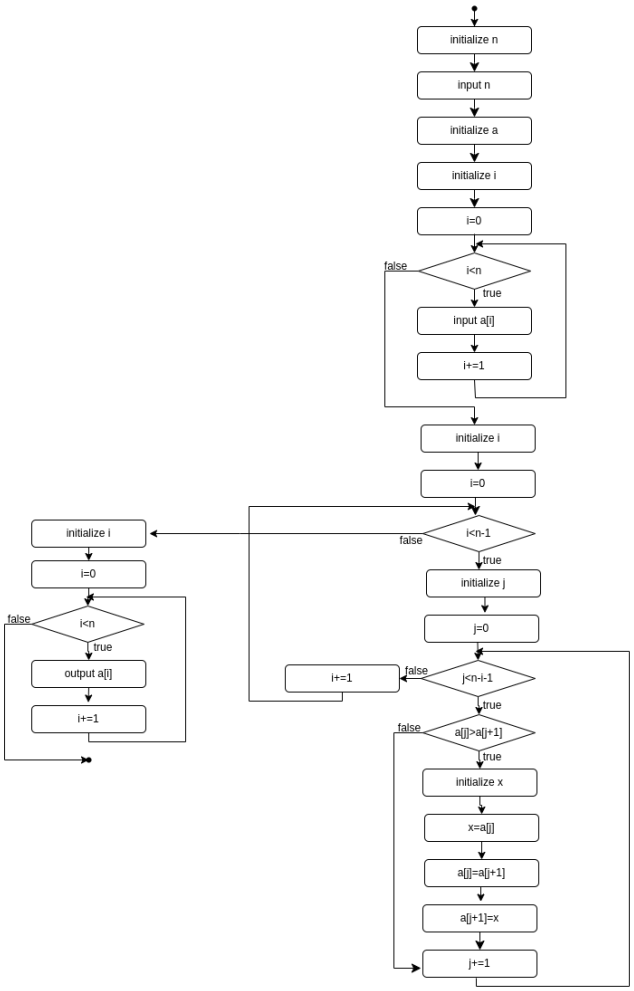

Image 2. The control flow graph (CFG) for program from code listing 1.

Programming object (PO) **[definition 5.13]** is a representation of object used in the program.

LA is expected time labor costs of programming language command or sequence of commands.

Each vertex corresponds to the sequence of operations. For each vertex $v \in V_G, G = G_c, c$ is CCFG.

- $\text{VR}_c(v)$ is set of all PO, which are used in commands that correspond to the vertex v and are not modified by these commands;
- $\text{VW}_c(v)$ is set of all PO, which are used in commands that correspond to the vertex v and are modified by these commands;
- $\text{LA}_c(v, \mathcal{A})$ is expected laboriousness of all commands, which corresponds v if these commands will be executed on EO;
- $\text{LA}_c(v, \mathcal{B})$ is expected laboriousness of all commands, which corresponds v if these commands will be executed on Rust.
- $\text{CO}_c(v) = \mathcal{A}$ if sequence of commands, which corresponds vertex v, should be executed on EO, and $co_c(v) = \mathcal{B}$ if these commands should be executed on Rust.

For $c = (G, \text{OBJ}, \text{RO})$, c is CCFG, RO_c is entry point of program. Sink vertices **definition 5.8** are places where program execution finishes.

Each c is CCFG can be associated with a program written in language $\mathcal{A}$, in which all code parts corresponding to the vertices of G_c colored $\mathcal{B}$ are replaced by code inserts in language $\mathcal{B}$ with using an foreign functional interface of language $\mathcal{A}$.

The purpose of the algorithm described in this article is to select for specified CFG G from the CCFG(G) one colored control flow graph, which is the most optimal by some criterion. A description of this criterion is given in the following sections.

5.4 Data transfering

Data transfering function assumes that the laboriousness of transporting data are proportional to the volume of data being transported. It is argued that the laboriousness of transporting data can be described by a function of the form $\text{LA} = \alpha + \beta x$, where x is the total amount of memory occupied by the transported data.

Definition 5.27 (Imported objects)**.** For $G = G_c$, c is CCFG, $e = (v, x) \in E_G$

$$\text{IOB}_c(v, x) = \Big(\bigcup_{a \in \text{AN}_{G,\text{RO}_c}(v)} (\text{VR}_c(a) \cup \text{VW}_c(a)) \Big) \cap \Big(\bigcup_{a \in \text{COM}_c(x)} (\text{VR}_c(a) \cup \text{VW}_c(a)) \Big)$$

This function is defined only for the case when $\text{CO}_c(v) = \mathcal{A}$ and $\text{CO}_c(x) = \mathcal{B}$.

Definition 5.28 (Exported objects)**.** For $G = G_c, c$ is CCFG, $e = (v, x) \in E_G$

$$\text{EOB}_c(v,x) = \left(\bigcup_{a \in \text{COM}_c(\text{CE}_c(v))} \text{VW}_c(a)\right) \cap \left(\bigcup_{a \in \text{DE}_G(x)} (\text{VR}_c(a) \cup \text{VW}_c(a))\right)$$

This function is defined only for the case when $\text{CO}_c(v) = \mathcal{B}$ and $\text{CO}_c(x) = \mathcal{A}$.

Definition 5.29 (transporting cost). For $G = G_c, c$ is CCFG denote $\text{TC}_c(\alpha, \beta, v, c) \rightarrow \mathcal{R}, \alpha \in \mathcal{R}, \beta \in \mathcal{R}, v \in V_G, c \in \text{CH}_G(v)$.

Definition 5.30 (transporting cost). For $x \in \text{CH}_G(v), v \in V_G, G = G_c, c$ is CCFG if $\text{CO}_c(v) = \text{CO}_c(x)$ then $\text{TC}_c(\alpha, \beta, v, x) = 0$.

Definition 5.31 (transporting cost). For $x \in \text{CH}_G(v), v \in V_G, G = G_c, c$ is CCFG if $\text{CO}_c(v) = \mathcal{A} \wedge \text{CO}_c(x) = \mathcal{B}$ then

$$\text{TC}_c(\alpha, \beta, v, x) = \alpha + \beta \sum_{o \in \text{IOB}_c(v,x)} \text{WEIGHT}_o$$

Definition 5.32 (transporting cost). For $x \in \text{CH}_G(v), v \in V_G, G = G_c, c$ is CCFG if $\text{CO}_c(v) = \mathcal{B} \wedge co_c(x) = \mathcal{A}$ then

$$\text{TC}_c(\alpha, \beta, v, x) = \alpha + \beta \sum_{o \in \text{EOB}_c(v,x)} \text{WEIGHT}_o$$

5.5 Target function

In this subsection, we will describe the target function that the algorithm described in this article should minimize.

Definition 5.33 (H function). For c is CCFG, Denote $H_c(g, v) : (\text{CCFG}(G), V_G) \rightarrow \mathcal{R}$.

Definition 5.34 (Hyper-parameters α, β, γ). Denote hyper parameters $\alpha \in \mathcal{R}, \beta \in \mathcal{R}, \gamma \in \mathcal{R}$.

Definition 5.35 (H function). For any sink vertex $v \in V_G, G = G_c, c$ is CCFG $H_c(g, v) = \text{LA}_C(v, \text{CO}_c(v))$.

Definition 5.36 (H function). For any non-sink vertex $v \in V_G, G = G_c, c$ is CCFG:

$$\text{H}_c(G, v) = \text{LA}_c(v, \text{CO}_c(v)) + \frac{1}{|\text{CH}_G(v)|} * \left(\sum_{q \in \text{CH}_G(v) \wedge \text{BE}_G(v,q)} (H_c(G, q) + \text{TC}_c(\alpha, \beta, v, q)) + \gamma \sum_{q \in \text{CH}_G(v) \wedge \text{BE}_G(v,q)} \text{TC}_c(\alpha, \beta, v, q)\right)$$

Definition 5.37 (L function). $G = G_c, c$ is CFG Denote $G : \text{CCFG}(c) \rightarrow \mathcal{R}$. $L(G) = H_c(G, \text{RO}_c)$.

Definition 5.38 (Ideal). $G = G_c, c$ is CFG $g \in \text{CCFG}(c)$ *ideal* if $\nexists q \in \text{CCFG}(c) : q \neq g \wedge L(q) < L(g)$.

5.6 Restrictions imposed on code fragments transformed into $\mathcal{B}$

For $E = E_G, V = V_G, G = G_c, c$ is CFG, consider $\text{S} \in V$, $\text{F} \in \text{DE}_G(a)$.

Definition 5.39 (Fragment). For $\text{S} \in V, \text{F} \in \text{DE}_G(\text{S})$, $\text{FR}_G(\text{S}, \text{F}) = \{v \mid v \in V \wedge \exists x \in V_G : (\text{P}_G(\text{S}, x) \wedge \text{P}_G(x, \text{F}))\}$. Each fragment has unique id (for fragment F denote as UID_F).

Definition 5.40 (Entry-point and Exit-point). For $\text{FR}_G(\text{S}, \text{F})$ denote *entry-point* = S and *exit-point* = F.

Definition 5.41 (Fout). For $f = \text{FR}_G(\text{S}, \text{F}), \text{FOUT}_G(f) = \bigcup_{v \in \text{FR}_G(\text{S,F})} \{x \mid x \in \text{CH}_G(v) \wedge x \notin \text{FR}_G(\text{S}, \text{F}) \wedge |\text{CH}_G(x)| \neq 0\}$.

Definition 5.42 (Fin). for $f = \text{FR}_G(S, F), \text{FIN}_G(f) = \bigcup_{v \in \text{FR}_G(\text{S,F})} \{x \mid x \in V \wedge x \notin \text{FR}_G(\text{S}, \text{F}) \wedge (x, v) \in E\}$.

Definition 5.43 (Correct Fragment). Fragment $f = \text{FR}_G(\text{S}, \text{F})$ is *correct* if

$$\left(\left(\sum_{v \in \text{FOUT}_G(\text{FR}_G(\text{S,F}))} \begin{cases} 1 \text{ if } \exists x \in \text{CH}_G(v) : |\text{CH}_G(x)| \neq 0 \\ 0 \text{ otherwise} \end{cases}\right) \leq 1\right) \wedge (|\text{FIN}_G(f)| \leq 1) \wedge (\nexists x \in f : \exists a, b \in f \cap \text{CH}_G(x) : a \neq b) \wedge (\nexists x \in f : \exists a, b \in f : a \neq b \wedge x \in \text{CH}_G(a) \wedge x \in \text{CH}_G(b))$$

Denote predicate $\text{CF}_G(f)$, where $f = \text{FR}_G(\text{S}, \text{F})$, which equals TRUE if f is correct.

In **definitions 5.39-5.43**, the concept of a fragment and a correct fragment was formulated. A fragment is a subset of CFG c vertices that must be converted into a single code insertion in $\mathcal{B}$ with using FFI. Since code insertion using FFI has only one entry point and only

one exit point, the subset of vertices whose operations are included in the part of the code transformed to $\mathcal{B}$ must have no more than one edge to a vertex outside the fragment (excluding edges to *sinks*) and no more than one edge included in the fragment from vertex out of fragment.

Definition 5.44 (Set of correct fragments)**.** $\mathrm{SCF}(G) = \{\mathrm{FR}_G(a,b) |\ a,b \in V \wedge \mathrm{CF}_G(\mathrm{FR}_G(a,b))\}$

Theorem 5.45 (Number of correct fragments)**.** $|SCF(G)| \leq |V|^2$.

Proof. For any vertex $v \in V$ $|\mathrm{DE}_G(v)| \leq |V|$. According to **definition 5.39**, fragment is uniquely defined by entry-point and exit-point. The total number of possible combinations of a vertex and its descendant is $\sum\limits_{v \in V} |\mathrm{DE}_G(v)| \leq \sum\limits_{v \in V} |V| = |V| * |V| = |V|^2$. □

Definition 5.46 (Control transfer point)**.** For $E = E_G, V = V_G, G = G_c, c$ is CFG, for fragment $f = \mathrm{FR}_G(\mathrm{S},\mathrm{F}), \mathrm{S} \neq \mathrm{RO}_c \wedge \mathrm{CF}_G(f)$, $x \in V : \ (x,\mathrm{S}) \in E \wedge x \notin f$, name x *control transfer point*. According to **definition 5.43**, when $\mathrm{S} \neq \mathrm{RO}_c$, for $v \in V \wedge v \neq \mathrm{RO}_c$ $!\exists$ *control transfer point*. For f denote it's control transfer point as $\mathrm{CTP}_G(f)$.

Definition 5.47 (Members of cycles)**.** For $V = V_G, E = E_G, G$ is graph, $\mathrm{MC}(G) = \{v \in V : \exists k \geq 1 : \exists a_1, ..., a_k \in V : v, a_1, ..., a_k$is cycle$\}$. Name it *set of members of cycle*.

Definition 5.48 (Control return point)**.** For $E = E_G, V = V_G, G$ is graph, for fragment $f = \mathrm{FR}_G(\mathrm{S},\mathrm{F}), \mathrm{S} \neq \mathrm{RO}_c \wedge \mathrm{CF}_G(f)$, denote $\mathrm{CRP}_G(f) = \{x \in V |\ (\mathrm{F},x) \in E \wedge x \notin f \wedge |\mathrm{CH}_G(x)| = 0\}$, name it *set of control return points*. According to **definition 5.43**, for correct fragment, $|\mathrm{CRP}_G(f)| \leq 1$.

Definition 5.49 (Profit)**.** For $f = \mathrm{FR}_G(\mathrm{S},\mathrm{F}) \wedge \mathrm{CF}_G(f)$, for x is CFG and $c \in \mathrm{CCFG}(x)$, where $\mathrm{CO}_c(v) = \begin{cases} \mathcal{A} & v \notin f \\ \mathcal{B} & v \in f \end{cases}$

$$\mathrm{PR}_x(f) = \left(\begin{cases} 1 & |\mathrm{MC}(G) \cup f| = 0 \\ \gamma & |\mathrm{MC}(G) \cup f| \neq 0 \end{cases}\right) * \left(\sum_{v \in f} \mathrm{LA}_c(v,\mathcal{A}) - \sum_{v \in f} \mathrm{LA}_c(v,\mathcal{B}) - \mathrm{TC}_c(\alpha,\beta,\mathrm{CTP}_G(f),\mathrm{S}) - \sum_{v \in \mathrm{CRP}_G(f)} \mathrm{TC}_c(\alpha,\beta,\mathrm{F},v)\right)$$

$\mathrm{PR}_x(f)$ means what expected gain in laboriousness costs of program execution will be obtained if the code of fragment f is transformed into $\mathcal{B}$.

Definition 5.50 (Nonoptimal fragments)**.** For $V = V_G, G = G_c, c$ is CFG, $\mathrm{SNF}(c) = \{f \in \mathrm{SCF} |\ \mathrm{PR}_c(f) > 0\}$. Denote this set - *set of nonoptimal fragments*.

Transformation any fragment from the set SNF to Rust with using FFI reduces the expected execution time of the source code.

Definition 5.51 (Optimal selection of fragments)**.** The same part of the source code cannot be converted to Rust simultaneously in two different fragments. Then, to maximize possible reduction of laboriousness, for c is CFG, need to find such a subset of $\omega \subseteq \mathrm{SNF}(c)$: $(\forall x,y \in \omega : \ (x \neq y) \implies (x \cap y = \emptyset)) \wedge$ $\left(\left(\sum\limits_{w \in \omega} \mathrm{PR}_c(w)\right) \to \max\right)$

5.7 Choosing the optimal subset of fragments to transform with using FFI. Reduction to the Maximal clique problem.

The problem statement given in **definition 5.51** can be reduced to the *maximal clique problem*. In this section, we will show how to reduce the problem to a weighted maximal clique problem.

Definition 5.52 (Clique graph)**.** For c is CFG, denote *clique graph* $\mathrm{CG}(c) = (V,E)$, $V_{\mathrm{CG}} = \{(\mathrm{UID},\mathrm{WEIGHT}) = (\mathrm{UID}_f, \mathrm{PR}_c(f)) |\ f \in \mathrm{SNF}(c)\}$, $((a,b) \in E_{\mathrm{CG}}) \Leftrightarrow (x,y \in \mathrm{SNF}(c) \wedge x \neq y \wedge |x \cap y| \neq 0)$. *Clique graph* is graph too.

Definition 5.53 (Clique)**.** For c is CFG, *clique* is $q \subseteq V_{\mathrm{CG}(c)}$.

Definition 5.54 (Independent clique)**.** For c is CFG, clique q is *independent* if $\forall a,b \in q : a \neq b \implies (a,b) \notin E_{\mathrm{CG}(c)}$.

Definition 5.55 (independent clique weight)**.** For c is CFG, for q is independent clique of c, denote $\mathrm{WE}_c(q) = \sum\limits_{v \in q} \mathrm{WE}_v$.

Definition 5.56 (Optimal selection of fragments)**.** For c is CFG, to maximize possible reduction of laboriousness, need to find *independent clique* $q : \mathrm{WE}_c(q) \to \max$. Denote that clique q *ideal independent clique*.

Definitions 5.51 and 5.56 are an equivalent description of which fragments to select to transformation to Rust. In addition, according to **definition 5.49**, for c is CFG, maximizing PR_c leads to a decrease the value of the L-function (**definition 5.37**). That is, the task of maximizing PR_c is equivalent to the task of reducing the expected laboriousness of program execution.

6 The algorithm for constructing the clique graph by CFG (α-algorithm) and maximal clique task solving

This section describes an algorithm that builds clique graph by CFG.

First of all, algorithm implementation is *here*.

Here and further, $c = (G, \text{OBJ}, \text{RO})$ is CFG, $V = V_G, E = E_G$.

Input data: α-algorithm takes as input CFG.

The result of the algorithm: according to **definitions 5.51 and 5.56**, algorithm must return set of *correct fragments*, having no intersecting CFG vertices and having maximum total PR.

6.1 α-algorithm: building a clique graph

Preprocessing takes place before the algorithm itself:

- For any $v \in V$, the list of programming objects used for reading or writing in all vertices-descendants of v is calculated. Denote objectsUsedInDescendants - array of hashsets, objectsUsedInDescendants[v] contains IDs of all programming objects, used in all descendants of v. The calculation of objectsUsedInDescendants[v] is performed using depth-first search (DFS). DFS is started for each vertex. If DFS is started from vertex v, entering new vertex u, all programming objects from VR_u and VW_u are added to objectsUsedInDescendants[v]. Since G in CFG is a rather sparse graph, the DFS single execution time complexity is $O(|V|+|\text{OBJ}|)$. Time complexity of calculation all objectsUsedInDescendants is $O(|V|(|V| + |\text{OBJ}|))$.
- Calculating fragmentRequirements. fragmentRequirements is array of hashsets, each hashset - vertices, color of which should be equal. For any vertex v, if $|\text{CH}_G(v)| > 1$, then to fragmentRequirements added hashset, concluding v and all vertices from $\text{CH}_G(v)$. Time complexity of calculation fragmentRequirements is $O(|V|)$.
- Calculating verticesInLoops. verticesInLoops is hashset of all vertices, which are members of loop. To calculate verticesInLoops, for each $v \in V$ starts depth-first search. If DFS returns to vertex v in some way, then v puts to verticesInLoops. Since G in CFG is a rather sparse graph, time complexity of calculating verticesInLoops is $O(|V|^2)$.
- Calculating graphReversed for G. graphReversed $= (V, E')$ is graph, where $(a, b) \in E' \Leftrightarrow (b, a) \in E$. Since G in CFG is a rather sparse graph, time complexity of calculating graphReversed is $O(|V|)$.

Functions definition:

findFragments:

The algorithm is based on a DFS. Name that recurcive function *findFragments*($v \in V$). Denote numberPrevObjectUsage - array, which maps programming object ID and number vertices on *findFragments* path to current vetrex v, in which VR or VW attended this programming object ID. Each time *findFragments* enters a vertex v, the number of uses of each object from VR_v and VW_v increases by 1, and when exiting, it decreases by 1. Depending on the traversal order, the values in numberPrevObjectUsage may differ, but regardless of the traversal order, if numberPrevObjectUsage$[x] = 0 \implies$ object x was never used before vertex v. Entering v, *findFragments* iterates through its descendants w, and checks if $\text{FR}_c(v, w)$ correct with using *checkSubFragment*. And after that, it runs recursively from all $\text{CH}_G(v)$.

checkSubFragment:

checkSubFragment checks if $f = \text{FR}_c(\text{S}, \text{F})$ correct. Firstly, *checkSubFragment* calculates currentFragment - hashset of vertices, included in f. After, *checkSubFragment* is interrupted prematurely if:

- currentFragment contains RO. According to **definition 5.21**, $\text{CO}_c(\text{RO}_c) = \mathcal{A}$;
- currentFragment contains sink. According to **definition 5.22**, $(|\text{CH}_G(v)| = 0) \implies (\text{CO}_c(v) = \mathcal{A})$;
- If it is FALSE, that for all $x \in$ fragmentRequirements, currentFragment contains all vertices from x, or contains no one of them. This check ensures that the all vertices in x, that must be the same color, belong same fragment, or do not belong any fragments. Since requirements are imposed, only on vertices a kind $(a, b \in x : (a, b) \in E) \vee (a, b \in x : \exists r \in x : (r, a) \in E \wedge (r, b) \in E)$, that they can belong only to one fragment, and not to different fragments;

- $\left(\sum_{v \in \text{FOUT}_G(f)} \begin{cases} 1 \text{ if } \exists x \in \text{CH}_G(v) : \\ |\text{CH}_G(x)| \neq 0 \\ 0 \text{ otherwise} \end{cases}\right) \geq 2) \vee (|\text{FIN}_G(f)| \geq 2)$. According to **definition 5.43**, that means F is not correct fragment. To check FOUT condition, *checkSubFragment* iterates by $v \in f$ and checks if v has vertex in $\text{CH}_G(v)$, which is not sink, and counts the number of such vertices v. Similarly, to check FIN condition, it iterates by $v \in f$ and counts the number of vertices v for which $|\text{CH}_{\text{graphReversed}}(v)| \neq 0$. Fulfilling this, as well as all the above conditions, ensures that f is correct fragment.
- $\nexists v \in$ currentFragment : $v \in$ prohibitedToTransform. Also, the algorithm provides an opportunity to prohibit the inclusion of certain vertices in fragments;
- $\text{PR}_c(f)$ is less or equals 0. The calculation of profit is described in *getFragmentProfit*.

If *checkSubFragment* have not interrupted, it adds fragment f to subFragments array - list of fragments, which will participate in the construction of the cliqueGraph.

getFragmentProfit:

getFragmentProfit returns expected profit of $f = \text{FR}_c(\text{S}, \text{F})$. Firstly, it checks if $\exists v \in f : v \in$ verticesInLoops. After that, it counts summarised weight of imported and exported objects (with using getFragmentImportObjectsWeight and getFragmentExportObjectsWeight) functions. Finally, it returns profit value according to **definition 5.49**.

getFragmentImportObjectsWeight:

For fragment $f = \text{FR}_c(\text{S}, \text{F})$, method calculates all objects, occuring in VR and VW in all vertices in f (denote fragmentObjects). After that, it counts and returns summarised WE of objects in intersection fragmentObjects and objects, for which numberPrevObjectUsage is greater 0. This intersection contains only those objects that need to be imported into the fragment (according to **definition 5.27**).

getFragmentExportObjectsWeight:

For fragment $f = \text{FR}_c(\text{S}, \text{F})$, method calculates all objects, occuring in VR and VW in all vertices in f (denote fragmentObjects). After that, it counts and returns summarised WE of objects in intersection fragmentObject and objects, for which objectsUsedInDescendants ≥ 1. This intersection contains only those objects that need to be exported from the fragment (according to **definition 5.28**).

buildCliqueGraph:

buildCliqueGraph builds clique graph by subFragments array. It builds graph CG = (V_{CG}, E_{CG}), $(a, b) \in E_{CG}$, if $\exists x : x \in \text{subFragments}_a \wedge x \in \text{subFragments}_b$.

The algorithm:

The algorithm starts with a call *findFragments*(RO). It is used to calculate subFragments (SNF(c)). After that it calls *buildCliqueGraph*, which builds clique graph for subFragments.

Time complexity:

Prerprocessing time complexity is $O(|V|^2 + |V| * |\text{OBJ}|)$.

findSubFragments time complexity is $O(|V|^3 + |V| * |\text{OBJ}|)$.

buildCliqueGraph time complexity is $O(|\text{subFragments}|^2 * |\text{OBJ}|)$. According to **theorem 5.45**, $|\text{subFragments}| \leq |V|^2$, so buildCliqueGraph time complexity is $O(|V|^4 * |\text{OBJ}|)$. In practice, when considering a large number of CFG examples, |subFragments| = $O(|V|)$.

So, all α-algorithm time complexity is $O(|V|^4 + |V|^2 |\text{OBJ}|)$.

6.2 Approaches to finding the maximal clique problem

The maximal clique problem is NP-complete task. The search for the maximal clique by a complete search of all options for clique graph with n vertices has time complexity $O(2^n * n^2)$. This is too much computational complexity. In addition, in the original formulation, clique does not have WE (we can assume that for each clique q $\text{WE}_q = 1$). So greedy algorithms will be used to search for independent clique with as much WE as possible. Greedy algorithms are searching for clique $q : \text{WE}_q \rightarrow \max$, but it is not guaranteed that they will find *ideal independent clique*.

Existing algorithms, solving maximal clique problem, which find the exact solution

Unsuccessful attempts were made to modify the following greedy algorithms to solve the weighted case of the maximal clique problem:

- Algorithm of Frank Harary and Ian C. Ross (doi:10.2307/2785673).

- Robson algorithm and modification of Robson algorithm. It also has exponential time complexity $O(2^{0.282n}n^2)$, but in practice it shows itself to be much better than a full search.
- Bron and Kerbosch algorithm. It has modification for weighted clique task, but it increases its time complexity to $O(3^{n/3}n)$.
- P. R. J. Ostergard algorithm. In weighted modification, its time complexity is $O(2^n)$, but this algorithm can easily be parallelized, unlike all of the above.

The considered algorithms are applicable in practice only for graphs containing no more than 50-70 vertices, which is a rather serious limitation.

Possible approaches for greedy problem solving:

- *Random walk through the graph:*
  A depth-first search is started from random vertex, at each even iteration (iterations are numbered from 0), the neighbors of the current vertex are added to *skiplist*, and the current vertex is added to the click, and the algorithm switches to a random neighbor of the vertex, which is not included in *skiplist*. On an odd iteration, there is simply a transition to a random neighbor. For a graph with n vertices, n^2 such DFS launches are performed, the one clique with the biggest weight is selected from all.
- *search random cliques:*
  Creates hashset *currennt_clique*. For graph G, while it's possible, select random vertex $v \in V_G$: $(v \notin$ *current_clique* $) \wedge (\nexists x \in$ *current_clique* $: (x, v) \in E_G)$ and add it to *current_clique*. This process repeats for $|V|^2$ times, after that selects clique with maximum WE.

The implemented algorithm takes a cliqueSolver object, which solves maximal clique problem. The implemented algorithm provides several algorithms for solving the maximal clique problem.

7 Proof of the correctness of the α-algorithm

The proofs of the correctness of the algorithm consist of proving several theorems.

7.1 The theorem that all returned by α-algorithm fragments are correct and do not intersect

Theorem 7.1. *For any $c = (G, OBJ, RO)$ is CFG, algorithm returns set of correct code fragments with no intersections.*
Proof.

- *No vertex should be included in more than one fragment for transformation to Rust.*

  According to **definition 5.56**, α-algorithm returns q - *independent clique* of $\mathrm{CG}(c)$. As clique q is independent, $\nexists x, y \in q : (x, y) \in E_{CG(c)}$. According to **definition 5.52**, that means $\nexists a, b : ((a \neq b) \wedge (x, y \in \mathrm{SNF}(c)) \wedge (\mathrm{UID}_x = a \wedge \mathrm{UID}_y = b) \wedge (|x \cap y| \neq 0))$.

- *Each fragment, which is needed to be transformed to Rust, must conclude only one control transfer point.*

  According to **definition 5.43**, for any fragment, which does not conclude RO_c, it has no more than one *control transfer point*. According to **definition 5.39**, fragments which conclude RO_c are not correct. According to **definition 5.13**, RO_c is *source*, so $\nexists v \in V_{G_c} : v \neq \mathrm{RO}_c \wedge \mathrm{RO}_c \in \mathrm{DE}_{G_c}(v)$. So, each $f \in \mathrm{SNF}(c)$ has only one control transfer point.

- Each fragment, which is needed to be transformed to Rust, must conclude only one control return point.

  According to **definition 5.48**, every correct fragment has no more than one *control return point*. According to **definition 5.22**, every sink has color $\mathcal{A}$, so no one correct fragment contain sink. That means, that for $f = \mathrm{FR}_G(\mathrm{S}, \mathrm{F})$ for specified S and F, F is not sink. That means, that $|\mathrm{CH}_G(\mathrm{F})| \geq 1$. So, $(1 \leq |\mathrm{CH}_G(F)| \leq 1) \implies |\mathrm{CH}_G(F)| = 1 \implies$ every correct fragment has only one *control return point*.

□

7.2 The theorem on the optimality of fragment selection by α-algorithm

Theorem 7.2. *Using the exact algorithm for solving the* maximal clique problem*, the α-algorithm makes the optimal selection of fragments (according to **definition 5.51**).*

Proof. According to **definitions 5.52-5.56**, solving of *maximal clique problem* equals to solving *selecting disjoint fragments with the maximum amount of* PR.

According to **definition 5.39**, every fragment uniquely defined by S and F ($\text{FR}_G(\text{S}, \text{F})$).

findFragments function, as it was described, iterates through all such pairs $(\text{S}, \text{F}) : \text{F} \in \text{DE}_G(S)$ and filters only *correct* fragments. That means, that *findFragments* finds all correct fragments for building *clique graph*.

α-algorithm returned $w \subseteq \text{SNF}(c)$. Suppose that exists $\omega \subseteq \text{SNF}(c) : \left(\left(\sum_{f \in \omega} \text{PR}_c(f)\right) > \left(\sum_{f \in w} \text{PR}_c(f)\right)\right) \wedge (\nexists a, b \in \omega : |a \cap b| \neq 0)$. That means, that in $\text{CG}(c)$ exists *independent clique*, which WE is more than WE of maximal independent clique. But that means, that algorithm to solve *maximal clique problem* is not exact, but this contradicts the original assumption. $\perp \implies$ on the contrary, it is shown that with the optimal solution of the *maximal clique problem*, the selection of fragments is optimal.

□

8 α-algorithm hyperparameters

As was mentioned in **subsection 5.4**, for transporting functions were used hyperparameters α, β, γ. This section is devoted to observations on how to select a hyperparameter.

8.1 α-hyperparameter

The hyperparameter α is responsible for the constant part of the cost of transporting data. It includes a constant part of the laboriousness of transporting data to a piece of code converted to the FFI language, and transporting data from a piece of code.

8.2 β-hyperparameter

The hyperparameter β is the coefficient of proportionality of the laboriousness of transporting data to and from a code fragment, depending on the total weight of imported and exported objects (according to **definitions 5.31 and 5.32**).

8.3 γ-hyperparameter

The hyperparameter γ is the coefficient of regularization for cycles in control flow graph. γ is used only in H-function (**definition 5.36**).

Possible values of γ:

- $\gamma > 1$ does not make sense;
- A value of 1 corresponds to the absence of regularization;
- $\gamma \in (0; 1)$ respond to using regularization. The higher the value of γ, the more the algorithm encourages the conversion of the entire cycle into control flow graph in its entirety, if the profit from one of its iterations is positive. Note that the parameter γ should be the larger the more iterations of the loop in control flow graph are expected on average.

The values of the hyperparameters α, β, γ should be selected experimentally.

9 Conclusion

In this article, a new α-algorithm was described that allows you to select code fragments that need to be converted into code inserts in the interface language of external functions to speed up the work of the source code of the program. In addition, the correctness of the developed algorithm was formally proved. In addition, a command-line tool has been developed based on this algorithm. Also, the developed command-line tool will be integrated into the code optimization module for the EO language.

References

Bugayenko, Yegor (2021). *EOLANG and φ-calculus*. arXiv: 2111.13384 [cs.PL].

Kudasov, Nikolai and Violetta Kim (2022). *φ-calculus: a purely object-oriented calculus of decorated objects*. arXiv: 2204.07454 [cs.PL].

West, David (2004). *Object Thinking*. Pearson Education.