Московский физико-технический институт
(государственный университет)
Физтех-школа радиотехники и компьютерных технологий
Кафедра информатики и вычислительной техники

Доработка статического анализа кода для выявления потенциальных ошибок в программах

Выпускная квалификационная работа
(бакалаврская работа)

Выполнил: Левченко Д.Н., 713 группа
Научный руководитель: к.т.н. Ермолицкий А.В.

Введение: анализ кода

Статический и динамический анализ

	Статический анализ	Динамический анализ
Преимущества	• Используется на ранних этапах разработки; • Позволяет анализировать существующую базу кода без компиляции; • Интегрируется в среду разработки.	• Высокая продуктивность по нахождению ошибок; • Для отслеживания причины ошибки может быть произведена полная трассировка стека и среды исполнения.
Недостатки	• Большая вероятность ложно-положительных срабатываний.	• Вмешательства в работу программы; • Полнота анализа ошибок зависит от степени покрытия кода.

Введение: инструменты статического анализа

	Статические анализаторы	Компиляторы
Преимущества	• Выявляют больше ошибок в программах.	• Простота использования; • Низкая стоимость нахождения ошибки.
Недостатки	• Требуют отдельной настройки; • Требуют много памяти и времени; • Хороший анализатор – коммерческий продукт с высокой стоимостью.	• Необходим баланс между временем работы и полнотой анализа. Зачастую последняя страдает.

Цель работы

Цель работы:

Разработать модуль в языковом компиляторе LCC, осуществляющий статический анализ с выдачей пользовательских предупреждений, для выявления потенциальных ошибок в программах для языков C и C++.

Задачи:

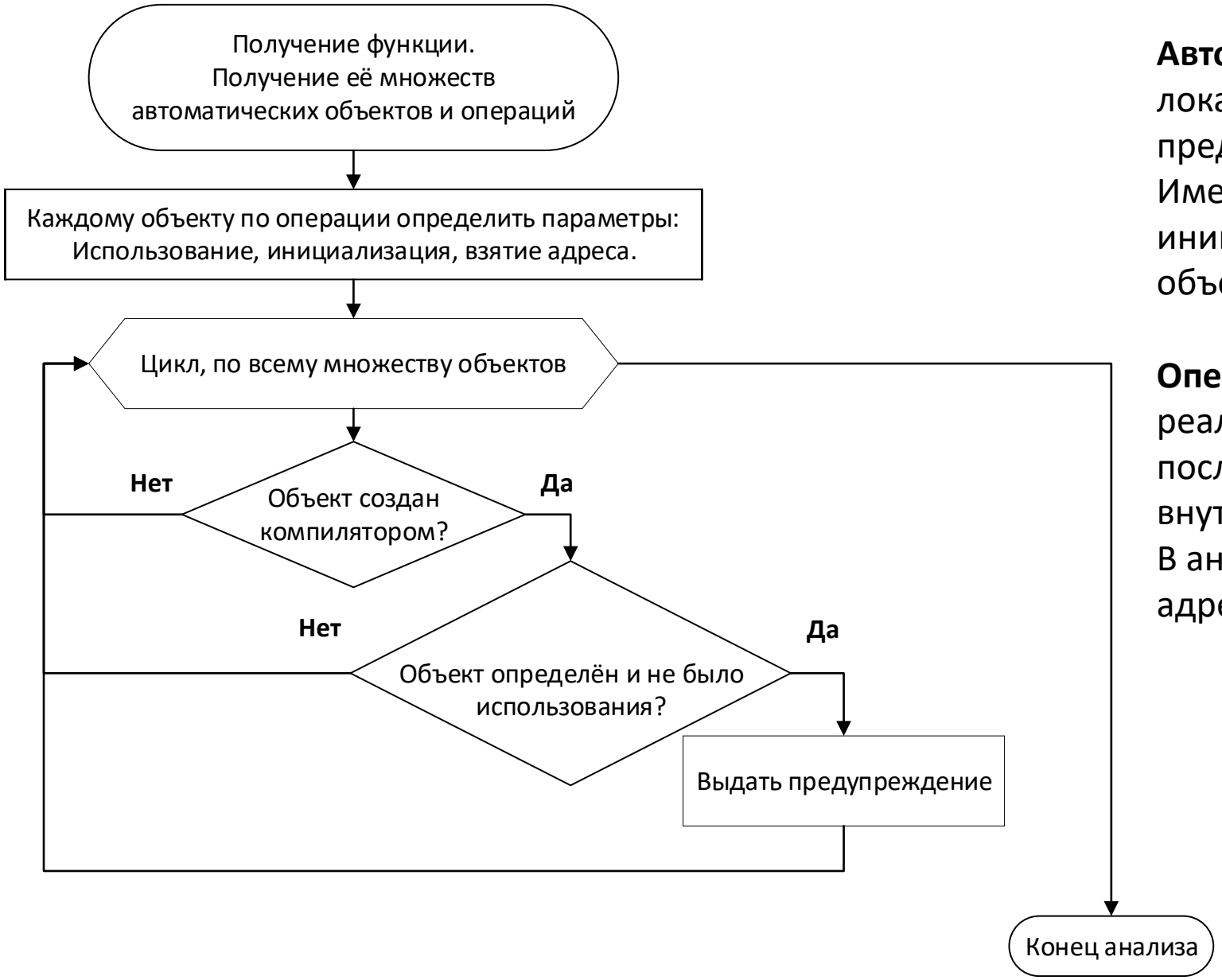
1. Разработать модуль в языковом компиляторе LCC, осуществляющий статический анализ с выдачей предупреждений по запросу пользователей:
   a) о неиспользуемых переменных;
   b) об использовании переменных в охватывающей области видимости;
   c) об использовании неинициализированных полей структур.
2. Отладить анализ на коротких направленных тестах, самосборке языкового компилятора и задачах SPEC CPU (95, 2000, 2006, 2017) для языков C и C++.
3. Собрать статистику по эффективности данного анализа для выявления потенциальных ошибок в целевых программах.

Задача 1a: анализ неиспользуемых переменных

Пример неиспользуемой переменной

```
1  double foo( double x1, double x2)
2  {
3    double a1, a2, b1, b2;
4
5    a1 = sin( x1);              Переменная определена, но не использована
6    a2 = cos( x2); <-----------------------------------------------------
7
8    b1 = bar( a1);            Здесь программист забыл изменить переменную
9    b2 = bar( a1); <-----------------------------------------------------
10
11   return b1 * b2;
12 }
13 ---------------------------------------------------------------------------
14 Выданное компилятором предупреждение:
15 lcc: "/home/levchenko_d/test.c", line 2: in function "foo":
16 lcc: "/home/levchenko_d/test.c", line 3: warning: variable "a2" is set, but not
17            used [-Wunused-but-set-variable]
```

Задача 1а: алгоритм поиска неиспользуемых переменных



Автоматический объект – описание каждой локальной переменной во внутреннем представлении компилятора LCC.
Имеет свойства: был ли объект использован, инициализирован, брался ли на него адрес, имя объекта и т.д.

Операция - высокоуровневая абстракция над реальной машинной инструкцией (или последовательным набором инструкций) для внутреннего представления компилятора.
В анализе используются операции чтения, записи, адресные операции.

Задача 1а: результаты поиска неиспользуемых переменных

Отладка анализа произведена на:

- Наборе самописных направленных тестов (27 шт.);
- Самосборке языкового компилятора;
- Исходниках бенчмарков SPEC CPU (95, 2000, 2006, 2017) для языков C и C++.

Результаты проведения анализа и сравнение с существующими решениями

Статистика собрана на исходниках бенчмарков
SPEC CPU (95, 2000, 2006, 2017) для языков C и C++

	GCC	Cppcheck	EDG	**LCC**
Количество предупреждений	941	3583*	1263	**1507**

*имеет много дублирующих предупреждений, статистика несколько завышена.

Задача 1b: анализ возможного использования переменных в охватывающей области видимости

Адрес локальной переменной нельзя записывать в переменную с охватывающей областью видимости

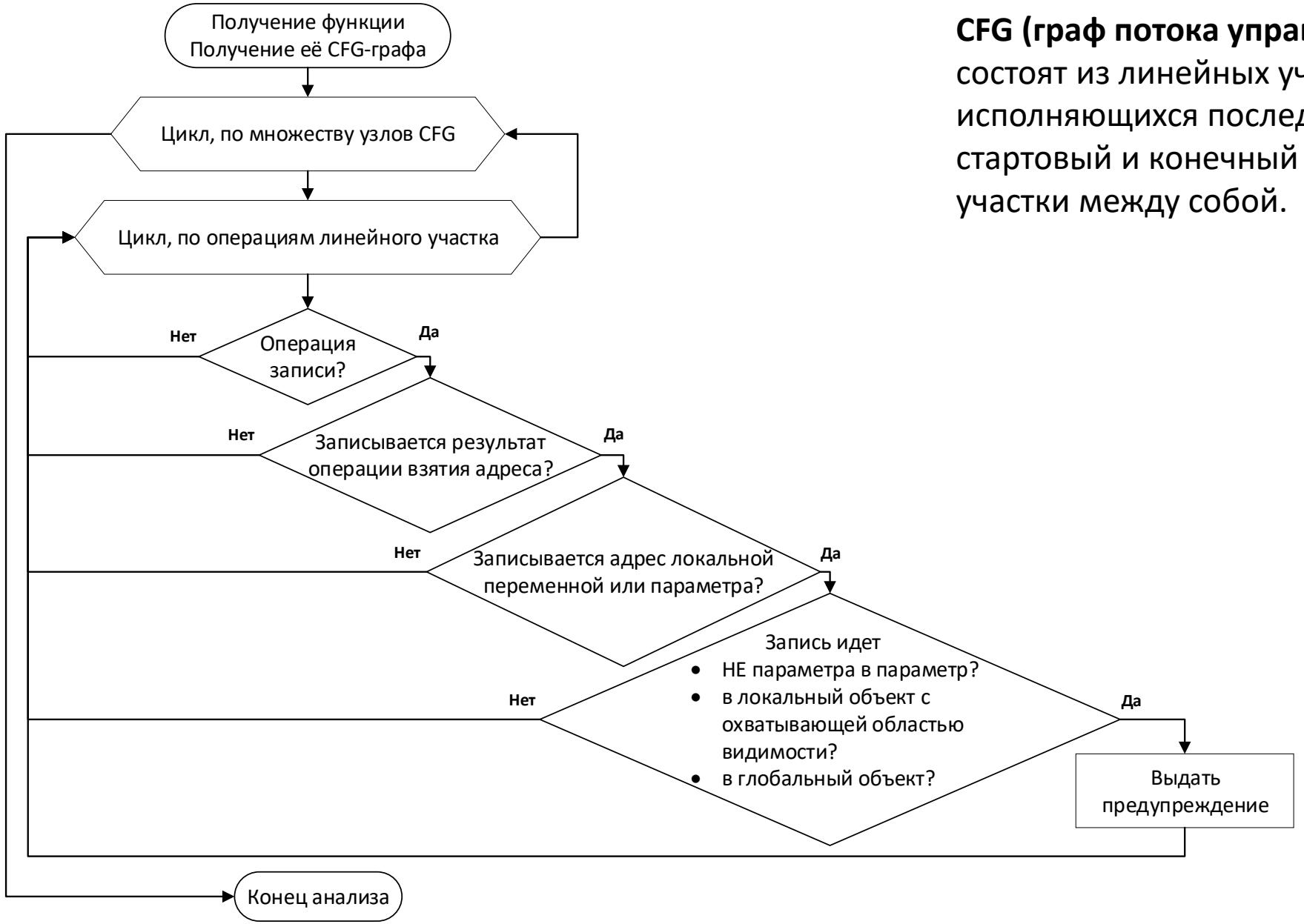
Пример с локальной переменной

```
1  void foo()
2  {
3      int * ptr;
4
5      if(1)
6      {                          Запись адреса локальной переменной
7          int var;             в указатель с большим временем жизни
8          ptr = &var; <-----------------------------------------------------
9      }
10                                  Недопустимое использование
11     *ptr = 1        <-----------------------------------------------------
12 }
13 ------------------------------------------------------------------------
14 Выданное компилятором предупреждение:
15 lcc: "/home/levchenko_d/test.c", line 2: in function "foo":
16 lcc: "/home/levchenko_d/test.c", line 8: warning: address of local variable
17           "var" is written into variable "ptr" with larger scope
18           [-Wmaybe-use-out-of-scope]
```

Пример с глобальной переменной

```
1  int * ptr;
2
3  void foo()
4  {
5      int var;        Запись адреса локальной переменной в глобальный указатель
6      ptr = &var; <-----------------------------------------------------------
7  }
8
9  void bar()
10 {
11     foo();                          Недопустимое использование
12     *ptr = 1;   <-----------------------------------------------------------
13 }
14 ------------------------------------------------------------------------
15 Выданное компилятором предупреждение:
16 lcc: "/home/levchenko_d/test.c", line 4: in function "foo":
17 lcc: "/home/levchenko_d/test.c", line 6: warning: address of local variable
18           "var" is written into global variable "ptr"
19           [-Wmaybe-use-out-of-scope]
```

Задача 1b: алгоритм поиска использования переменных в охватывающей области видимости



CFG (граф потока управления) – граф, узлы которого состоят из линейных участков операций, исполняющихся последовательно. Обязательно содержит стартовый и конечный узел. Рёбра соединяют линейные участки между собой.

Задача 1b: Результаты поиска использования переменных в охватывающей области видимости

Отладка анализа произведена на:

- Наборе самописных направленных тестов (6 шт.);
- Самосборке языкового компилятора;
- Исходниках бенчмарков SPEC CPU (95, 2000, 2006, 2017) для языков C и C++.

Результаты проведения анализа и сравнение с существующими решениями

Статистика собрана на исходниках бенчмарков
SPEC CPU (95, 2000, 2006, 2017) для языков C и C++.

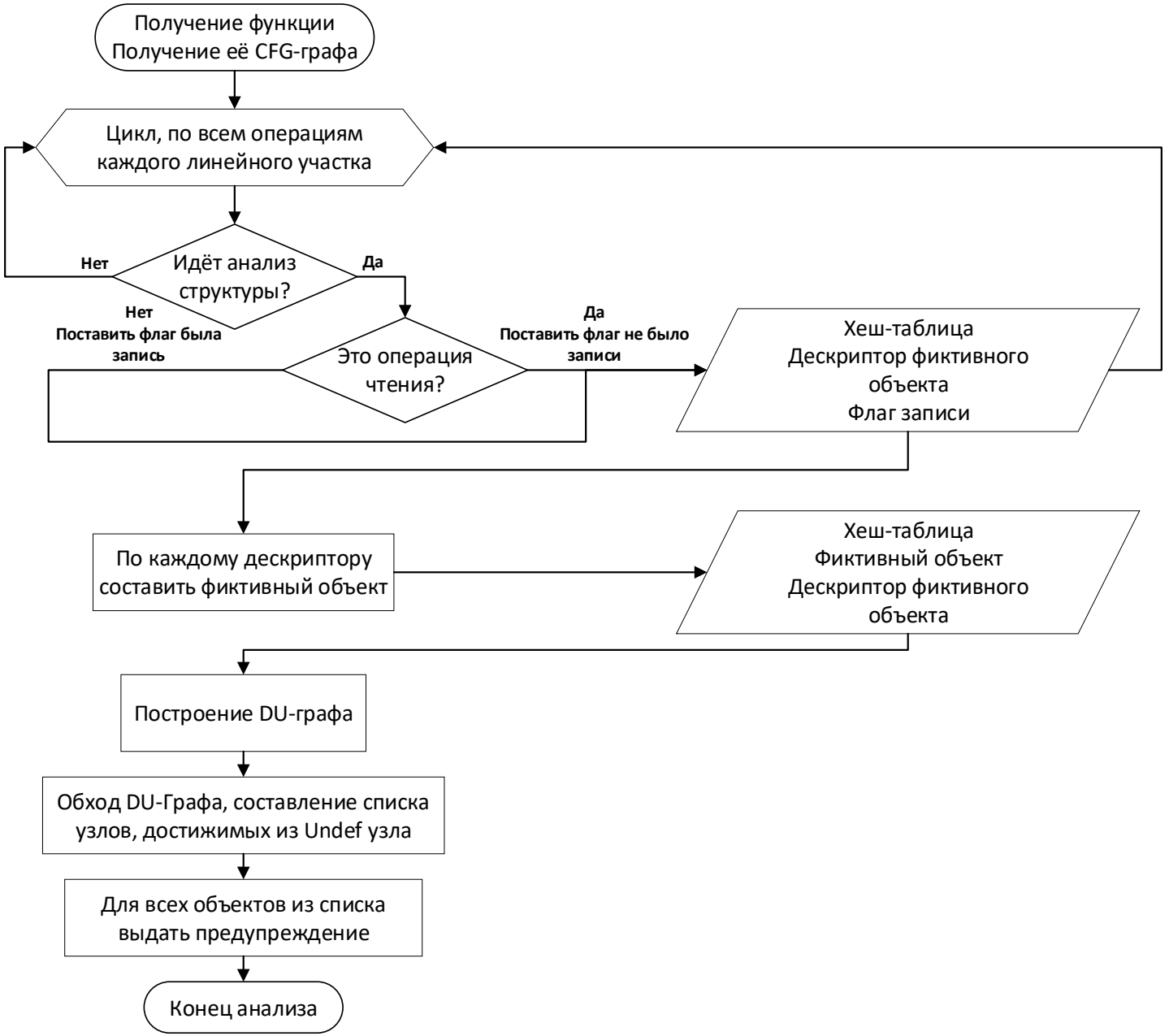
	GCC	Cppcheck	EDG	**LCC**
Количество предупреждений	-	1	-	**96**

Задача 1c: анализ неинициализированных полей структур

Пример

```
1  typedef struct point
2  {
3    int x, y;
4  } point;
5
6  void foo( point * p)
7  {
8    point p_temp;
9                                  Программист забыл инициализировать поле y
10   p_temp.x = 1; <-----------------------------------------------------------
11
12   (*p).x = p_temp.x;          А затем прочитал неинициализированное поле
13   (*p).y = p_temp.y; <------------------------------------------------------
14 }
15 ---------------------------------------------------------------------------
16 Выданное компилятором предупреждение:
17 lcc: "/home/levchenko_d/test.c", line 7: in function "foo":
18 lcc: "/home/levchenko_d/test.c", line 12: warning: struct field "p_temp.y" is used
19            uninitialized [-Wuninitialized]
```

Задача 1с: алгоритм поиска неинициализированных полей структур



DU-граф (граф определений и использований) – граф, в узлах которого идет либо определение объекта, либо его использование. В местах схождения потоков данных строятся phi-узлы.

Phi-узел - это инструкция, используемая для выбора значения в зависимости от того, откуда пришли данные на вход.

Undef узел – phi узел, соответствующий неопределённому значению.

Фиктивный объект – объект, который временно строится для каждого поля структуры для дальнейшего дополнения DU-графа.

Задача 1с: результаты поиска неинициализированных полей структур

Отладка анализа произведена на:

- Наборе самописных направленных тестов (14 шт.);
- Самосборке языкового компилятора;
- Исходниках бенчмарков SPEC CPU (95, 2000, 2006, 2017) для языков Си C++.

Результаты проведения анализа и сравнение с существующими решениями

Статистика собрана на исходниках бенчмарков
SPEC CPU (95, 2000, 2006, 2017) для языков C и C++

	GCC	Cppcheck	EDG	**LCC**
Количество предупреждений	30	-	-	**47**

Результаты

1. Реализован модуль в составе компилятора LCC для статического анализа программ с выдачей пользовательских предупреждений:
   - о неиспользуемых переменных;
   - об использовании переменных в охватывающей области видимости;
   - об использовании неинициализированных полей структур.
2. Анализ был отлажен на коротких направленных тестах, самосборке языкового компилятора и задачах SPEC CPU (95, 2000, 2006, 2017) для языков C и C++.
3. Статистика эффективности анализа собрана на задачах SPEC CPU (95, 2000, 2006, 2017) для языков C и C++ . Продемонстрирован прирост найденных ошибок по сравнению с фронтендом EDG.

Приложение

Найденные неиспользуемые переменные, которые не нашли другие компиляторы

523.xalancbmk_r: TraverseSchema.cpp: 5359

```
1  if (!typeInfo)
2  {
3    if (!XMLString::equals(typeURI, SchemaSymbols::fgURI_SCHEMAFORSCHEMA) ||
4        XMLString::equals(fTargetNSURIString, SchemaSymbols::fgURI_SCHEMAFORSCHEMA))
5    {
6
7      DOMElement* typeNode = fSchemaInfo->getTopLevelComponent(SchemaInfo::C_ComplexType,
8                                                               SchemaSymbols::fgELT_COMPLEXTYPE,
9                                                               localPart,
10                                                              &fSchemaInfo);
11     if (typeNode)
12     {
13       // fBuffer is reused by traverseComplexTypeDecl, so we have to store its current value
14         XMLBuffer buffCopy(fBuffer.getLen()+1, fMemoryManager);
15         buffCopy.set(fBuffer.getRawBuffer());                Переменная объявлена, но неиспользована
16         int typeIndex = traverseComplexTypeDecl(typeNode); <----------------------------------------
17         typeInfo = fComplexTypeRegistry->get(buffCopy.getRawBuffer());
18     }
19   }
20 }
```

Приложение

Найденные возможные использования переменных за областью их жизни, которые не нашли другие компиляторы

177.mesa: eval.c: 2065

```
1  void gl_EvalCoord1f(GLcontext* ctx, GLfloat u)
2  {
3    ...
4    GLubyte *colorptr;
5
6  /** Color **/
7    if (ctx->Eval.Map1Color4) {
8       ...
9    }
10   else {
11      GLubyte col[4];                          Здесь происходит запись адреса локальной переменной
12      COPY_4V(col, ctx->Current.ByteColor );       в переменную с большей областью определения
13      colorptr = col;                        <-----------------------------------------------
14   }
15   ...
16 }
```