Ассамблея

Мультиязычный обфускатор программ и компилируемая библиотека непрозрачных предикатов

Антон Баглий, Борис Штейнберг, Елена Алымова, Константин Гуфан

4 апреля 2017 г.

ЮФУ, ФГАНУ Спецвузавтоматика

Первоначальные цели и ограничения

- **Предназначение:** Пооператорное запутывание байт-кода MSIL, LLVM и исходного текста JavaScript
- **Функции:**
  - Описание преобразований операторов общее для всех входных языков
  - Поиск подходящих операторов и применение преобразований к ним
  - Вставка вызовов или inlining непрозрачных функций, указанных в описании преобразований
- **Ограничения:**
  - Обработка каждой инструкции (оператора) отдельно
  - Одно представление для всех входных языков

Сложности

- Сложно создать новое универсальное представление для очень разных входных языков
- Структура исходной программы должна представляться одинаково для всех языков

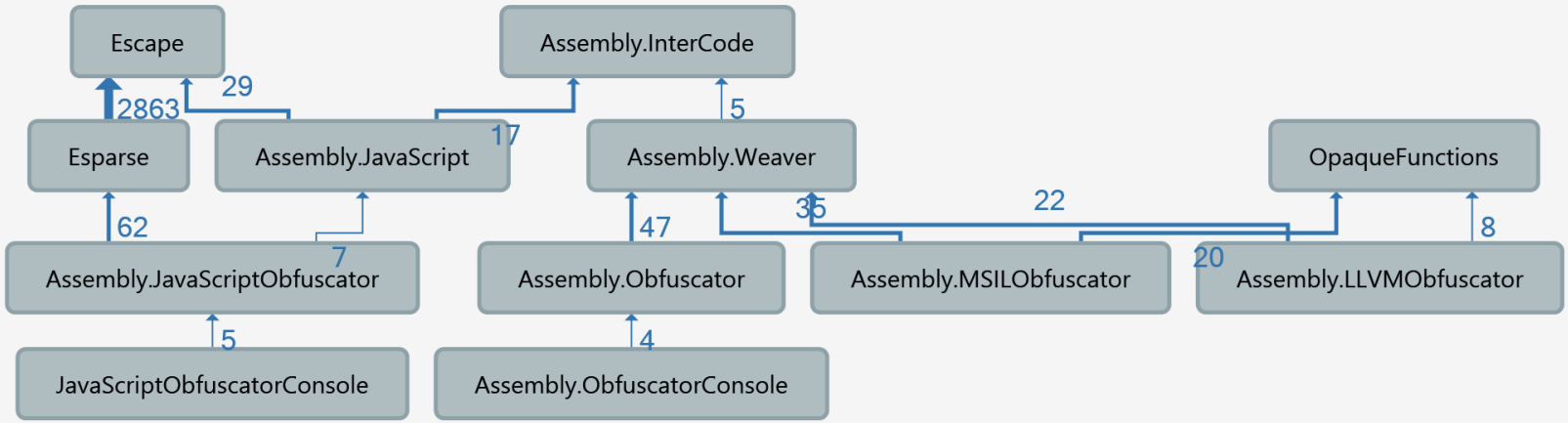
решения

- Используется готовое AST с аннотациями произвольного типа, в которые помещаются части исходной программы и другая информация (например, типы)
- Структура графа потока управления с вершинами-простыми блоками подходит больше всего

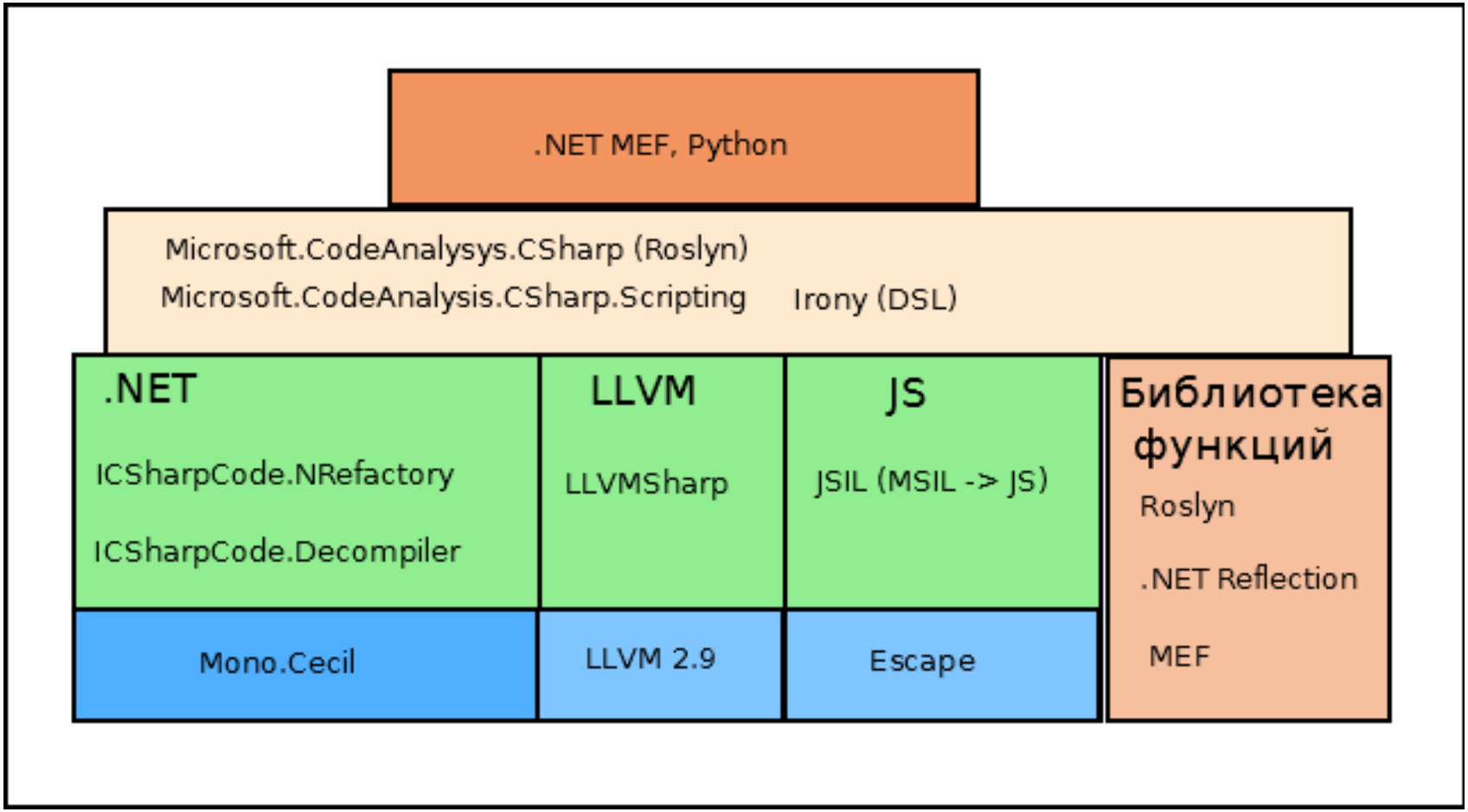
Реализованные средства и алгоритмы

- Описание преобразований - InterCode
- Применение преобразований к арифметическим операторам в MSIL, LLVM, JavaScript
- Проверка типов при применении преобразований в MSIL и LLVM
- Вставка непрозрачных функций, если они указаны в преобразовании
- Библиотека непрозрачных функций

Модули


Escape
Assembly.InterCode
29
2863
5
17
Esparse
Assembly.JavaScript
Assembly.Weaver
OpaqueFunctions
62
7
47
35
22
8
20
Assembly.JavaScriptObfuscator
Assembly.Obfuscator
Assembly.MSILObfuscator
Assembly.LLVMObfuscator
5
4
JavaScriptObfuscatorConsole
Assembly.ObfuscatorConsole

Инструменты



Общая схема

- Исходная программа обрабатывается в несколько проходов
- В каждом проходе программа разбивается на блоки операторов, например на простые блоки-вершины управляющего графа.
- Посещаются операторы каждого блока, проверяется применимость преобразований к ним.
- Для операторов, к которым применимы преобразования, они применяются.
- Записывается полученный байт-код или исходный код модифицированной программы.

InterCode

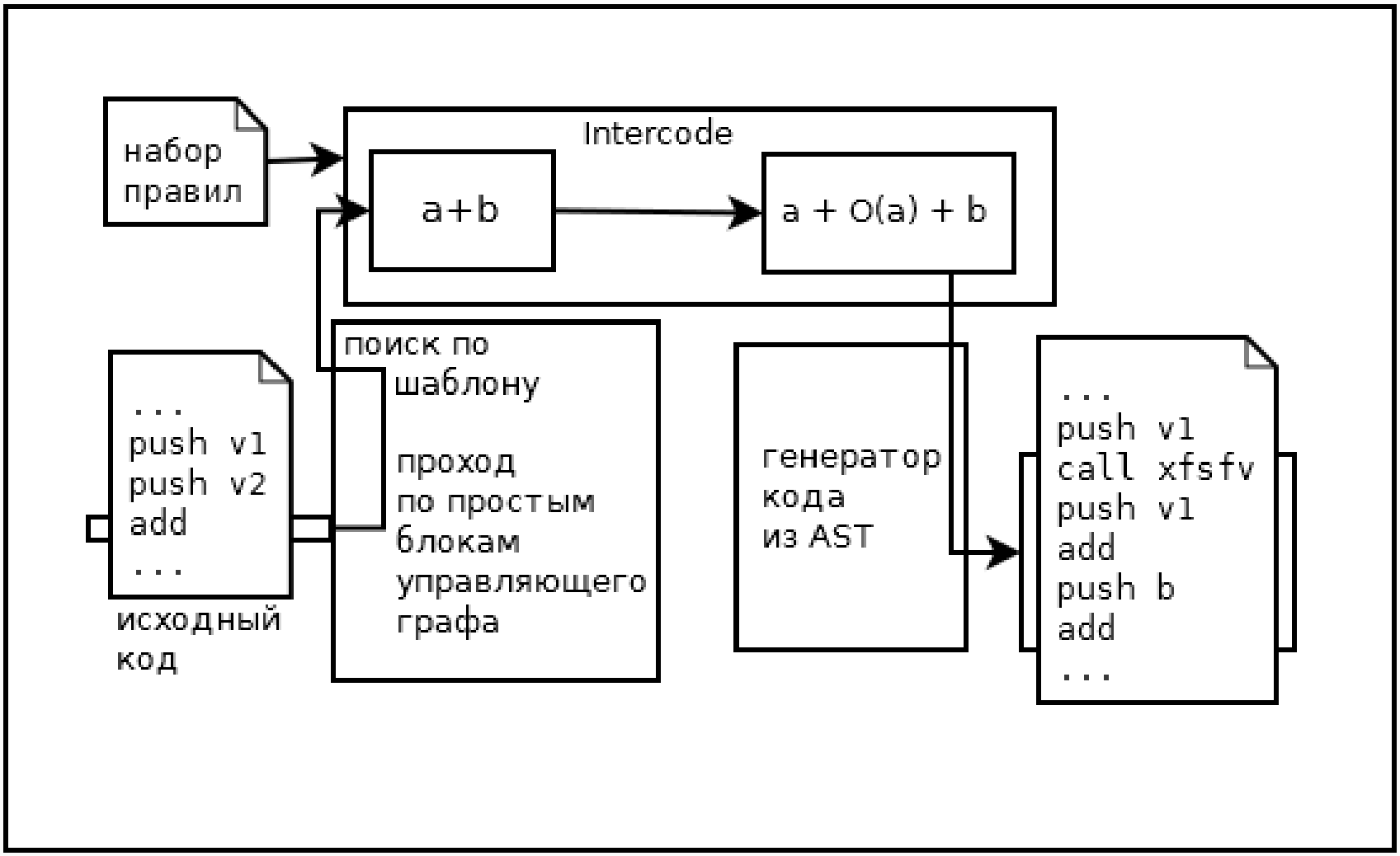
пример

SumTransform: a + b => a + 1 + b - 1 + c { var c = FormIntOpaqueValue(0);}

здесь

- SumTransform, DivTransform - имена преобразований
- выражение1 => выражение2 означает замену оператора (или его части) эквивалентного левой части на правую
- текст в скобках { } задает значения новых переменных, в примере это вызов непрозрачной функции

Применение правил



Пример запуска

```
o = PyAssembly.Manager(..., "LLVM")
p = o.GetPass("all_cfg")
o.AddParameter("useCodeGenerator", "true")
o.AddParameter("constant1", "Opaque1LoopInt")
p.CreateRule(name = "1", fromExpr = "a+b",
to = "a+b", action = "␣")
p.CreateRule(name = "2", fromExpr = "a-b",
to = "a-b+1-1", action = "␣")
p.CreateRule(name = "3", fromExpr = "a*b",
to = "a*b*c",
action = "var␣c=FormIntOpaqueValue(1);␣")
o.AddPass(cfgPass)
o.Open(path); o.Run()
o.Save(path)
```

Дополнительные условия в правилах

```
c = System.Func[IAttributable, bool](
lambda function: function.HasAttribute("ObfuscateThis"))
extraConditions = List[System.Object]()
extraConditions.Add(extraCondition)
p.CreateRule("1", "a+b", "a+b+0", "", extraConditions)
p.CreateRule("2", "a-b", "a-b+0", "", extraConditions)
p.CreateRule("3", "a*b", "a*b+0", "", extraConditions)
o.AddPass(cfgPass)
```

Это применит замены только к функциям
с атрибутом [ObfuscateThis]

Типы

пример

SumTransform: a + b => a + 1 + b - 1 { int a; int b;}

- можно указать типы для всех участвующих переменных
- типы проверяются без неявных преобразований
- расширить проверки типов достаточно просто

Ветвления

пример

```
1: _FunctionStart=>hub{var hub=AddControlHub();}
2: _Break=>entry{var entry=AddHubEntryAfter(current);}
3: _FunctionEnd=>nop{var nop=FinilizeHub();}
```

- можно перенаправлять операторы ветвлений
- нужны проверки корректности
- не достаточно удобно добавлять правила

Пример MSIL

правило: a-b => a - 24 - b + 24

sub =>

результат

```
stloc a_backup
ldloc a_backup
ldc.i4 24
sub
ldloc b_backup
sub
ldc.i4 24
add
```

Пример вставки функций MSIL

правило: a+b => a + 3 + b - 3 + f(), f = sin(pi/3) + sin(-pi/3)

add

=>

результат

```
stloc a_backup          ldc.r8 1,047197551966
ldloc a_backup          call Double Sin(Double)
ldc.i4 3                ldc.r8 -1,047197551966
add                     call Double Sin(Double)
ldloc b_backup          add
add                     stloc V_22
ldc.i4 3                ldloc V_22
sub                     add
```

Пример LLVM

правило: a * b => a * b * 3 / 3

%tmp2 = mul
i32 %1,
%tmp1

результат

```
%tmp0012 = alloca i32
  store i32 3, i32* %tmp0012
  %tmp0023 = load i32, i32* %tmp0012
  %tmp0034 = mul i32 %tmp0023, %tmp0001
  %tmp0045 = alloca i32
  tore i32 3, i32* %tmp0045
  %tmp0056 = load i32, i32* %tmp0045
  %tmp0067 = sdiv i32 %tmp0056, %tmp0034
```

Пример вставки функций LLVM

правило: a * b => a * b * c, c = Opaque1LoopInt()

%tmp2 = mul i32 %1, %tmp1

результат

```
%tmp1 = mul i32 %tmp1, %1
%tmp2 = call i32 @Opaque1LoopInt()
%tmp3 = mul i32 %tmp2, %tmp1
%tmp4 = mul i32 %1, %tmp0
...
define i32 @Opaque1LoopInt() {
"3":
%loc0 = alloca double
...
"4": ...
```

Пример JavaScript

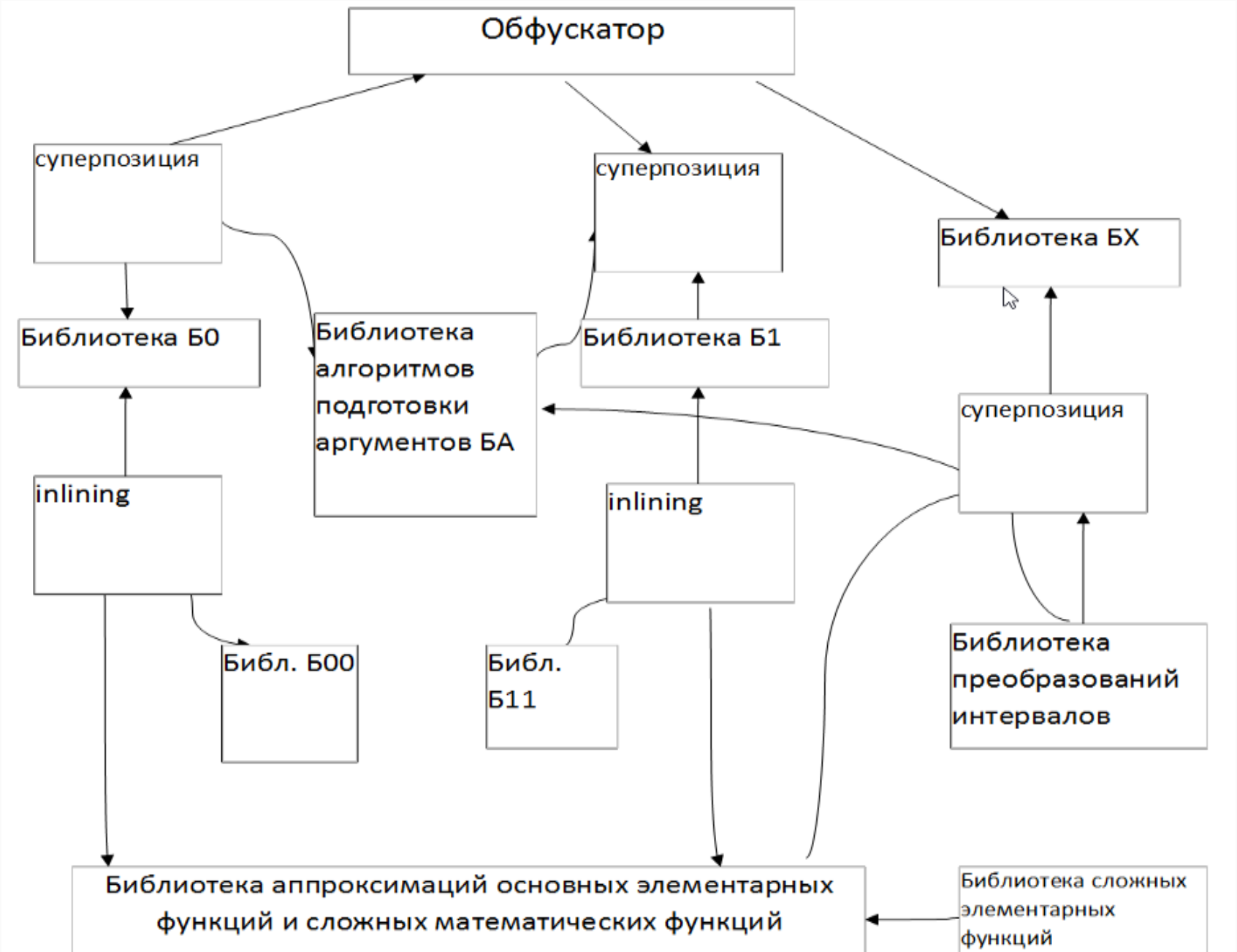
вход

```
var res1 = (a - b) | 0; // result = 3
var res2 = (a * b) | 0; // result = 18
var res3 = (a / b) | 0; // result = 2
var res4 = (((a + b) | 0) - 1) | 0;
```

результат

```
var res0 = ((((a + 2) + b) - 2)) | 0;
var res1 = ((((a - 42) - b) + 42)) | 0;
var res2 = ((((a * b) * 2) / 2)) | 0;
var res3 = (((a * 1) / b)) | 0; // result = 2
var res4 = ((((((a + 3) + b) - 3)) | 0) - 1) | 0;
```

Структура библиотеки непрозрачных функций



Пример простой функции

$$\ln(1+x) = \frac{2x}{2+x} - \frac{x^2}{3(2+x)} - \frac{4x^2}{5(2+x)} - \ldots$$

$$\ldots - \frac{m^2x^2}{(2m+1)(2+x)} - \ldots;$$

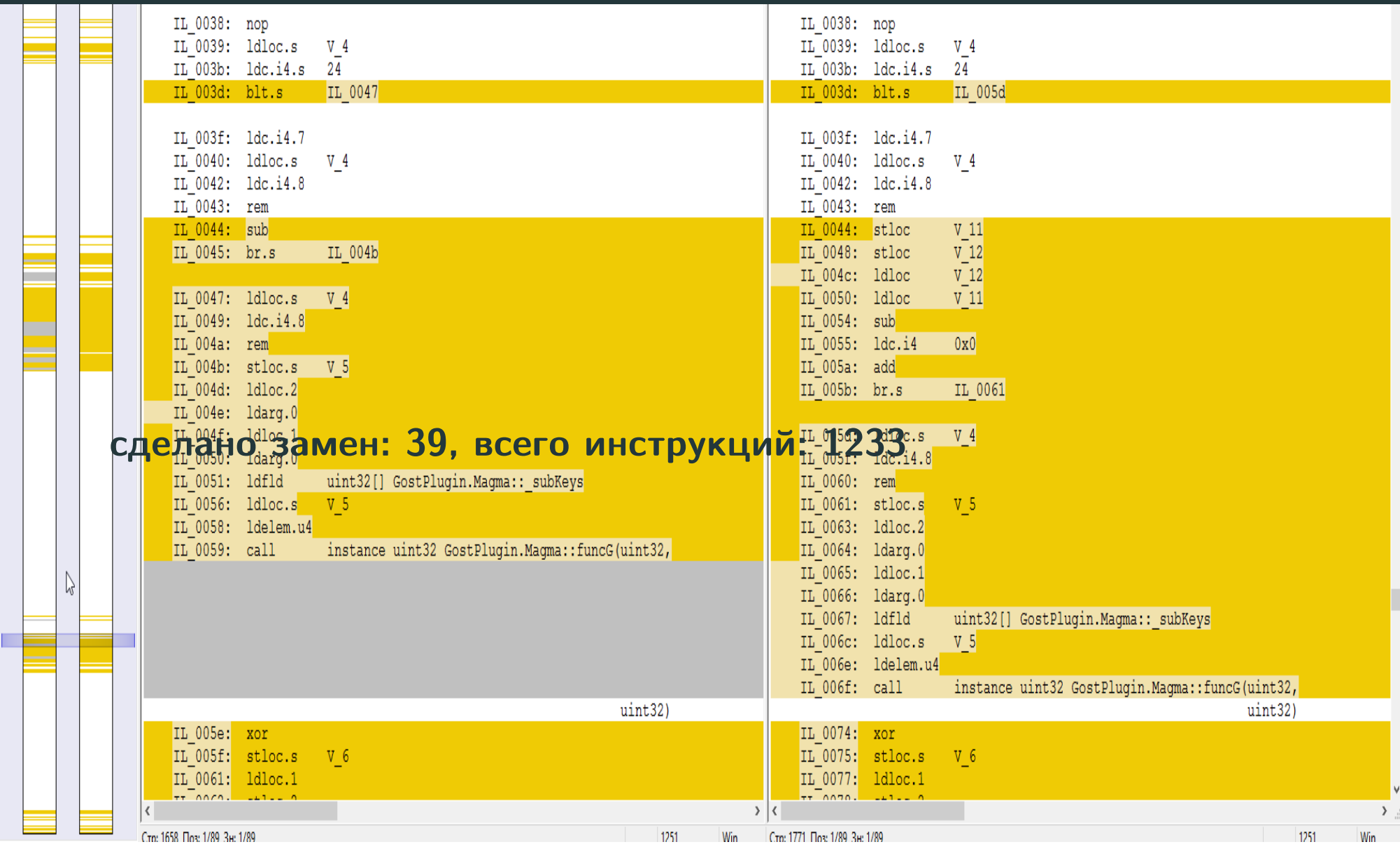
$$\ln(1+x) = x - \frac{x^2}{2} + \frac{4x^3}{3(4+3x)} - \frac{27x^3}{5(12+7x)} - \ldots$$

$$\ldots - \frac{(m^2-1)^3 x^2}{(2m+1)(2m(m+1)+(m^2+m+1)x)} - \ldots;$$

$$\ln(1+x) = x - \frac{3x^2}{6+4x} - \frac{10x^2}{30+16x} - \frac{126x^2}{70+36x} - \ldots$$

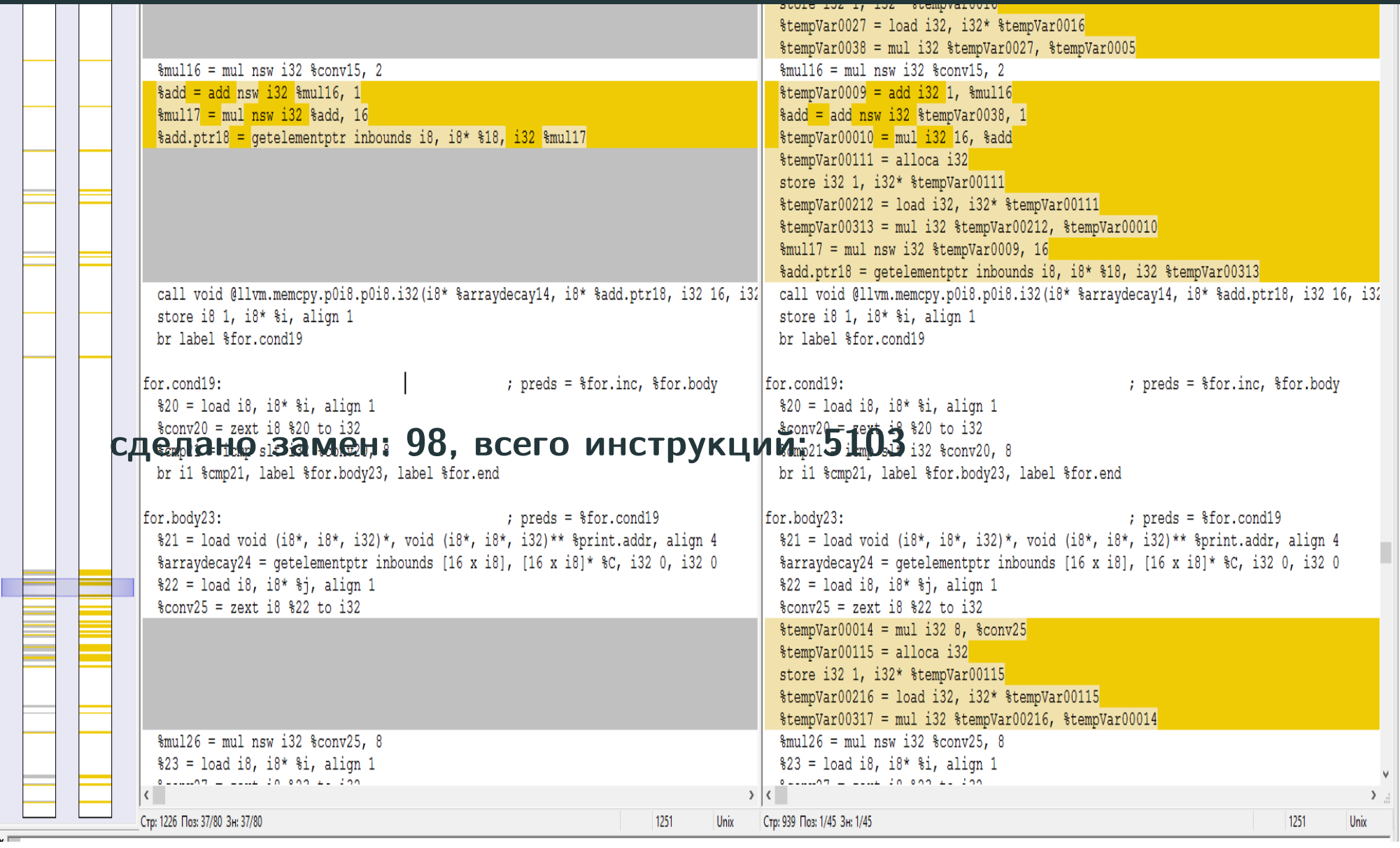
$$\ldots - \frac{m(m-1)(2m-3)(2m+1)x^2}{2(4m^2-1)+4m^2x} - \ldots;$$

Рабочий пример MSIL - ГОСТ Р 34.12-2015.



сделано замен: 39, всего инструкций: 1233

Рабочий пример LLVM - ГОСТ Р 34.12-2015.



сделано замен: 98, всего инструкций: 5103

Другие примеры и большие сборки.

1. Linpack и Whetstone для .NET (сборка 20-30кб) сделано замен: 252, всего инструкций: 2628, время работы около минуты
2. MathNet.Numerics (сборка 1.5мб) время работы больше часа, потребляет больше 10Гб памяти...

Нерешенные проблемы

1. обход управляющего графа в JavaScript и использование общего интерфейса
2. вывод типов в JavaScript - сложная задача в общем случае
3. оптимизация времени работы обфускатора MSIL (использование "тяжелых"инструментов)
4. неоптимальная организация обфускатора LLVM из-за использования C-интерфейсов
5. как не замедлить программу модификациями? Пример для ГОСТ с MSIL может легко замедлить его в 5-10 раз из-за вставки кода во вложенные циклы.
6. разработка эквивалентных преобразований

Сценарии развития

1. семейство обфускаторов, объединенных общим подходом (описанием преобразований, алгоритмом их выбора)
   1.1 улучшенная работа с динамическими языками за счет использования уже имеющихся инструментов
   1.2 сложнее обеспечить какой-то общий способ модификации потока управления
2. многоязыковая среда с обработкой нескольких языков через обобщенный метаязык - сильнее ограничивает работу с каждым из языков и сложнее в реализации