SAINT-PETERSBURG STATE UNIVERSITY

Software Engineering

Evgeniy Auduchinok

Implementation of F# language support in JetBrains Rider IDE

Graduation Thesis

Scientific supervisor:
Senior lecturer Iakov Kirilenko

Reviewer:
Software engineer Dmitry Ivanov

Saint-Petersburg
2017

Contents

Introduction

F# [5] is an open-source functional-first programming language that was designed to be used in .NET, a software framework developed by Microsoft. .NET Framework contains a large class library and a powerful runtime called Common Language Runtime (CLR). F# was highly influenced by OCaml, an ML family language. Code written in F# may inter-operate with code written in other .NET languages such as C# and VB.NET and, being a general-purpose programming language, is especially popular in data analysis, web-development and as a language for rapid prototyping and scripting. F# was designed by Don Syme, Microsoft Research and is developed by F# Software Foundation, Microsoft and open-source contributors.

An integrated development environment (an IDE) provides an easy navigation through source code, shows warnings and errors found with offering possible fixes and allows users to do refactorings across multiple files along with other useful features. In addition to code reading and editing features, an IDE usually includes tools for building, unit-testing and publishing code and version control systems integration.

JetBrains Rider[1] is a cross-platform .NET IDE based on JetBrains IntelliJ Platform[2] and JetBrains ReSharper[3].

IntelliJ Platform is an open-source platform developed by JetBrains and is used to develop IDEs. It is most known for JetBrains IntelliJ IDEA[4], an IDE for JVM languages such as Java and Kotlin and for Google Android Studio aiming mobile development. Its core components include rich text-editor, UI framework, virtual file system, version control integration, debugger framework and may be further extended with plugins [2].

ReSharper was initially developed as an extension for Visual Studio, a .NET IDE from Microsoft. It provides additional code inspections along

[1] jetbrains.com/rider
[2] github.com/JetBrains/intellij-community
[3] jetbrains.com/resharper
[4] jetbrains.com/idea

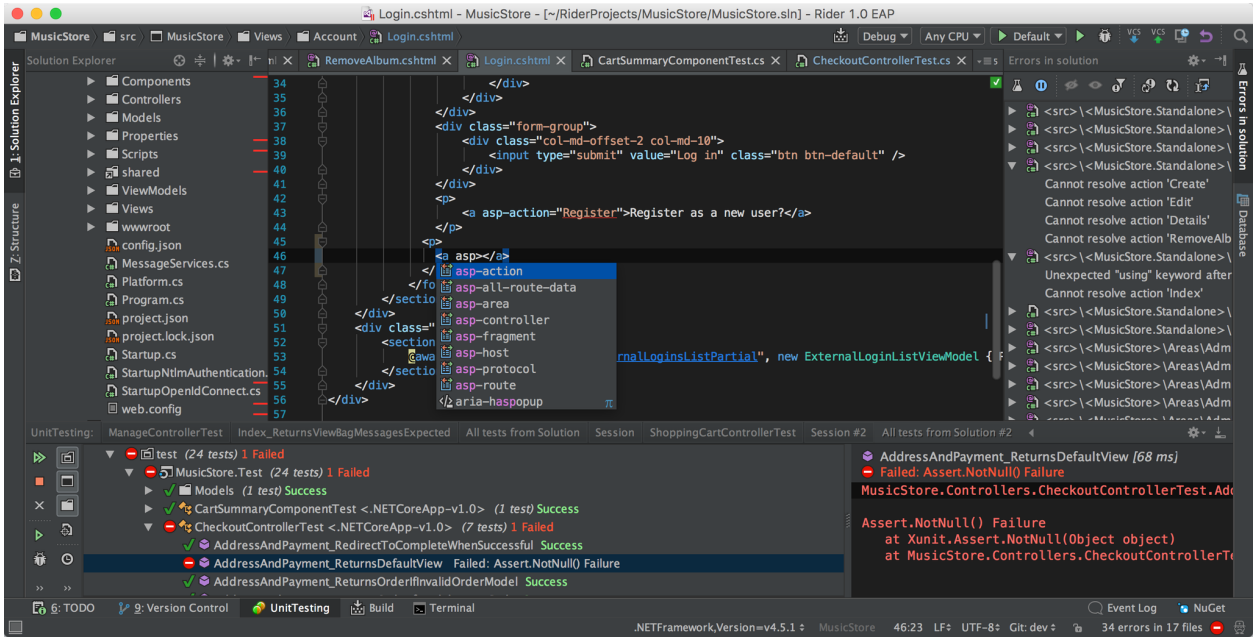

Figure 1: A screenshot of JetBrains Rider with invoked code completion feature, test run results and a list of errors found in solution.

with possible fixes, complex refactorings and a better source code navigation. ReSharper supports C#, VB.NET and languages used in web-development such as JavaScript, CSS and HTML. In Rider it offers the same features it has when is used in Visual Studio.

Rider supports languages that ReSharper or IntelliJ support but neither had F# support before this work and many Rider preview users wanted to use the language in it. This work is aimed to bring an F# support to Rider.

1 Statement of the problem

A language support in an IDE consists of several parts. Usually code being analyzed belongs to a project, so an IDE should be able to work with project systems used by developers using that language. An IDE text editor should help user reading the code by highlighting it syntactically and semantically. Further analysis should result in showing warning and errors, offering possible fixes. Additionally, the editor should have features like commenting a line of code or extending selection range according to the language constructs. Usually an IDE has refactorings support allowing users to rename defined symbols, extract and inline expressions. Another important part of such support is a navigation in a project. It includes a global search of symbol definitions as well as a navigation to their uses across the codebase.

The purpose of this work is to implement an initial F# support in JetBrains Rider which would consist of several tasks:

- design an implementation,
- implement language specific features including:
  - code analysis and editor features,
  - working with F# projects,
  - code analysis and navigation,
  - building, running and debugging code,
- publish an F# support plugin.

2 Overview

Rider is built on top of IntelliJ platform, a frontend, which is written in Java and Kotlin and runs on JVM, and ReSharper, a backend, mostly written in C# and executed in CLR. These parts run in separate virtual machines and communicate using RdProtocol, a custom asynchronous stateful protocol designed for Rider. The protocol uses a DSL written in Kotlin and generates Kotlin and C# files for corresponding Rider parts.

F# Software Foundation maintains an open-source F# compiler library called FSharp.Compiler.Service[5] (FCS) that provides APIs designed to be used by editors and IDEs. FCS is already used by existing F# support implementations such as Visual F# (Visual Studio plugin), FsAutoComplete (used by Ionide[6] plugins for Visual Studio Code and Atom editors) and other tools.

While support for most languages in ReSharper and IntelliJ was implemented from scratch, FCS and other tools may be used for implementing F# support in Rider. Reusing existing tools allows us to offer features other F# implementations already have. FCS is actively maintained by the F# community and no substantial extra work would be needed in the future to support newer versions of the language because FCS shares its codebase with the actual language compiler and updates accordingly.

FCS provides a variety of APIs that produce ready-to-use info like data needed for source code navigation, based on parse trees, signature tooltips or semantic highlighting ranges. At the same time FCS offers APIs providing data to be further processed to implement other features on top of it. For example, it includes APIs like getting all symbol uses in a project, as seen by F# language, which may be used to implement a rename refactoring.

Several APIs are implemented in both ways. Such APIs include two ways of getting possible code completions at a location: there is an option to return a list of completions with ready-to-insert text and preformatted

[5] github.com/fsharp/FSharp.Compiler.Service
[6] ionide.io

signatures tooltips and an option returning list of symbols available in the context. The latter option provides more useful data but involves a need of processing these symbols manually. Tools such as Visual F# mostly use APIs that yield prepared data and filter or modify it when needed. When FCS does not provide an API with info needed it is possible to offer a patch to the project thanks to its being open-source.

Other F# support implementations are open-source and may be used as an FCS usage reference.

3 Implementation design

Both platforms used in Rider, IntelliJ and ReSharper, provide SDKs allowing to implement new features and languages support. There are several possible ways of implementing an F# support in Rider. One way is to use FsAutoComplete tool that itself uses FCS and provides command line interface, thus it may be used by programs written in an arbitrary language. If FsAutoComplete was used it would be possible to implement an IntelliJ plugin in Kotlin that could be used not only in Rider but in other IntelliJ-based IDEs as well. However, a downside of this approach would be a lack of F# integration with other .NET languages supported by ReSharper.

Using FCS in a ReSharper plugin referencing ReSharper.SDK may be used to add an F# support to ReSharper and eventually getting it in Rider. In that case an IntelliJ plugin targeting Rider should be implemented as well to notify the platform that ReSharper should be used for files in this language and to implement some features on the frontend part.

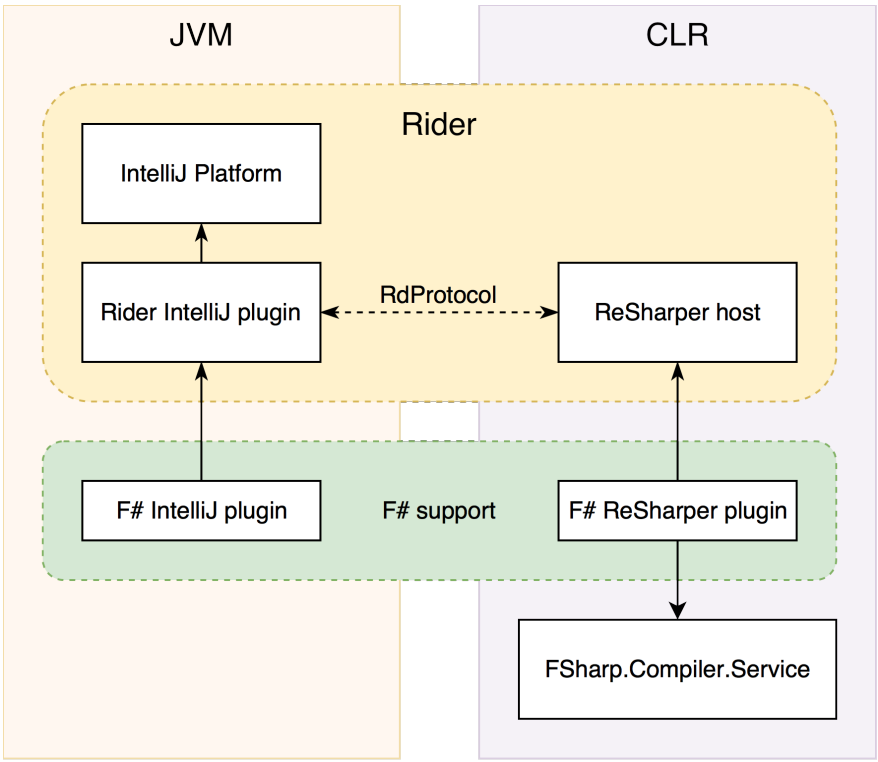

Figure 2: A simplified integration design.

FCS has an internal background compiler and uses its checks results. The checking is done in two stages: a parsing and a type-check. Each stage yields data that may be used for implementing various features. The parsing stage is fast and may be used for creating navigation cache, highlighting syntax errors and validating possible breakpoint locations. The type-checking stage uses parse results and involves inferring types, checking other files in the project and a process of attempting to generalize defined symbols. The results of the latter stage include, but are not limited to, semantical errors and warnings, uses of symbols found in a file. FCS provides APIs like getting possible completions at a location, showing tooltips and methods overload signatures using these type-check results as well.

The FCS package also contains other types used by the compiler which may be reused as well. For example, the lexer included may be used for basic syntax highlighting. Lexer helpers may be used to check whether an identifier is valid or to get the list of keywords defined in the language.

4 Language specific features implementation

4.1 Code analysis and editor features

ReSharper provides many extension points using inversion of control [7] and dependency injection [6] patterns. A language support must implement language-specific types and services to be used by other ReSharper parts. It includes a definition of a language and file extensions associated with it, lexer and parser factories, feature-specific services like a searcher provider or a reference processor.

Lexer factory needed by ReSharper must create a lexer based on a text buffer passed in. Later, a token buffer, created by resulting lexer, is used to create a cached lexer which in turn is passed to a parser factory. Configuration-defined constants used by conditional compilation process do not appear before the parsing stage. However, these constants are needed by FCS lexer but are not passed to ReSharper lexer factories. This limitation resulted in implementing a dummy lexer factory to satisfy ReSharper requirement and swapping its results later with a proper lexer implementation results during a parsing stage.

Language-specific daemon, a ReSharper background code analysis task, was implemented. It is used to highlight source code semantically and to present warnings and errors found by FCS. The implemented daemon is also used to map symbols resolved by FCS to ReSharper representations allowing to navigate to linked elements like base and derived symbols, or to navigate between signatures and implementations of symbols.

F#-aware editor actions were implemented on top of FCS APIs and other parts introduced during this work. For example, commenting a line is done using the implemented F# lexer as well as highlighting matching braces. At the same time, highlighting usages of a symbol under a caret is done using FCS API which is based on a file type-check results.

4.2 Working with F# projects

ReSharper and Rider, being very extensible, have rare places which were not previously needed to be extended with plugins. For example, both have a concept of a project language for projects targeting .NET. Supported project languages are defined internally and there was no way to define one in a plugin. This lead to problems like being unable to create caches for cross-language navigation and resolve process or even having no way to run an F# console application or add it to existing solutions. Patches allowing defining new .NET project languages externally were pushed to ReSharper and Rider.

Types defining F# project language, project properties and configurations were implemented as well.

FCS performs parsing and type-checking in a context of some project. A context is set by project options containing files to check, paths to referenced assemblies, configuration-specific compiler arguments [4] and referenced projects options. Files in a project are checked in the order defined in the project file which is different from C# and VB.NET projects and is needed to be handled properly. In the case of F# scripts single-file project options should be used.

FCS project includes a tool for producing suitable project options using project files called ProjectCracker[7]. It starts an MSBuild instance, a build engine used in .NET projects, and creates project options using its results. Rider itself does the same job of reading these project files with MSBuild, adding parsed projects (with their properties including files names and references) to its project model so running another MSBuild instance did not seem reasonable. Additionally, using ProjectCracker on macOS and Linux from a ReSharper plugin would make it run on Rider's bundled Mono which differs from Mono installed in the user system and due to modified environment may produce wrong results or not work at all. Rider includes a part to communicate with MSBuild called MsBuildHost, which is run in a different process using .NET runtime

[7] nuget.org/packages/FSharp.Compiler.Service.ProjectCracker/

installed in the system and communicates with ReSharper using RdProtocol.

An implemented project options provider tracks changes to the project model, updates and invalidates project options notifying FCS when needed. It creates project options using results from MsBuildHost by obtaining project files paths in the right order and other properties needed by FCS.

4.3 Source code navigation

FCS provides API for creating navigation cache and this API is used in other implementations such as Visual F#. This API uses parsing stage results, thus it is fast enough, and provides info about symbol names and places they are defined at. However, this cache does not provide enough information needed by ReSharper because the very same cache is later used for analysis as well.

For each type defined in source code or in an assembly ReSharper solution cache also includes its kind (class, interface, struct, etc), type parameters, names of types this type implements or is derived from, its attributes and modifiers (like abstract or static in C#) and so on. This cache is built on top of Program Structure Index (PSI) [3], ReSharper abtract syntax trees with symbol representations (called declared elements in there) and powerful mechanisms of references and types. Generally speaking, most of features in ReSharper are implemented on top of PSI, not only this particular cache. Types of PSI tree nodes implement various interfaces defined in ReSharper allowing a great part of analysis to be language-independent.

Initial plugin prototype did not have a PSI implementation for F#. For most features it had it used offsets of tokens in files that were converted from FCS APIs results which use line and column coordinates. By using this technique it was possible to implement features like errors highlighting or finding symbol uses across F# projects but it did not integrate well into cross-language navigation and analysis in ReSharper. It was clear then that

PSI was needed for a better user experience.

ReSharper.SDK includes lexer and PSI-compatible parser generators that are used in ReSharper itself. Implementing an F# parser from scratch did not seem reasonable in the amount of time and other ways were considered. As the most of code analysis is being done by FCS, ReSharper does not actually need a true PSI for F# files. This lead to idea of creating a partial PSI for F# files that would contain crucial info needed by navigation system and cross-language analysis. During the parsing stage FCS produces abstract syntax trees that contain ranges of tree nodes and these trees can be converted to PSI with a needed structure.

Parser generator bundled with ReSharper.SDK was used to generate PSI data structures using a simplified F# grammar written for this case. ReSharper.SDK also contains a base class for tree builders that suites well for this task and was used to implement an F# AST walker that creates a needed PSI structure.

Some of the info needed for types cache is not contained in FCS ASTs and, subsequently, in converted PSI trees. For example, in F# object types are defined using 'type' keyword and actual type kinds are computed later during the type-check. The F# 4.0 specification [1] includes rules that are used during kind inference process and are applied to type members and attributes. Unfortunately, applying these rules to ASTs is not correct in general case due to a possibility to abbreviate attribute types making types with such attributes compile with a different kind. However, after analyzing popular F# projects it was clear that this approach still may work well in the most cases. Proper resolving type abbreviations in cases like this can be done using a separate cache for abbreviated types in the future. In addition to a partial structure of types and members, F# cache provider was implemented. It is used to declare these symbols in the solution cache that is used by ReSharper navigation and the type system for other analysis.

ReSharper type system includes representations for all compiled types defined in source code and assemblies. It is used to compute types of expressions in code and parameters and return types of type members, substituting type parameters when needed. These types are later used by

other analysis including type usage inspections. To make F# work with other languages in ReSharper, types inferred by F# compiler must be properly mapped to their ReSharper representations. As for now, a significant amount of types is converted properly making it possible to use these types from projects using other languages. This part is considered to be work in progress but in its current state it already allows it to be used.

ReSharper uses compiled assemblies to resolve found symbols to when source code is not available or the language used is not supported. It means that when a project in an unsupported project is changed it has to be compiled to make changes visible from projects using supported languages. Implementing the F# cache made it possible to work with symbols defined in F# without building F# projects first.

FCS maintains its own type-check results cache. Many of its public APIs provide a subset of this data limited by what is mostly needed for F# analysis. Additional APIs providing internal data like compiled symbols representations may be used for better ReSharper integration and to avoid duplication of some logic. One such patch was pushed to the FCS project. At the same time, reusing internal FCS cache may improve performance on very large F# projects.

4.4 Building, running and debugging code

F# projects use the same build system as C# and VB.NET do, so building these projects has been supported in ReSharper before this work.

Running and debugging code is handled by corresponding Rider subsystems. Pushing patches to Rider and ReSharper allowing to add .NET project languages externally made it possible to reuse these mechanisms.

ReSharper bundled with Rider contains types that aim to replace missing features previously offered by Visual Studio. Some of these types are not public until a separate Rider SDK release. This was the case for an interface to provide possible breakpoint ranges on a line. This interface was made public and implemented in the plugin allowing to set

breakpoints in F#. For debugging F# code the same debugger is used as for other .NET languages.

5 Conclusion

During this work an initial F# support plugin was implemented which is now bundled in JetBrains Rider allowing its users to use F# language. The following tasks were done:

- an FCS integration was designed,
- language specific features were implemented including:
  - code analysis and editor features,
  - working with F# projects,
  - source code navigation,
  - running and debugging F# code.
- an F# support plugin is now bundled in Rider.

The plugin source code is going to be available at github.com/jetbrains/resharper-fsharp.

References

[1] F# 4.0 specification. — URL: http://fsharp.org/specs/language-spec/4.0/FSharpSpec-4.0-latest.pdf.

[2] JetBrains. IntelliJ Plugin Development Guidelines. — URL: https://www.jetbrains.com/help/idea/2017.1/plugin-development-guidelines.html.

[3] JetBrains. ReSharper DevGuide. — URL: https://www.jetbrains.com/help/resharper/sdk/Architecture/PSI.html.

[4] Microsoft. F# compiler options. — URL: https://docs.microsoft.com/en-us/dotnet/articles/fsharp/language-reference/compiler-options.

[5] Syme Don, Granicz Adam, Cisternino Antonio. Expert F#. — Apress, 2007. — ISBN: 978-1-4302-0285-1. — URL: http://dx.doi.org/10.1007/978-1-4302-0285-1_19.

[6] Wikipedia. Dependency injection. — URL: https://en.wikipedia.org/wiki/Dependency_injection.

[7] Wikipedia. Inversion of control. — URL: https://en.wikipedia.org/wiki/Inversion_of_control.