

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Математико-Механический факультет

Кафедра Системного программирования

Алефиров Алексей Андреевич

Декомпиляция MSIL в F#.

Курсовая работа

Научный руководитель:
аспирант кафедры системного программирования Григорьев С. В..

Санкт-Петербург
2013

Оглавление

<u>Введение</u>	3
1. <u>Постановка задачи</u>	4
2. <u>Обзор существующих решений</u>	4
3. <u>Реализация</u>	5
4. <u>Апробация разработанного декомпилятора</u>	7
<u>Заключение</u>	9

Введение.

Декомпиляция — это преобразование низкоуровневого объектного кода компиляции в код языка программирования высокого уровня. У программиста необходимость в подобной трансформации может возникнуть в следующих случаях:

- при разработке собственного кода. Изучая результаты декомпиляции своих сборок, программист может найти пути оптимизации программы, декомпиляция может помочь найти ошибки и отладить код;
- при работе с чужим кодом. Декомпиляция является полезным методом исследования приложений и библиотек с целью изучения решений и расширения функциональности;
- при изучении платформы, с которой работает программист. Исследуя декомпилированный объектный код, программист как эффективнее программировать под данную платформу, чтобы программы работали с наибольшей производительностью.

F# [1] — мультипарадигмальный язык программирования из семейства языков .NET Framework, поддерживающий функциональное программирование в дополнение к императивному и объектно-ориентированному. Помимо F# в семейство .NET входят такие языки, как C#, Visual Basic и другие. Программы, написанные на языках семейства, компилируются в единый для .NET байт-код Common Intermediate Language (CIL, MSIL) [2].

На данный момент существуют декомпиляторы из MSIL в C# и Visual Basic. Таким образом, сборки кода, написанного на одном языке семейства, можно изучать на другом языке, предварительно продекомпилировав. Декомпилятора из MSIL в F# на момент начала данной курсовой работы не существовало. Это представляло собой интересную и актуальную проблему, в связи с чем целью курсовой работы стало написание декомпилятора из MSIL в F#.

F# — современный, развивающийся язык программирования. F# позволяет писать лаконичный код программ, а парадигмы программирования, которые он поддерживает, признаны более подходящими в различных направлениях разработки ПО, например, лексические и синтаксические анализаторы. Язык F# набирает популярность, и, в то время как результаты декомпиляции программ, написанных на F#, в другие языки платформы .NET сильно отличаются от исходного кода и могут вызывать затруднения в изучении, создание декомпилятора в F# составляет действительно актуальную проблему.

Основная проблема при решении такой задачи состоит в том, что F# со всеми своими функциональными конструкциями компилируется в MSIL, представляющий объектно-ориентированную и императивную парадигму языков программирования. Таким образом, исходный код проходит нетривиальные преобразования, в то время как с кодом, написанном на C# или Visual Basic, такое происходит в меньшей степени.

Постановка задачи.

Разработать необходимый декомпилятор возможно было двумя общими способами — разработать свою самостоятельную программу, охватывающую все этапы декомпиляции сборки .Net, такие как дизассемблирование байт-кода и формирование деревьев

синтаксического разбора, или найти программные средства, занимающиеся данными этапами, и затем использовать их в создании декомпилятора; такие программные средства, например, как ПО, занимающееся восстановлением .NET сборок, к которому можно было бы разработать дополнение, расширяющее множество целевых высокоуровневых языков. Нахождение такого программного средства значительно облегчило бы решение задачи.

Для выполнения курсовой работы были поставлены следующие задачи.

- Произвести обзор программного обеспечения, занимающегося декомпиляцией .NET сборок.
- Изучить язык программирования F#.
- Изучить общие основы декомпиляции.
- Изучить, каким образом F# компилируется в MSIL.
- Если искомое ПО будет найдено, изучить его архитектуру.
- Разработать декомпилятор из MSIL в F# в виде дополнения для найденного ПО или как самостоятельное приложение.

Обзор существующих решений.

При подготовке к работе был произведен обзор средств декомпиляции сборок .NET. Ни одно из существующих и представленных ниже не поддерживает F#, большинство поддерживают C# и VB .NET.

- **.NET Reflector** [3] — в прошлом open source, ныне коммерческий проект с закрытым кодом, поддерживает декомпиляцию C# и Visual Basic.
- **ILSpy** [4] — open source проект, запущенный после закрытия свободной версии Reflector.
- **Dotnet IL Editor** [5] — open source, однако, не поддерживает разработку плагинов.
- **MonoDevelop** [6] — open source, однако, хоть и поддерживает разработку плагинов, однако, в отличие от ILSpy, не дает возможности “на лету” навигироваться и просматривать декомпилированный код.

ILSpy представил для данной работы наибольший интерес, поскольку это ПО с открытым кодом, предоставляющее удобную платформу для разработки плагина, и, кроме того, дает средства навигации и просмотра дизассемблированного и декомпилированного кода.

Реализация.

Реализовывать задачу было решено в виде плагина к ILSpy. Для этого нужно было изучить архитектуру этого инструмента и понять, как он работает с различными целевыми языками декомпиляции.

ILSpy представляет собой определенное количество библиотек и, собственно, приложение ILSpy.exe. В проекте ILSpy определен абстрактный класс Language, который представляет интерфейс для реализации декомпиляции конкретного языка. При запуске

декомпиляции в представлении приложения вызываются методы `Language`, перегруженные в соответствующем данному языку потомке. Методы принимают дизассемблированный IL и вызывают средства для его декомпиляции. Например, класс `CSharpLanguage` обращается к библиотеке `ICSharpCode.Decompiler`. Для представления дизассемблированного IL используется библиотека `Mono.Cecil`.

Таким образом, разрабатываемый плагин представлял собой библиотеку, содержащую класс-потомок `Language`, ответственный за предоставление инструменту `ILSpy` декомпиляции MSIL в F# - `FSharpLanguage` (рис. 1)

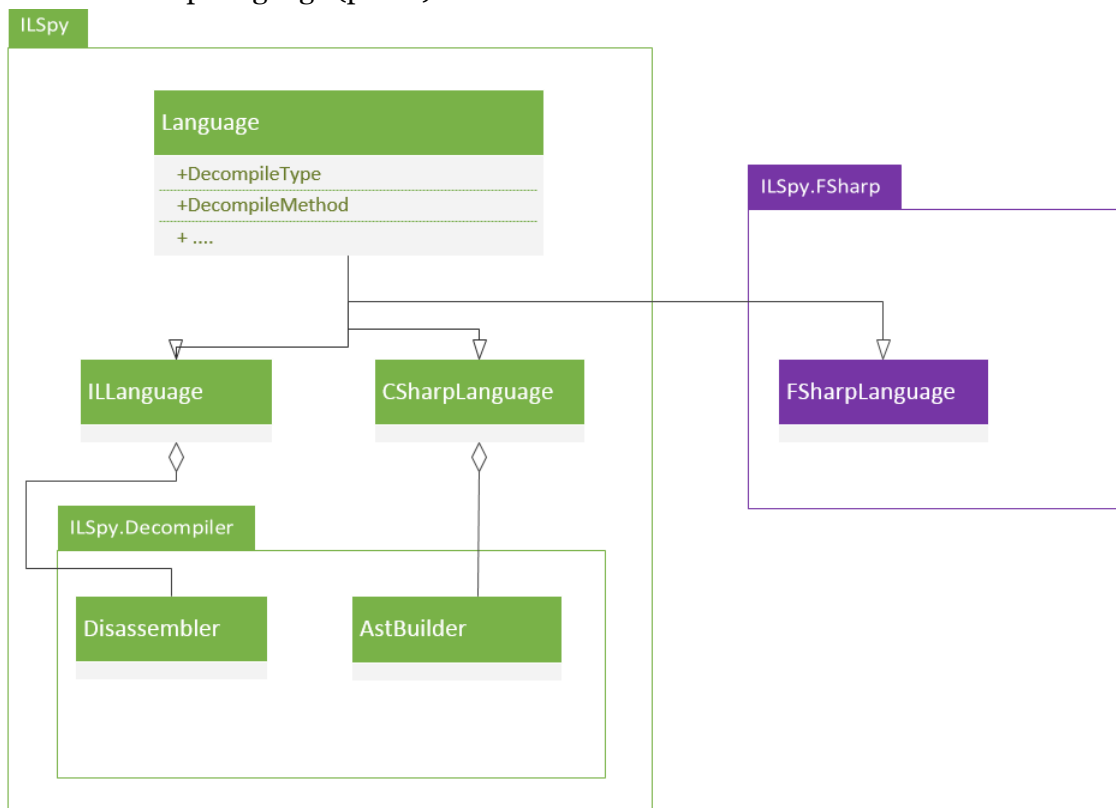


Рис. 1: `Language` - “точка прикрепления” плагина для декомпиляции нового языка

Для декомпиляции в C# в библиотеке `ICSharpCode.Decompiler` реализованы деревья синтаксического анализа (абстрактный класс `AstNode` и его потомки), класс для их создания `AstBuilder`, набор трансформаций (интерфейс `IAstTransform`, а так же класс-посетитель `DepthFirstVisitor`) и конвейер трансформаций `TransformationPipeline`. Также для удобства работы с деревьями разработчиками `ILSpy` были созданы роли узлов дерева (`Roles`). Поскольку это удобная конструкция, в работе над декомпилятором в F# было реализовано расширение типов узлов `Ast` (`FSharp.AST`), наследник `AstBuilder` (`FSAstBuilder`), добавлены новые трансформации и создан новый конвейер трансформаций для создания деревьев, в последующем генерируемых в F#-код.

Для генерации кода новых деревьев был создан класс `FSharpASTPrinter` и модуль `PrinterWrapper`. Печать играет важную роль для кода F#, поскольку форматирование в F# несет синтаксическое и семантическое значение. В качестве выхода `FSharpPrinter` использует интерфейс `ITextOutput` библиотеки `ICSharpCode.Decompiler`.

Как уже было сказано, основной проблемой в работе являлось то, что конструкции функционального программирования, поддерживаемые F#, нетривиально отображаются в MSIL при компиляции. Решение проблемы их восстановления состояло в том, что бы определить шаблоны, по которым происходит это отображение, научиться распознавать их в скомпилированном коде и отображать их обратно в F#.

Имеются в виду такие конструкции, как использование анонимных функций. В исходном коде они выглядят лаконично и занимают немного места, но их компиляция порождает вызов внутреннего класса, специально определенного тут же с конструктором, полями и методом Invoke().

Необходимо было найти ссылку на класс, понять, что эти ссылка и класс порождены компиляцией анонимной функции, создать узел дерева, соответствующий данной функции и подставить его в дерево вместо ссылки на класс. Узел дерева, представляющий собой ссылку на класс, называется MemberType. Чтобы узнать, на что ссылается этот узел, использовался метод Annotation интерфейса AbstractAnnotable, который вызывался у узла MemberType. Данный метод возвращает дизассемблированные конструкции библиотеки Mono.Cecil.

1. Прежде всего был создан класс AnonymousFunction — потомок AstNode для представления в дереве анонимных функций.
2. Был создан класс SubstituteFunction — потомок DepthFirstVisitor и реализация интерфейса IAstTransform для обхода деревьев синтаксического анализа ILSpy и их трансформаций.
3. У SubstituteFunction перегружен метод предка DepthFirstVisitor VisitMemberType, чтобы производить следующие изменения при нахождении узла MemberType.
4. Получить Annotation узла в виде TypeDefinition — класс-конструкция Mono.Cecil, соответствующая определению класса.
5. Полученный класс проверяется на наследование от класса Microsoft.FSharp.Core.FSharpFunc — это признак того, что класс и ссылка на него в коде были порождены при компиляции анонимной функции.
6. Создается экземпляр AstBuilder для порождения дерева синтаксического анализа из TypeDefinition, полученный узел дерева представляет класс TypeDeclaration.
7. У него находится метод Invoke.
8. У метода Invoke сохраняются параметры и тело — это параметры и тело анонимной функции, которую необходимо создать.
9. Создается экземпляр класса AnonymousFunction с сохраненными параметрами и телом.
10. Полученный экземпляр заменяет узел — вызов конструктора, предка узла ссылки на класс с помощью метода ReplaceWith класса AstNode.

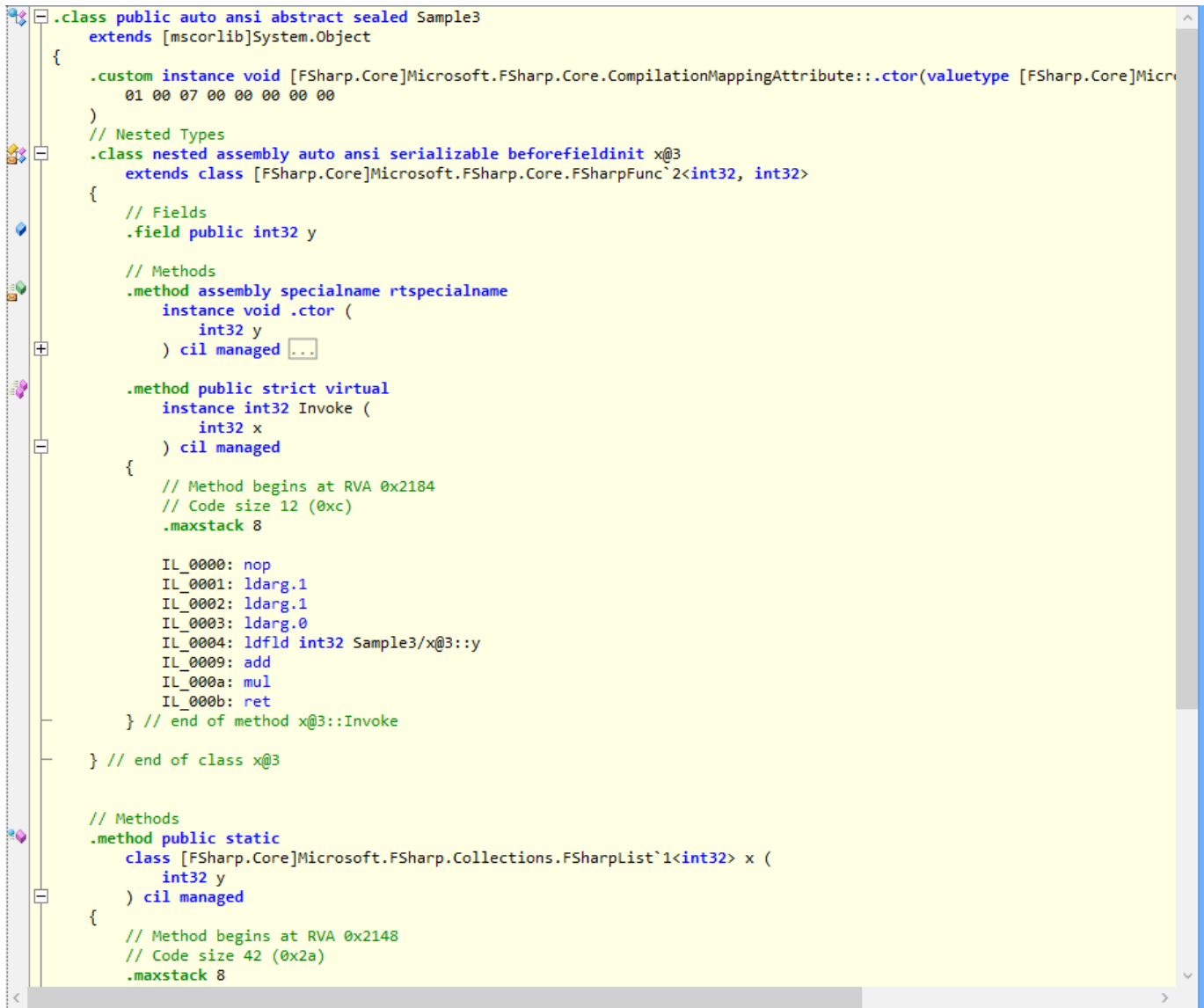
Апробация разработанного декомпилятора.

Ниже приведен пример декомпиляции MSIL-кода несложной F# программы (рис. 2) в F# (рис. 5) и, для сравнения, в C# (рис. 4), а так же IL-представление (рис. 3).

```
module Sample3
```

```
let x y = [1;2;3;4] |> List.map (fun x -> x*(x+y))
```

Рис. 2: Исходный код программы



```
.class public auto ansi abstract sealed Sample3
  extends [mscorlib]System.Object
{
  .custom instance void [FSharp.Core]Microsoft.FSharp.Core.CompilationMappingAttribute::.ctor(valuetype [FSharp.Core]Micro
    01 00 07 00 00 00 00 00
  )
  // Nested Types
  .class nested assembly auto ansi serializable beforefieldinit x@3
    extends class [FSharp.Core]Microsoft.FSharp.Core.FSharpFunc`2<int32, int32>
    {
      // Fields
      .field public int32 y

      // Methods
      .method assembly specialname rtspecialname
        instance void .ctor (
          int32 y
        ) cil managed ..

      .method public strict virtual
        instance int32 Invoke (
          int32 x
        ) cil managed
    {
      // Method begins at RVA 0x2184
      // Code size 12 (0xc)
      .maxstack 8

      IL_0000: nop
      IL_0001: ldarg.1
      IL_0002: ldarg.1
      IL_0003: ldarg.0
      IL_0004: ldfld int32 Sample3/x@3::y
      IL_0009: add
      IL_000a: mul
      IL_000b: ret
    } // end of method x@3::Invoke
  } // end of class x@3

  // Methods
  .method public static
    class [FSharp.Core]Microsoft.FSharp.Collections.FSharpList`1<int32> x (
      int32 y
    ) cil managed
  {
    // Method begins at RVA 0x2148
    // Code size 42 (0x2a)
    .maxstack 8
```

Рис. 3: IL-представление скомпилированного кода

Список литературы

1. Syme Don, Granicz Adam, Cisternino Antonio. Expert F# 2.0.
2. ECMA. Common Language Infrastructure (CLI), Partition III: CIL InstructionSet. – URL: <http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-335.pdf>.
3. Reflector .NET. – URL: <http://www.red-gate.com/products/dotnet-development/reflector/>.
4. ILSpy. – URL: <http://ilspy.net/>.
5. Editor Dotnet IL. – URL: <http://sourceforge.net/projects/dile/>.
6. MonoDevelop. – URL: <http://monodevelop.com/>.