А.В. Макаров, С.Ю. Скоробогатов, А.М. Чеповский

Учебный курс “CIL и системное программирование в Microsoft .NET”

Лекция 14.

Преобразование последовательности инструкций в граф потока управления

Введение

- Разработчик любого метаинструмента, использующего граф потока управления в процессе анализа CIL-кода, сталкивается с проблемой преобразования линейной последовательности инструкций в граф потока управления. В данной лекции мы рассмотрим алгоритм такого преобразования
- Наш алгоритм будет работать на уровне метода, то есть в качестве входных данных для него будут выступать тело метода и массив предложений обработки исключений для этого метода. На выходе алгоритма будет построенный граф потока управления

Постановка задачи

- Пусть дан массив инструкций P размера N и массив предложений обработки исключений EH размера M
  - Мы будем предполагать, что в массивах P и EH адреса инструкций предварительно заменены на их номера (это касается встроенных операндов инструкций перехода, а также границ областей в предложениях обработки исключений). Кроме того, все массивы, которые мы будем рассматривать при описании алгоритма, включая массивы P и EH, будут индексироваться, начиная с нуля
- Требуется построить граф потока управления и возвратить ссылку на блок тела метода построенного графа

Набор полей каждого предложения в массиве EH

- Flags.
  - Задает тип обработчика исключений: обработчик с фильтрацией по типу, обработчик с пользовательской фильтрацией, обработчик finally или обработчик fault.
- TryOffset.
  - Номер инструкции в массиве P, с которой начинается защищенная область.
- TryLength.
  - Количество инструкций, входящих в защищенную область.
- HandlerOffset.
  - Номер инструкции в массиве P, с которой начинается область обработчика.
- HandlerLength.
  - Количество инструкций, входящих в область обработчика.
- ClassToken.
  - Токен метаданных, обозначающий тип исключения (используется в случае обработчика с фильтрацией по типу).
- FilterOffset.
  - Номер инструкции в массиве P, с которой начинается область фильтра (используется в случае обработчика с пользовательской фильтрацией)

14.1. Создание массива узлов (слайд 1)

- На первом этапе работы алгоритма мы создаем узел графа для каждой инструкции и формируем из созданных узлов массив
- На входе первого этапа мы имеем массив P. Для каждой инструкции, входящей в массив P, мы создаем соответствующий ей узел графа. В этот узел записывается все данные об инструкции, кроме информации о передаче управления и принадлежности блокам
- Узлы записываются в массив Nodes, состоящий из N элементов

Создание массива узлов (слайд 2)

```
Nodes = новый массив узлов размера N;
for (int i = 0; i < N; i++)
{
    Nodes[i] = новый узел, содержащий информацию
        об инструкции P[i];
}
```

- Следует особо отметить, что для инструкций безусловного перехода также создаются отдельные узлы. Эти узлы являются временными и на последнем этапе алгоритма удаляются из графа
- Инструкцию **nop** мы будем считать инструкцией безусловного перехода по относительному адресу 0.

14.2. Создание дерева блоков

- На втором этапе работы алгоритма мы строим дерево блоков на основе информации, находящейся в массиве предложений обработки исключений
- На входе второго этапа мы имеем массив EH. На выходе мы получаем дерево блоков и вспомогательный массив B, связывающий блоки с информацией о диапазонах входящих в них инструкций (другими словами, с информацией об областях кода).
- Каждый элемент массива B будет состоять из трех полей:
- Поле block.
  - Это поле содержит ссылку на блок.
- Поле offset.
  - Содержит целое число, обозначающее индекс первой инструкции блока в массиве P.
- Поле length.
  - Содержит количество инструкций, входящих в блок (длина блока).

Создание дерева блоков: инициализация

- Сначала создадим блок тела метода (назовем его MBB). Этот блок будет являться корнем дерева, которое мы строим. К нему в дальнейшем будут "прицеплены" все остальные узлы графа, и именно его наш алгоритм будет возвращать в результате своей работы. Для блока MBB в массив B добавляется запись, поле start которой содержит значение 0, а поле length – значение N.

```
MBB = новый блок тела метода, в который записана информация о мет
    имя метода, сигнатура и данные о типах локальных переменных;
B = новый массив размера 2*M+1 для хранения информации о блоках;
B[0].block = MBB;  B[0].start = 0;  B[0].length = N;
BN = 1;
```

Создание дерева блоков: главный цикл

- Напомним, что предложения обработки исключений расположены в массиве EH в определенном порядке, гарантирующем, что более вложенный блок находится ближе к началу массива, чем объемлющий его блок.
- Так как нам требуется строить дерево блоков в направлении от корня к листам, то есть от менее вложенных блоков к более вложенным, то мы будем просматривать массив EH от конца до начала

```
for (int i = M-1; i >= 0; i--)
```

Создание дерева блоков: поиск в массиве B блока, диапазон инструкций которого содержит защищенную областью i-го предложения

- Мы перебираем элементы массива B в обратном порядке, начиная с последнего элемента, поэтому найденный блок будет самым вложенным из блоков, диапазон инструкций которых содержит защищенную область i-го предложения

```
int tryOffset = EH[i].TryOffset, tryLength = EH[i].TryLength;
for (int j = BN-1; j >= 0; j--)
    if (tryOffset >= B[j].offset &&
        tryOffset+tryLength <= B[j].offset+B[j].length)
        break;
```

Создание дерева блоков: создание нового защищенного блока и добавление его в дерево

```
if (tryOffset != B[j].offset || tryLength != B[j].length)
{
    B[BN].block = новый защищенный блок;
    Сделать блок B[j].block родителем блока B[BN].block;
    B[BN].offset = tryOffset;
    B[BN].length = tryLength;
    j = BN;  BN++;
}
```

Создание дерева блоков: создание блока обработки исключений

```
B[BN].block = новый блок обработки исключений, тип которого
    определяется значением EH[i].Flags;
Сделать родителем блока B[BN].block тот же блок, который являет
    родителем блока B[j].block;
Добавить блок B[BN].block в конец списка обработчиков,
    ассоциированных с блоком B[j].block;
B[BN].offset = EH[i].HandlerOffset;
B[BN].length = EH[i].HandlerLength;
BN++;
```

Создание дерева блоков: создание блока фильтрации

```
if (EH[i].Flags обозначает блок с пользовательской фильтрацией)
{
    B[BN].block = новый блок фильтрации;
    Сделать блок B[BN-1].block родителем блока B[BN].block;
    B[BN].offset = EH[i].FilterOffset;
    B[BN].length = EH[i].HandlerOffset - EH[i].FilterOffset;
    BN++;
}
```

14.3. Присвоение родительских блоков узлам графа

- На третьем этапе узлы, соответствующие инструкциям, получают родителей. Тем самым завершается построение дерева блоков

```
for (int i = 0; i < BN; i++)
{
    for (int j = B[i].offset; j < B[i].offset+B[i].length; j++)
        Сделать блок B[i].block родителем узла Nodes[j];
}
```

14.4. Формирование дуг (слайд 1)

- На четвертом этапе в граф потока управления добавляются дуги, обозначающие передачу управления между инструкциями

```
blockNodes = новый массив ссылок на узлы размера N, изначальнo
    заполненный нулевыми ссылками;
for (int i = 0; i < BN; i++)
{
    Провести дугу от B[i].block к Nodes[B[i].offset];
    blockNodes[B[i].offset] = B[i].block;
}
```

Формирование дуг (слайд 2)

```
for (int i = 0; i < N; i++)
{
    int[] Flow = новый массив, в который добавляются номера
        инструкций, на которые может быть передано управление
        от инструкции, соответствующей узлу Nodes[i];
    for (int j = 0; j < Flow.Length; j++)
    {
        int n = Flow[j];
        if (blockNodes[n] != null && blockNodes[n] не является
            непосредственно или транзитивно родителем узла Node
            Провести дугу от Nodes[i] к blockNodes[n];
        else
            Провести дугу от Nodes[i] к Nodes[n];
    }
}
```