

# Кросс-процедурный анализ потока управления

**Дятлов Андрей**

**C# Developer, JetBrains**

# Обо мне

- Занимаюсь поддержкой языка C# в ReSharper с 2015 года
  - Анализаторы кода, рефакторинги
  - Поддержка новых версий языка
- Ищу баги в Roslyn

—  **Julien Couvreur**  
@jcouv

Still trying to catch up on the series of nullable issues filed by @a\_tessenr  
They're interesting cases, clear and detailed.  
Thanks!  
[github.com/dotnet/roslyn/](https://github.com/dotnet/roslyn/) ...

# План доклада

- Анализ потока данных
- Сложность кросс-процедурного анализа
- Локальные функции
- Алгоритм сбора данных для анализа
- Пишем свою инспекцию
- Применение для анализа всего проекта
- Работа с графом вызовов

# Где используется кросс-процедурный анализ?

- В компиляторе
  - Nullable reference types
- В рефакторингах
  - Extract method
  - Inline
- В статических анализаторах кода
- В небольших модификациях кода
  - Закашировать значение переменной
  - Объединить переменные

```
private void ReplayReadsAndWrites(  
    LocalFunctionSymbol localFunc,  
    SyntaxNode syntax,  
    bool writes)  
{  
    // https://github.com/dotnet/roslyn/issues/27233  
}
```



<https://bit.ly/2VlXvO4>

# Примеры кросс-процедурных инспекций

```
void M(SomeType arg) {  
    if (arg == null) return;
```

```
    LocalFunction();
```

```
    arg = GetValue();
```

```
    LocalFunction();
```

```
    // possible NRE because of the second call
```

```
    void LocalFunction() => arg.DoSomething();
```

```
}
```

```
[CanBeNull] SomeType GetPossibleNullValue() => null;
```

# Примеры кросс-процедурных инспекций

```
object Method(bool arg) {  
    if (!arg)  
        return Local();  
  
    return arg && Local2();  
  
    // expression is always false  
    object Local() => arg ? GetSomething() : Nothing();  
  
    // expression is always true  
    bool Local2() => arg || SomeCondition();  
}
```

# Примеры кросс-процедурных инспекций

```
IEnumerable<IEnumerable<T>> GetEnumerables<T>(Func<Resource, T> generator)
{
    using (var disposable = new Resource())
    {
        IEnumerable<T> NestedIterator()
        {
            // access to disposed closure
            yield return generator(disposable);
        }

        yield return NestedIterator();
    }
}
```



<https://bit.ly/2H2zFy1>

# Что такое data-flow анализ?

- Возможные наборы значений переменных и выражений в каждой точке программы
  - Possible NRE / Expression is never null
  - Expression is always / never of type
  - Unreachable code
- Границы достижимости значений
  - Assigned value is never used
  - Access to modified closure

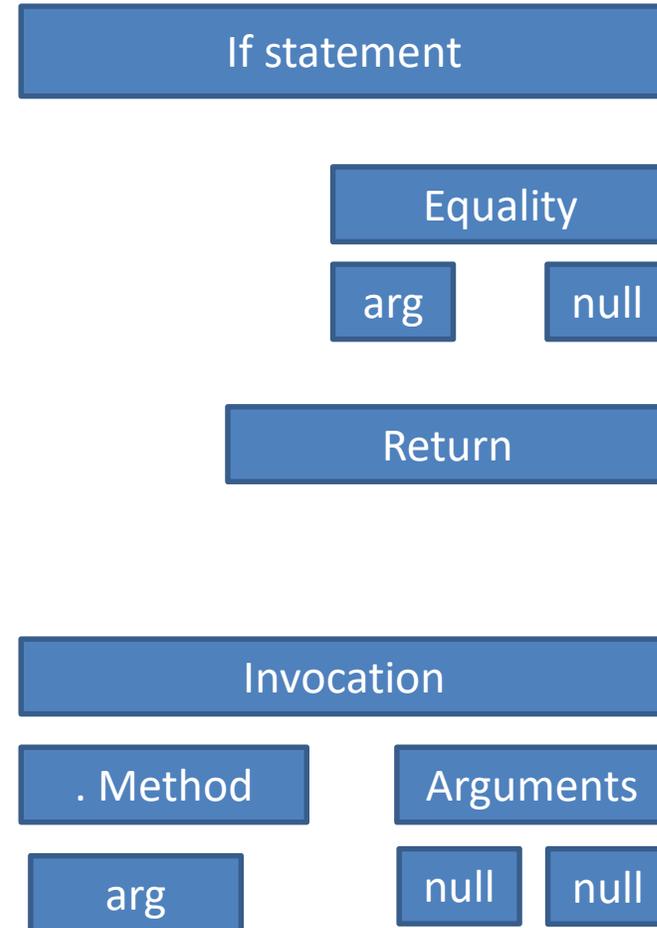
# Как data-flow анализ работает в ReSharper?

- Абстрактные значения переменных
  - [NotNull], [CanBeNull], true, false
  - [NotNull, ItemCanBeNull]
- Построение графа потока управления
- Интерпретация графа в терминах абстрактных значений
- Поиск фиксированной точки анализа

# AST – abstract syntax tree

```
void Method(T arg)
{
  if (arg == null)
  {
    return;
  }

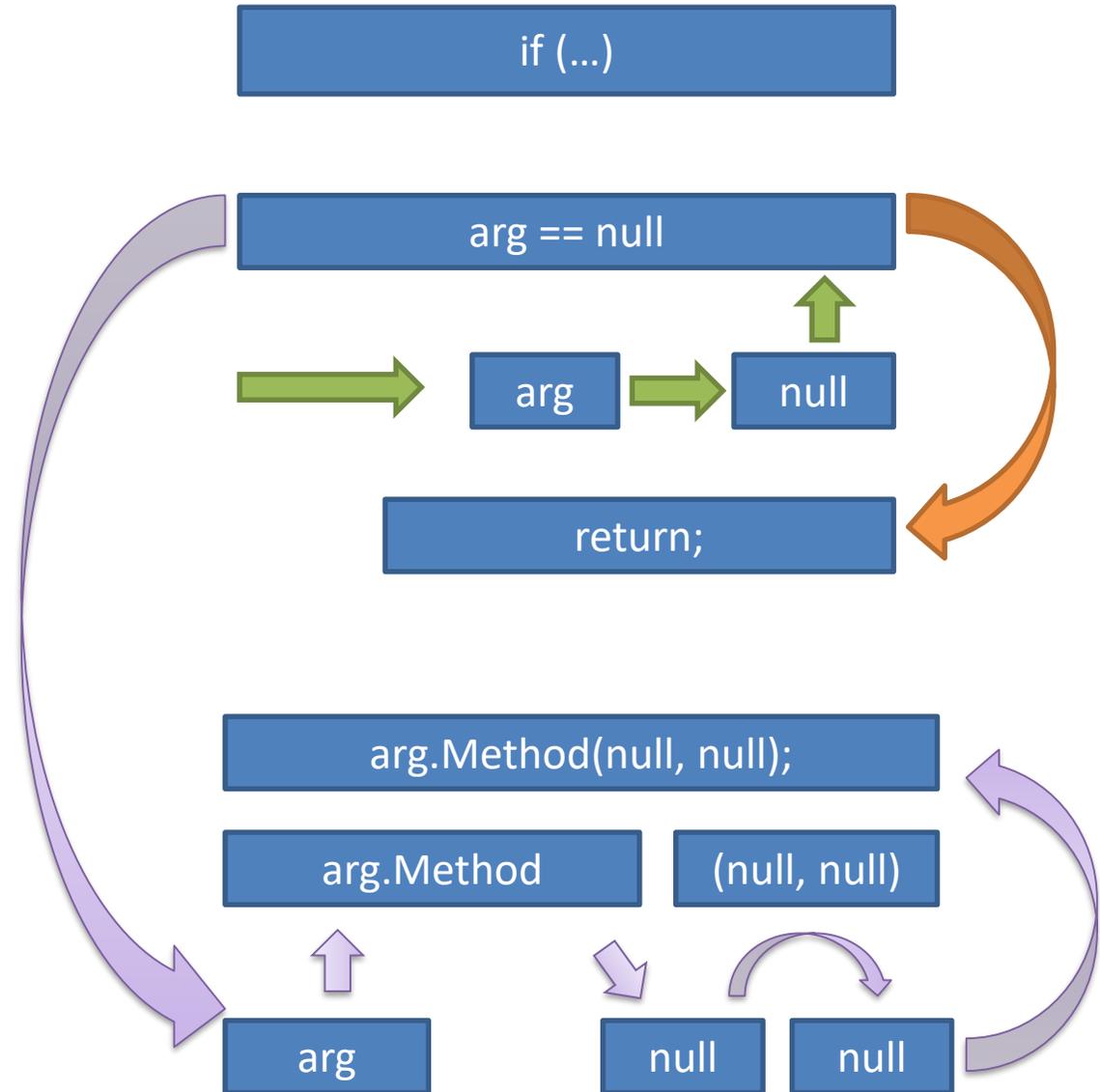
  arg.Method(null, null);
}
```



# Построение графа потока управления

```
void Method(T arg)
{
    if (arg == null)
    {
        return;
    }

    arg.Method(null, null);
}
```



# Обработка ветвлений

```
void Method([NotNull] T arg) {
```

$\text{null} \notin \mathit{arg}$



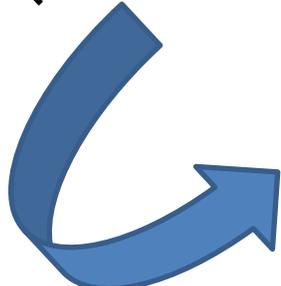
```
if (condition) → { arg = null; }
```

$\text{null} \notin \mathit{arg}$



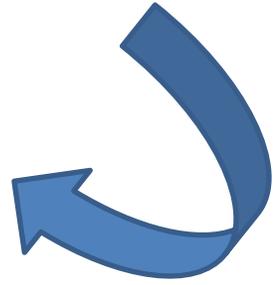
```
{ arg = null; }
```

$\text{null} \notin \mathit{arg}$



Join arg states

$\text{null} \in \mathit{arg}$



$\text{null} \in \mathit{arg}$

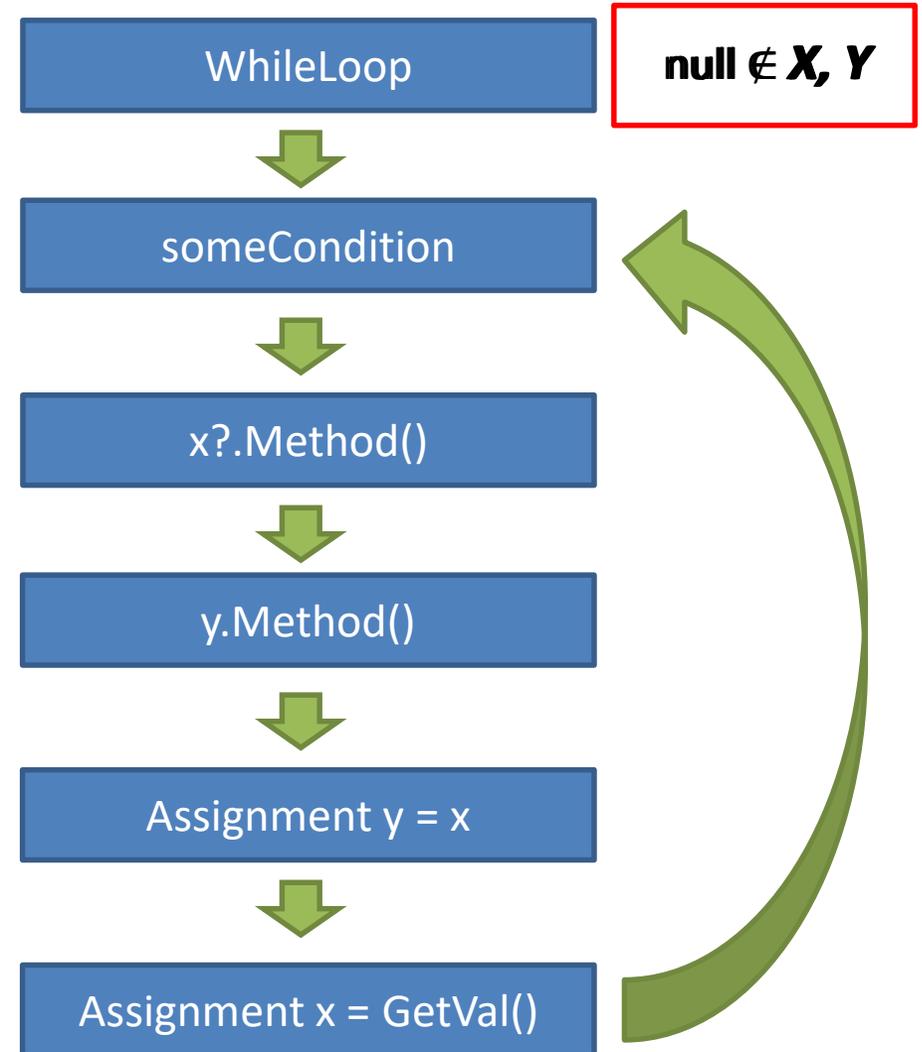


```
arg.SomeMethod(); // possible NRE
```

```
}
```

# Фиксированная точка анализа

```
void M([NotNull] T x,  
      [NotNull] T y) {  
    while (someCondition) {  
        x?.Method(); // redundant ?  
        y.Method(); // possible NRE  
        y = x;  
        x = GetNewValue();  
    }  
}  
[CanBeNull] T GetNewValue() => null;
```



# Фиксированная точка анализа

```
void M([NotNull] T x,  
       [NotNull] T y) {  
    while (someCondition) {  
        x?.Method();  
        y.Method(); // possible NRE  
        y = x;  
        x = GetNewValue();  
    }  
}  
[CanBeNull] T GetNewValue() => null;
```

- 1 итерация
  - [NotNull] x
  - [NotNull] y
- 2 итерация
  - [CanBeNull] x
  - [NotNull] y
- 3-∞ итерация
  - [CanBeNull] x
  - [CanBeNull] y

# Визитор для выражения графа

```
void VisitMethodCall(ICall call) {  
    Visit(call.Qualifier);  
    foreach (var arg in call.Arguments) {  
        Visit(arg);  
    }  
    InspectTheCallItself(call);  
}
```

# Визитор для выражения графа

```
namespace Microsoft.CodeAnalysis.CSharp
```

```
{
```

```
    internal class ControlFlowPass : AbstractFlowPass<ControlFlowPass.LocalState>
```

```
    {
```

```
        private rea
```

```
        private rea
```

```
        protected b
```

Base types of 'ControlFlowPass'

 **AbstractFlowPass<TLocalState>** (in Microsoft.CodeAnalysis.CSharp) Microsoft.CodeAnalysis.CSharp

 BoundTreeVisitor (in Microsoft.CodeAnalysis.CSharp) Microsoft.CodeAnalysis.CSharp

# А зачем тогда вообще граф?

## Граф

- нужно строить граф
- +/- состояние на ребро
- + таймстепы ребер  
повторный анализ  
только циклов

## Визитор

- + не требуется строить граф
- +/- одно «текущее»  
состояние переменных
- дополнительные контексты  
в переменных на стеке  
и словарях
- нельзя отследить что  
переанализировать

В чем



Grzegorz Galezowski  
@GGalezowski

Follow

рным

Why does @roslyn issue a warning in this case? The ItemGroups is a nullable ref type, but it's assigned in the constructor and there is no way in my class to nullify it...

```
public class XmlProjectBuilder
{
    private readonly XmlProject _xmlProject;

    public XmlProjectBuilder(string assemblyName)
    {
        _xmlProject = new XmlProject
        {
            AbsolutePath = AbsolutePathTo(assemblyName),
            PropertyGroups = new List<XmlPropertyGroup>
            {
                new XmlPropertyGroup
                {
                    AssemblyName = assemblyName
                }
            },
            ItemGroups = new List<XmlItemGroup>()
        };

        public void WithReferences(params string[] names)
        {
            _xmlProject.ItemGroups.Add(
                new XmlItemGroup { Name = "References", From(names) }
            );
        }

        public void WithPa
    {

```

4:43 PM - 27 Apr 2019

<https://bit.ly/2Wrrf8U>



Но ведь я присвоил только `[NotNull]` в конструкторе?

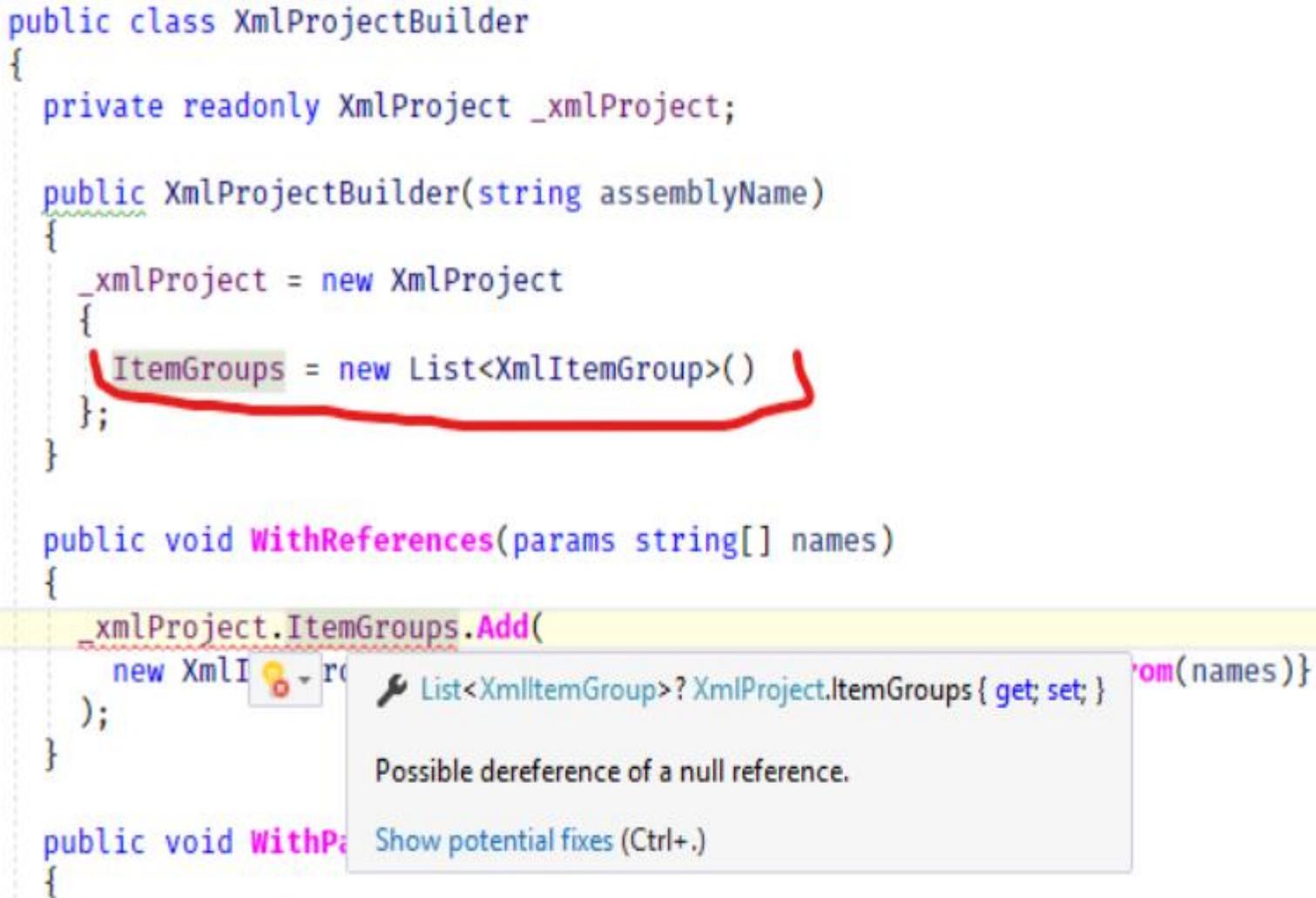
```
public class XmlProjectBuilder
{
    private readonly XmlProject _xmlProject;

    public XmlProjectBuilder(string assemblyName)
    {
        _xmlProject = new XmlProject
        {
            ItemGroups = new List<XmlItemGroup>()
        };
    }

    public void WithReferences(params string[] names)
    {
        _xmlProject.ItemGroups.Add(
            new XmlItemGroup { Name = names[0], Path = Path.Combine(names) }
        );
    }

    public void WithPath(string path)
    {

```



# В чем проблема с кросс-процедурным анализом?

- Кто мог поменять это свойство?

```
[XmlElement(ElementName = "ItemGroup")]  
public List<XmlItemGroup>? ItemGroups { get; set; }
```

- Методы на объекте вызывались?
- Сам объект из методов возвращался?
- В конструкторе исключения возникали?
- Все конструкторы инициализируют одинаково?
- Присвоение было через инициализатор?
- Свойство автоматическое?
- Есть наследники перегружающие сеттер?

# В чем проблема с кросс-процедурным анализом?

- Большой объем кода
- Неизвестные статически вызовы
  - interface
  - virtual / abstract
  - dynamic
- Рекурсия

# Как не работают кросс процедурные анализы в ReSharper?

```
[CanBeNull] SomeType field = null;
void Method() {
    if (field != null) {
        AnotherMethod(); // can it override field?
        field.SomeMethod(); // can it be null here?
    }
}
```



: Let's hope not!

<https://bit.ly/2WqXDYW>

# Как не работают кросс процедурные анализы в Roslyn?



```
struct MyStruct {  
    int x, y;  
    public MyStruct() {  
        x = 0;  
        Console.WriteLine(x); // this works  
        UseX(); // but this doesn't  
        y = 0;  
    }  
  
    void UseX() => Console.WriteLine(x);  
}
```

# Language design notes

```
class C {  
    string? Field1;  
    void M1(C c) {  
        if (c.Field1 != null)  
            c.Field1.Equals(...);  
    }  
}
```



<https://bit.ly/2VeBsZE>

# Language design notes

```
class C {  
    string? Field1;  
    void M1(C c) {  
        if (c.Field1 != null)  
            M2(c);  
    }  
}
```

```
void M2(C c) {  
    // The null checking from M1 is lost here and M2 has to  
    // check again for null to avoid a warning  
    c.Field1.Equals(...);  
}
```



<https://bit.ly/2VeBsZE>

# Существующие кросс-процедурные анализы

```
class C {  
    // no warnings, both field initialized  
    readonly string field1;  
    readonly string field2;  
  
    public C() : this("a") {  
        field2 = "b";  
    }  
  
    private C(string f1) { field1 = f1; }  
}
```



# Кросс процедурный анализ на примере локальных функций

- Анализируется как часть метода
  - Инициализирует переменные
  - Изменение кода может сломать компиляцию
- Замыкания
  - Зависимости между функцией и внешним методом
  - Неожиданное изменение значений

# Что такое локальные функции?

```
void M() {  
    int x = 0;  
    Console.WriteLine(x); // 0  
    Local();  
    Console.WriteLine(x); // 1  
  
    void Local() => x++;  
}
```

# Что изменилось с появлением локальных функций?

```
void Method() {  
    int x;  
    AssignX(); ← x is assigned now  
    Console.WriteLine(x);  
  
    void AssignX() => x = 0;  
}
```

## В чем отличие от лямбд?

- Лямбда не является частью метода
- Может быть обобщенной
- Может быть итератором
- Допускает рекурсию
  - Без замыкания на временную переменную

# Отличия локальных функций от методов

- Все вызовы известны статически – нет виртуальных вызовов
- Ограниченный объем кода – нет вызовов в соседние классы и файлы которые могут быть еще не проанализированы

# План доклада

- Анализ потока данных
- Сложность кросс-процедурного анализа
- Локальные функции
- ***Алгоритм сбора данных для анализа***
- Пишем свою инспекцию
- Применение для анализа всего проекта
- Работа с графом вызовов

# Объединить графы метода и функции?

```
void Method(out int x) {
```

```
    Local();
```

```
    x = 0;
```

```
    Local();
```

```
}
```

```
void Local() => Smth();
```

X не инициализирован

# Заинлайнить граф функции?

```
void Method() {  
    Fibonacci(5);  
}
```



```
int Fibonacci(int n) {  
    if ( n == 0 ) return 0;  
    else if ( n == 1 ) return 1;  
    else return Fibonacci(n-1)  
           + Fibonacci(n-2);  
}
```

```
int Fibonacci(int n) {  
    if ( n == 0 ) return 0;  
    else if ( n == 1 ) return 1;  
    else return Fibonacci(n-1)  
           + Fibonacci(n-2);  
}
```



```
int Fibonacci(int n) {  
    if ( n == 0 ) return 0;  
    else if ( n == 1 ) return 1;  
    else return Fibonacci(n-1)  
           + Fibonacci(n-2);  
}
```



```
int Fibonacci(int n) {  
    if ( n == 0 ) return 0;  
    else if ( n == 1 ) return 1;  
    else return Fibonacci(n-1)  
           + Fibonacci(n-2);  
}
```



# Составление резюме о функции

```
void Method() {  
    int x;  
    if (condition) {  
        x = 0;  
        ReadX();  
    }  
    AssignX();  
    Console.WriteLine(x);  
  
void AssignX() => x = 0;  
void ReadX() => Console.WriteLine(x);  
}
```



**AssignX:**  
Запись: { X }  
Чтение: ∅ { }



**ReadX:**  
Запись: ∅ { }  
Чтение: { X }

# Составим требования

- Информация о переменных
  - какие переменные были записаны?
  - какие переменные были прочитаны?
  - какие переменные попали в замыкания?
  - какие переменные возможно были записаны?
- Обработка рекурсии
- Производительность

# Как собирается информация о функции?

```
void Local() {
```

```
  a = GetValue();
```

← a присвоено

```
  if (a != null)
```

← Нет записи/чтения

```
    b = GetValue();
```

← a, b присвоено

← a присвоено,  
b возможно присвоено

```
  c++;
```

← Запись / чтение c

```
}
```

Local:

Запись: { a, c }

Чтение: { c }

Возможная запись: { b }

# Как обрабатывать вложенные вызовы и несколько точек выхода?

- `void Local() => Local2();`
  - Повторный вызов функции сбора резюме
  - Применение данных из дочерней функции в месте вызова
- `object Local() { if (...) return x; else return y; }`
  - Создание информации о точке выхода
  - Объединение информации в резюме (аналогично объединению при ветвлении)

# Пишем сбор данных о функции

```
Dictionary<Function, Resume> _resumes;
```

```
void OnLocalFunctionCall(Function func)  
=> ApplyResume(GetOrCreateResume(func));
```

```
Resume GetOrCreateResume(Function func)  
=> _resumes.GetOrCreate(  
    func, () => CollectResume(func));
```

# Работает?

```
void Local(int x)
{
    if (x > 0)
        Local(x - 1);
}
```

- GetOrCreateResume(Local)
- CollectResume(Local)
- OnLocalFunctionCall(Local)

# Игнорируем проблему

- Переменные затронутые рекурсивным вызовом эквивалентны тем которые встретятся при нормальном исполнении функции

```
void Local() { ← Запись X, чтение Y
  x = 0;
  if (someCondition) { + = Запись X, чтение Y
    Local(); ← Запись X, чтение Y
  }
  y.Read();
}
```

```
HashSet<Function> _callStack;
```

```
Resume GetOrCreateResume(Function func)  
{  
    if (_resumes.TryGetValue(func, out var resume))  
        return resume;
```

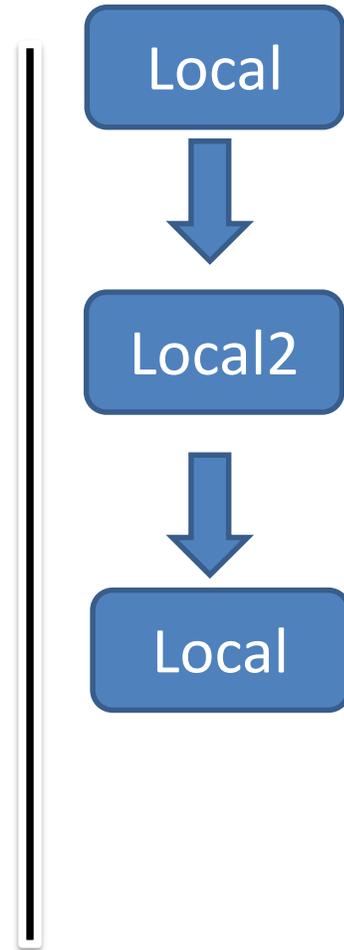
```
    if (_callStack.Add(func)) {  
        _resumes[func] = CollectResume(func);  
        _callStack.Remove(func);  
        return _resumes[func];
```

```
    } else  
        return Resume.RecursiveInfo;
```

```
}
```

# Работает?

```
void Local() {  
    if (smth) Local2();  
  
    closure = GetValue();  
}  
  
void Local2() => Local();
```



запись в 'closure'  
↑  
нет записи  
в переменные  
↑  
=> рекурсия

# Один стэк не работает? Добавь два!

```
void Local() => Local2();  
void Local2() => Local3();  
void Local3() => Local4();  
  
void Local4() {  
    if (smth) Local2();  
    closure = GetValue();  
}
```

Стек вызовов

Local

Local2

Local3

Local4

Стек анализа

Local

Local2

Local3

Local4



```
Stack<Function> _calls, _analysisStack;
```

```
Resume GetOrCreateResume(Function func) {  
    if (_resumes.TryGetValue(func, out var resume))  
        return resume;  
    if (!_calls.Contains(func)) {  
        _calls.Push(func);  
        resume = TryCacheResume(func);  
        _calls.Pop();  
        return resume;  
    } else {  
        UnwindAnalysisStack();  
        return Resume.RecursiveInfo;  
    }  
}
```

# Добавляем стек анализа

```
Resume TryCacheResume(Function func)
```

```
{
```

```
    if (_calls.Count == _analysisStack.Count)  
        _analysisStack.Push(func);
```

```
    var resume = CollectResume(func);
```

```
    if (_analysisStack.Peek() == func) {  
        _resumes[func] = resume;  
        _analysisStack.Pop();  
    }
```

```
    return resume;
```

```
}
```

# Добавляем стек анализа

```
void UnwindAnalysisStack(Function func) {  
    if (_analysisStack.Contains(func)) {  
        while (_analysisStack.Peek() != func)  
            _analysisStack.Pop();  
    }  
}
```

# Работает? Да, только медленно...

```
void Local1() {  
    Local2();  
    Local3();  
    Local2();  
    Local3();  
}  
  
void Local2() {  
    if (smth) Local3();  
}  
  
void Local3() => Local1();
```

Текущий стек

Local1

Local2

Local3

Закэшировано

Local1

Проанализировано

Local1

Local2

Local3

Local3

Local2

Local3

Local3

# Алгоритмическая сложность

```
void Local1() {  
    Local1();  
    ...  
    LocalN();  
}
```

... ..

```
void LocalN() {  
    Local1();  
    ...  
    LocalN();  
}
```

{ 1 ... N } top level functions

TopLevel = RecursiveInfo

{ 2 ... N } calls each has

SecondLevel = RecursiveInfo

{ 3 ... N } calls...

# Алгоритмическая сложность

```
void Local1() {  
    Local1();  
    ...  
    LocalN();  
}
```

... ..

```
void LocalN() {  
    Local1();  
    ...  
    LocalN();  
}
```

1 -> 2 -> 3 -> 4

1 -> 2 -> 4 -> 3

1 -> 3 -> 2 -> 4

1 -> 3 -> 4 -> 2

1 -> 4 -> 2 -> 3

1 -> 4 -> 3 -> 2

В стек вызовов попадут все перестановки из N по N без повторений

$$P(N, N) = N!$$

# Добавим временный кэш

```
void Local1() {  
    Local1();  
    ...  
    LocalN();  
}
```

... ..

```
void LocalN() {  
    Local1();  
    ...  
    LocalN();  
}
```

1 -> 2 -> 3 -> 4 -> 1

1 -> 2, 3, 4 (cached)

Кэшируем: 1, Сбрасываем: 2, 3, 4

2 -> 3 -> 4 -> 2

2 -> 1, 3, 4 (cached)

Кэшируем: 1, 2, Сбрасываем: 3, 4

3 -> 4 -> 3

3 -> 1, 2, 4 (cached)

```
Resume TryCacheResume(Function func)
{
    if (_tempCache.TryGetValue(func, out var resume))
        return resume;

    if (_calls.Count == _analysisStack.Count)
        _analysisStack.Push(func);

    resume = CollectResume(func);
    if (_analysisStack.Peek() == func) {
        _resumes[func] = resume;
        _analysisStack.Pop();
        _tempCache.Clear();
    }
    else _tempCache[func] = resume;
    return resume;
}
```

# Алгоритмическая сложность с кэшем

```
void Local1() {  
    Local1();  
    ...  
    LocalN();  
}
```

... ..

```
void LocalN() {  
    Local1();  
    ...  
    LocalN();  
}
```

- Для заполнения временного кэша потребуется не более 1 прохода каждой функции
- Анализ 1 функции = N функций пройдено 1 раз
- Анализ 2 функции = N-1 функций пройдено 1 раз
- $n + (n-1) + \dots + 2 + 1$
- $n(n+1)/2 = O(n^2)$

# Когда это перестает работать?

- Не анализируется зависимость между состоянием замыканий и возвращаемым значением

```
void M([NotNull] SomeType s) {  
    s = ReturnClosure();  
    s.CanItBeNullOrNot();  
    SomeType ReturnClosure() => s;  
}
```

# Как можно улучшить анализ?

- Кросс-процедурный поиск фиксированной точки
- Уравнение зависимости выходных параметров от входных

# Окей, и зачем все это нужно?

- Рефакторинги
- Статический анализ
- Модификации кода
  - Объединить переменные
  - Поменять местами инструкции
  - Закэшировать значение

- `Transform.position`



<https://bit.ly/2VPXGjY>

# Маленькая но кросс процедурная инспекция

```
Vector3 Method(Transform t) {  
    if (t.position.x > 0)  
        DoSomething(t.position.y);  
    else {  
        var sum = t.position.x + t.position.y;  
        ProcessSum(sum);  
        UpdatePosition(t.position);  
    }  
    return t.position;  
    void DoSomething(int y) => ProcessSum(y);  
    void ProcessSum(int sum) {  
        if (sum < 5) t.position = new Vector3(0,0,0);  
    }  
}
```

# Как написать кэширование?

- Найти точку инвалидации кэша
  - Пройти дальше по методу анализируя код
  - Искать запись в кэшируемую переменную
- Создать переменную под кэш
- Заменить вызовы на обращения к ней

# Инвалидация кэша

```
Vector3 Method(Transform t) {  
    if (t.position.x > 0)  
        DoSomething(t.position.y);  
    else {  
        var sum = t.position.x + t.position.y;  
        ProcessSum(sum);  
        UpdatePosition(t.position);  
    }  
    return t.position;  
}  
void DoSomething(int y) => ProcessSum(y);  
void ProcessSum(int sum) {  
    if (sum < 5) t.position = new Vector3(0,0,0);  
}
```

# Где еще можно применить подобный анализ/модель?

```
void Method(object arg) {  
    if (arg is string) {  
        DoSmth();  
    }  
  
    if (arg is int) {  
        DoSmthElse();  
    }  
  
    if (arg is bool) { }  
}
```

```
void DoSmth() => DoSmthElse();  
void DoSmthElse() => Console.WriteLine(arg);  
}
```

```
switch (arg) {  
    case string _:  
        DoSmth();  
        break;  
    case int _:  
        DoSmthElse();  
        break;  
    case bool _:  
        break;  
}
```

# Это все только про локальные функции?

- Локальные функции:
  - Внешний метод
  - Переменные
  - Локальные функции
- Любая программа
  - Класс
  - Поля и свойства
  - Методы класса

# Это все только про локальные функции?

```
void Method() {  
  
    var x = 1;  
    F1();  
  
    void F1() => F2();  
    void F2() => WriteLine(x++);  
}
```

```
class C  
{  
    int x;  
  
    void Root()  
    {  
        x = 1;  
        M1();  
    }  
  
    void M1() => M2();  
    void M2() => WriteLine(x++);  
}
```

# Что еще нужно учесть для обычных методов?

- Виртуальные вызовы
  - virtual / abstract
  - Interface
  - Dynamic
- Граф вызовов может включать в себя всю программу

# Виртуальные вызовы

```
abstract class C
{
    protected string x;

    public void Root() {
        if (x == null) return;

        Virtual();
        WriteLine(x.Length);
    }

    protected abstract void Virtual();
}
```

```
class D1 : C {
    override Virtual()
        => x = null;
}

class D2 : C {
    override Virtual()
        => Console.WriteLine(x);
}
```

# Виртуальные вызовы

```
class D1 : C {  
    override Virtual()  
        => x = null;  
}
```

```
class D2 : C {  
    override Virtual()  
        => Console.WriteLine(x);  
}
```

Запись x

abstract Virtual();

Запись x  
Чтение x

Чтение x

# Какие еще есть проблемы?

- Публичные наследуемые классы в сборке
  - Виртуальный вызов потенциально может изменить любое поле к которому имеет доступ
- Динамические вызовы
- Доступ по ссылке (ref / out)
- Алиасы – зависимости между объектами

```
void M(C c, C other) {  
    if (c.field != null) {  
        other.field = null;  
        c.field.ToString();  
    }  
}
```

# План доклада

- Анализ потока данных
- Сложность кросс-процедурного анализа
- Локальные функции
- Алгоритм сбора данных для анализа
- Пишем свою инспекцию
- Применение для анализа всего проекта
- ***Работа с графом вызовов***

# В каком порядке анализировать программу?

- Для анализа самих функций потребуется собрать данные об их замыканиях на момент вызова
  - Можно сохранять данные для каждого вызова и объединять их аналогично объединению при ветвлении
- Но эта информация может быть не полной для функций которые вызываются из других локальных функций

# Как данные попадают в модель?

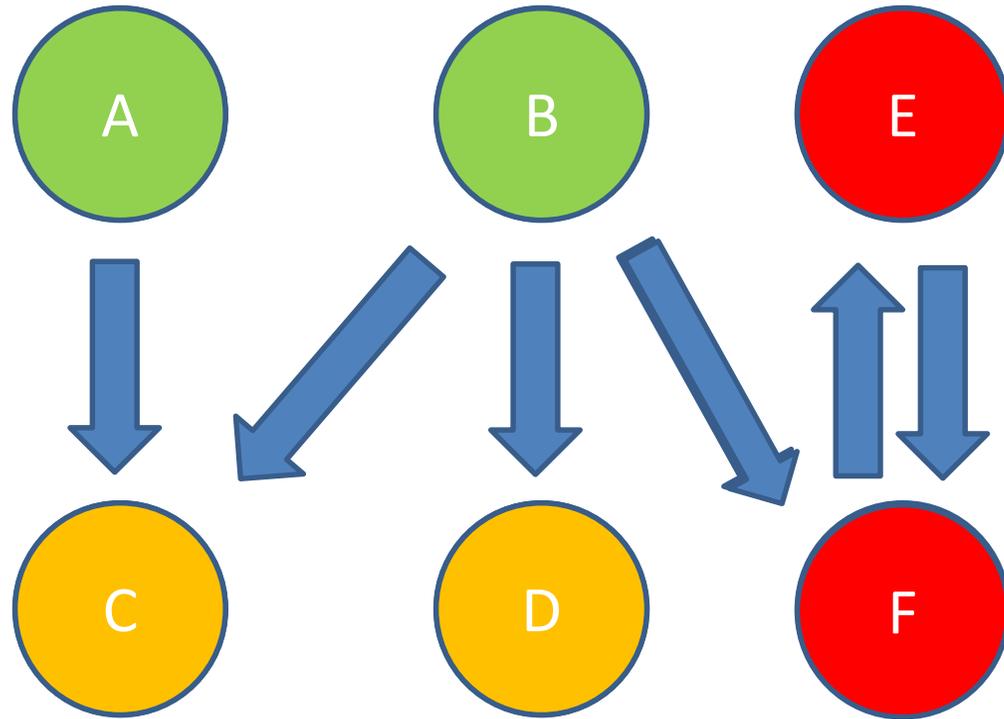
```
class LocalFunctionAnalysis<TDataSink> {  
    public TDataSink GetCurrentAnalysisFrame();  
}  
  
void OnVariableAccess() {  
    var sink = _analysis.GetCurrentAnalysisFrame();  
    sink.RegisterVarAccess(variable, accessType);  
}
```

# Добавим граф вызовов

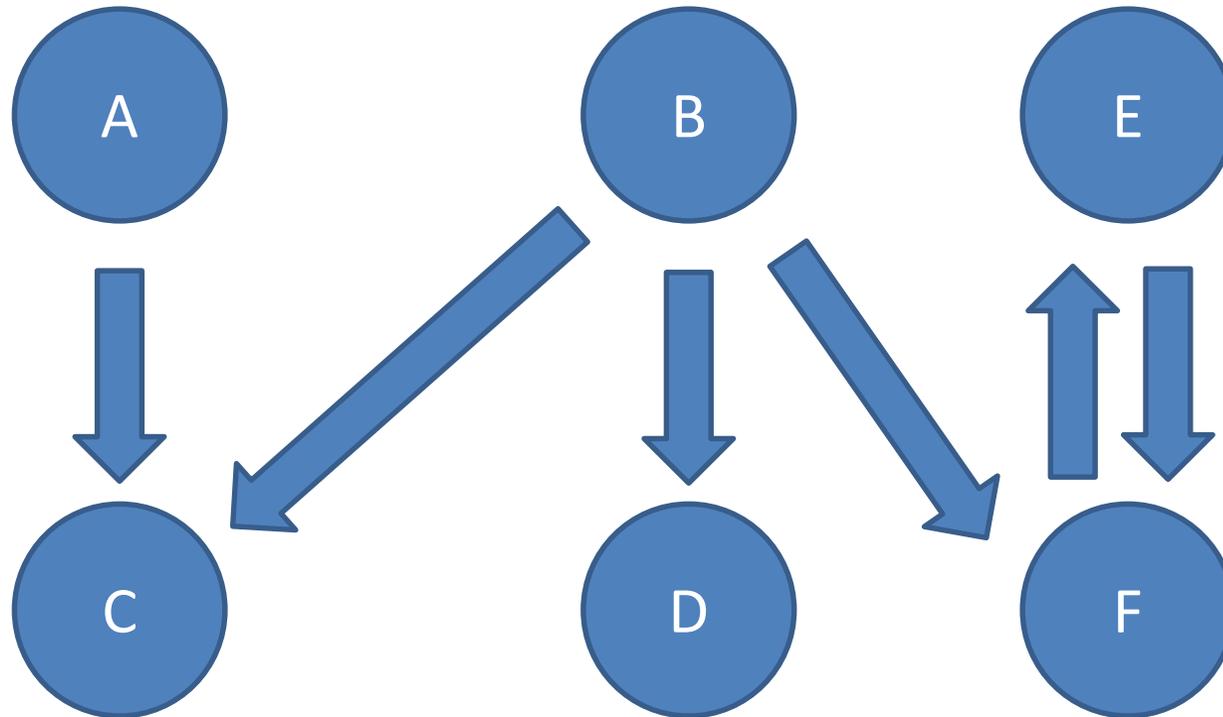
```
void OnLocalFunctionCall(Function f) {  
    _analysis.GetCurrentAnalysisFrame()  
        .RegisterLocalFunctionCall(f);  
    var resume = _analysis.GetOrCreateResume(f);  
    ApplyResume(resume);  
}
```

# Анализ самих локальных функций

```
void A() => C();  
void B() { C(); D(); F(); }  
void C() { }  
void D() { }  
void E() => F();  
void F() => E();
```



# Топологическая сортировка с циклами



Roots

~~A, B~~

Result

A B C D

Unsorted

E, F

# Где можно применить подобный анализ/модель?

- Взаимосвязь между участками кода
  - Поиск выражений которые могли изменить значение переменной
  - Анализ кода сквозь вызовы локальных функций
    - Кэширование, рекурсия, стек временной информации...
- Любой статический анализ
  - Real time / non real time

# Где применить анализ

- Локальные функции
  - Real-time статические анализаторы
  - Модификации кода
    - Кэширование переменных
    - Изменение порядка исполнения
- Кросс-процедурный анализ программы
  - CI server
  - Более точный анализ

# Спасибо за внимание



[tessenr@gmail.com](mailto:tessenr@gmail.com)



[github.com/TessenR](https://github.com/TessenR)



[twitter.com/a\\_tessenr](https://twitter.com/a_tessenr)