МИНОБРНАУКИ РОССИИ

Федеральное государственное автономное образовательное
учреждение высшего образования
«Южный федеральный университет»

Институт математики, механики
и компьютерных наук им. И. И. Воровича

Кафедра алгебры и дискретной математики

Пузиков Дмитрий Владимирович

РАЗРАБОТКА ГЕНЕРАТОРА КОДА ДЛЯ PASCALABC.NET В ИНФРАСТРУКТУРЕ ROSLYN

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
по направлению подготовки
02.03.02 – Фундаментальная информатика и информационные технологии

Научный руководитель –
доц., к. ф.-м. н. Михалкович Станислав Станиславович

Допущено к защите:
руководитель
направления_____ Михалкович С.С.

Ростов-на-Дону – 2024

Оглавление

1. Постановка задачи

Целью данной работы является разработка и внедрение генератора IL-кода для языка PascalABC.NET в существующий компилятор, с использованием бэкенда Roslyn для генерации IL-кода. Для достижения цели были сформулированы следующие задачи:

- Изучить устройство компилятора PascalABC.NET в части генерации IL-кода;
- Углубить познания архитектуры Roslyn;
- Разработать и реализовать архитектуру генератора кода в рамках компилятора PascalABC.NET;
- Интегрировать новый генератор кода в существующий код компилятора PascalABC.NET

Важным условием для создаваемой архитектуры является требование как можно меньше изменять существующий код компилятора PascalABC.NET при внедрении.

2. Введение

.NET Framework - платформа, включающая в себя инструменты разработки, стандартную библиотеку и общеязыковую среду выполнения (CLR). Ее главной особенностью является свободное взаимодействие между собой программных модулей, написанных на разных языках программирования, поддерживаемых этой платформой: C#, F#, Visual Basic, Visual C++ и т. д. [1]

Однако, на данный момент, Microsoft прекращает разработку новых возможностей для .NET Framework, отказываясь от нее в пользу более современной платформы .NET (бывш. .NET Core), которая, к тому же, является кроссплатформенной. [2][3]

В .NET Framework Microsoft предоставляла открытые методы API для разработки компиляторов под свою платформу, включающие в себя методы для генерации IL-кода и создания сборок. Однако, в .NET API было изменено таким образом, что была убрана возможность сохранять сгенерированные сборки, что делает невозможным создание полноценного компилятора на данной платформе, в том числе, компилятора PascalABC .NET.

Решением данной проблемы может стать использование API Roslyn – стандартного open-source компилятора языков C# и Visual Basic.

3. Общие сведения о компиляторах

Компилятор – это программа, переводящая текст на понятном для человека языке программирования в форму, пригодную для исполнения компьютером (машинный код), или виртуальной машиной (байт-код). Такой процесс называется компиляцией. Процесс компиляции можно разделить на несколько основных фаз [4]:

1. Лексический анализ
2. Синтаксический анализ
3. Семантический анализ
4. Генерация кода

3.1. Лексический анализ

Лексический анализ – первый этап компиляции, служащий для разбиения входного текста на поток (список) токенов, пригодный в дальнейшем для осуществления синтаксического анализа. Токены являются составными частями языка, такими как идентификаторы, ключевые слова, операторы, служебные символы, и т.д. Также они называются лексемы. [4]

Данный этап компиляции выполняется лексическим анализатором, также называемым лексером. Как правило, лексер реализует некоторый конечным автоматом, однако, для более сложных языков, он может реализовывать более сложные автоматы. Например, лексический анализатор языка Python реализует автомат с магазинной памятью.

3.2. Синтаксический анализ

Синтаксический анализ – это проверка корректности программы с точки зрения грамматики языка программирования. Он выполняется синтаксическим анализатором, также называемым парсером.

На этапе синтаксического анализа используется формальная грамматика языка, определенная в общем виде в форме контекстно-свободной грамматики

(КС-грамматики). КС-грамматика определяет множество правил, описывающих структуру предложений в языке, а также все возможные пути их производства. [4]

Для реализации синтаксического анализа компиляторы используют различные алгоритмы обработки грамматик. Одним из часто используемых алгоритмов парсинга является алгоритм рекурсивного спуска. Этот алгоритм начинает работу с начального символа грамматики и выбирает продукцию исходя из текущего (и, в некоторых случаях, следующего) токена. [4]

Результатом работы парсера является абстрактное синтаксическое дерево (AST — Abstract Syntax Tree), являющееся представлением синтаксических конструкций языка в программе. В это дереве внутренними узлами являются операции, а их дочерними узлами — аргументы этих операций.

3.3. Семантический анализ

Семантический анализ — этап компиляции, во время которого программа проверяется на соответствие семантическим правилам языка, таким как соответствие типов и корректность операций. Также могут определяться некоторые вспомогательные признаки, например используемость кода.

Также в процессе семантического анализа строится таблица символов. Это таблица, в которой хранится семантическая информация обо всех встреченных символах. Символами называются классы, функции, методы и т.д. Таблица символов затем активно используется в процессе семантического анализа, а также в процессе генерации кода.

Создаваемые в процессе семантического анализа символы зачастую создают древовидную структуру, именуемую семантическим деревом программы. Эта структура используется совместно с таблицей символов для более удобного анализа программы, а также для последующей генерации кода.

4. Архитектура Roslyn

Roslyn следует стандартной структуре компилятора, описанной в Разделе 3. Однако, в его архитектуре присутствует ряд особенностей.

Два основных пространства имен из Roslyn, используемых в данной работе – это `Microsoft.CodeAnalysis` и `Microsoft.CodeAnalysis.CSharp`. Они также доступны в виде NuGet-пакетов, однако в таком виде они предоставляют лишь функционал по анализу кода на C# [5], так как классы и методы, необходимые для генерации кода, в том числе описанные в этом разделе, скрыты под модификаторами `internal`. По этой причине Roslyn был использован в виде исходного кода.

Для того, чтобы открыть доступ к `internal` классам и методам для своего компилятора было необходимо изменить файлы соответствующих проектов Roslyn.

Листинг 1. Изменение `.csproj` файлов.

```
<ItemGroup>
  . . .
  <InternalsVisibleTo Include="NETGenerator" />
  . . .
</ItemGroup>
```

После такого изменения, все `internal` члены пространств имен `Microsoft.CodeAnalysis` и `Microsoft.CodeAnalysis.CSharp` становятся доступны в проекте NETGenerator.

Важным классом в архитектуре Roslyn (по крайней мере, той части, которая ответственная за компиляцию языка C#) является `CSharpCompilation`. Процесс компиляции начинается с создания экземпляра этого класса, и он участвует во всех стадиях компиляции, включая генерацию IL-кода. Он хранит в себе информацию о текущем состоянии компиляции.

Так как исследование лексического и синтаксического анализаторов Roslyn не входило в цели данной работы, могу лишь сказать, что этими процессами управляет `CSharpCompilation`, делегируя другим частям компилятора и получая в результате синтаксическое дерево программы. Исходный код на C# для узлов синтаксического дерева генерируется в процессе сборки Roslyn одним из его проектов.

После того, как получено синтаксическое дерево программы (оно может быть получено не только с помощью `CSharpCompilation`), по нему строится таблица символов, содержащая семантическую информацию о пространствах имен, классах и методах, после чего следует этап семантического анализа и кодогенерации.

В Roslyn семантический анализ и генерация кода выполняются параллельно. Программа не подвергается семантическому анализу целиком перед тем, как начать генерировать код. Вместо этого, анализ выполняется для каждого метода отдельно непосредственно перед тем, как сгенерировать его код.

В Roslyn существует множество отдельных семантических анализаторов, каждый из которых называется `Binder`, реализует Visitor и отвечает за проверку одного семантического правила. Из сказанного выше следует, что каждый `Binder` получает на вход минимальную часть дерева, которую имеет смысл проверять на конкретное правило. В результате работы они производят части семантического дерева, называемые «связанными» (каждый класс узла такого дерева имеет префикс Bound).

После того, как для метода произведен семантический анализ, для него генерируется код. Для генерации кода служит класс `CodeGenerator`, представляющий собой Visitor по «связанному» дереву. Для того, чтобы непосредственно генерировать инструкции IL-кода он использует класс `ILBuilder`, обладающий набором методов `Emit*` для генерации инструкций. Особенность

`ILBuilder` состоит в том, что он не только генерирует инструкции, но и частично следит за сохранением инвариантов IL-кода. Так, например, он не позволяет сгенерировать последовательность инструкций, потенциально приводящую к нарушению стека если не указать такое намерение явно. Забегая вперед, скажу, что эту проверку пришлось отключить из-за особенностей алгоритмов компиляции PascalABC.NET, которые генерируют такую последовательность IL-команд, которая фактически корректна, но локально нарушает инвариант стека. Однако, сгенерировать невалидный, или даже приводящий к Segmentation Fault IL-код все еще возможно, хоть `ILBuilder` и частично предотвращает это.

Для сохранения сгенерированного кода и другой информации о сборке служит класс `PEModuleBuilder`. Его главная особенность состоит в том, что он хранит в себе таблицу сопоставления символов методов их сгенерированным телам.

Генерацией исполняемого бинарного файла занимается `CSharpCompilation`, используя для этого ранее упомянутый `PEModuleBuilder`. При этом, для создания файла требуется не только сгенерированный IL-код методов, но и таблица символов, не имеющих непосредственного представления в IL-коде и являющихся метаданными. Соответственно, при неправильном создании символов, компиляция может либо завершиться ошибкой, либо завершиться аварийно, либо будет сгенерирован некорректный бинарный файл, который будет падать с ошибкой при попытке запуска.

5. Устройство генератора кода PascalABC.NET

В архитектуре компилятора PascalABC.NET за генерацию кода отвечает класс `ILConverter`, являющийся Visitor'ом по семантическому дереву программы, а также несколько вспомогательных классов, представляющих таблицу символов, операции над ней, а также сущности, инкапсулирующие символы вместе с соответствующими им Builder'ами.

Так как компилятор был написан с расчетом на платформу .NET и имеющиеся в ней средства и API, в процессе генерации кода используются встроенные в платформу классы и методы для генерации кода.

Так, процесс начинается с создания сборки – экземпляра класса `AssemblyBuilder`, и модуля – экземпляра класса `ModuleBuilder`. В модуле затем объявляются все создаваемые типы, каждому соответствует экземпляр класса `TypeBuilder`. В каждом типе объявляются методы, поля и свойства, представляемые классами `MethodBuilder`, `FieldBuilder` и `PropertyBuilder` соответственно.

Все эти классы унаследованы от соответствующих классов с постфиксом Info (например, `MethodInfo` для `MethodBuilder`). Builder представляет собой изменяемую сущность, которая позволяет конструировать объект (тип, метод, поле, свойство), в то время как Info представляет информацию об этом объекте.

Таким образом, вся работа, связанная с генерацией кода, производится с помощью API, предоставляемого платформой .NET.

Генерация кода происходит в процессе прохода по семантическому дереву, т.е. тело метода генерируется в тот момент, когда посещается соответствующий узел дерева, при этом на этот момент могут быть не посещены типы и методы, используемые текущим методом. Аналогичная ситуация происходит и с самими типами, и с полями, и со свойствами.

6. Архитектура нового генератора кода

6.1. Выбор архитектуры

В процессе исследования было принято решение встраиваться в процесс на уровне API .NET, т.е. перехватывать вызовы к API платформы и исполнять свой код. Такой подход был принят, так как это повлечет минимальное изменение исходного кода PascalABC.NET и с меньшей вероятностью создаст непредвиденные проблемы при разработке.

Также был рассмотрен альтернативный вариант – изменение кода процесса генерации кода PascalABC.NET и перестройка его под использование Roslyn. Однако, от данного подхода было решено отказаться ввиду слишком большого объема работы, низкой эффективности и высокой вероятности появления различного рода непредвиденных ошибок.

Однако, у выбранного подхода есть существенный недостаток – несоответствие используемых парадигм .NET и Roslyn. Так, в .NET широко используется паттерн Builder, предоставляющий изменяемые объекты, для представления создаваемых сущностей. В то же время в Roslyn широко используются неизменяемые структуры данных, в том числе символы являются неизменяемыми. Соответственно, для создания дерева символов Roslyn необходимо заранее знать всю информацию о создаваемых символах, в то время как .NET позволяет работать с неполной информацией и дополнять сущности в процессе.

Данную проблему мог бы решить второй предложенный подход, однако это повлекло бы значительное изменение существующей кодовой базы и значительно больший объем работы, что видится нецелесообразным, особенно, учитывая то, что данную проблему удалось решить.

6.2. Архитектура

API генератора кода спроектировано так, чтобы повторить исходно используемое API .NET. Для этого каждый используемый в коде класс из API .NET заменяется на соответствующий тип интерфейса, повторяющий интерфейс этого класса. Данные типы интерфейсов называются адаптерами. Хоть большая часть из них не реализует одноименный паттерн, такое название прижилось в процессе работы и не было необходимости его менять.

Описанный подход позволяет максимально облегчить интеграцию новой системы в существующий код, так как для интеграции понадобится всего лишь заменить имена используемых типов.

Для упрощения, адаптеры не повторяют полностью интерфейс соответствующих классов, а реализуют только те методы и свойства, которые действительно используются в коде. Примеры интерфейсов адаптеров приведены в Листингах 2 и 3.

Листинг 2. Адаптер класса `LocalBuilder`.

```
public interface ILocalBuilderAdapter: ILocalInfoAdapter
{
    ITypeAdapter LocalType { get; }

    void SetLocalSymInfo(string name);
}
```

Листинг 3. Адаптер класса `FieldInfo`.

```
public interface IFieldInfoAdapter: IMemberInfoAdapter
{
    ITypeAdapter FieldType { get; }
    bool IsLiteral { get; }
    bool IsStatic { get; }
    string Name { get; }
    ITypeAdapter DeclaringType { get; }
    FieldAttributes Attributes { get; }

    object GetRawConstantValue();
}
```

Типы интерфейсов адаптеров являются связующим звеном двух частей архитектуры – части, предназначенной для .NET Framework, и части предназначенной для .NET. Все классы адаптеров реализуют соответствующие интерфейсы. Кроме того, иерархия наследования интерфейсов и классов адаптеров повторяет соответствующую иерархию исходных классов .NET.

Для создания адаптеров используется фабрика – синглтон абстрактного класса `AdapterFactory`. Фабрика необходима для простоты создания адаптеров. Существует две фабрики – `FrameworkAdapterFactory` и `RoslynAdapterFactory` для частей .NET Framework и .NET соответственно.

Выбор того, какая часть реализации будет использоваться определяется во время компиляции компилятора, по сути, выбором фабрики, так как корневой объект, с которого начинается генерация кода, `AssemblyBuilder` всегда запрашивается у фабрики. Сам же выбор фабрики происходит с помощью условной компиляции в зависимости от того, под какую платформу собирается компилятор – .NET Framework или .NET. Код выбора приведен в Листинге 4.

Листинг 4. Выбор фабрики с помощью условной компиляции

```
private static AdapterFactory Instance()
{
    if (!(_instance is object))
    {
#if !NETCOREAPP
        _instance = new FrameworkAdapterFactory();
#else
        _instance =  new RoslynAdapterFactory();
#endif
    }

    return _instance;
}
```

Представленный в Листинге 3 код представляет процесс создания экземпляра синглтона фабрики с помощью условной компиляции. Так, при компиляции под .NET компилятором определен флаг `NETCOREAPP`, в то время как при

компиляции под .NET Framework этот флаг не определен. Условная компиляция также используется для того, чтобы не компилировать под .NET большинство классов адаптеров Framework, а также некоторых мест в коде, которые под .NET не скомпилируются из-за различий в API.

6.3. Адаптеры для .NET Framework

В части, предназначенной для .NET Framework, все классы имеют префикс Framework в имени. Они являются прокси-классами, направляющими вызовы соответствующим объектам исходных классов, и не имеют в себе бизнес-логики. Пример представлен в Листинге 5.

Листинг 5. Адаптер класса `LocalBuilder`.

```
public class FrameworkLocalBuilderAdapter:
             ILocalBuilderAdapter
{
    public ITypeAdapter LocalType =>
                        Adaptee.LocalType.GetAdapter();
    public LocalBuilder Adaptee { get; }

    public FrameworkLocalBuilderAdapter(LocalBuilder adaptee)
    {
        Adaptee = adaptee;
    }

    public void SetLocalSymInfo(string name)
    {
        Adaptee.SetLocalSymInfo(name);
    }

    // …
}
```

В Листинге 5 можно заметить вызов `.GetAdapter()` при возврате значения из свойства `LocalType`. `GetAdapter` – набор методов расширения, широко используемых при компиляции для .NET Framework. Любой результат вызова методов API .NET требует вызова `GetAdapter` для преобразования его в адаптер перед тем как вернуть его из адаптера. Об этом речь пойдет чуть позже.

У `FrameworkAdapterFactory`, в отличие от `RoslynAdapterFactory`, являющейся фабрикой для второй части кодогенератора, все методы создания

возвращают валидный объект. Об этом речь пойдет в разделе про часть, предназначенную для .NET.

Так как адаптеры запрашиваются большое количество раз в процессе компиляции, для того чтобы сэкономить память и ресурсы на создание объектов, ведется учет созданных адаптеров. Также, для корректной работы кода, который опирается на равенство ссылок объектов, для каждого объекта .NET с уникальной ссылкой создается уникальный адаптер, который затем возвращается по запросу. Эта механика реализована при помощь статического класса `AdapterExtensions`, который хранит словарь созданных адаптеров, а также определяет методы расширения `GetAdapter` для всех используемых исходных классов .NET, которые возвращают соответствующий адаптер. При запросе адаптера с помощью одного из этих методов происходит проверка, был ли уже создан адаптер для этого объекта. Если адаптер уже был создан, то он возвращается, иначе создается новый. Реализация методов `GetAdapter` немного различается в зависимости от того, запрашивается ли объект Builder или объект Info. Так, если `GetAdapter` вызывается для Builder'а, значит этот объект точно является Builder'ом, т.к. все классы Builder'ов являются `sealed`. В то же время, если `GetAdapter` вызывается для объекта Info, это не значит, что объект на самом деле является Info, а также может быть и Builder'ом. Это не имеет значения если адаптер уже был создан, так как при поиске по ссылке ссылки объектов будут совпадать. Однако, при создании адаптеров, если не учитывать этот момент, то адаптеры будут создаваться неправильно, и объект InfoAdapter будет хранить в себе Builder, что может привести к некорректному поведению. Для обработки этого случая используется рефлексия и проверяется имя фактического типа объекта, для которого вызван `GetAdapter`. Примеры в Листингах 6 и 7.

Листинг 6. Метод расширения `GetAdapter` для класса `Constructor-Builder`

```
public static IConstructorBuilderAdapter
GetAdapter(this ConstructorBuilder builder)
{
     if (builder is null)
    return null;

if (_adaptees.TryGetValue(builder, out var adaptee))
    return adaptee as IConstructorBuilderAdapter;

var instance = AdapterFactory.ConstructorBuilder(builder);
_adaptees.Add(builder, instance);
return instance;
}
```

Листинг 7. Метод `GetAdapter` для класса `ConstructorInfo`

```
public static IConstructorInfoAdapter
GetAdapter(this ConstructorInfo info)
{
    if (info is null)
        return null;

    if (_adaptees.TryGetValue(info, out var adaptee))
        return adaptee as IConstructorInfoAdapter;

    var typeName = info.GetType().Name.Replace("Runtime", "");
    var method = typeof(AdapterFactory)
                 .GetMethod(typeName, BindingFlags.Static |
                                      BindingFlags.Public)
                 ?? typeof(AdapterFactory)
                    .GetMethod("ConstructorInfo",
                                      BindingFlags.Static |
                                      BindingFlags.Public);
    var instance = method.Invoke(null, new [] { info })
                   as IConstructorInfoAdapter;

    _adaptees.Add(info, instance);
    return instance;
}
```

6.4. Адаптеры для .NET

Адаптеры для части, предназначенной для .NET, также, как и в части для .NET Framework, создаются фабрикой, `RoslynAdapterFactory`. Однако, использование фабрики и методов `GetAdapter` в этой части значительно меньше,

по большей части из-за того, что нет необходимости для учета адаптеров, создаваемых моим кодом, так как они, в отличие от адаптеров Framework, не содержат ссылок на внешние объекты API .NET. За некоторыми исключениями, адаптеры создаются только другими адаптерами.

При компиляции с использованием Roslyn, ключевую роль играют символы. Так как парадигмы компиляции .NET Framework и Roslyn несовместимы, о чем было сказано в разделе 6.1, процесс компиляции разделяется на две части, разделенные вызовом метода `Save` класса `RoslynAssemblyBuidlerAdapter`.

В течение первой фазы создаются все адаптеры для всех сущностей компилируемой программы и сохраняются команды кодогенератора (это будет описано подробнее в разделе 7). Во второй фазе создаются символы, фактически генерируется код и происходит сохранение сборки в файл. При этом активное использование Roslyn происходит именно во время второй фазы.

7. Использование Roslyn

7.1. Создание адаптеров сущностей

Как было сказано в разделе 6.4, во время первой фазы происходит создание адаптеров для всех сущностей компилируемой программы. Этими сущностями являются типы компилируемой программы и их члены, а также используемые типы .NET. Для типов компилируемой программы используется класс `RoslynTypeBuilderAdapter`. Для типов .NET переиспользуется класс `FrameworkTypeAdapter`. Отдельного внимания заслуживают generic-типы, о них речь пойдет в одном из следующих разделов.

Примеры – создание адаптера типа `RoslynTypeBuilderAdapter` классом `RoslynModuleBuilderAdapter` и создание адаптера метода `RoslynMethodBuilderAdapter` классом `RoslynTypeBuilderAdapter` представлены в Листингах 8 и 9.

Листинг 8. Создание `RoslynTypeBuilderAdapter`

```
public ITypeBuilderAdapter DefineType(string name,
                      TypeAttributes attr, ITypeAdapter parent,
                      ITypeAdapter[] interfaces)
{
    if (_members.ContainsKey(name))
    {
        throw new InvalidOperationException("Type exists");
    }

    var type = new RoslynTypeBuilderAdapter(this, name, attr,
                                            parent, interfaces);
    _members.Add(name, type);
    return type;
}
```

Стоит заметить, что `name` в Листинге 8 обязательно должно являться полным квалифицированным именем типа, т. е. содержать в себе также полное пространство имен, в котором содержится тип.

Листинг 9. Объявление метода в классе `RoslynTypeBuilderAdapter`

```
public IMethodBuilderAdapter DefineMethod(string name,
           MethodAttributes attributes, ITypeAdapter returnType,
      ITypeAdapter[] parameterTypes)
{
     if (Declaration is object)
    {
        throw new InvalidOperationException(
                  "Type has already been created");
    }

    var method = new RoslynMethodBuilderAdapter(this, name,
                 attributes, returnType, parameterTypes);
    if (_members.TryGetValue(name, out var members))
    {
        members.Add(method);
    }
    else
    {
        _members.Add(name,
                  new List<IMemberInfoAdapter> { method });
    }

    return method;
}
```

В Листинге 9 можно увидеть несколько важных вещей, общих для всех методов класса `RoslynTypeBuilderAdapter`, объявляющих новые члены типа.

Во-первых, проверка того, что объект `Declaration` был создан. Эта проверка преследует цель повторить поведение класса `TypeBuilder`, который кидает исключение если тип уже был создан с помощью вызова `CreateType`. Также, объект `Declaration` используется при создании символов Roslyn для типов.

Во-вторых, все классы, представляющие члены класса (методы, поля, свойства) реализуют интерфейс `IMemberInfoAdapter` и хранятся в поле `_members` класса `RoslynTypeAdapter`. Реализация интерфейса повторяет иерархию наследования исходных типов, а также позволяет однообразно хранить и работать с членами типа, не обращая внимания на то, чем они являются. Поле `_members` в свою очередь представляет собой словарь, где ключом является имя, а значением – список адаптеров, имеющих данное имя.

Аналогичным образом, как и в Листинге 9, происходит создание остальных членов типа.

Отдельно происходит создание ref-типов, типов массивов и generic-типов. Создание ref-типов и типов массивов показано в Листингах 10 и 11. Generic-типы будут описаны отдельно в одном из следующих разделах.

Листинг 10. Создание ref-типа

```
public override ITypeAdapter MakeByRefType()
{
    return new RoslynTypeBuilderAdapter(Module, FullName + "&",
                Attributes, BaseType, _interfaces.ToArray());
}
```

Листинг 11. Создание типа массива

```
public virtual ITypeAdapter MakeArrayType()
{
    return new RoslynArrayTypeAdapter(this);
}
```

Создание ref-типов и типов массивов происходит на основе существующих типов – тип ссылки для ref-типа и тип элемента для типа массива. Таким образом, представленные в Листингах 10 и 11 методы принадлежат классам `RoslynTypeBuilderAdapter` и `RoslynTypeAdapter` соответственно.

Стоит заметить, что, в то время как API .NET Framework различает ref и не-ref типы (ref-типы имеют символ “&” в конце имени), Roslyn подобного разделения не производит. Вместо этого, разделение на ref и не-ref производит символ метода, предоставляя список значений, указывающих на то, является параметр на данной позиции ref или нет.

7.2. Создание символов

Когда происходит вызов `RoslynAssemblyBuilderAdapter.Save()`, все сущности уже созданы и происходит создание символов.

Процесс создания символов начинается с создания символа глобального пространства имен, которое в свою очередь создается из декларации. Декларация глобального пространства имен создается с помощью вспомогательного

класса `DeclarationsUtility` на основе информации о созданных типах и имеет древовидную структуру, в которой корнем является декларация самого глобального пространства имен, а узлами – декларации вложенных пространств имен и типов. Декларации пространств имен представлены классом `SingleNamespaceDeclaration`, декларации типов представлены классом `SingleTypeDeclaration`.

Сам процесс создания дерева деклараций реализован в виде создания вспомогательного дерева объектами абстрактного класса `Node`, от которого наследуются классы `NamespaceNode` и `TypeNode` для пространств имен и классов соответственно. Это работает следующим образом: получая на вход список всех типов программы, для каждого типа выделяется содержащее его пространство имен и ищется его положение в дереве. Если данное пространство имен не присутствует в дереве, то оно создается. Когда пространство имен найдено, ему добавляется обрабатываемый тип. Код метода `CreateDeclarations`, реализующий данный алгоритм приведен в Приложении 1. Стоит заметить, что `CreateDeclarations` возвращает дерево без корневого узла, т. е. список детей корневого узла. Сам корневой узел создается отдельно позже.

Декларации пространств имен создаются методом `CreateNamespaceDeclaration` класса `DeclarationsUtility`, в то время как декларации типов создаются самими типами методом `CreateType` класса `RoslynTypeBuilderAdapter`. Создание декларации типа приведено в Приложении 2.

После того, как вспомогательное дерево создано, оно конвертируется в дерево деклараций, из которого затем создается символ пространства имен класса `SourceNamespaceSymbol`. Созданный символ затем устанавливается объекту `SourceModule` как символ глобального пространства имен. Таким образом мы получаем созданные автоматически символы всех типов, присутствовавших в дереве деклараций, однако эти символы все еще не полны, т. к.

не содержат информации о членах данных типов, а также информации о базовых типах, реализуемых интерфейсах, типе данного типа (класс, интерфейс, абстрактный класс и т. д.) и generic параметрах. Эта информация добавляется символам с помощью метода `CreateTypes` класса `RoslynAssemblyBuildlerAdapter`.

Данный метод получает на вход символ пространства имен и рекурсивно обходит его, дополняя созданные символы типов. На этом этапе происходит создание символов для методов, конструкторов, полей и свойств, а также символу типа устанавливается базовый класс, список реализуемых интерфейсов и generic-параметры при наличии.

Список членов типа представлен в виде словаря, где ключом является имя члена, а значением – список символов с данным именем. После создания всех символов для типа, данный словарь устанавливается символу с помощью метода `SetMembersDictionary` класса `SourceMemberContainerSymbol`.

Стоит заметить, что в исходном виде Roslyn берет всю упомянутую в этом разделе информацию от типе из синтаксического дерева. Так как в моей работе синтаксические деревья Roslyn не используются, пришлось модифицировать исходный код Roslyn, добавив методы установки базового типа, списка реализуемых интерфейсов, generic-параметров и словаря членов в класс `SourceNamedTypeSymbol` либо в один из его базовых классов (которым, например, является класс `SourceMemberContainerSymbol`).

После установки словаря членов типа, для того чтобы изменения «вступили в силу», требуется вызвать методы `ResetMembersAndInitializers` и `GetMembersAndInitializers` класса `SourceMemberContainerSymbol`. После выполнения описанной процедуры символы типов можно считать завершенными.

В процессе создания символов часто возникает задача «разрешать» различные сущности – типы, методы, поля и т. д., то есть сопоставлять адаптер с

соответствующим ему символом. Для решения этой задачи был создан вспомогательный статический класс `ResolveHelper`, предоставляющий набор методов `Resolve*`, выполняющих данную задачу. Для работы `ResolveHelper` необходимо инициализировать, передав ему символ генерируемой сборки, представленный классом `AssemblySymbol`. Инициализация происходит дважды за время компиляции программы – первый раз при создании `RoslynAssemblyBuidlerAdapter`, и второй раз при вызове метода `Save` и создании `ModuleBuilder`. Первая инициализация нужна преимущественно для generic-типов, предоставляемых платформой .NET (например `List<T>`). Вторая инициализация нужна для создания моих символов и генерации IL-кода так как в процессе меняется объект `AssemblySymbol`.

Любое разрешение начинается с разрешения типа (если разрешается тип, то на этом и заканчивается). Ключевую роль в этом играет метод `AssemblySymbol.GetTypeByMetadataName`, возвращающий символ типа по его квалифицированному имени.

Для удобства были написаны два метода расширения `AssemblySymbol`: `GetType` и `GetNamedType`, к которым обращается `ResolveHelper` для разрешения типа. `GetType` представляет собой более общий метод, способный разрешить типы generic-параметров, массивов и указателей. В свою очередь, `GetNamedType` разрешает именованые типы (в терминологии Roslyn типы массивов, указателей и generic-параметров не являются именованными). К нему обращается `GetType` если запрашиваемый тип не является ни одним из упомянутых. Код, создающий символы типов массивов и указателей приведен в Листингах 12 и 13. В обоих Листингах `type` имеет тип `ITypeAdapter`.

Листинг 12. Создание символа типа массива.

```
if (type.IsArray)
{
    var elementType = GetType(assembly, type.GetElementType());
    var arrayType = TypeWithAnnotations
                    .Create(false, elementType);
    return ArrayTypeSymbol
```

```
            .CreateCSharpArray(assembly, arrayType);
}
```

Листинг 13. Создание символа типа указателя.

```
if (type.IsPointer)
{
    var name = type.FullName.Replace("*", "");
    var t = assembly
            .GetTypeByMetadataName(name, true, false, out _);
    return new PointerTypeSymbol(TypeWithAnnotations.Create(t));
}
```

Как было упомянуто в разделе 7.1, Roslyn не различает ref и не-ref типы, поэтому перед поиском из имени типа необходимо удалить символ "&". Также, для generic-типов необходимо добавить к имени типа символ обратного апострофа, за которым следует количество generic-параметров типа.

Для разрешения методов используется метод `ResolveMethod` класса `ResolveHelper`. Он обходит дерево наследования типа, объявляющего метод и ищет метод по сигнатуре. Были реализованы две стратегии поиска метода: по точному совпадению типов параметров и по совпадению, учитывая приведения типов.

Исходные классы символов Roslyn для членов типа реализованы как неизменяемые, и, как и `SourceNamedTypeSymbol`, берут большое количество информации из синтаксического дерева. В то же время мне нужна изменяемость символов, а также я не использую синтаксические деревья Roslyn, следовательно, не могу предоставить эту информацию исходным классам. Для того, чтобы не изменять исходный код этих классов, а также для того, чтобы иметь больше контроля над символами, были созданы собственные классы символов для этих сущностей, унаследованные от соответствующих базовых типов (так как сами классы являются `sealed`). Также был создан класс символа параметра типа для generic-типов и методов.

7.3. Generic типы

При использовании generic-типов может возникнуть четыре ситуации: generic-тип .NET с параметром типа .NET (например `List<int>`), generic-тип

.NET с параметром типа компилируемой программы (например `List<Test>`), generic-тип компилируемой программы с параметром типа .NET (например `Test<int>`), generic-тип компилируемой программы с параметром типа компилируемой программы (например `Test1<Test2>`).

Для первых двух случаев был написан класс `RoslynGenericTypeAdapter`. Так как содержащийся тип является типом .NET, в конструкторе для него сразу находится символ, и, если возможно, инстанцируется. Инстанцирование символов типа и метода происходит вызовом метода `ConstructIfGeneric` с символами типов, которыми будет инстанцирован символ, в качестве аргумента. Если же инстанцирование в момент создания объекта невозможно, то создается "Unbound" символ с помощью метода `ConstructUnboundGenericType`, с отсутствующими аргументами типа. Такие символы используются в таких конструкциях, как `typeof(List<>)`, либо когда аргументы типа неизвестны на момент создания. Также стоит заметить, что метод `ConstructIfGeneric` нельзя вызывать у символов, созданных с помощью `ConstructUnboundType`. Для того, чтобы инстанцировать Unbound символ, необходимо получить символ, из которого он был создан с помощью свойства `ConstructedFrom` и инстанцировать уже его.

Адаптеры `RoslynGenericTypeAdapter`, которые были созданы неинстанцированными, можно инстанцировать с помощью метода `MakeGenericType`. При этом инстанцирование не происходит сразу, так как инстанцировать сразу может быть невозможно. Это касается случаев, когда тип инстанцируется типами компилируемой программы, так как, в момент создания адаптера, создание сущностей еще не завершено, соответственно не созданы символы этих типов. Вместо того, чтобы инстанцировать generic-тип сразу, типы его аргументов сохраняются в адаптере, а инстанцирование происходит только тогда, когда это возможно. За это отвечает свойство `Adaptee` класса `RoslynGenericTypeAdapter`, возвращающее содержащийся в нем символ.

При каждом обращении к этому свойству, происходи проверка, инстанцирован ли тип. За это отвечает флаг `_instantiated`. Если тип не инстанцирован и у него имеются сохраненные типы аргументов, то он пытается получить их символы. Если все символы получены успешно, то символ инстанцируется и возвращается, в противном случае возвращается исходный символ. Также особым случаем является случай, когда тип .NET инстанцируется другим generic-параметром. Пример такого случая приведен в Листинге 14.

Листинг 14. Пример инстанцирования generic-типа generic-параметром

```
procedure Def<C>(var a: C);
begin
  var cc := new List<C>;
  // …
end;
```

В таком случае создание символа параметра типа (в данном случае `C`) происходит сразу в конструкторе `RoslynGenericTypeAdapter`. Несмотря на то, что этот символ уже существует, получить его в этот момент невозможно из-за особенностей цепочки вызовов, приводящей в конструктор. То же происходит при разрешении типа generic-параметра.

Для корректной работы символам параметра типа необходимо, чтобы их свойство `ContainingSymbol` имело значение не равное `null`, однако при создании символа оно равно `null`. Для этого с помощью метода `SetContainingSymbol` им устанавливается необходимый символ при создании символа типа либо метода соответственно. Однако, это не работает для символов параметров типа, созданных искусственно, т.е. образом, описанным в предыдущем абзаце. Для этого, при генерации IL-кода для всех используемых типов дополнительно вызывается метод класса `ResolveHelper` `FixGenericTypeArgumentsIfNeeded`. Он служит для того, чтобы установить правильный содержащий символ для символов параметров типов, для которых он не установлен. Он рекурсивно обходит тип и для каждого символа параметра типа устанавливает нужный символ.

Также для методов и конструкторов generic-типов .NET был написаны классы-обертки `RoslynNativeGenericTypeMethodInfoAdapter` и `RoslynNativeGenericTypeMethodInfoAdapter` соответственно. Они наследуются от соответствующих классов `Framework*`, и нужны для того, чтобы возвращать правильное значение свойства `DeclaringType`.

Для generic-типов компилируемой программы ситуация обстоит проще. Создание generic-типа происходит с помощью вызова метода `DefineGenericParameters` класса `RoslynTypeBuilderAdapter`. При этом сразу создаются объекты `RoslynGenericTypeParameterBuilderAdapter`, представляющие созданные параметры типа. Инстанцирование типа затем происходит с помощью метода `MakeGenericType`, возвращающего новый инстанцированный тип. Созданный тип, однако, хранит ссылку на оригинальное определение.

Члены инстанцированного типа не инстанцируются сразу. Вместо этого, они инстанцируются при запросе. Это реализовано так, потому что оригинальный тип может измениться, и необходимо каждый раз получать его актуальные члены. Для инстанцирования членов generic-типа был написан метод `Instantiate`, объявленный в интерфейсе `IMemberInfoAdapter`. Он получает список аргументов типа и инстанцирует ими член, от которого он был вызван, возвращая новое определение.

При создании символов типов, если тип generic, то для него также создаются символы параметров типа, и для них вызывается `SetContainingSymbol` для того, чтобы установить правильный содержащий символ.

7.4. Генерация IL-кода

Для генерации IL-кода был написан класс `RoslynILGeneratorAdapter`. Его ключевая особенность – отложенная генерация кода. В Roslyn для генерации кода используется класс `ILBuilder`, однако он оперирует символами, которые могут быть еще не созданы на момент использования `RoslynILGeneratorAdapter`. Для этого генератор не использует `ILBuilder` и не генерирует код сразу, а вместо этого запоминает все использованные команды и переданные аргументы.

Команды и аргументы сохраняются в два отдельных списка, `_instructions` и `_arguments` соответственно. В списке команд хранятся элементы перечисления `Instruction`, представляющие вызванный метод генератора, например `Emit`, `EmitCall`, `DeclareLocal` и т.д. В списке аргументов хранятся переданные данным методам аргументы. Это могут быть: объекты классов `EmitInstruction` и `EmitCallInstruction`, хранящие `OpCode` и его аргументы; адаптеры типов, методов, конструкторов, полей; числовые и строковые значения; `null`. Каждой команде соответствует аргумент даже при его отсутствии, т.е. `null`. Таким образом, длины `_instructions` и `_arguments` всегда равны.

Для того, чтобы фактически сгенерировать код и получить объект `MethodBody`, представляющий тело метода со сгенерированным кодом, используется метод `Realize` класса `ILConverter`. Этот метод проходится по всем сохраненным командам и, в зависимости от типа команды, выполняет соответствующие действия с `ILBuilder`. Сам же `ILConverter` создается с помощью метода `Realize` класса `RoslynILGeneratorAdapter`, получая на вход список команд и аргументов, а также ссылку на текущий компилируемый метод.

Так как в .NET и Roslyn используются разные перечисления для опкодов, перед использованием полученные опкоды формата .NET необходимо преобразовать в формат Roslyn. Это делается с помощью прямого приведения

значения `OpCode.Value` типа short к типу `ILOpCode`. Для этого служит метод `GetOpCode`.

Также существуют различия в используемых метках переходов между .NET и Roslyn. В Roslyn в качестве метки может использоваться любой объект, в то время как в .NET в качестве метки используются объекты типа `Label`, имеющие приватное поле идентификатора метки, а также приватный конструктор, позволяющий создать метку с заданным идентификатором. Так как интерфейс `IILGeneratorAdapter` обязывает возвращать `Label`, а метки должны отличаться друг от друга и быть уникальными, то для создания меток в `RoslynILGeneratorAdapter` используется упомянутый конструктор `Label`, полученный с помощью рефлексии. При создании метки ей передается уникальный числовой идентификатор. Счетчик меток, являющийся уникальным идентификатором каждой метки хранится в поле `_labelsCount` класса `RoslynILGeneratorAdapter`, которое увеличивается с каждой созданной меткой. За создание меток отвечает метод `CreateLabel`.

При генерации кода метки, используемые `ILBuilder` представляют собой пустые объекты. Также, `ILConverter` хранит словарь соответствия числовых идентификаторов объектам-меткам. Здесь идентификаторы меток получаются из объектов Label также через рефлексию, получая поле `m_label`. Код получения идентификатора метки приведен в Листинге 15.

Листинг 15. Получение числового идентификатора метки.

```
(int)label
     .GetType()
     .GetField("m_label", BindingFlags.NonPublic |
                          BindingFlags.Instance)
     .GetValue(label);
```

Для многих команд в классе `ILBuilder` есть выделенные методы. Например, опкоды `ret` и `br*` не генерируются напрямую, для них есть вызовы `EmitRet` и `EmitBranch` соответственно. Также выделенные методы есть для инструкций `throw`, `switch`, `ldarg`, `ldarga`, `ldloc`, `ldloca` и других, а также для загрузки констант различных типов на стек.

Для блоков try-catch-finally существуют методы `OpenLocalScope` и `CloseLocalScope`. Метод `OpenLocalScope` получает на вход тип открываемого блока, при этом они могут быть вложены друг в друга. Код, необходимый для начала блока try приведен в Листинге 16.

Листинг 16. Открытие блока try.

```
builder.OpenLocalScope(ScopeType.TryCatchFinally);
builder.OpenLocalScope(ScopeType.Try);
```

Для начала блока catch необходимо закрыть блок try (но не `TryCatchFinally`), выровнять стек и открыть блок catch. Код, выполняющий данную операцию приведен в Листинге 17.

Листинг 17. Открытие блока catch.

```
builder.CloseLocalScope();
builder.AdjustStack(1);
builder.OpenLocalScope(ScopeType.Catch,
                       GetToken(exceptionType));
```

Начало блока finally выполняется аналогично, но без выравнивания стека. В конце конструкции try-catch-finally необходимо закрыть блок `TryCatchFinally`.

8. Заключение

В рамках данной работы была разработана архитектура генератора кода, использующего Roslyn, для компилятора PascalABC.NET. По разработанной архитектуре был реализован генератор кода с использованием API и архитектуры Roslyn. За счет этого была получена возможность компилировать программы на языке PascalABC.NET под кроссплатформенную платформу .NET, работающую в том числе на Linux.

Для реализации было изучено устройство генератора кода PascalABC.NET, а также более подробно изучен Roslyn.

Таким образом был реализован генератор кода для подмножества языка PascalABC.NET, достаточного для компиляции стандартной библиотеки PABCSystem, способный генерировать код для платформы .NET, который в последствие можно запускать на различных ОС, в т.ч. Windows и Linux.

Ознакомиться с исходным кодом можно по следующим ссылкам:

- Форк компилятора PascalABC.NET: https://github.com/Lyrai/pascalabcnet
- Модифицированный компилятор Roslyn: https://github.com/Lyrai/roslyn-compilers

9. Список литературы

1. Общие сведения о платформе .NET – URL: https://learn.microsoft.com/ru-ru/dotnet/framework/get-started/overview (дата обр. 21.05.2024)
2. Обзор .NET Core – URL: https://learn.microsoft.com/ru-ru/dotnet/core/introduction (дата обр. 21.05.2024)
3. .NET Framework Support Policy – URL: https://dotnet.microsoft.com/en-us/platform/support/policy/dotnet-framework (дата обр. 21.05.2024)
4. Компиляторы: принципы, технологии и инструментарий / Альфед В. Ахо, Моника С. Лам, Рави Сети, Джеффри Д. Ульман. — М.: ООО «И. Д. Вильямс», 2008. — 1178 с.
5. Roslyn Wiki – URL: https://github.com/dotnet/roslyn/blob/main/docs/wiki/NuGet-packages.md (дата обр. 21.05.2024)

11. Приложение

Приложение 1. Создание дерева деклараций.

```
public static ImmutableArray<SingleNamespaceOrTypeDeclaration>
CreateDeclarations(IEnumerable<RoslynTypeBuilderAdapter> types)
{
    var rootNamespace = new NamespaceNode("");
    foreach (var type in types)
    {
        var namespaces = type.Namespace.Split('.');
        var current = rootNamespace;
        foreach (var ns in namespaces)
        {
            var next = Find(current, ns);
            if (next is null)
            {
                var newNode = new NamespaceNode(ns);
                current.Children.Add(newNode);
                current = newNode;
            }
            else
            {
                Debug.Assert(next is NamespaceNode);
                current = next as NamespaceNode;
            }
        }

        Debug.Assert(current is object);
        current.Children.Add(new TypeNode(type));
    }

    return ToImmutable(rootNamespace);
}
```

Приложение 2. Создание декларации типа.

```
private SingleTypeDeclaration CreateDeclaration()
{
    var memberNames = _members
                  .Keys
                  .Select(name => (name, new VoidResult()).ToKeyValuePair())
                  .ToImmutableSegmentedDictionary();

    var declarationModifiers = DeclarationModifiers.None;
    if (IsAbstract) declarationModifiers |= DeclarationModifiers.Abstract;
    if (IsPublic) declarationModifiers |= DeclarationModifiers.Public;
    if (IsSealed) declarationModifiers |= DeclarationModifiers.Sealed;

    var typeDeclarationFlags = SingleTypeDeclaration
                               .TypeDeclarationFlags.None;
    if (_members.Any())
        typeDeclarationFlags |= SingleTypeDeclaration
                                .TypeDeclarationFlags.HasAnyNontypeMembers;

    DeclarationKind declarationKind = DeclarationKind.Struct;
    if (IsClass) declarationKind = DeclarationKind.Class;
    if (IsInterface) declarationKind = DeclarationKind.Interface;

    var nestedTypes = _netstedTypes
```

```
                    .Values
                    .Select(type => (type as RoslynTypeBuilderAdapter)
                                    .CreateDeclaration())
                    .ToImmutableArray();

    Declaration = new SingleTypeDeclaration(
                declarationKind,
                Name,
                _genericParameters.Length + _genericArguments.Length,
                declarationModifiers,
                typeDeclarationFlags,
                null,
                new RoslynSourceLocation(0, 0),
                memberNames,
                nestedTypes,
                ImmutableArray<Diagnostic>.Empty,
                QuickAttributes.None
    );

    return Declaration;
}
```