University of Applied Sciences and Arts Northwestern Switzerland FHNW

Master of Science in Engineering

Project 8

Improving the Frege developer experience

Autor	Tobias Wyss, tobias.wyss@students.fhnw.ch
Supervisor	Prof. Dierk König
Start	17.02.2025
Submission	15.08.2025

Abstract

The Language Server Protocol [1] standardizes the interface between text editors (clients) and programming language-specific tooling (servers). To improve the developer experience for Frege users, we created a new language server for Frege [2] in this thesis. It builds upon ideas from the existing implementation [3] but is now written entirely in Frege. In addition to the features from the previous server, it offers code completion, code navigation, and faster response times by reacting to document change notifications [4] from the client. The implemented server is easily extensible and thoroughly testable, providing a solid foundation for future development.

Table of Contents

1 Introduction

This chapter gives a brief overview over the achieved results during this project work, the initial problem statement and describes the structure of the document.

1.1 Results

During this project we created a new implementation of the Language Server Protocol (LSP) [1] for the Frege programming language [2]. Compared with the existing language server [3], the new implementation offers more features and is written entirely in Frege.

1.1.1 Frege Language Server features

The language server developed in this project follows the LSP life cycle (Section 2.2.3 *"Protocol life cycle"*) and offers the following features:

1. Go to definition: jump to the definition of the symbol under the cursor
2. Code completion: suggest code for the symbol preceding a dot
3. Show compiler information: compile the open source file and forward compiler messages to the text editor
4. Hover: show type signatures and documentation for the symbol under the cursor

All features rely on tight cooperation with the Frege compiler. The language server uses the compiler to compile files opened and edited by the user, then extracts the required data from the compiler results and presents it to the developer.

Figure 1 shows the Frege Language Server in action, suggesting methods and properties accessible with the dot on a `String`:

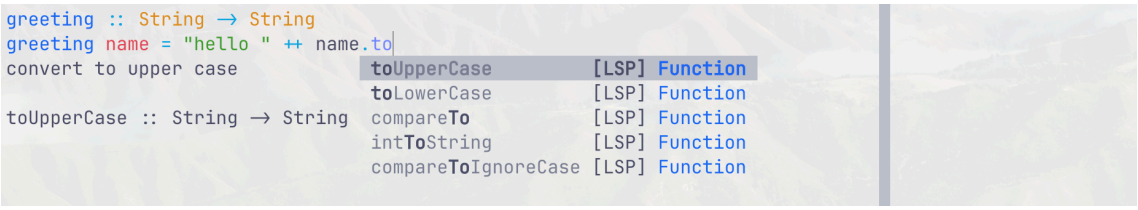


Figure 1: The Frege Language Server auto completes based on the symbol preceding the dot.

The implemented features greatly benefit Frege developers and also solve many issues that will recur in the further development of this language server. This foundation allows adding new features quickly.

1.2 Project environment

1.2.1 Frege

"Frege" is a functional programming language that compiles to Java source code and runs on the JVM [2]. Frege's syntax and features closely match Haskell Standard 2010 [5], [6], making it easy for Haskell developers to learn. However, Frege's native declarations enable developers to integrate existing Java code and libraries. Frege thus combines strong typing and functional programming with Java's extensive ecosystem [7].

Listing 1 shows the entry point of a Frege program. When executed, the program prints "Hello Frege" to standard output.

```
1 main :: IO ()
2 main = println "Hello Frege"
```

Listing 1: A small Frege program

1.2.2 Project goals

Although Frege offers significant benefits [8], it remains largely unknown with only a small user base. As a result, it lacks modern and stable tooling [9].

This project aims to improve the developer experience for Frege users. It enhances IDE integration by providing a new implementation of the Language Server Protocol for Frege.

1.3 Reader guidance

The report begins in Chapter 2 *"The Language Server Protocol"* with explaining the basic structure of the Language Server Protocol. Then it continues with Chapter 3 *"Frege Language Server features"* and explains the features the Frege language server supports. Chapter 4 *"Frege Language Server implementation"*

describes the architecture and implementation of the Frege Language Server. The detailed view of the project results provides Chapter 5 *"Discussion"*.

2 The Language Server Protocol

This chapter outlines why the Language Server Protocol exists and how it works.

2.1 What is the Language Server Protocol?

The Language Server Protocol specifies how text editors and programming language-specific tooling communicate [10]. Many editors and languages support the protocol [11], [12].

2.1.1 The n:n problem

Creating editor support for a programming language requires significant effort. Developers must write an adapter that communicates with the language's compiler and adheres to each text editor's interface. Supporting another editor means writing a new adapter. This approach produces many similar adapters that quickly become unmaintainable.

Figure 2 illustrates how supporting n editors requires n dedicated adapters, each performing similar tasks tailored to the specific editor:

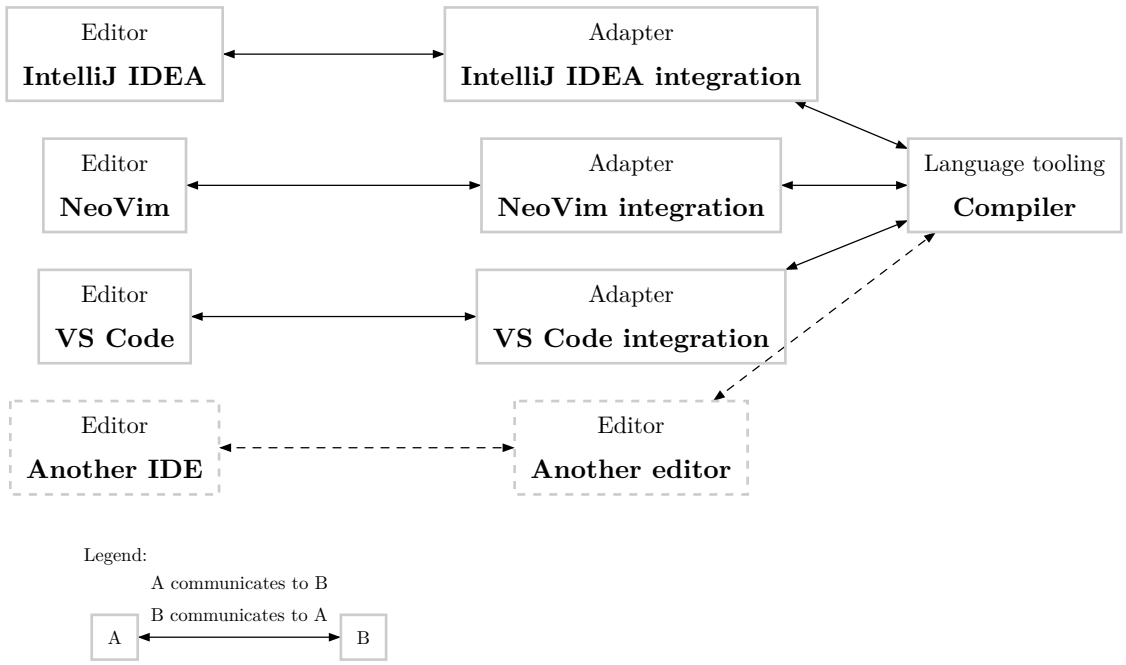


Figure 2: Supporting different editors leads to a lot of work.

2.1.2 LSP solves this problem

That is exactly where LSP comes in to play: it standardizes the way how an editor (client) and an adapter (server) communicate. This leads to reusable servers for different clients.

Figure 3 shows that tooling creators adhering to the Language Server Protocol, only need to create one server [13]. This sole server is able to communicate to different clients:

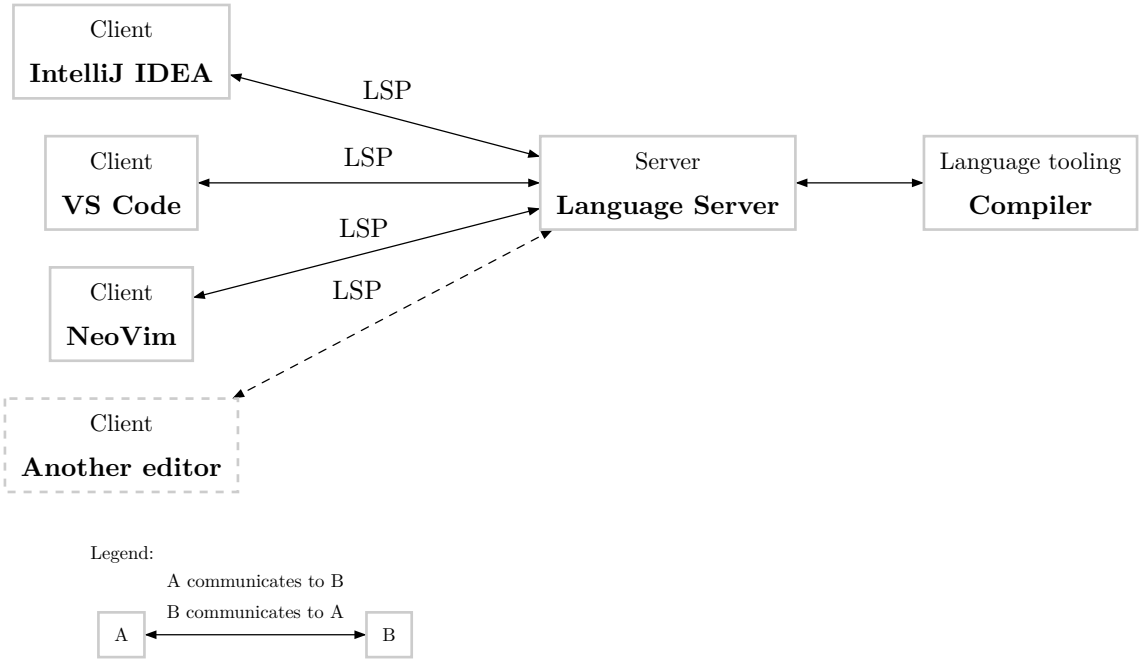


Figure 3: How the Language Server Protocol works

Supporting another text editor is easy, as tooling creators only need to bridge from the client to the server. In NeoVim, for example, this is achieved with approximately five lines of code [14].

2.2 Protocol Architecture

The client usually starts the language server as a subprocess and sends messages to the server's standard input. These messages use JSON-RPC to describe their content [13].

2.2.1 Communication

The Language Server Protocol defines a set of structured messages and a server life cycle. The next sections provide a brief overview of these topics.

2.2.1.1 Message Structure

Messages following the Language Server Protocol consist of a header and a content part. The header contains metadata, while the content holds the actual message [15].

Listing 2 shows a completion message sent from the client to the server. The Content-Length field specifies the message length in bytes [16].

```
Content-Length: ...\r\n
\r\n
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "textDocument/completion",
  "params": {
    ...
  }
}
```

Listing 2: An example LSP message

2.2.2 Message types

The Language Server Protocol knows three types of messages [17]:

1. Request
2. Response
3. Notification

The client sends a request when it expects a result from the server. The server processes the message and responds to the client. Each request includes a unique message identifier (ID), which the server uses to link responses to their corresponding requests.

Notification messages on the other hand do not expect a response. They therefore not include an ID. Both the client and the server use notifications to inform the

receiver about an event, such as “the user opened a file” (client to server) or “these are the compiler messages to file XY” (server to client) [13].

2.2.3 Protocol life cycle

The protocol does not define when and how the client starts and shuts down the server. However, the protocol specifies messages which the client and the server exchange during these events, called life cycle messages [18].

Figure 4 outlines the life cycle in a sequence diagram:

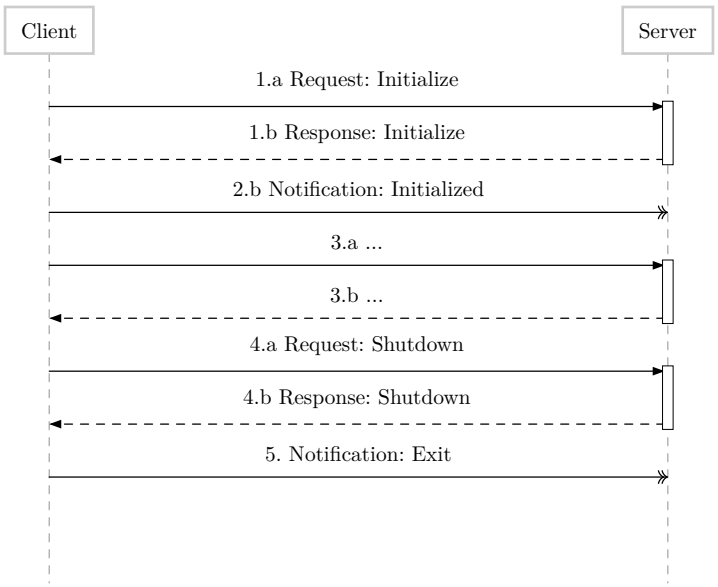


Figure 4: The Language Server Protocol’s life cycle messages

1. a. The client notifies the server about its capabilities. Capabilities define what parts of the LSP a client or a server support.
1. b. The server responds with its capabilities. Now client and server know which message and notifications the other end supports
2. The client states the successful retrieval of the server capabilities.
3. a. The client is able to send any requests and notifications to the server.
3. b. The server on its side responds to messages or sends notifications to the client.
4. a. The client indicates that it wants to shutdown the server. The server prepares itself to shutdown

4. b. Once the server is ready to shutdown, it informs the client.
5. The client sends one last notification, allowing the server to shutdown and the server terminates itself.

The protocol defines how to handle messages outside the life cycle in [18].

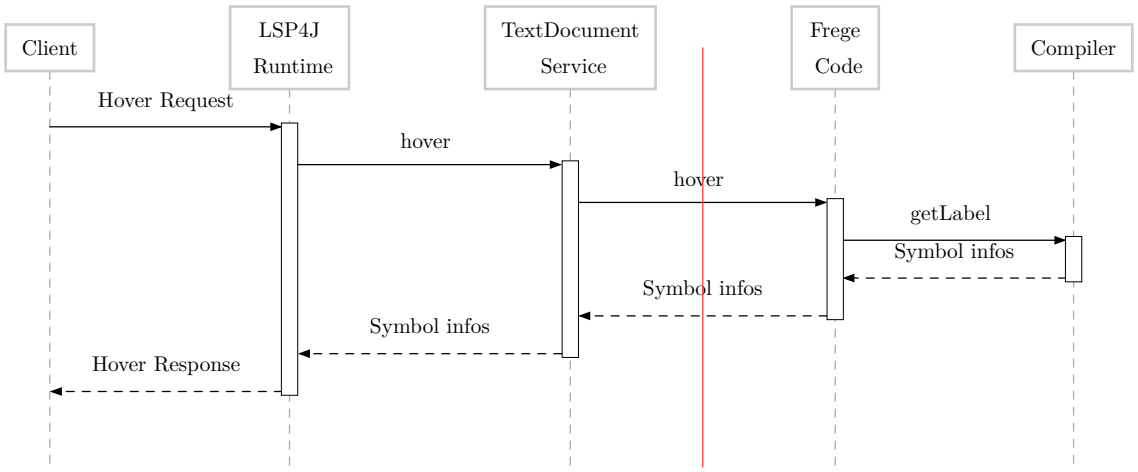
3 Frege Language Server features

This chapter explains why we developed a new language server for Frege from scratch and describes its supported features. It concludes by outlining the current limitations of the Frege Language Server.

3.1 Why we created a new language server

Frege already has a language server [19], which uses the Java library LSP4J [20] to implement the protocol. This avoids implementing the protocol from scratch, but requires handling LSP messages in Java, bridging them to Frege for compiler interaction, then translating the results back to Java to send them via LSP4J.

Figure 5 shows the conceptual architecture of the current Frege language server at the example of a hover request:



Note: Everything on the left of the red line is Java code, everything on the right is Frege.
Figure 5: The architecture of the previous Frege Language Server

When the LSP4J runtime receives a hover request, it passes it to a Java class implementing the `TextDocumentService` interface. This interface then forwards the request to Frege code, which communicates with the compiler.

This setup has drawbacks:

1. Each LSP feature adds a new Java-to-Frege boundary crossing (red line), leading to code which is difficult to navigate using IDE features, since Java only understands the transpiled Frege code. Ideally, only one crossing point should exist when combining Java with Frege.
2. Frege offers strong type safety. Crossing the boundary weakens it. For example, the ADT `Maybe` loses enforced handling of the `Nothing` case.
3. Testing becomes harder due to the involvement of the LSP4J runtime.

For these reasons, we chose to rewrite the language server entirely in Frege. However, the existing server served as a blueprint for compiler communication. All features known from the existing server are supported by the new server as well.

3.2 Features

This section describes the features the new Frege Language Server supports by now.

3.2.1 Document synchronization

The editor tracks code changes made by the developer. To keep the language server in sync, the editor must notify it of any updates. The Language Server Protocol defines synchronization messages for this purpose [4].

The Frege Language Server supports the most important ones:

1. `textDocument/DidOpen`: The client sends this notification when a developer opens a text document for the first time. This enables the server to check the opened document for errors and provide diagnostic information to the developer.
2. `textDocument/DidChange`: When developers make changes to a document, the client sends the in-memory version (not yet written to the file system) to the server. This allows the Frege Language Server to react to the most recent changes, which is especially necessary for code completion.
3. `textDocument/DidSave`: When a file is saved, the client sends a message indicating this event occurred. The client may optionally include the document

content with this message. Since the file has been saved, the server can also read it directly from disk if needed.

3.2.2 Definition

The definition request allows developers to navigate to the definition of the symbol under the cursor. When the client sends the request to the Frege Language Server, it resolves the symbol to a specific location in a file (line number and column). Developers typically initiate a definition request by using `ctrl+click` on a symbol [21].

3.2.3 Hover

A hover request asks the language server to provide additional information about the symbol under the cursor [22].

Figure 6 shows that the Frege Language Server provides both the symbol's type and its documentation when responding to hover requests:

Figure 6: The Frege Language Server provides type information and Markdown documentation on hover.

3.2.4 Diagnostics

When the client syncs a document to the server (see Section 3.2.1 *"Document synchronization"*), the server sends a `textDocument/publishDiagnostics` notification to the client [23].

The Frege Language Server retrieves diagnostics by compiling a file and extracting compiler messages from the result.

Figure 7 shows that the client then provides the compiler messages to the developer:

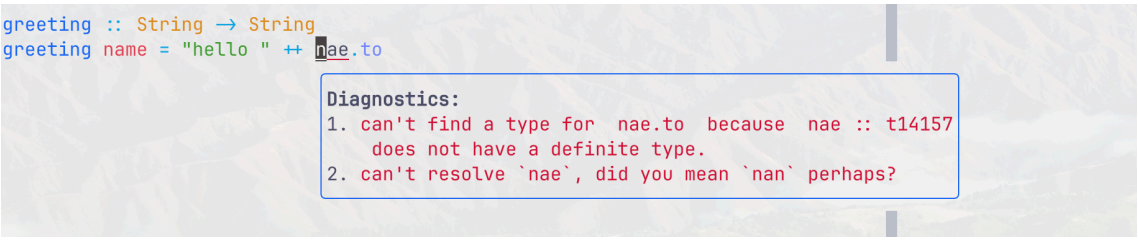


Figure 7: The Frege Language Server extracts diagnostics for a document from the compiler.

3.2.5 Completion

As the developer types, the client syncs the document to the server (via `textDocument/didChange` notification), which triggers a recompilation of the file (see Section 3.2.1 *"Document synchronization"*).

After the change event, the client sends a `textDocument/completion` request to the server containing the developer's cursor position. If the symbol at the cursor is a "`.`" (dot), the Frege Language Server searches for code completions.

This allows the server to provide auto-completion information for imported modules:

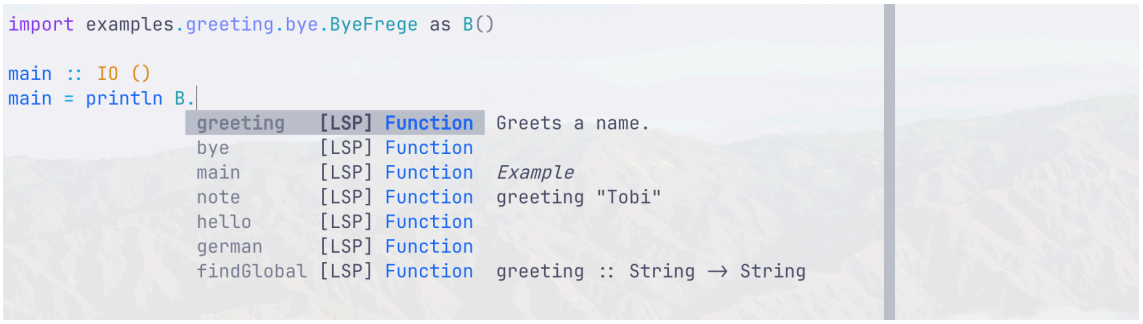


Figure 8: The Frege Language Server provides code completion for the members of the module imported as `B`.

3.2.5.1 The power of the dot

Code completion after a dot is particularly valuable in Frege, as the language features a distinctive dot syntax for both members of data declarations and methods of type classes [24].

Listing 3 shows a function accessing fields of a record using the dot-notation:

```
1 data X = X { memberOfX :: Y }
2
3 data Y = Y { memberOfY :: String }
4
5 upperCase :: X -> String
6 upperCase x = x.memberOfX.memberOfY.toUpperCase
```

Listing 3: Accessing record fields with the `.` syntax

Listing 4 demonstrates how Frege extends this capability further, enabling developers to update record properties through the dot notation as well:

```
1
2 data Y = Y { memberOfY :: String }
3
4 changeY :: Y -> Y
5 changeY y = y.{memberOfY = "new value"}
```

Listing 4: Changing the value of a record property

Figure 9 shows how the Frege Language Server leverages this feature to suggest record-based code completions via dot notation, significantly enhancing its usability:

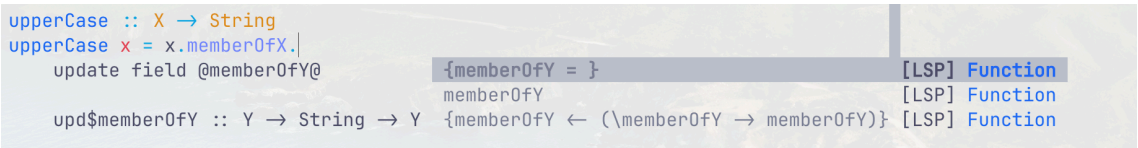


Figure 9: The Frege Language Server auto completes record fields.

3.3 Current Limitations

This section outlines the current limitations of the Frege Language Server's implemented features.

3.3.1 Document synchronization

Document synchronization supports the essential features needed to keep the server current after each text addition or deletion.

However, the Frege Language Server requires the client to send the entire document after every change. The Language Server Protocol also supports "incremental sync," which sends the complete text document only when initially opened, then transmits just the changes thereafter [25].

Additionally, the server lacks support for renaming text documents, leading to inconsistent state.

3.3.2 Definition

The definition feature has two cases where developers may not receive the expected response:

1. Requesting definition on an import: The Frege Language Server currently does not handle this case. The cursor remains in place.
2. Requesting definition for a library member: The Frege Language Server returns the position of the first import statement. For `Prelude` members, the server provides no position.

3.3.3 Diagnostics

The Frege Language Server forwards all compiler messages to the client. However, since the server disables Java compilation of generated Java code for performance reasons, errors in native definitions remain undetected in the client interface.

Listing 5 shows a native definition for `Char.toString`. This definition is incorrect because `toString` must return type `String`:

```
1 module examples.Example where
2
3 pure native toString "Char.toString" :: Char -> Bool
4
5 main :: IO ()
6 main = println $ toString 'a'
```

Listing 5: A wrong definition of the `toString` method

As Figure 10 shows, the Frege Language Server reports no errors, so the editor displays no diagnostic messages:

Figure 10: The Frege Language Server does not report incorrectly defined native declarations

As Listing 6 shows, compiling to Java bytecode produces an error that identifies the incorrect native definition:

```
./frege/main/examples/Example.java:79: error: cannot find symbol
                      return  Prelude.<Boolean>println(PreludeText.IShow_Bool.it,
Char.toString('a'));
                                                                              ^
  symbol:   variable Char
  location: class Example
1 error
E ./frege/src/main/frege/examples/Example.fr:6: java
    compiler errors are most likely caused by erroneous native
    definitions
examples.Example: build failed because of compilation errors.
Build failed.
```

Listing 6: The Frege compiler states native definition errors

3.3.3.1 Mitigation

We could mitigate this issue by compiling to Java bytecode when saving. This approach maintains good performance during document changes while catching all errors at save time.

3.3.4 Completion

While dot completion works well for records, there is room for improvement:

1. Type class methods do not yet support dot completion
2. Dot completion requires the receiver to be a single symbol. Expressions like `"hello world".` do not produce completions yet.

Additionally, the compiler cannot provide necessary type information in certain situations.

Listing 7 demonstrates Frege's "underscore-dot notation" [26], a syntactic sugar for accessing a function's single argument. Line 3 shows the `_.` notation, while line 4 shows its desugared form:

```
main :: IO ()
main = do
  let strings = map (_.toString) [1,2,3]
  let strings = map (\el -> el.toString) [1,2,3]
  ...
```

Listing 7: Changing the value of a record property

The compiler cannot determine the underscore's (_) type in isolation, preventing us from accessing its available properties.

Similarly, the compiler cannot resolve types within `let`-expressions. Listing 8 illustrates a case where dot completion fails because the symbol originates from a `let`-expression:

```
1 letDefintions :: String -> IO ()
2 letDefintions str = do
3   let uppercased = str.toUpperCase
4
5   println $ show uppercased.
```

Listing 8: Let-expressions lack type information

The same limitation applies to `where` declarations without type annotations.

4 Frege Language Server implementation

This chapter describes the architecture and code structure of the Frege Language Server, drawing on relevant parts of the arc42 template [27].

4.1 Context scope

4.1.1 Technical context

Figure 11 shows the context of the Frege Language Server. It communicates using the Language Server Protocol (see Chapter 2 *"The Language Server Protocol"*) with the client. This documentation mainly focuses on the server itself (gray box). As usual in the Language Server Protocol all participants run on the same computer and therefore have access to the same file system.

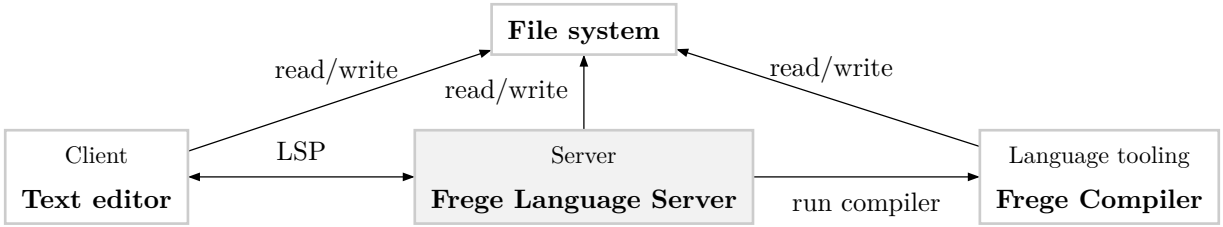


Figure 11: The context of the Frege Language Server

Table 1 describes the participating systems briefly:

System	Description
Text editor	Serves as interface to the developer and owns the server process.
File system	Contains the source code and the compiled artifacts
Frege Language Server	Bridges between client and compiler
Frege Compiler	Compiles the source code and provides code insights to the server.

4.2 Building block view

Figure 12 shows the central components of the Frege Language Server and their interconnections. The following sections further break down each component.

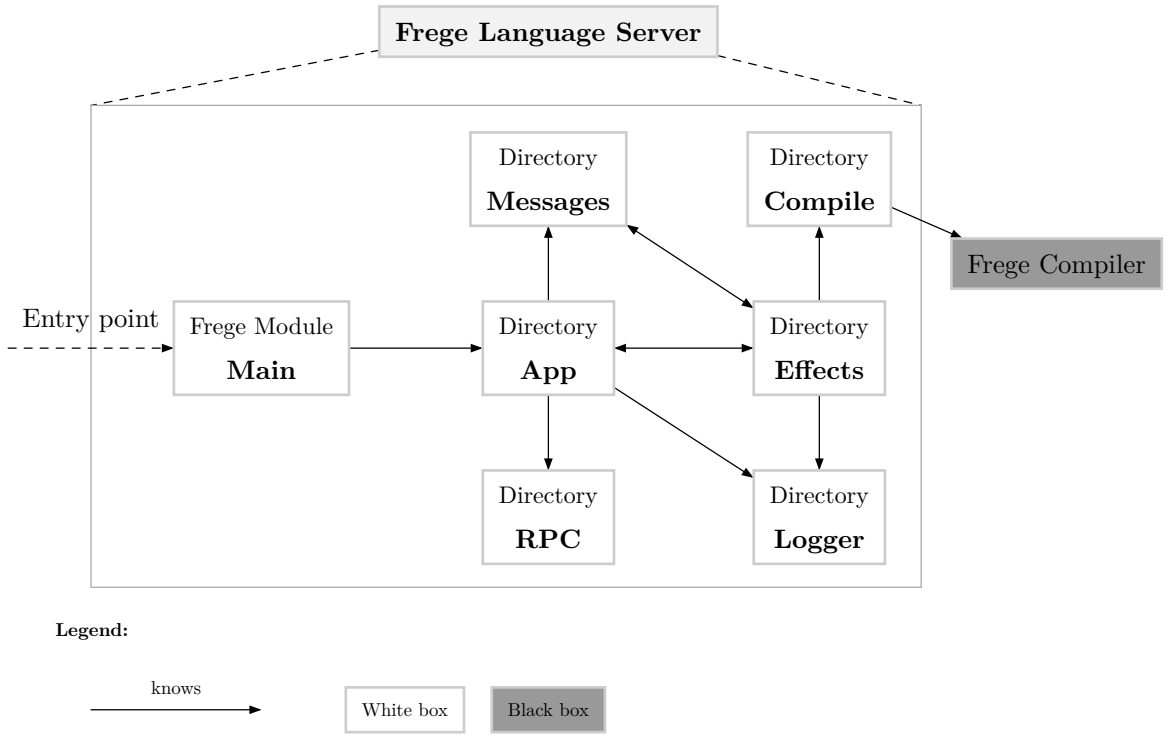


Figure 12: Frege Language Server module overview

4.2.1 Black box Frege compiler

The Frege compiler itself is written in Frege. We have easy access to it, since it is shipped with the Frege standard library. However, we treat it as a black box since we don't want to touch the compiler itself. But we provided some helpers which contain excerpts of the Frege compiler (see Section 4.2.6 *"White box* `Compile`*"*).

The Frege Compiler reads source files from the file system and accepts configuration through compile options [28].

When compiling a file, the Frege compiler returns a data structure of type `Global` containing all discovered information: compiler messages, symbol positions, imported modules, and more. The Frege Language Server relies heavily on this data structure to provide information to clients

4.2.2 White box `Main`

The Main module serves as the main entry point of the system it starts the program located in the `App` directory.

4.2.3 White box `App`

Figure 13 shows the module overview of the `App` directory:

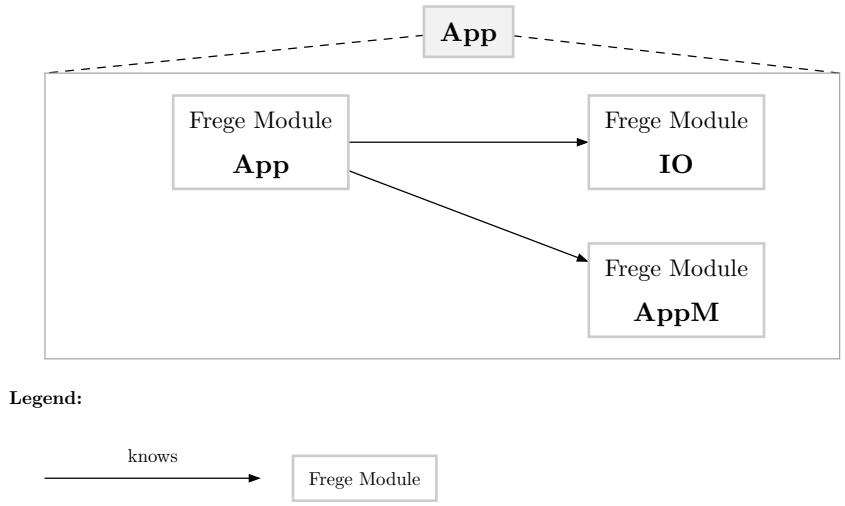


Figure 13: White box `App` structure

Table 2 describes the modules of the `App` component:

Module	Description
`App.fr`	Contains the main loop Processes LSP messages
`IO.fr`	Read client messages from stdin Write response messages to stdout File system access
`AppM.fr`	Contains the transformer stack and the app state

4.2.4 White box `RPC`

RPC decodes JSON-RPC messages (see Section 2.2.1.1 *"Message Structure"*) into Frege data structures and encodes data structures as JSON-RPC messages, serving as a bridge between client and server.

4.2.5 White box `Messages`

Figure 14 shows the structure of the white box `Messages`:

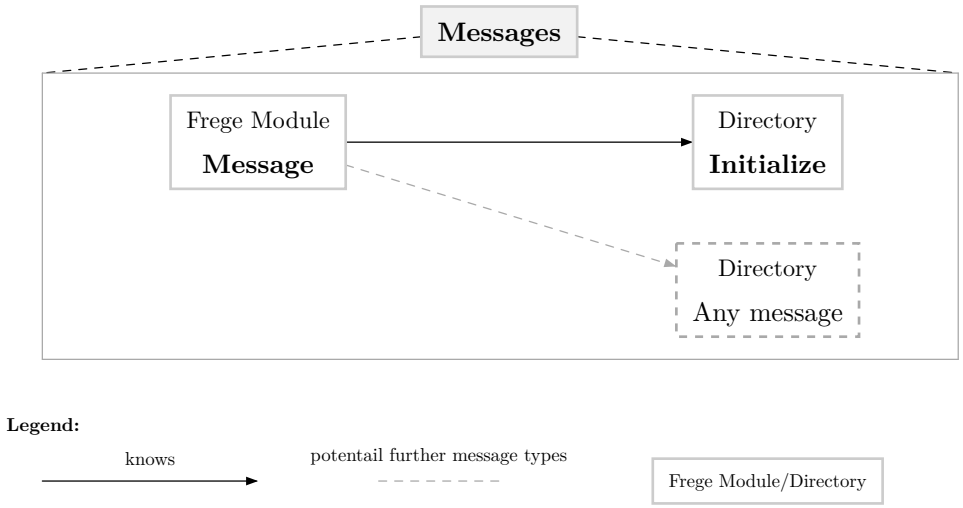


Figure 14: White box `Messages` structure

Every supported LSP message [13] has a specific directory in `messages`. The `Message` module distributes received messages to their corresponding handlers in the submodules.

Since the Frege Language Server has a dedicated `Messages` component, supporting a new LSP message only requires modifying this component.

4.2.6 White box `Compile`

Figure 14 shows the structure of the white box `Compile`:

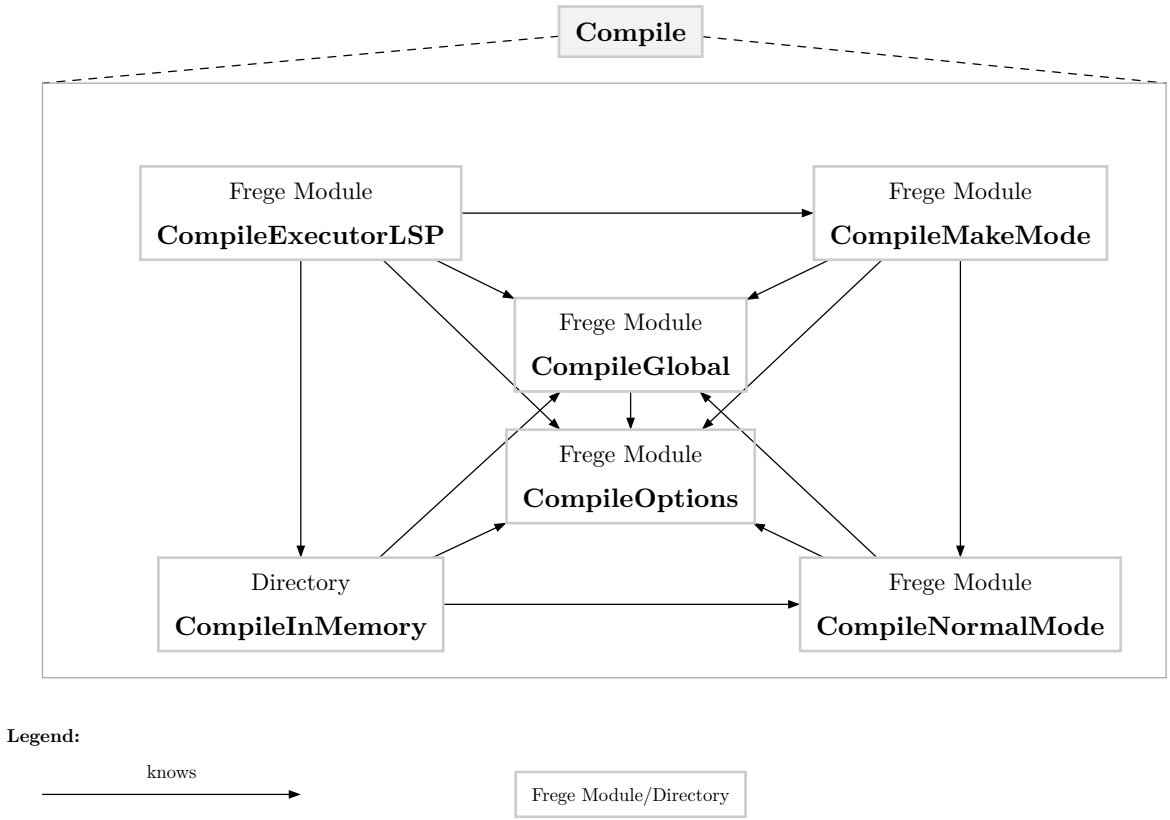


Figure 15: White box `Compile` structure

Table 3 briefly describes each inhabitant of the `Messages` component:

Module/Directory	Description
`CompileExecutorLSP`	Bridges the server to the compiler.
`CompileMakeMode`	Modified excerpt of the Frege compiler which is more graceful to source code errors and logs less.
`CompileInMemory`	Allows to compile an in memory representation of a file.
`CompileNormalMode`	Modified excerpt of the Frege compiler, defines the compiler passes.
`CompileGlobal`	Creates a `CompileGlobal` which provides access to the Java class loader and serves as an input for the compiler when compiling.
`CompileOptions`	Defines the options which the Frege compiler uses to compile the code.

We reused most of the code in this directory from the previous Frege Language Server [29].

The only new addition is the `CompileInMemory` component. The Frege Compiler typically reads files from the file system during compilation. However, change events have no file system representation. We therefore adapted the compiler to support in-memory compilation by simply bypassing file system access.

4.2.7 White box `Effects`

As Section 4.4.3 *"Handling effects"* describes, we implement side effects through type classes. Figure 16 shows each effect and their interactions:

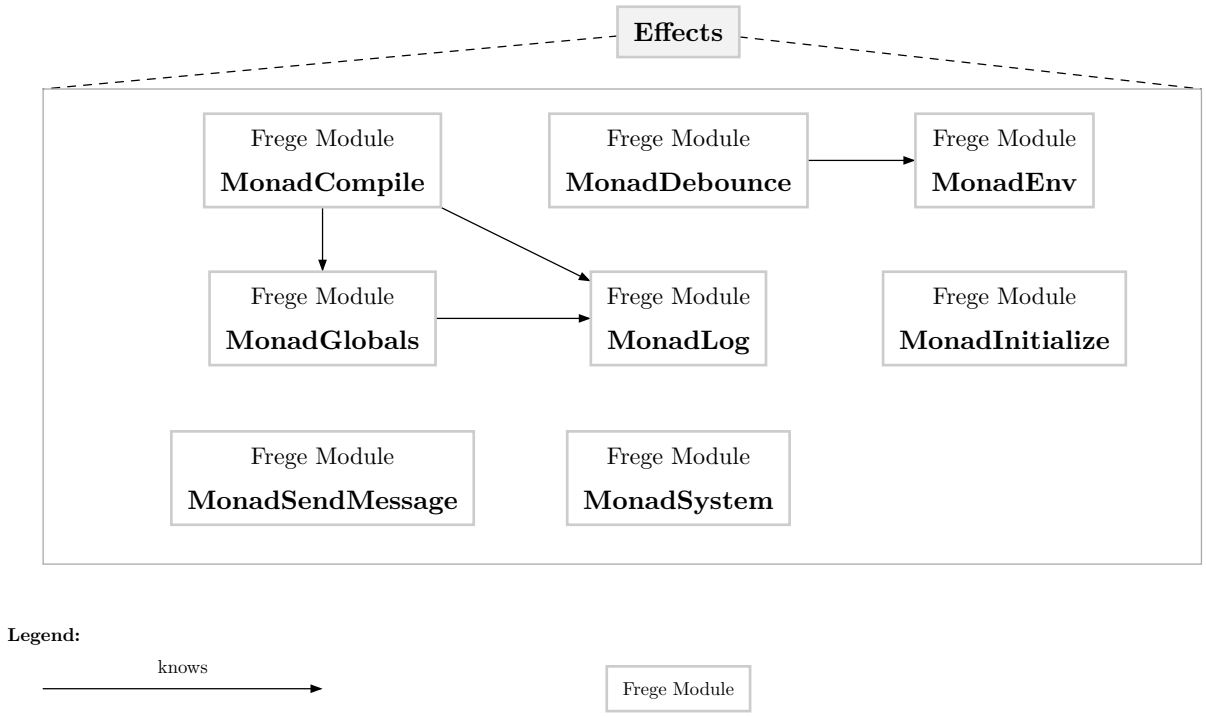


Figure 16: White box `Effects` structure

Each module contains the definition and implementation of the corresponding type class.

The Frege Language Server decomposes effects based on the following criteria:

1. Usage: Functionality used together belongs in the same effect.

2. Expressiveness: Small effects clearly indicate what a function does. For example, while Globals are part of the Environment, accessing them requires a separate effect.

See Section 4.4.3 *"Handling effects"* to learn more about the effects themself.

4.2.8 White box `Logger`

The white box `Logger` contains a single module that logs messages with timestamps to a specified log file.

4.3 Runtime view

This section outlines the most complex run time scenarios. The diagrams usually cover the happy path. Each scenario outlines the interaction within the Frege Language Server boundary. If not stated differently, actors in the sequence diagrams in this section correspond to a Frege module.

4.3.1 Processing a client request

Figure 17 shows at the example of the initialize request [30], how the Frege Language Server handles client requests:

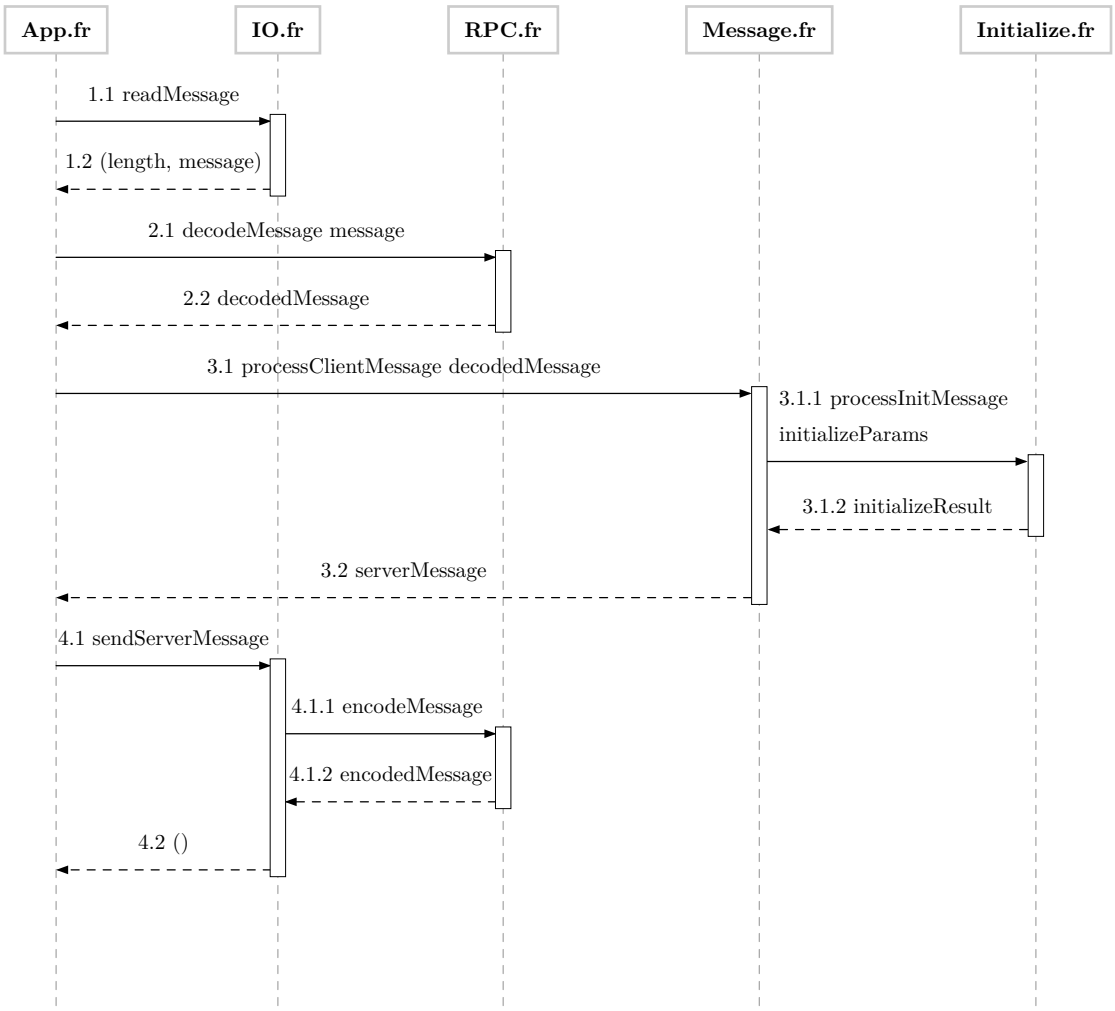


Figure 17: Processing a message

1. 1. The app calls `IO.readMessage` and waits for the next client message
   2. When a new message enters, `IO` responds with the message and its length
2. 1. The app tries to decode the message using `RPC.decodeMessage`
   2. The RPC returns a decoded message. Now, the server knows that it received a structural valid message from the client
3. 1. The app forwards the decoded message to `Message`
      1. `Message` matches on the message type and calls the actual message handler, in this case it is `Initialize`
      2. `Initialize` handles the message and maps it into an `InitializeResult`

2. `Message` turns the `InitializeResult` into a `ServerMessage` and returns it back to `App`.

4. 1. The app sends the message to `IO` to return it back to the client
      1. `IO` sends the message to `RPC` to encode it
      2. `RPC` returns the encoded message
   2. `IO` sends the message to the client and returns back to the app

4.3.2 Processing a client notification

A client notification has no response (see Section 2.2.2 *"Message types"*). Figure 17 describes how the Frege Language Server handles requests. It handles notifications in the same manner, but in step 3.2 the `Message` module returns `EmptyResponse`. In this case the app does send any response back to the client.

4.3.3 Compiling a file

The ability to compile is abstracted in the `MonadCompile` type class (see Section 4.4.3 *"Handling effects"*).

Figure 18 shows how the Frege Language Server compiles a file.
Note: actors annotated with `Instance` are implemented instances of the specified type class for the `AppM` transformer stack (see Section 4.4.2 *"AppM computation"*).

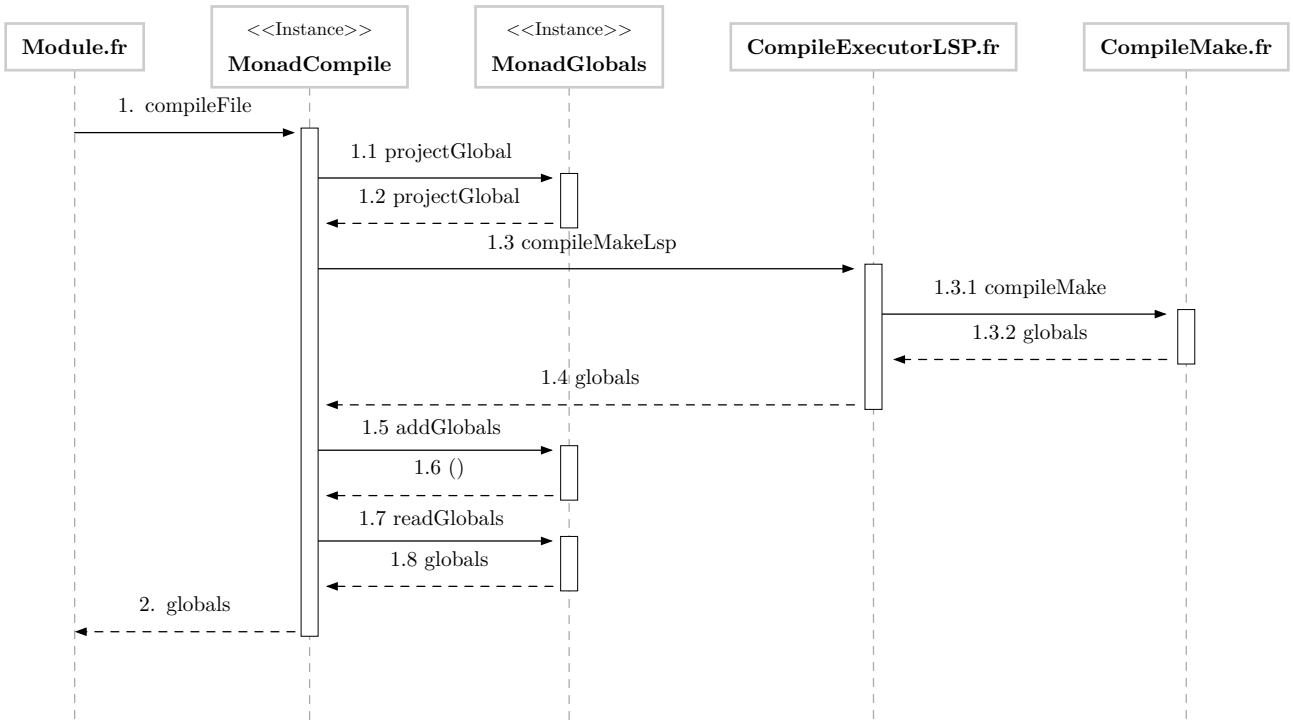


Figure 18: Compiling a file

1. Any Frege module requests compiling a given file from the `MonadCompile` instance.
   1. The `MonadCompile` instance requests a `CompileGlobal` from the `MonadGlobal` instance to gain access to the class loader.
   3. The `MonadCompile` instance calls `compileMakeLsp` which bridges to the compiler.
      1. `CompileExecutorLSP` calls the actual compiler.
      2. Each compiled file results in a global.
   5. The `MonadCompile` instance adds the newly retrieved globals to the already existing ones, using the `MonadGlobals` instance.
   7. The `MonadCompile` instance reads all globals from the `MonadGlobals` instance.
2. The `MonadCompile` instance returns all globals back to the caller.

4.3.4 Debounce Compiling

To provide completions and diagnostics after changes (see Section 3.2.1 *"Document synchronization"*), the Frege Language Server invokes the Frege compiler when receiving file changes.

Since clients send many events rapidly while developers type, the server must defer compilation until change notifications stop. We accomplish this using a debouncer that runs actions only after a specified delay. If the debouncer receives a new action before the previous one starts, it cancels the pending action and schedules the new one.

Figure 19 shows how the `AppM` instance of `MonadDebounce` (see Section 4.4.3 *"Handling effects"*) triggers actions on the Java object `CompileDebouncer`. The debouncer maintains a synchronized reference to a `ScheduledFuture` and schedules the given action in step 1.1.1. With 2.1, a new action arrives, causing `CompileDebouncer` to cancel the pending action (2.1.1) and schedule the new one. When no further messages arrive, the `ScheduledFuture` finally executes the action (2.1.3).

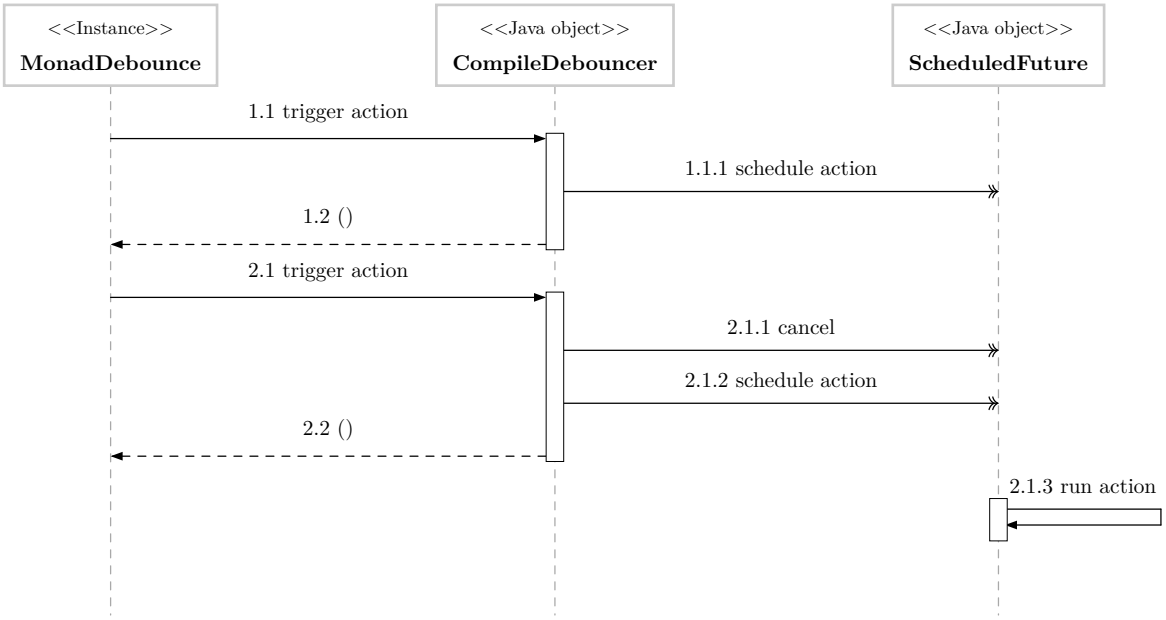


Figure 19: Using the compile debouncer

4.3.5 Accessing globals

Since compilation is deferred for change events (see Section 4.3.4 *"Debounce Compiling"*), access to compile results must be synchronized. The Frege Language Server uses Frege's STM [31] to synchronize access to globals.

4.3.5.1 Waiting for compilation

To provide meaningful completion results, the server must work with the compilation from the newest file changes. Therefore, the type class `MonadGlobals` (see Section 4.4.3 *"Handling effects"*) provides `waitForCompilation`. This function blocks for a given maximum time period when the server currently runs a compilation or returns immediately when no compilation is running, ensuring the server always uses the latest file changes.

This requires an additional synchronized variable to track compilation status. `MonadEnv` provides `setWaitForCompilationRef` for this purpose.

4.4 Crosscutting concepts

4.4.1 The App state

The Frege Language Server uses a state transformer [32, p. 338] to manage global app state and provide `IO` access. Listing 9 shows this transformer, called `AppStateM`:

```
1 type AppStateM  = StateT ServerState IO
```

Listing 9: The definition of the `AppStateM` state transformer

Listing 10 defines the underlying state, while Table 4 explains each component.

```
1 data ServerState = ServerState {
2   rootPath           :: Maybe URI
3 , globals            :: TVar (HashMap URI Global)
4 , waitForCompilation :: TVar Bool
5 , projectGlobal      :: Maybe Global
6 , logFilePath        :: String
7 }
```

Listing 10: The definition of the Frege Language Server state

Component	Description
`rootPath`	The source root, passed by the client's initialize request [18]
`globals`	A dedicated global for each source file
`waitForCompilation`	`True` if the server is currently compiling a file asynchronously
`projectGlobal`	The project global
`logFilePath`	A path to a log file to collect server logs

The Frege Language Server accesses `globals` and `waitForCompilation` from different threads. For synchronization, it uses Frege STM's `TVars` [31]. (See Section 4.3.5 *"Accessing globals"*)

4.4.2 AppM computation

To handle failing computations, the Frege Language Server uses the `EitherT` transformer [32, p. 338]. Combined with `AppStateM`, this creates the following transformer stack:

```
1 data ServerError = Error String
2 type AppStateM   = StateT ServerState IO
3 type AppM        = EitherT ServerError AppStateM
```

Listing 11: The Frege Language Server transformer stack

The server currently represents errors as a simple data type containing only a `String`. We can easily extend this for more advanced error handling.

4.4.2.1 Each message handler is an `AppM` computation

When the server receives a new client message, it runs an `AppM` computation using `runAppM` defined in Listing 12. The server passes either the initial state or the state from the previous message, enabling easy error handling and state persistence across computations.

```
1 runAppM :: ServerState -> AppM a -> IO (ServerResult a, ServerState)
2 runAppM st env = runStateT (EitherT.run env) st
```

Listing 12: `runAppM` runs an `AppM` computation

4.4.3 Handling effects

Section 4.4.2 *"AppM computation"* defines a monad transformer stack over `IO`. This allows us to annotate the hover request [22] handler's function signature as follows:

```
1 processHoverRequest :: HoverRequestParams -> AppM HoverResult
```

Listing 13: Hover request handler as `AppM` computation

This states clearly, that `processHoverRequest` is an `AppM` computation from `HoverRequestParams` to `HoverResult`. While this provides convenient access to `IO` and internal state in every computation, it also means that *any* computation could do *anything*. This leads to impure functions with poor testability and meaningless signatures.

4.4.3.1 Making effects explicit

The Frege Language Server therefore uses type classes to make side effects explicit. This approach enables providing mock instances when testing functions that use these effects.

Table 5 lists the effects used in the Frege Language Server:

Type class	Description
`MonadCompile`	Allows to compile code
`MonadDebounce`	Allows to debounce an action
`MonadEnv`	Allows access to the environment
`MonadGlobals`	Allows accessing compile results
`MonadLog`	Allows logging
`MonadInitialize`	Allows initializing the language server
`MonadSendMessage`	Allows sending messages to the client
`MonadSystem`	Allows executing system commands (e.g. shutdown)

Table 5: The Frege Language Server's effects

In addition, the Frege Language Server makes use of `MonadFail` and `MonadIO` from Frege's standard library [33], [34].

Given instances of these effects for `AppM`, we can transform the function signature from Listing 13 to:

```
processHoverRequest ::
  (
    MonadFail m
  , MonadGlobals m
  , MonadLog m
  ) => HoverRequestParams -> m HoverResult
```

Listing 14: Hover request handler with explicit effects

This clearly states that the monad `m` must be able to fail, access globals, and log. However, it also shows that this function cannot send notifications to the client or compile code. With `IO` completely absent from the signature, this function is now pure.

Mock instances for the effects not only enable testing and asserting side effects, but also simplify providing proper test data.

4.4.4 Testing

For testing the Frege Language Server uses Frege's QuickCheck [35]. Where possible and meaningful, we use properties to test the functions. QuickCheck also provides `once` to run a test one single time. We make extensive use of it as very often we want to test how a message handler behaves in a certain state when a specific message appears.

4.4.4.1 Asserting side effects

To assert side effects, we created an alternative `AppM` implementation that uses mock state with dedicated dummy instances for each effect.

Listing 15 defines this alternative implementation of `AppM`. It uses a transformer stack with simplified error handling and a state specific for testing:

```
newtype TestAppM st a = TestAppM (EitherT String (TestState st) a) where
  run = ...
```

Listing 15: The definition of `TestAppM`

Listing 16 shows a property that asserts the language server sends diagnostics when a user opens a file:

```
-- set up the state for this test properly
initialEnvirionment = ...
-- mock client notification parameters
didOpenNotificationParams = ...

p_ShouldSendDiagnosticsAfterOpening :: Property
p_ShouldSendDiagnosticsAfterOpening =
  let
    -- the resulting environment should contain the sent message after the test
    (_, env) = TestAppM.run initialEnvironment $
      processDidOpenNotification didOpenNotificationParams
  in
    once $
      env.sentMessages.length == 1
```

Listing 16: Asserting that diagnostics are sent to the client after opening a file

Using this pattern we can easily test highly complex functions in isolation.

5 Discussion

This chapter summarizes the project's main insights. It first analyzes the achieved results, then explores possible improvements, and finally evaluates whether we met our initial goals.

5.1 Results

This report demonstrates that Frege greatly benefits from a new language server. The new features significantly enhance the developer experience. In contrast to the already existing language server, we provide IDE support on document changes (not only on document saves), enabling much faster development. Additionally, jump-to-definition functionality allows developers to navigate code rapidly.

We also successfully leverage Frege's dot syntax to provide context-aware suggestions based on cursor position. However, type inference limitations sometimes cause frustration, as Section 3.3.4 *"Completion"* describes. These limitations affect other features as well: insufficient type information from the compiler causes hover requests and diagnostic messages to fail in certain cases. Since it is predictable when this is the case, developers can work around them.

As Section 4.4.3 *"Handling effects"* explains, explicit effect annotations in function signatures showcase the strengths of Frege's type system. Expressive function types improve code readability and comprehension while greatly enhancing testability. The Frege Language Server also serves as a valuable reference implementation of a production-ready Frege project.

5.2 Improving the Frege Language Server

There are many ways to improve the Frege Language Server. The Language Server Protocol offers numerous features that the Frege implementation does not yet support. One major pain point is that developers must manually manage module imports. Auto-import functionality for standard library and project modules would provide significant value and should be prioritized. Frege's underscore-dot

notation would also benefit from auto completion, making this another important area to explore.

Frege's unique selling point is the seamless integration with the Java ecosystem. The language would greatly benefit from better IDE support for this interaction. Definition jumps from Frege to Java code or vice-versa, for example, would overcome cumbersome file navigation when combining the two technologies.

Also code actions [36] are low-hanging fruits with high impact. Since the compiler already provides type information and suggestions, implementing simple actions would require minimal effort.

As Section 3.3.1 *"Document synchronization"* states, document synchronization also needs improvement.

5.3 Did we achieve the goals?

We focused on Frege's IDE integration during this project, excluding other aspects of developer experience. This focus produced a strong language server that significantly improves the Frege development workflow. Dividing attention across multiple areas would have resulted in a weaker language server and less satisfying outcome.

This project successfully delivers a modern development experience that helps Frege realize its potential as a practical functional language for the JVM.

List of Figures

List of Listings

Bibliography

[1] Microsoft, "Official page for language server protocol." Accessed: Jul. 18, 2025. [Online]. Available: https://microsoft.github.io/language-server-protocol/

[2] Frege, "GitHub - Frege/frege: Frege is a Haskell for the JVM. It brings purely functional programing to the Java platform.." Accessed: Jul. 18, 2025. [Online]. Available: https://github.com/Frege/frege/

[3] T. Gagnaux, "Frege IDE: Supported language features." Accessed: Jul. 18, 2025. [Online]. Available: https://tricktron.github.io/frege-lsp-server/frege-ide/1.0/functional-overview.html#_supported_language_features

[4] Microsoft, "LSP - Text document synchronization." Accessed: Jul. 18, 2025. [Online]. Available: https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#textDocument_synchronization

[5] S. Marlow, "Haskell 2010 Language Report." Accessed: Jul. 18, 2025. [Online]. Available: https://www.haskell.org/onlinereport/haskell2010/

[6] I. Wechsung, *The Frege Programming Language (Draft)*. Accessed: Jul. 18, 2025. [Online]. Available: http://www.frege-lang.org/doc/Language.pdf

[7] M. Madasamy, "Frege: Hello Java · mmhelloworld." Accessed: Jul. 18, 2025. [Online]. Available: https://mmhelloworld.github.io/blog/2013/07/10/frege-hello-java/

[8] "Frege Github: What's in it for me?." Accessed: Jul. 18, 2025. [Online]. Available: https://github.com/Frege/frege?tab=readme-ov-file#whats-in-it-for-me

[9] "Frege: Github: Frege project state." Accessed: Jul. 18, 2025. [Online]. Available: https://github.com/Frege/frege?tab=readme-ov-file#project-state

[10] Microsoft, “LSP - Official page for Language Server Protocol.” Accessed: Jul. 18, 2025. [Online]. Available: https://microsoft.github.io/language-server-protocol/

[11] Microsoft, “LSP - Tools supporting the LSP.” Accessed: Jul. 18, 2025. [Online]. Available: https://microsoft.github.io/language-server-protocol/implementors/tools/

[12] Microsoft, “LSP - Language Servers implementations.” Accessed: Jul. 18, 2025. [Online]. Available: https://microsoft.github.io/language-server-protocol/implementors/servers/

[13] Microsoft, “LSP - Overview Language Server Protocol.” Accessed: Jul. 21, 2025. [Online]. Available: https://microsoft.github.io/language-server-protocol/overviews/lsp/overview/

[14] Vonheimken, “LSP setup | Learn nvim.” Accessed: Jul. 18, 2025. [Online]. Available: https://vonheikemen.github.io/learn-nvim/feature/lsp-setup.html

[15] Microsoft, “LSP - Base protocol.” Accessed: Jul. 21, 2025. [Online]. Available: https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#baseProtocol

[16] Microsoft, “LSP - Specification Header.” Accessed: Jul. 21, 2025. [Online]. Available: https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#headerPart

[17] Microsoft, “LSP - Protocol.” Accessed: Jul. 21, 2025. [Online]. Available: https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#languageServerProtocol

[18] Microsoft, “LSP - The server life cycle.” Accessed: Jul. 18, 2025. [Online]. Available: https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#lifeCycleMessages

[19] T. Gagnaux, “Frege IDE: Software Architecture.” Accessed: Jul. 18, 2025. [Online]. Available: https://tricktron.github.io/frege-lsp-server/frege-ide/1.0/software-architecture.html#_frege_language_server

[20] E. Foundation, “eclipse-lsp4j/lsp4j: A Java implementation of the language server protocol intended to be consumed by tools and language servers implemented in Java..” Accessed: Jul. 21, 2025. [Online]. Available: https://github.com/eclipse-lsp4j/lsp4j

[21] Microsoft, “LSP - Text document defintion.” Accessed: Jul. 21, 2025. [Online]. Available: https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#textDocument_definition

[22] Microsoft, “LSP - Hover.” Accessed: Jul. 21, 2025. [Online]. Available: https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#textDocument_hover

[23] Microsoft, “LSP - Publish diagnostics.” Accessed: Jul. 21, 2025. [Online]. Available: https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#textDocument_publishDiagnostics

[24] D. König, “Frege Goodness - Power of the dot.” Accessed: Jul. 21, 2025. [Online]. Available: https://dierk.gitbooks.io/fregegoodness/content/src/docs/asciidoc/dot_notation.html

[25] Microsoft, “LSP - Incremental Sync.” Accessed: Jul. 23, 2025. [Online]. Available: https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#textDocumentSyncKind

[26] D. König, “Frege Goodness - Underscore-dot notation.” Accessed: Jul. 21, 2025. [Online]. Available: https://dierk.gitbooks.io/fregegoodness/content/src/docs/asciidoc/underscore_dot_notation.html

[27] D. G. Starke, “arc42 Template Overview.” Accessed: Jul. 25, 2025. [Online]. Available: https://arc42.org/overview

[28] Frege, “frege/frege/compiler/common/CompilerOptions.fr at 3.25 · Frege/frege.” Accessed: Jul. 25, 2025. [Online]. Available: https://github.com/Frege/frege/blob/3.25/frege/compiler/common/CompilerOptions.fr

[29] tricktron, “Github compile at 6038d56c64f11a02c67e4b64ae9f749eed68e18c · tricktron/frege-lsp-server.” Accessed: Jul. 23, 2025. [Online]. Available: https://github.com/tricktron/frege-lsp-server/tree/6038d56c64f11a02c67e4b64ae9f749eed68e18c/src/main/frege/ch/fhnw/thga/fregelanguageserver/compile

[30] Microsoft, “LSP - Protocol.” Accessed: Jul. 23, 2025. [Online]. Available: https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#initialize

[31] Frege, “Control.concurrent.STM - frege documentation.” Accessed: Jul. 25, 2025. [Online]. Available: http://www.frege-lang.org/doc/frege/control/concurrent/STM.html

[32] S. Liang, P. F. Hudak, and M. Jones, “Monad transformers and modular interpreters,” Jan. 1995, doi: https://doi.org/10.1145/199448.199528.

[33] frege, “Prelude.PreludeMonad - frege documentation.” Accessed: Jul. 25, 2025. [Online]. Available: http://www.frege-lang.org/doc/frege/prelude/PreludeMonad.html#MonadFail

[34] Frege, “Control.monad.trans.MonadIO - frege documentation.” Accessed: Jul. 25, 2025. [Online]. Available: http://www.frege-lang.org/doc/frege/control/monad/trans/MonadIO.html

[35] frege, “Getting Started - QuickCheck.” Accessed: Jul. 25, 2025. [Online]. Available: https://github.com/Frege/frege/wiki/Getting-Started#quickcheck

[36] Microsoft, “LSP - Code actions.” Accessed: Jul. 23, 2025. [Online]. Available: https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#textDocument_codeAction