Санкт-Петербургский государственный университет

Бережных Алексей Владимирович

Выпускная квалификационная работа

Кросс-проектный анализ для генеративных поставщиков типов F#

Уровень образования: магистратура

Направление *09.04.04 «Программная инженерия»*

Основная образовательная программа *ВМ.5666.2020 «Программная инженерия»*

Научный руководитель:
доцент кафедры системного программирования, к.т.н., Ю.В. Литвинов

Консультант:
инженер-программист JetBrains, Е.П. Аудучинок

Рецензент:
инженер-программист JetBrains, И.Е. Мигалев

Санкт-Петербург
2022

Saint Petersburg State University

Berezhnykh Aleksey Vladimirovich

Master's Thesis

Cross-project analysis for F# generative type providers

Education level: master

Speciality *09.04.04 «Software Engineering»*

Programme *BM.5666.2020 «Software Engineering»*

Scientific supervisor:
Assistant Professor, Phd Litvinov Y. V.

Consultant:
Software engineer at JetBrains, Auduchinok E. P.

Reviewer:
Software engineer at JetBrains, Migalev I. E.

Saint Petersburg
2022

Оглавление

1. Введение **5**

2. Постановка задачи **8**

3. Обзор **9**

3.1. Поставщики типов 9
3.2. Генеративные поставщики типов 10
3.3. ReSharper PSI 11
3.4. Кросс-языковой анализ в Rider 12
3.5. Изолированный запуск поставщиков типов в Rider 13

4. Архитектура **14**

4.1. Кросс-языковой анализ генеративных типов 14
4.2. Генеративные типы в модели ReSharper PSI 15

5. Реализация **17**

5.1. Обнаружение созданных поставщиками генеративных типов 17
5.2. Кэш генеративных типов 18
5.3. Представление генеративных типов в ReSharper PSI . . . 20
5.4. Алгоритм разрешения генеративных типов 23
5.5. Дополнительная функциональность в IDE 25

6. Тестирование и апробация **27**

6.1. Прототип 27
6.2. Написание автоматических тестов 27
6.3. Ручное тестирование с помощью FSharp.Data и OpenApiProvider 29
6.4. Выводы 30

7. Ограничения **31**

8. Заключение **32**

Список литературы **33**

1. Введение

Современные программные системы оперируют большими данными, реализуя сложные задачи по их обработке [8]. Известно, что данные реальных источников, как правило, слабо структурированы, в то время как большинство промышленных языков программирования имеют строгую статическую типизацию. Традиционно для реализации доступа программных систем к внешним данным, таким как веб-сервисы или базы данных, требовалось переписывать вручную или генерировать с помощью специальных инструментов программный код. Однако практика кодогенерации в большинстве случаев считается неприемлемой из-за сложности в понимании и поддержке сгенерированного кода.

Поставщики типов языка F# [6] позволяют упростить получение данных из внешних источников, предоставляя необходимые типы данных «на лету» во время написания кода и компиляции, загружая структуру данных из внешнего описания или схемы. Для программиста поставщик типов представлен в виде подключаемой библиотеки, включающей набор определенных типов, при использовании которых во время написания кода и компиляции генерируются требуемые статические типы данных. Типы, созданные генеративными поставщиками, на этапе компиляции записываются в метаданные сборки, в которой они были объявлены, что позволяет использовать полученные типы во всех .NET-совместимых языках.

Библиотека сервисов компилятора (FSharp.Compiler.Service, FCS) [3] с открытым исходным кодом[1] является частью компилятора языка F# и обеспечивает поддержку этого языка в различных средах разработки. В частности, данная библиотека позволяет пользователю подключать поставщики типов в проекты на F# в качестве обычных NuGet-пакетов и использовать сгенерированные ими типы данных в своем коде.

Совместимость F# с языками и библиотеками .NET, способность

[1]FSharp.Compiler.Service – библиотека языковой поддержки F#. URL: https://github.com/dotnet/fsharp (дата обращения: 03.05.2022).

справляться со сложностью обработки и анализа больших данных, а также удобства системы типов при проектирования домена приложения предлагают привлекательные возможности для бизнеса даже в тех проектах, где большая часть кодовой базы написана на C#. Современные IDE позволяют из проектов на C# обращаться к коду, написанному на F# – и наоборот, однако использование генеративных поставщиков типов в подобных проектах приводит к проблемам: FCS не анализирует код проектов на C#, а средства анализа C#-кода (такие как Roslyn[2] или ReSharper[3]) ничего не знают о внутреннем устройстве поставщиков типов из F#. В итоге генеративные типы, как правило, не имеют явных деклараций в коде и, следовательно, недоступны для анализа. Данная проблема приводит к ошибкам анализа кода в редакторе и, как следствие, не позволяет программисту во время написания кода использовать в C#-проектах сгенерированные поставщиками типы. Единственный существующий на данный момент способ обойти проблему заключается в компиляции проекта, содержащего генеративные поставщики типов. После этого средства анализа C#-кода смогут прочитать информацию о типах из метаданных физической сборки, а не из исходного кода проекта. Очевидны проблемы при использовании такого подхода: при малейшем изменении кода, к которому могут обращаться проекты на C#, требуется перекомпиляция проекта с поставщиками типов; между проектами не будет работать базовая функциональность IDE, такая как навигация к декларации символа из проекта с поставщиками типов или его переименование. Данная проблема актуальна[4] и для Rider[5] – одной из популярных IDE для .NET компании JetBrains. Движок ReSharper, на основе которого выполняется анализ кода в Rider, не содержит совместимых со своей моделью кода PSI (Program Structure Interface) представлений для генеративных типов, как это реализовано

[2]Roslyn – платформа для .NET-компиляторов. URL: https://github.com/dotnet/roslyn (дата обращения: 05.05.2022).

[3]ReSharper – инструмент повышения производительности для .NET-разработчиков. URL: https://www.jetbrains.com/ru-ru/resharper/ (дата обращения 02.03.2022)

[4]Описание проблемы при анализе генеративных поставщиков типов в C#. URL: https://youtrack.jetbrains.com/issue/RIDER-7912 (дата обращения: 03.05.2022).

[5]JetBrains Rider – интегрированная среда разработки для .NET. URL: https://www.jetbrains.com/ru-ru/rider (дата обращения: 03.05.2022).

для остальных элементов языка F#. В то же время, для принудительного считывания сгенерированных поставщиками типов из скомпилированного проекта необходимо дополнительно выгружать его из IDE, что не позволяет одновременно работать над этим проектом вместе с проектами, его использующими. Наличие таких критических неудобств при работе с генеративными поставщиками типов в Rider является главной мотивацией данной работы и требует разработки подсистемы, обеспечивающей честный анализ генеративных поставщиков типов из F# в проектах на C#.

2. Постановка задачи

Целью данной работы является реализация подсистемы анализа генеративных поставщиков типов F# в C#-проектах для среды разработки Rider.

Для достижения этой цели были поставлены следующие задачи:

1. Исследовать особенности компиляции и анализа кода в проектах с генеративными поставщиками типов.
2. Разработать архитектуру подсистемы взаимодействия генеративных поставщиков типов и механизмов анализа кода среды разработки Rider.
3. Реализовать необходимые объектные представления для генеративных типов и интегрировать их в систему типов Rider.
4. Создать тестовое покрытие и произвести апробацию разработанной подсистемы.

3. Обзор

3.1. Поставщики типов

Поставщики типов предоставляют пространства имён, которые, в свою очередь, содержат набор заранее заданных предоставляемых типов. Такие типы, обычно, содержат конструкторы, принимающие литеральные формальные параметры: строку подключения к базе данных, путь к файлу и т.д. На основе переданных в конструктор аргументов поставщик типов выполняет описанный в нём код и генерирует типы, которые с точки зрения компилятора являются обертками (ProvidedType) над экземплярами настоящих типов среды выполнения .NET и позволяют получить информацию, необходимую для отображения программисту при написании им кода, такую как:

- вложенные типы;
- конструкторы + список формальных параметров;
- методы + список формальных параметров;
- свойства;
- поля;
- события.

Предоставляемые типы в своем содержимом могут ссылаться на типы из стандартных библиотек .NET. Для унификации обработки предоставляемых типов и их содержимого в FCS обертки ProvidedType создаются как для типов, созданных поставщиками, так и для типов из стандартных библиотек, используемых в содержимом ProvidedType. В дальнейшем в данной работе понадобится умение отличать настоящие сгененированные поставщиками типы от оберток над библиотечными типами.

Написать свой поставщик типов [5] можно с помощью библиотеки с открытым исходным кодом FSharp.TypeProviders.SDK[6], после чего его можно подключить в свой проект в качестве библиотеки напрямую с диска или через менеджер пакетов NuGet.

3.2. Генеративные поставщики типов

Генеративный поставщик типов создаёт отдельную сборку с .NET-метаданными для генерируемых типов. Содержимое такой сборки при компиляции записывается в метаданные сборки проекта, в котором объявлены поставщики типов – это позволяет использовать сгенерированные типы в использующих сборку проектах на .NET-совместимых языках.

Для статического связывания типов между проектом с генеративным поставщиком типов и сгенерированной поставщиком сборкой необходимо для сгенерированных типов создать в исходном коде соответствующий им тип-аббревиатуру, который в дальнейшем будет корневым для группы вложенных сгенерированных типов. Так, например, на рис. 1 для группы сгенерированных из файла yaml-конфигурации типов создаётся тип-аббревиатура GeneratedConfig, содержащая сгенерированные типы Level1_Type, Level2_Type и т.д.

При анализе генеративных типов в FCS им передается контекст ProvidedTypeContext, содержащий отображение корневых и вложенных генеративных типов в соответствующее ILTypeRef-представление[7] в окончательной сборке.

Для генеративных поставщиков типов существует «поздняя стадия» в процессе подготовки метаданных сборки, которая статически связывает сгенерированные типы в окончательной .NET-сборке на основе отображений в ProvidedTypeContext. Этот этап выполняется при настоящей компиляции, но не выполняется в соответствующем сценарии

[6]Библиотека для создания поставщиков типов. ULR: https://github.com/fsprojects/FSharp.TypeProviders.SDK (дата обращения 03.04.2022)

[7]ILTypeRef в компиляторе F#. URL: https://github.com/dotnet/fsharp/blob/4990f64fa5951b6214c94691845518edad1e6b82/src/Compiler/AbstractIL/il.fsi#L202-L228 (дата обращения: 22.05.22)

```
type GeneratedConfig = YamlConfig<YamlText = """
Level1:
  Level12:
    Level13:
      -
        name: Jack
        age: 32
      -
        name: Claudia
        age: 25
Level2:
  Level21: 2""">


GeneratedConfig.Level1_Type.
                          Level12_Type
                          Press Enter to insert, Tab to replace

type Level12_Type =
new: unit -> Level12_Type
member Level13:
IList<Level13_Item_Type>
member Changed: EventHandler
nested type Level13_Item_Type
```

Рис. 1: Пример типа-аббревиатуры для сгенерированного типа

анализа проектов в IDE [7].

3.3. ReSharper PSI

ReSharper использует собственный вывод типов для C#, основанный на модели кода ReSharper PSI [4], позволяющей установить связь между декларациями в коде и семантическими элементами, которые порождают эти декларации в определенном контексте. Одними из ключевых элементов в модели ReSharper PSI являются интерфейс ITypeElement, который представляет совместимые с CLR типы, такие как класс, структура, интерфейс, делегат и т.д., и ITypeMember, представляющий их содержимое: методы, конструкторы, вложенные типы, операторы и т.д. Интерфейс IType используется для представления различных конструкций типов – не только отдельных типов, но и таких конструкций, как анонимные типы, массивы, указатели и другие. Интерфейс IPsiModule представляет единицу компиляции проекта (например, сборка).

Данная модель позволяет универсальным способом рассматривать и

использовать информацию о типах из разных языков, используемых в проекте. Для большей части определенных в F#-коде элементов в Rider уже реализовано совместимое с ReSharper PSI представление, благодаря чему для них работает кросс-проектный анализ.

И хотя содержимое большинства элементов в языках F# и C# схожи, однако для некоторых F#-специфичных элементов (таких как размеченные объединения) в ITypeElement генерируются дополнительные элементы ITypeMember (например, дополнительные конструкторы, методы и свойства). Это необходимо, чтобы эмулировать структуру элемента как после компиляции. И поскольку данные элементы не описаны явно в F#-коде и являются искуственно сгенерированными на основе базового элемента, в языковой поддержке F# для Rider создан интерфейс ISecondaryDeclaredElement. Данный интерфейс содержит свойство OriginElement, позволяющее определить базовый элемент, из которого сгенерированы дополнительные элементы.

На данный момент для генеративных поставщиков типов не реализовано представление ReSharper, а сгенерированные типы, по большей части, не имеют деклараций в коде, вследствие чего ReSharper ничего не знает об их содержимом.

3.4. Кросс-языковой анализ в Rider

3.4.1. Анализ F#-проектов

Основными этапами анализа кода в проектах на F# являются синтаксический разбор и вывод типов с помощью FCS [1]. Во время синтаксического разбора пространства имён, модули и типы добавляются в глобальный символьный кэш ReSharper. Символьный кэш представляет собой префиксное дерево, где в качестве префиксов используются полное имя элемента (путь к элементу), и используется для разрешения типов во всех проектах (и таких контекстных действий как импорт пространства имен).

Вывод типов происходит для файлов в последовательном порядке, определенном в файле конфигурации проекта. Результаты вывода ти-

пов для файлов записываются в кэш FCS-символов. В случае изменения кода в определенном файле, для него и следующих за ним файлов данные кэша инвалидируются.

3.4.2. Анализ C# проектов

Когда во время анализа C#-кода встречается ссылка на какой-либо символ, ReSharper ищет его по имени в символьном кэше с учетом используемых в текущем анализируемом файле пространств имен. Поиск считается успешным, если в символьном кэше обнаружен запрашиваемый единственный элемент, обьявленный в одном из используемых пространств имён. При обращении к его содержимому (создании обьекта через конструктор, вызов метода, чтения свойства и т.д.) создается ITypeElement, который на основе FCS-символов из кэша возвращает свое содержимое в виде набора ITypeMember.

3.5. Изолированный запуск поставщиков типов в Rider

Поддержка поставщиков типов инкапсулирована в FCS, и внутреннее API для получения данных из генеративных типов недоступно для пользователей FCS. В то же время предоставляемые FCS после анализа кода символы не содержат важной информации о генеративных типах (например, список конструкторов). Однако в рамках бакалаврской работы [2] была проведена работа по изолированному запуску поставщиков типов в отдельном от FCS и Rider процессе. Данный этап можно считать подготовительным в рамках текущей работы, поскольку в ней был реализован доступ к внутреннему API генеративных типов и механизмы получения, хранения и передачи содержимого предоставляемых типов от изолированного процесса к FCS.

4. Архитектура

4.1. Кросс-языковой анализ генеративных типов

На рис. 2 представлена схема кросс-проектного анализа генеративных поставщиков типов, состоящая из следующих этапов:

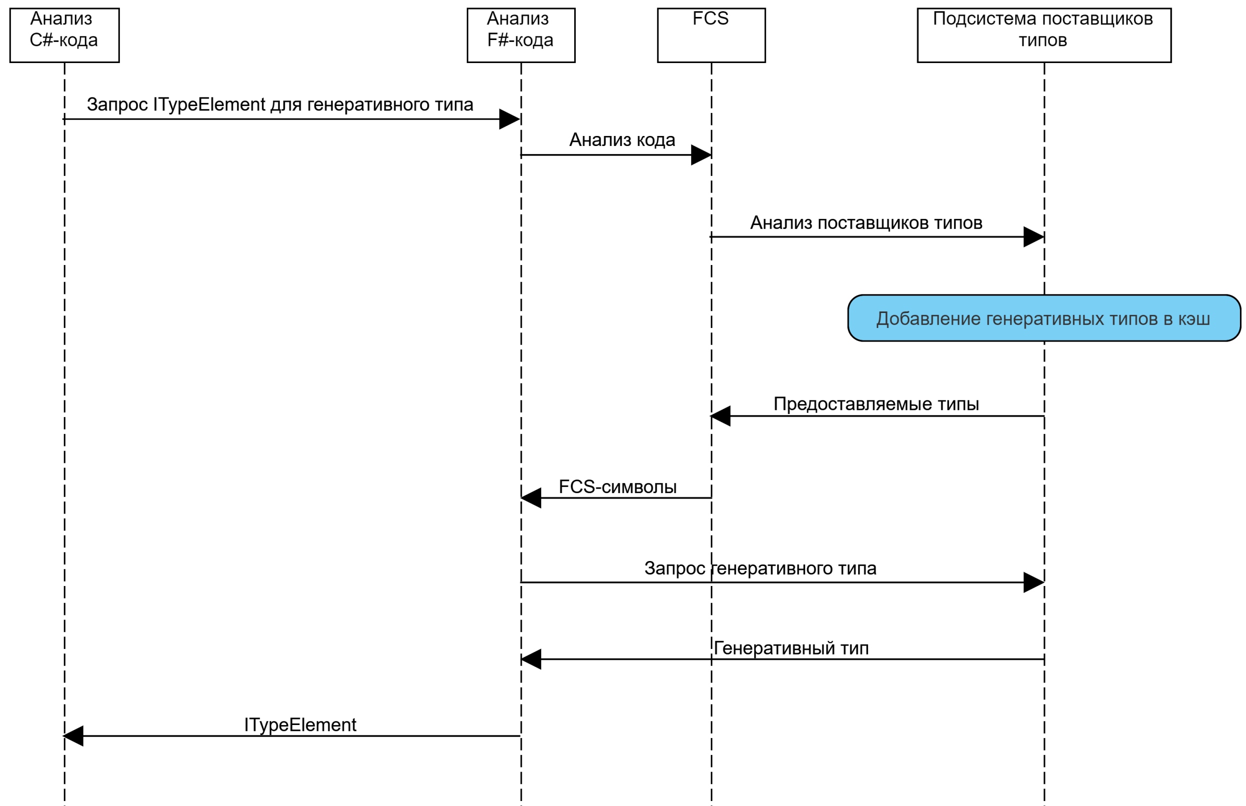

Рис. 2: Схема кросс-проектного анализа генеративных типов.

1. При обнаружении в C#-коде ссылки на тип из символьного кэша, для него запрашивается ITypeElement и его содержимое.
2. Если актуальные результаты анализа F# кода отстуствует в кэше FCS-символов, выполняется анализ с помощью FCS.
3. FCS запрашивает у поставщиков типов данные.
4. Генеративные типы, созданные в процессе анализа, записываются в кэш генеративных типов.
5. FCS-символы после анализа записываются в кэш FCS-символов.

6. Если для запрошенного типа FCS-символ соответствует генеративному типу, из кэша генеративных типов запрашивается необходимый генеративный тип.

7. Полученный генеративный тип отображается в соответствующий ему ITypeElement, содержимое которого отображается в соответсвующий набор ITypeMember/IType.

Аналогично, если генеративный тип используется в качестве, например, параметра/возвращаемого значения F#-функции, для него создается требуемый IType, как описано в пунктах 6 и 7.

В рамках данной работы были реализованы кэш генеративных типов, отображение FCS-символа в генеративный тип и создание на его основе необходимых представлений ITypeElement/ITypeMember/IType.

4.2. Генеративные типы в модели ReSharper PSI

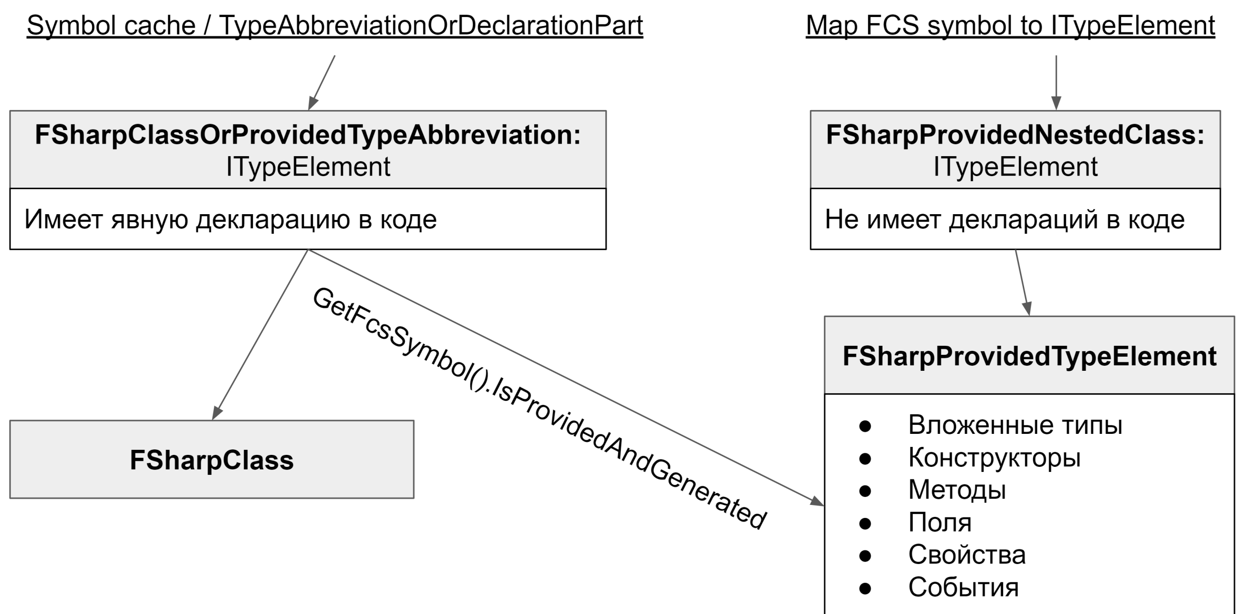

Рис. 3: ReSharper PSI для генеративных типов

Генеративные типы можно разделить на две категории:
типы-аббревиатуры и вложенные генеративные типы. Типы-аббревиатуры имеют явную декларацию в коде и добавляются в символьный кэш

ReSharper в виде экземпляра существующего класса TypeAbbreviationOrDeclarationPart. Вложенные генеративные типы содержатся в типе-аббревиатуре (а также могут быть вложены друг в друга) и не имеют явной декларации в коде, а потому должны иметь отличную от типа-аббревиатуры реализацию. В то же время, генеративный тип-аббревиатура и вложенные в него генеративные типы имеют общую логику по конвертации содержимого генеративных типов в совместимые с ReSharper PSI представления. Поэтому были реализованы:

- класс FSharpProvidedTypeElement, реализующий логику конвертации содержимого генеративных типов;
- класс FSharpClassOrProvidedTypeAbbreviation, реализующий ITypeElement для типа-аббревиатуры;
- класс FSharpProvidedNestedClass, реализующий ITypeElement для вложенного типа;
- набор классов, реализующих ITypeMember для содержимого генеративных типов (методов, полей и т.д.).

Класс FSharpProvidedNestedClass агрегирует FSharpProvidedTypeElement и делегирует запросы на получение содержимого ему. Класс FSharpClassOrProvidedTypeAbbreviation по FCS-символу может определить, является ли он генеративным, и делегировать запросы на получение содержимого к FSharpProvidedTypeElement или базовому существующему классу FSharpClass.

5. Реализация

5.1. Обнаружение созданных поставщиками генеративных типов

В построении символьного кэша ReSharper участвуют не только типы из исходного кода проекта, но и типы из метаданных библиотек, на которые есть ссылки в проекте. В то же время, предоставляемые поставщиками типы так же могут быть оберткой над каким-либо библиотечным типом, используемым внутри по-настоящему созданного поставщиком типа. Если для настоящих сгенерированных типов и оберток над библиотечными типами использовать одинаковое представление ReSharper PSI, то это может привести к неконсистентности, когда один и тот же библиотечный тип при анализе имеет стандартно используемое в ReSharper представление типа из метаданных и представление для сгенерированного поставщиком типа. Потому для вторых могут выдаваться не самые релевантные советы автодополнения, не будут работать некоторые инспекции/рефакторинги/быстрые исправления. Необходимо отличать по-настоящему созданные поставщиками типы от библиотечных типов в обертке предоставляемого типа – и создавать для первых отдельное представление, а для вторых брать готовое представление из символьных кэшей ReSharper.

Для описания по-настоящему генерируемых поставщиками типов среди авторов библиотек поставщиков типов стандартным является использование класса ProvidedTypeDefinition из библиотеки для создания поставщиков типов FSharp.TypeProviders.SDK . Поскольку поставщики типов в Rider создаются в отдельном контролируемом процессе, который обменивается данными с бэкендом Rider, при передаче данных о предоставляемом типе возможно с помощью рефлексии узнать, является ли предоставляемый тип экземпляром ProvidedTypeDefinition. Была реализована[8] соответствующая проверка, результат которой записыва-

[8]Определение созданных поставщиком типов. URL: https://github.com/JetBrains/resharper-fsharp/pull/331 (дата обращения 22.05.2022)

ется во флаг IsCreatedByProvider в передаваемых данных о предоставляемом типе.

5.2. Кэш генеративных типов

5.2.1. Контекст генеративных типов

Генеративные типы, в отличие от стираемых предоставляемых типов, содержат контекст ProvidedTypeContext. Данный контекст хранит отображение генеративного типа-аббревиатуры и вложенных в него генеративных типов в соответствующие ILTypeRef-представления в проекте, в котором объявлен тип-аббревиатура. ILTypeRef содержит CLR-имя генеративного типа в итоговой сборке после компиляции проекта с поставщиками типов, позволяющее уникально определить тип внутри проекта.

Предоставляемые типы иммутабельны, а генеративные типы создаются посредством вызова на предоставляемом типе метода ApplyContext с передачей необходимого ProvidedTypeContext. Изначально для экономии памяти и поддержания ссылочной структуры в реализации изолированных поставщиков типов в Rider новые экземпляры генеративных типов не создавались при вызове ApplyContext, вместо чего данные применяемых контекстов объединялись в один контекст, а все предоставляемые типы хранились в единственном экземпляре в кэше предоставляемых типов. Однако, как показала практика, такой подход ненадёжен и может привести к некорректной инвалидации содержимого ProvidedTypeContext и ошибкам при анализе кода.

Для корректного анализа генеративных типов понадобилось в некоторой степени сохранить логику из FCS: при вызове ApplyContext создаётся легковесная обертка ProxyProvidedTypeWithContext, содержащая переданный ProvidedTypeContext и делегирующая вызовы получения данных обернутому предоставляемому типу из общего кэша предоставляемых типов. Такой подход так же позволяет использовать уже полученные и закэшированные из другого процесса типом данные без дополнительных запросов. Данный контекст передается от

типа-аббревиатуры во вложенные генеративные типы и их содержимое, поэтому соответствующие обертки были реализованы[9] для всей иерархии генеративных элементов. В то же время, данный контекст требуется лишь типам с флагом IsCreatedByProvider и типам-обобщениями, рекурсивно содержащим типы-параметры с флагом IsCreatedByProvider, поэтому данные обертки создаются лишь для них.

Однако польза от такого подхода не ограничивается исправлением существующих проблем в реализации генеративных типов в Rider и позволяет при создании ProxyProvidedTypeWithContext добавлять генеративный тип в кэш генеративных типов. Данный кэш реализован глобально для всех проектов и позволяет получить генеративный ключ по уникальному ключу.

5.2.2. Добавление генеративных типов

Генеративный тип ProxyProvidedTypeWithContext при создании добавляется/перезаписывается в потокобезопасную коллекцию по ключу, состоящему из IPsiModule и CLR-имени генеративного типа из соответствующего ему ILTypeRef-представления в контексте. Данный ключ однозначно определяет генеративный тип на уровне всех проектов.

5.2.3. Инвалидация

Инвалидация кэша генеративных типов потокобезопасна и возможна в двух случаях: при изменении/удалении генеративного типа-аббревиатуры в коде на F# и при инвалидации поставщика типов.

Для инвалидации кэша при удалении генеративного типа-аббревиатуры был реализован компонент ProvidedAbbreviationTypePartInvalidator, который подписывается на событие удаления типа из символьного кэша ReSharper. Данное событие содержит IPsiModule и ITypeElement измененного/удаленного типа. При возникновении события обработчик в

[9] Реализация ProxyProvidedTypeWithContext. URL: https://github.com/JetBrains/resharper-fsharp/pull/300 (дата обращения: 22.05.2022)

ProvidedAbbreviationTypePartInvalidator проверяет, что удаляемый тип является аббревиатурой и отмечает в кэше генеративных типов ключ IPsiModule + CLR-имя из ITypeElement как требущий инвалидации. Кэш генеративных типов добавляет данный ключ в потокобезопасную очередь на удаление требуемых типов и выполняет очистку типа-аббревиатуры и всех содержащихся в его контексте вложенных генеративных типов перед следующим добавлением нового генеративного типа в кэш. Отложенность в данном случае требуется, поскольку событие удаления типа из символьного кэша выполняется на главном потоке и должно занимать как можно меньше времени.

При инвалидации поставщика удаляются все типы, содержащие соответствующий идентификатор поставщика типов.

5.3. Представление генеративных типов в ReSharper PSI

В данной главе рассматриваются особенности реализации совместимых с ReSharper PSI представления для генеративных элементов (генеративных типов и их содержимого).

5.3.1. Генеративный ITypeElement

Класс FSharpProvidedTypeElement содержит общую для генеративных типов-аббревиатур и вложенных типов логику и реализует:

- получение генеративных конструкторов, методов, свойств, полей, событий, операторов, вложенных типов;
- получения имен вышеобозначенных элементов для поиска и отображения в списке для автодополнения;
- получение XML-документации;
- получение дополнительных флагов о наличии тех или иных элементов в содержимом типа (например, наличие публичного конструктора без параметров);

- получение базового типа и реализуемых интерфейсов.

Особым образом реализована конвертация генеративных методов, поскольку список методов у ProxyProvidedTypeWithContext включает в себя так же список методов-ацессоров для свойств и событий, которые в модели ReSharper PSI должны содержаться внутри представлений свойств и событий. Для всех ITypeMember вычисляется XmlDocId, позволяющий уникально идентифицировать элемент содержимого типа. Методы-ацессоры фильтруются из общего списка методов на основе XMLDocId ацессоров свойств и событий генеративного типа.

5.3.2. Тип-аббревиатура

При обращении к содержимому FSharpClassOrProvidedTypeAbbreviation запрашивается соответствующий ему FCS-символ, у которого проверяется свойство IsProvidedAndGenerated, определяющее, является ли тип-аббревиатура генеративным типом. Если проверка истинна и в кэше генеративных типов содержится соответствующий ProxyProvidedTypeWithContext, то на его основе создается FSharpProvidedTypeElement, к которому делегируются запросы на получение содержимого генеративного типа. Если проверка не выполняется, значит тип является обычным типом из F# и запросы на получение содержимого делегируются базовой существующей реализации FSharpClass. Остальные методы, например получение деклараций типа в исходном коде, наследуются от базового класса FSharpClass.

5.3.3. Генеративный элемент

Для остальных генеративных элементов реализован базовый класс FSharpProvidedMember с общей логикой, такой как:

- получение имени элемента;
- получение XML-документации;

- получение генеративного типа, содержащего генеративный элемент;
- дополнительные флаги элемента, такие как область видимости, виртуальность и др.

Остальную логику, требуемую по умолчанию для сгенерированных элементов языка F#, FSharpProvidedMember наследует от существующего класса FSharpGeneratedMember.

На основе FSharpProvidedMember созданы следующий наследники для генеративных элементов:

- абстрактный FSharpProvidedMethodBase с последующими наследниками FSharpProvidedMethod (для методов) и FSharpProvidedConstructor (для конструкторов), реализующими IMethod и IConstructor соответственно;
- FSharpProvidedProperty, реализующий IProperty для свойств;
- FSharpProvidedField, реализующий IField для полей;
- FSharpProvidedEvent, реализующий IEvent для событий;
- FSharpProvidedParameter, реализующий IParameter для параметров методов и конструкторов;
- FSharpProvidedNestedClass, реализующий IClass для вложенных генеративных типов.

5.3.4. Вложенные генеративные типы

Для вложенных генеративных типов реализован класс FSharpProvidedNestedClass, который делегирует получение содержимого FSharpProvidedTypeElement и реализует дополнительные методы, такие как получение пространства имен типа, область видимости, флаги типа и др.

5.4. Алгоритм разрешения генеративных типов

В языковой поддержке F# в Rider есть набор утилит, позволяющий по FCS-символу и IPsiModule создать представление IType/ITypeElement для элементов, объявленных в F#. Данные утилиты были дополнены и для генеративных типов. На рис. 4 представлен алгоритм конвертации FCS-символа в IType.

Если FCS-символ имеет флаг IsProvidedAndGenerative, то есть является символом для генеративного типа, то по его CLR-имени и переданному IPsiModule ищется соответствующий генеративный тип в кэше генеративных типов. Если по какой-то причине такой тип не обнаруживается в кэше (что считается исключительной ситацией), то выполняется логгирование и возвращается неопределенный тип с помощью TypeFactory.CreateUnknownType(). При ожидаемом поведении из кэша возвращается требуемый генеративный тип, для которого осуществляется ряд последовательных проверок:

1. если тип создан поставщиком типов (IsCreatedByProvider) и он:
   - содержится в другом типе – это вложенный генеративный тип, необходимо создать его тип из FSharpNestedProvidedClass;
   - не содержится в другом типе – это генеративный тип-аббревиатура, необходимо получить его из символьного кэша.
2. тип является массивом или указателем – это стандартный тип, требуется рекурсивно получить тип его элемента и создать тип-массив/тип-указатель;
3. тип не является обобщенным – это библиотечный тип, необходимо получить его тип из символьного кэша;
4. тип является обобщенным – требуется рекурсивно получить обобщенное определение типа, типы-параметры и создать обобщенный тип.

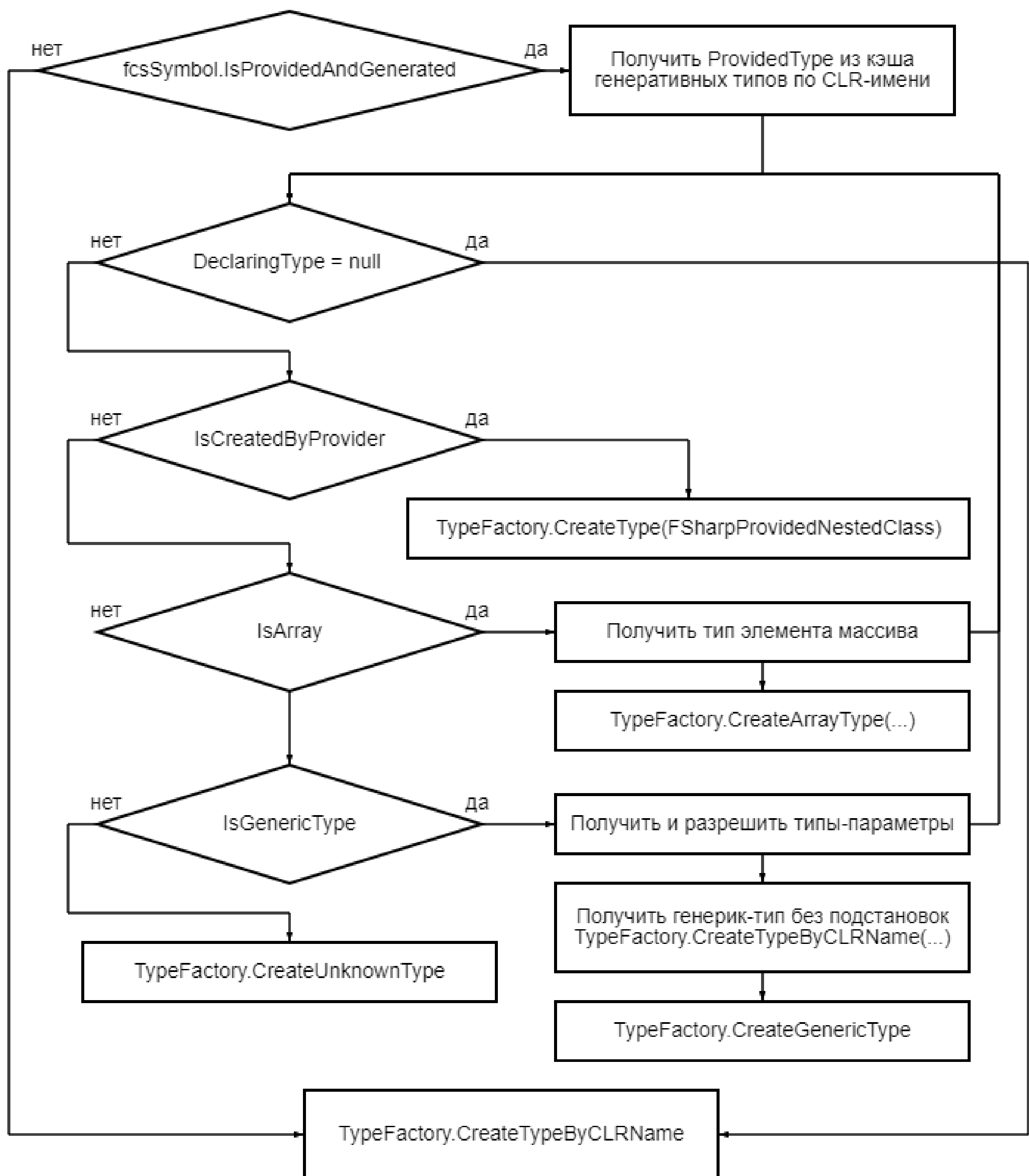

Рис. 4: Отображение символа FCS в IType

Из полученного в результате IType можно получить ITypeElement вызовом соотвествующего метода IType.GetTypeElement().

5.5. Дополнительная функциональность в IDE

Генеративные элементы должны поддерживать в проектах на C# такие действия в IDE как навигация к декларации элемента (позволяет отобразить пользователю тип, в котором определен соответсвующий элемент) и рефакторинг переименования, как это сделано для остальных элементов языка F#.

5.5.1. Навигация к декларации

Явная декларация в коде имеется у типа-абревиатуры и отсутствует для вложенных в него генеративных типов и их содержимого. В то же время тип-аббревиатура содержится в символьном кэше ReSharper. Поэтому навигация для типа-аббревиатуры будет работать автоматически, а для вложенных генеративных элементов была реализована навигация в соотвествующий тип-аббревиатуру. Для этого потребовалось реализовать в базовом для вложенных генеративных элементов классе FSharpProvidedMember интерфейс ISecondaryDeclaredElement, свойство OriginElement. OriginElement, будучи типом-аббревиатурой, определяется как первый родительский тип в иерархии содержащих друг друга вложенных типов, который затем запрашивается из символьного кэша.

5.5.2. Переименование

Алогично с навигацией к декларации, переименование автоматически работает для типов-аббревиатур. Для остальных генеративных элементов не может быть доступен рефакторинг переименования, а потому его необходимо ограничить. Для этого интерфейс
ISecondaryDeclaredElement был расширен булевым флагом IsReadonly, позволящим определить, доступна ли модификация элемента, реализующего интерфейс, в рефакторингах. Реализация интерфейса в

FSharpProvidedMember возвращает false для данного флага, механизм переименования символов в F# был дополнен соответствующей проверкой.

6. Тестирование и апробация

6.1. Прототип

В ходе данной работы был реализован прототип для плагина языковой поддержки F# в Rider. На рис. 5 показано как элементы, сгенерированные поставщиком типов YamlConfig для типа-аббревиатуры GeneratedConfig, без ошибок анализируются в проекте на C#.

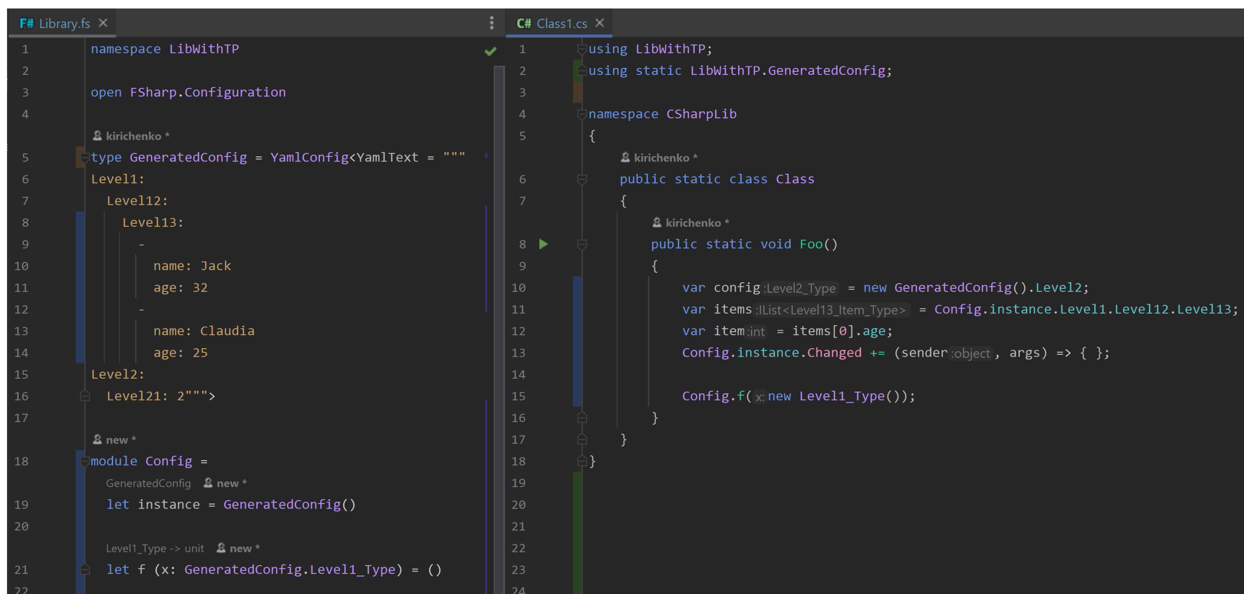

Рис. 5: Анализ поставщика типов YamlConfig в C#-проекте

6.2. Написание автоматических тестов

Для проверки корректности реализованной подсистемы и отслеживания возможных регрессий в дальнейшем были написаны автоматические интеграционные тесты для различных сценариев использования генеративных поставщиков типов в проектах на C#. В интеграционных тестах в изолированном окружении запускается полноценный Rider, позволяющий эмулировать действия пользователя. В качестве тестовых данных используется подготовленный проект на C#, который ссылается на один или несколько проектов на F#, использующих генеративные поставщики типов.

6.2.1. Анализ генеративных типов и их содержимого

В данной категории тестов открывается файл с исходным кодом на C#, в котором используются генеративные типы и проверяется:

- отсутствие ошибок анализа кода при обращении к генеративным элементам;
- отображение корректной XML-документации, сгенерированной поставщиками типов для генеративных элементов;
- отображение корректных подсказок автодополнения для генеративных элементов.

6.2.2. Инвалидация генеративных типов

В данной категории тестов сместе с файлом исходного кода на C# открывается файл с исходным кодом на F#, содержащий используемые генеративные типы. Проверяется состояние кэша генеративных типов, возникновение/отсутствие ошибок в файле на C# при изменении статических аргументов генеративного типа (в том числе, если в качестве аргументов используется константа, определенная в отдельном файле на F#), изменении/удалении типа-аббревиатуры.

6.2.3. Загрузка и выгрузка проектов

В данном тесте проверяется, что после выгрузки и последующей загрузки проектов на C# и F# в IDE анализ в проекте на C# происходит успешно, данные в кэше генеративных типов корректно перезаписываются.

6.2.4. Дополнительная функциональность в IDE

В данной категории тестов открывается файл с исходным кодом на C#, выполняется одно из действий и проверяется, что:

- при навигации из типа-аббревиатуры/вложенного генеративного типа/генеративного метода открывается декларация типа-аббревиатуры в F#-файле;

- переименование типа-аббревиатуры в файле на C# переименовывает тип-аббревиатуру в файлах на C# и F#, ошибки анализа кода отсутствуют;

- переименование недоступно для вложенных генеративных типов и других генеративных элементов.

6.2.5. Одинаковые ссылки на генеративные типы

В данном тесте проверяется, что в случае одинаковых CLR-имен у двух типов-аббревиатур в разных F#-проектах в файле на C# используется правильный тип из проекта, на который есть ссылка в C#-проекте.

6.3. Ручное тестирование с помощью FSharp.Data и OpenApiProvider

Для ручного тестирования был создан проект на C#, ссылающийся на несколько F#-проектов, содержащих поставщики типов OpenApi и YamlConfig. В интерактивном режиме проводилось добавление и удаление NuGet-пакетов с провайдерами, выгрузка проектов, редактирование кода в проекте на C# и F#, включающая в себя изменение статических параметров генеративных типов, изменение типов-аббревиатур, добавление и удаление ссылок на проекты, доступность рефакторинга переименования и навигации к декларации. Для корректного кода на C# проверялась успешная компиляция проекта, при возникновении ошибок в редакторе проверялось наличие соответсвующих ошибок и при компиляции.

6.4. Выводы

По результатам автоматического и ручного тестирования были обнаружены и исправлены некоторые недоработки. В конечном итоге разработанная подсистема анализа генеративных поставщиков типов продемонстрировала свою работостопособность и позвляет достичь поставленную в данной работе цель.

7. Ограничения

В ходе выполнения данной работы не было реализовано совместимое с ReSharper PSI представление для генеративных типов-интерфейсов, из-за чего на данный момент они не могут участвовать в кросс-проектном анализе – данное поведение зафиксировано соответствующими проверками в коде и может быть доработано в случае необходимости после апробации на реальных пользователях.

8. Заключение

В рамках данной работы были достигнуты следующие результаты:

- Изучены механизмы статического связывания генеративных поставщиков типов в FCS, исправлены ошибки кэширования генеративных типов в Rider.
- Разработана архитектура подсистемы кросс-языкового анализа генеративных поставщиков типов с учетом механизмов анализа кода и системы типов ReSharper.
- Для Rider реализованы кэш генеративных типов, алгоритмы обнаружения и разрешения созданных поставщиками генеративных типов, а также совместимые с ReSharper PSI представления для генеративных типов, типов-аббревиатур и их содержимого.
- Реализована функциональность навигации к типу-аббревиатуре для генеративных типов и их содержимого, ограничена функциональность переименования содержимого генеративных типов.
- Выполнено тестирование и апробация реализованной подсистемы: написаны автоматические тесты для основных сценариев использования генеративных поставщиков типов в кросс-проектном анализе и произведено ручное тестирование работоспособности на поставщиках типов из бибилиотек FShar.Data и OpenApiProvider.

В дальнейшем планируется внедрение реализованного в данной работе решения[10] в новую версию Rider и его апробация пользователями в рамках использования в реальных индустриальных проектах.

[10]Кросс-проектный анализ генеративных поставщиков типов. URL: https://github.com/JetBrains/resharper-fsharp/pull/355 (дата обращения: 22.05.2022).

Список литературы

[1] Auduchinok Evgeniy. Implementation of F# language support in JetBrains Rider IDE. — 2017. — URL: http://se.math.spbu.ru/SE/diploma/2017/pi/Auduchinok.pdf (дата обращения: 14.03.2022).

[2] Berezhnykh Alexey. Hosting F# type providers out-of-process. — 2020. — URL: https://oops.math.spbu.ru/SE/diploma/2020/bmo/Berezhnykh-report.pdf (дата обращения: 14.03.2022).

[3] F#Software Foundation. F# Compiler Services. — 2019. — URL: https://fsharp.github.io/FSharp.Compiler.Service/ (дата обращения: 07.05.2022).

[4] JetBrains. ReSharper Declared Elements. — 2022. — URL: https://www.jetbrains.com/help/resharper/sdk/PSI/DeclaredElements.html (дата обращения: 14.04.2022).

[5] Microsoft. Учебник. Создание поставщика типов. — 2022. — URL: https://docs.microsoft.com/ru-ru/dotnet/fsharp/tutorials/type-providers/creating-a-type-provider (дата обращения: 06.04.2022).

[6] Syme Don. The Early History of F#. — 2020. — URL: https://fsharp.org/history/hopl-final/hopl-fsharp.pdf (дата обращения: 14.05.2022).

[7] Syme Don. Generative type providers analysis in F# compiler service. — 2022. — URL: https://github.com/dotnet/fsharp/issues/3234#issuecomment-579396231 (дата обращения: 22.05.2022).

[8] Сошников Дмитрий. F# – компромисс между академическим миром и реальной жизнью. — 2013. — URL: http://samag.ru/archive/article/2371 (дата обращения: 07.05.2022).