HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

Bachelor Thesis

Modern IDE Support for Functional Programming

June 15, 2017, Spring Term

Cyrill Schenkel

Advisor
Prof. Dr. Farhad Mehta

Co-Examinor
Dr. Simon Meier

Proofreader
Mirko Stocker

Department of Computer Science

University of Applied Science Rapperswil

Abstract

Functional programming concepts have recently received a lot of attention, as many mainstream programming languages, such as C#, C++ and Java have adopted concepts like lambda expressions and lazy streams. Despite this development, the growth of adoption of functional programming languages by industry still lags behind.

By researching literature on this topic, interviewing students and industry professionals, who work with functional programming, and analysing the state of the art of functional programming IDEs, a set of requirements could be deduced. Based on this set of requirements an architecture for functional programming IDEs is proposed. Additionally a proof of concept was implemented in Haskell to show how the results could be applied and what difficulties arise during the implementation of the proposed architecture.

During the conceptual stage of the project, multiple architectural problems were found in existing functional programming IDEs. Based on the acquired set of requirements and these deficiencies, a new micro-services based architecture is proposed along with a proof of concept implementation.

There is still a lot of work to do in this area. Specifically in the case of Haskell, where the tools are very fragmented.

Contents

Management Summary

Motivation

Purely functional programming languages have multiple advantages. First and foremost, they allow the developers to focus on the problem from a higher level of abstraction than programming languages, which are more widely used.

Despite the advantages of purely functional languages, they have never reached the popularity and distribution of their imperative counterparts. The reasons for this are widely disputed. The arguments range from technical questions to primarily social ones. This is not the place to discuss the reasons of the low adoption rates of functional programming languages.

This thesis proposes a solution for one of the problems which are sometimes mentioned, as one of many causes, effects or both, for low adoption rates of functional programming: integrated development environment (IDE) support for functional programming.

IDEs are tools which are used by some developers. IDEs aim to improve the efficiency of developers. It does this by cutting down on routine activities.

Method

This thesis is structured into three main parts. Each part represents a specific phase of the project.

1. In the first phase, the situation needed to be analyzed. This was done by studying other works on this topic, interviewing four industry professionals and three students and analyzing existing IDEs for functional programming. A list of functional and non-functional requirements was synthesized from this research.
2. In the second phase, a proposal for an architecture for a new generation of functional programming IDEs was created, by taking into consideration the previously compiled requirements. The proposed architecture is fairly abstract, since it is not limited to any specific programming language.
3. In the third phase, a proof of concept was implemented for the Haskell programming language.

Findings

In the first phase of the project, it was found that IDE support for functional programming is still lacking in many areas. This is especially true for Haskell. Multiple issues were found which are inherent in the approach that most existing Haskell IDEs take. The proposed architecture aims to solve these fundamental problems.

During the implementation of the proof of concept multiple issues were found, which make the implementation of a modern IDE for Haskell more difficult than it was anticipated. These issues were primarily related to the diverging language implementations and frequent updates to the most common compiler: The Glasgow Haskell Compiler (GHC).

Prospects

One project which is based on efforts by the Haskell community, Haskell IDE Engine (HIE), looks very promising. It does not have any fundamental flaw in its architecture. There are some unsolved problems however, which are outlined in this thesis.

Chapter 1

Introduction

With the recent adoption of concepts borrowed from functional programming by mainstream imperative languages like C++ (lambda expressions), C# (LINQ) and Java (stream API), the interest in the functional paradigm and, with it, in purely functional programming languages has grown. One might ask why the adoption of purely functional programming languages by industry and open source projects would not increase along with the interest in its concepts. While there are a multitude of reasons for this, one of them is the state of tooling support for such languages. Very high-level tooling is important for large and critical projects to succeed. While there are a lot of useful tools for commonly used functional programming languages, there is usually no simple and unified experience of using those; such as the one an integrated development environment (IDE) would provide, for example. For the purely functional programming paradigm to become widely adopted, the IDE support for purely functional programming languages needs to be improved.

In order to alleviate the lack of IDE support for purely functional programming languages, one must first gain a clear understanding of the requirements for such an IDE. This was achieved by interviewing a group of three students who recently attended a course which also provided an introduction to functional programming and another group of four professionals who work with purely functional programming languages in their jobs. In this way, the needs of both groups, the students and the professionals, could be captured. The analysis of the developers needs, together with an analysis of the current state of functional programming IDEs or similar tools, provide a solid basis for a set of requirements that all useful functional programming IDEs must fulfill.

Based on these requirements, an architecture is proposed. While being open for extension, this architecture seeks to be a basis on which all the mentioned requirements can be fulfilled. Another design influence is the premise that an IDE should be as non-disruptive to the pre-existing processes and practices of a developer as possible.

An implementation of the proposed architecture is provided for the purely functional programming language Haskell. Haskell was chosen for this because of its maturity, large community and adequate library support. The implementation is a proof of concept and not production ready.

Chapter 2

Requirements

To gain a better understanding of the requirements for IDEs for functional programming, developers of various levels of skill were consulted. Four of them work with functional programming, professionally. The other three recently attended a course which also provided an introduction to the functional paradigm and Haskell specifically.

The comparison of the results of the aforementioned interviews, with currently available IDE solutions and other tools for functional programming, allows the compilation of a set of requirements, which all IDEs for functional programming must fulfill.

The requirements are divided into two general categories: functional requirements and non-functional requirements. The former describe what functionalities shall be implemented and the latter constrain the how.

2.1 Interviews

One general observation which can be made from the interviews[1] is that all the interviewees enjoy their freedom to choose the tools they are working with. Some of them stated that they would be prepared to change to an IDE, which is more advanced, if that would significantly increase their efficiency. One of them mentioned that they do not know of any IDE feature which would let them abandon their current work environment.

Some features, are just expected to be there. This includes, for instance, *syntax highlighting* and *compiler error highlighting*, the latter being especially useful for strongly typed languages: Others are specific to functional programming. *Read-eval-print loops* (REPL) or *point-free refactoring*, for example.

The main difference between the two groups of interviewees, students and professionals, is that the latter struggle most with their tools not being able to cope with their large code bases. This can be attributed to a common design flaw in certain tools, which will be discussed further down the line.

Access to the REPL turned out to be more important for the students than the professionals. While the students test their code by evaluating expressions in the REPL, the professionals prefer to use automated testing.

[1] See Appendix A

2.2 IDE Comparison

Because functional programming has been around for a long time, since the invention of LISP in 1958, there has been a multitude of integrated development environments for functional programming. Some of those, who are deemed to have had the most impact, are discussed below.

After that the state of tooling and IDE support for Haskell is examined. This is to answer the question: *How IDE support for Haskell differs from IDE support for other functional programming languages.* Haskell is used substitutionally for functional programming languages which lack mature IDE support.

2.2.1 SLIME

Superior lisp interaction mode for Emacs (SLIME) is an Emacs extension which is used for Common Lisp programming. SLIME has a client-server architecture. On the client side there is *slime* that is written in Emacs Lisp. The server side is called *swank* and is written in Common Lisp itself. [10, 16]

This approach allowed other developers to reuse the SLIME front end an write their own swank back ends. Two notable projects, which have done this, are *swankr*[14] for the R programming language and *swank-js*[15] for JavaScript.

As for features, SLIME provides: online documentation, jump to definition, macro expansion, code evaluation, symbol and package name completion, a debugger interface, a REPL interface and an interactive object-inspector.[10]

2.2.2 Light Table

Light Table[8] came in to being as a crowdfunded project on Kickstarter and is now available under the MIT license on GitHub. The project reached its funding goal in 2012 and was open sourced in 2014.

Light Table is based on Google's Chromium browser. It can, therefore, render everything Chromium can, including 3D graphics with WebGL. Light Table focuses heavily on interactive programming and debugging. The relevance of Light Table lies in these new UI concepts.

Light Table supported Clojure, a functional programming language which is either compiled to *Java virtual*

machine (JVM) byte code or JavaScript, from the outset. Later support for JavaScript and Python was added.

2.2.3 Scala IDE for Eclipse

Scala IDE[5] is an Eclipse plugin for the multi-paradigm programming language Scala. It has a very complete feature set, including some features which are specific to Scala, for example the highlighting of implicit conversions.

Scala IDE provides two features that are especially useful for strongly typed functional programming languages, which have type inference. The first shows the inferred type of the identifier the mouse pointer hovers over. The second shows the inferred type of a selected expression.

2.2.4 Haskell IDEs and Tools

Haskell is the most commonly used purely functional programming language. Nevertheless, IDE support for Haskell is still lacking. This could also be seen in a study conducted by FPComplete in 2015[2], in which about 52% of the participants indicated that improved IDE support is important or crucial to them. About 74% stated that improved IDE support would at least be helpful. Over half of the participants would like to use their existing tool set with Haskell. The comments to this question often refer to compatibility of the respective participants editors of choice as a desirable feature of Haskell tools. Others would like more tooling for development and operations (DevOps) or integration with their current technology stack (.NET, JVM, NixOS, AWS, etc.). Some of these issues have already been addressed since. This includes the problems with the build system and the NixOS integration of such.

GHCi Forks

A common way of providing some sort of IDE support for Haskell is to fork GHCi and make some modifications to it. Two of the most successful Haskell IDEs, Intero[4] and GHCid[13], are implemented by interfacing with GHCi in one way or another.

Intero is an Emacs plugin for Haskell development, which is very similar to SLIME. It is also based on a client-server architecture. The original client is written in Emacs Lisp. The server is a GHCi fork and thus written in Haskell. Intero only works well with projects based on Stack.

The Intero plugin uses Stack to launch Intero servers for each package, which is being edited. This can be a problem for working on large projects with a lot of packages. An Intero server instance can take up several hundred megabytes of memory.

Dependencies not being automatically reloaded when they have changed, is another annoyance when working with Intero. If there are, for instance, two packages *a* and *b* where *b* depends on *a* and *a* changes, the package *a* needs to be rebuilt and the new build needs to be registered in the package database, before the Intero instance for package *b* is restarted. Only after this procedure, Intero is in the expected state.

Intero works out of the box for most small and medium sized projects. It however has the aforementioned problems, which are consequences of its architecture.

GHCid watches the file system for source code changes, reloads changed files in GHCi automatically and displays compiler warnings and errors. It is an extremely simple but popular tool.

Other Haskell IDEs

There are many Haskell IDEs which are not (yet) usable or abandoned. EclipseFP for instance is not maintained anymore.

Ghc-mod comes as an executable and a library. It aims to provide the infrastructure for building IDEs. It does this by interfacing directly with GHC and Cabal. Ghc-mod is known to break very often, because it relies on APIs which tend to change very frequently.

Haskell IDE engine (HIE) tries to integrate with existing tools such as ghc-mod and the Haskell refactorer (HaRe). The Development of HIE is in an early development stage. HIE is a very promising project, although it suffers from some of the same flaws as some of the other tools do. It actually inherits them by integrating with those tools. By leveraging the plugin-API to replace the problematic components, this can however be solved.

HIE is also based on a client-server model, as is SLIME. Its architecture allows for multiple *'haskell-ide-client'* components, for interfacing with different editors. This way the *'haskell-ide-deamon'* and its plugins can be reused across different tool sets.

Ide-backend provides a simplified interface to the GHC API. The stack-ide package provides a JSON interface to ide-backend. The project has been abandoned in favor of haskell-ide-engine.

Haskell Build Tools

Most Haskell IDEs support different build tools. Intero for instance supports Stack, whereas ghc-mod only supports cabal-install.

The Glasgow Haskell compiler (GHC) is the most commonly used Haskell implementation. GHC is sometimes used directly to build projects. For the purpose of building projects, GHC supports the *stringstyle−−stringstylemake-* option. This option can be used for building multi-module projects. All dependencies of the given root module will automatically be resolved.

GHC implements the Haskell 2010 Language Report. This report states that "[s]ome compiler implementations support compiler pragmas, which are used to give additional instructions or hints to the compiler, but which do not form part of the Haskell language proper and do not

change a program's semantics."[9] GHC provides various such extensions to the standard. A large set of LANGUAGE pragmas were added to GHC over time. They are heavily used by the Haskell developers. This leads to GHCs of different versions being incompatible with one another. This is an obstacle for implementing IDEs, because it means they cannot just be linked to GHC or they would need to be recompiled for every compiler version the developers might want to use. It also means that it is a very tedious feat to build and maintain compiler independent parsers and typecheckers, because they would have to be maintained for every compiler version one might want to use. Now the first option is clearly the less expensive.

Cabal refers to three things: (1) the Cabal package metadata format and the Cabal specification[7], (2) the reference implementation of the specification as a library and (3) *cabal-install* a command-line tool which uses the Cabal-library.[1]

The cabal-install tool is used for developing, building and installing Haskell projects. The tool supports sandboxes, which solve a problem known as cabal hell. This is a state where two packages cannot be installed simultaneously because they depend on different versions of the same package.

Cabal-install acquires and installs packages from Hackage, which is a large repository of open source software written in Haskell.

Stack is a tool which is based on Cabal the library and thus complies with the cabal specification. It aims to solve several shortcomings of cabal-install. Stack aims at making builds as reproducible as possible. This means that building a package in one environment should theoretically yield the same result, as when it is built in another environment. Note that this is not entirely possible until GHC produces deterministic output.[17] Stack however at least ensures that the auxiliary inputs of the build, specifically the compiler, the dependencies and the compiler flags are always the same for the same package metadata, *stack.yaml*-file and OS.

Stack uses Stackage as its package repository. This eliminates the need for *cabal freeze* which freezes the dependencies in a package metadata file (".cabal"-file) to their specific versions. Stackage provides what they call snapshots. A snapshot is a set of packages with certain versions which are guaranteed to build and pass all their tests. There are two types of snapshots: nightly and long term support (LTS). The former contain all the latest releases. The latter are more stable. A minor version increment from one LTS snapshot to another suggests no API changes between them. A LTS snapshot with a new major version may entail API changes, package additions and package removals.

Shake is a build system, which is written in Haskell as a replacement for make. There was an attempt to rewrite Cabal with Shake for dependency resolution. This project was superseded by Stack however. There was another interesting project which tried to speed up compile times with GHC by integrating GHC with Shake.[12] There also is a proposal to move the GHC build system from make to Shake.[18]

Other Haskell Tools

There are other tools that provide some support for Haskell development. These tools are often integrated into existing Haskell IDE solutions.

LambdaBot was developed as a chat bot for the #haskell IRC channel. It supports a variety of features, like showing the source code of a specific function or point-free refactoring of terms.

GHCi on Acid is a GHCi extension that brings the LambdaBot functionality to GHCi.

HLint provides suggestions for improving code style. HLint is especially useful for new haskell developers.

Stylish Haskell helps with automatically formatting code, so that it is easier to read and more beautiful to look at. Stylish Haskell inherits some bugs from *'haskell-src-exts'*, on which it depends.

HIndent is similar to Stylish Haskell but has a different feature set. It also sometimes breaks the code, by rearranging documentation comments too aggressively.

2.2.5 Conclusion

As can be seen, there are a lot of IDEs for purely or partially functional programming languages. The ones above are only a small collection, of course. A common theme among them is the reliance on some sort of a client-server architecture. In the cases, where the REPL is used as an IDE backend, there is no alternative. This separation allows for two distinct advantages over the model in which editors and IDEs are integrated tightly: (1) the user can use their editor of choice and (2) the IDE can be reused for other purposes.

One big obstacle for providing IDE support for Haskell is the rapid changes in GHC. Intero solves this problem by needing to be installed with stack and thus using the same compiler version as the code for which it is used.

A lot of the tools mentioned above use different parsers than GHC. This goes for Stylish Haskell, HLint and HIndent. The result is that a file may parse with GHC and HLint but not with Stylish Haskell. There are multiple bug reports related to this in the Stylish Haskell issue tracker. Therefore one requirement for any Haskell IDE must be to provide an AST and semantic model which can be used for all its features; consequently all the tools rely on the same information.

One of the problems which need to be addressed by every new Haskell IDE, is the excessive resource consumption that Intero has. For large projects it is no option to

have a GHCi instance for every package, because that implies that all the module dependencies of the files, that are currently being worked on, are constantly loaded.

2.3 Functional Requirements

The following functional requirements were derived from the interviews, the 2015 study by FPComplete[2] and the IDE comparison above.[2] The functional requirements are written in the form of use cases.

All use cases are prioritized as MUST, SHOULD or COULD requirements. MUST requirements shall be implemented first. Without these requirements being satisfied, an IDE is of no practical use. The SHOULD requirements need to be implemented so that the IDE is at least as good or better than existing IDEs for functional programming. The COULD requirements describe functionality which is non-essential but agreeable.

UC-01: Find the Origin of a Compiler Error and fix it

Priority
MUST

Primary Actor
Developer

Precondition
Compilation of the code, that is currently being worked on yields an error.

Scenario

1. The IDE shows the user where the error originates from
2. The user fixes the error
3. The IDE shows the error no more

Note
The full error message of the Haskell compiler must be shown.

UC-02: Start a new indented Line or indent an existing Line

Priority
SHOULD

Primary Actor
Developer

Scenario I

1. The user starts a new line
2. The IDE automatically moves the cursor to the proposed indentation level

Scenario II

1. The user tries to change the indentation of an existing line
2. The user uses the 'tab' key to cycle through different indentation levels

UC-03: View the Type of an Expression

Priority
SHOULD

Primary Actor
Developer

Scenario I

1. The user wants to know the type of an identifier
2. The IDE shows the type of that identifier

Scenario II

1. The user selects an expression they want to know the type of
2. The IDE shows the type of that expression (if possible)

UC-04: Debug a Project

Priority
COULD

Primary Actor
Developer

Scenario

1. The user wants to find the cause of a bug and fix it
2. The IDE lets them set breakpoints, view the bindings in the current environment and lets them step through the evaluation

UC-05: Evaluate an Expression interactively

Priority
COULD

Primary Actor
Developer

Scenario I

1. The user wants to learn how to use a new library
2. The IDE lets them execute commands in an environment in which the library is loaded

Scenario II

1. The user wants to try some generic algorithm outside of their current project
2. The IDE lets them load some code into an interactive environment

[2] See section 2.2

3. The user uses the interactive environment to try out their algorithm

Note

An interactive environment is usually provided by a REPL.

UC-06: Lookup Documentation

Priority

SHOULD

Primary Actor

Developer

Scenario

1. The user selects the identifier of which they want to look up the documentation
2. The IDE shows the documentation to the user

UC-07: Search something on Hoogle

Priority

COULD

Primary Actor

Developer

Scenario

1. The user tries to find something (function, type class, datatype, etc.)
2. The IDE lets them query Hoogle in order to find what they're looking for

UC-08: Search something on Hackage/Stackage

Priority

COULD

Primary Actor

Developer

Scenario

1. The user wants to find a package
2. The IDE lets them query Hackage or Stackage respectively

UC-09: Rename an Identifier

Priority

COULD

Primary Actor

Developer

Scenario

1. The user selects an identifier for renaming
2. The IDE lets them enter a new name
3. The IDE renames all occurrences of the identifier which are semantically equal

UC-10: Organize Imports

Priority

COULD

Primary Actor

Developer

Scenario I

1. The user tells the IDE to remove all unused imports
2. The IDE removes all unused imports

Scenario II

1. The user uses an identifier which is not defined
2. The IDE suggests adding an import for any of the modules which define that identifier
3. The user selects the module they want to use
4. The IDE adds the import for that module

Scenario III

1. The user wants to automatically remove all unused imports and order the remaining imports alphabetically
2. The IDE removes all unused imports
3. The IDE orders the remaining imports alphabetically

UC-11: Format a whole Source File

Priority

COULD

Primary Actor

Developer

Scenario

1. The user wants to format the source file they are working on
2. The IDE formats the whole source file according to a set of rules

Note

In the future the set of rules according to which the code is formatted should be customizable.

UC-12: View the Code with Syntax highlighting

Priority

MUST

Primary Actor

Developer

Scenario

1. The IDE color codes the syntax to help the user understand the code

UC-13: Get a continuous Stream of relevant Test Results

Priority
COULD

Primary Actor
Developer

Scenario

1. The user wants to always be aware if their current code passes their tests
2. The IDE continuously runs the relevant tests for the code that has been modified
3. The IDE shows the most recent test results to the user

2.4 Non-Functional Requirements

Most of the deficiencies of current IDEs belong to the category of non-functional requirements (NFR). This means that the important features are mostly present, but the quality of their implementation is lacking.

NFR-01: Scalability

The IDE MUST work well with very small and very large code bases. The size of the code base should impact the response time of the IDE features as little as possible. The resource consumption may not increase more than linearly in respect to the size of the code base.

Resource	Max. Consumption
Disk Space	2× Project Size
Memory	512MB

Table 2.1: NFR-01: Resource Consumption Constraints

The values in Table 2.1 are for medium sized projects. Memory consumption may be linearly higher for larger projects. The values are estimations of what is deemed to be an acceptable level of resource consumption.

NFR-02: Performance

The UI MUST never be blocked by any IDE feature. The response time of all IDE components should be adequate for their task. For type on hover and code completion, for instance, this means showing a response to the user in less than a second. The initial load time of a project shall not be higher than five seconds.

NFR-03: Ease of Use

The IDE MUST be easy to use. Features of the IDE have to be easily accessible. Commands should thus be callable through keyboard shortcuts. Commands that are not practicable in certain contexts shall only be available when the users focus allows it.

Starting a new project must be as easy as possible, so that beginners don't have to understand cabal nor stack before starting to program.

NFR-04: Easy Setup

The IDE MUST be easy to setup. The installation process is limited to the following steps:

1. Install stack.
2. Add stack to the PATH environment variable.
3. Install an editor.
4. Install the editor plugin.

NFR-05: No Clutter

The Developer MUST not be hindered by the IDE in his work. The developer is free to use the IDE features if and how they choose. The IDE must not distract the developer from their task.

NFR-06: Useful User Interface

All error messages and warnings, especially the ones from the compiler, MUST be displayed in full and with proper line breaks.

NFR-07: Integration with existing Tool set

The IDE SHOULD integrate with the developers' existing tool set. This doesn't mean, that an IDE must have an integration with every editor. But it means that the architecture of the IDE should allow the developer to integrate it with their tools, if they like to do so.

NFR-08: Extensibility

The IDE SHOULD be extendable by some kind of plug-in API. Projects like Emacs, Atom and VS Code have shown, that an extensible architecture vastly increases the amount of community contributions, because it lowers the barrier.

Chapter 3

Architecture

The proposed architecture is described using a top-down approach. At first an overview and rationale of all the components is given. Then each component is explained in detail.

3.1 Overview

From the requirements specified in chapter 2 and the examination of different IDEs in section 2.2 a number of technical constraints can be derived, to which the architecture of an IDE must adhere.

In order to meet NFR-07 (integration with existing tool set) the architecture must follow some kind of client-server model. This can also be seen in existing IDEs.[1] With a client-server model the editor is required to ensure that the UI does not block. This addresses the first part of NFR-02 (performance, non-blocking).

Constraint 1 *The architecture must follow a client-server model, where the client is the editor and the server is the IDE itself.*

The second part of NFR-02 (performance, response time) requires the IDE to provide intelligence to the user within a second or less. This also means that the user should not be required to save in order to see a response from the IDE. This means that the IDE must at least have a copy of the dirty document buffers[2] of the editor. In order to ensure that the states of IDE, editor and file system are never out of sync, the IDE must not be allowed to access the file system directly. This means the IDE must have a complete copy of all the files, needed by the IDE to provide its assistance, as they are perceived by the user.

Constraint 2 *The IDE must not access the file system directly, but only through communication with the editor.*

NFR-08 suggests some sort of a plug-in system, in order to make the IDE extensible. A micro-service architecture can be used in order to provide the necessary extensibility. The advantages of micro-services over other extension mechanisms, are that (1) the IDE does not need to be recompiled to extend it and (2) the extensions do not need to be compiled with the same compiler or even written in the same language.

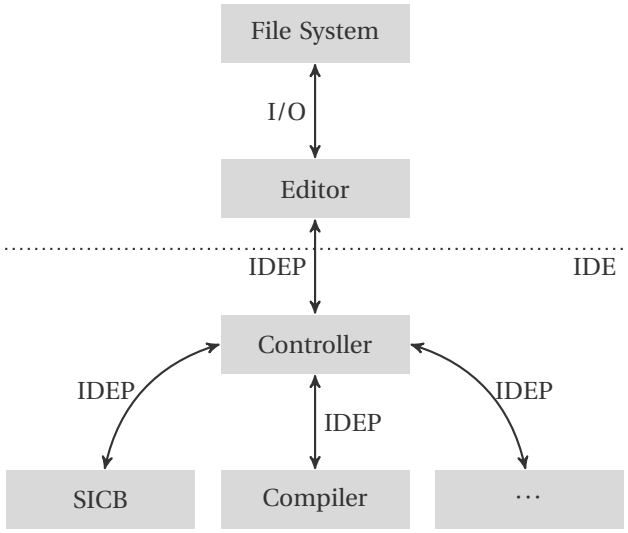


Figure 3.1: Architecture Overview

Constraint 3 *IDE features should be provided as micro-services.*

Another prerequisite for NFR-08 (extensibility) is a shared syntactic and semantic model. The syntactic and semantic model should contain all the information about the code on which the IDE is used. That is necessary for implementing the functional requirements.[3]

Constraint 4 *There must be a single source for all necessary syntactic and semantic information about the code.*

NFR-03 (ease of use), NFR-05 (no clutter) and NFR-06 (useful user interfaces) are user experience (UX) requirements and need to be met by the editor. NFR-01 (scalability) must be addressed by every particular implementation.

Figure 3.1 gives an overview of the overall structure which stems from the constraints which are described above. The IDE (everything below the dotted line in Figure 3.1) consists of a controller component and various micro-services, two of which — shared immutable code

[1] See section 2.2

[2] A *document buffer* is an in-memory byte array that represents a document. A document may be a source file or any other object which can be edited with the editor. *Dirty document buffers* are document buffers which are either not assigned to any persistent file or contain changes to a persistent file.

[3] See section 2.3

base (SICB) and compiler — are required. The makeup of these components is described in detail below. All components communicate over a protocol which, from here onwards, is called *integrated development environment protocol (IDEP)*. IDEP is really just a placeholder for a concrete protocol an IDE implementer would use — for instance an extended variant of the language server protocol (LSP).[11]

The *controller* component is necessary to satisfy constraints 3 and 1. It controls the communication among the micro-services and between the micro-services as well as the editor. The *SICB* micro-service keeps the view of the IDE on the file system in sync with the state of the editor, by relying only on the IDEP and not accessing the files of the user directly. It is required for satisfying constraint 2. The *compiler* component makes the architecture adhere to constraint 4 by providing a common representation of the syntax and semantics of the user's source code.

3.2 Controller

The controller component is the backbone of the proposed IDE architecture. It manages the communication flow among the micro-services as well as between the micro-services and the editor. Furthermore it controls the life-cycle of the services.

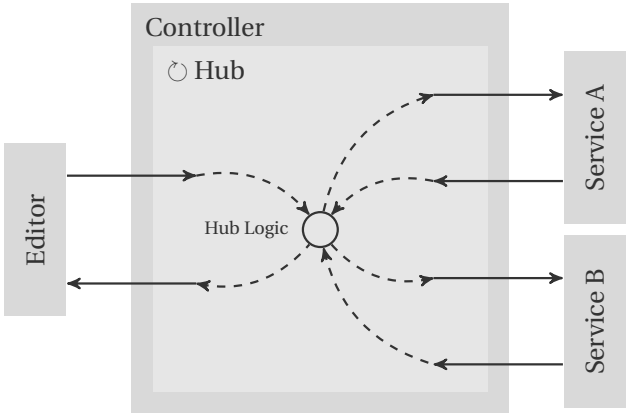


Figure 3.2: Controller Overview

Figure 3.2 shows how the IDEP messages flow through the controller. Services A and B are placeholders for arbitrary micro-services. The hub logic does the actual routing of messages.

Every service supports a certain set of requests and notifications, the difference between request and notification being, that requests need to be answered by responses or errors and notifications do not need to be answered. Notifications can be supported by multiple services.

When the hub logic receives a request, it saves its origin and redirects the request to the service, which supports that particular message, or to the editor if it supports that message. If the request is neither supported by any service nor by the editor, an error is returned to the origin of the request. If the hub receives a response, it recalls the origin of the request which triggered the response and redirects the response to that origin. If the hub does not receive a response within a certain time limit, the hub responds with an error.

The handling of notifications is much easier than that of requests. Notifications are redirected to all services, which support them, and the editor if it supports them. If notifications are not supported by any service connected to the hub or the editor, they are just dropped.

3.3 SICB Service

The shared immutable code base (SICB) service handles the server side synchronization of the code base. Figure 3.3 shows an overview of how this is accomplished.

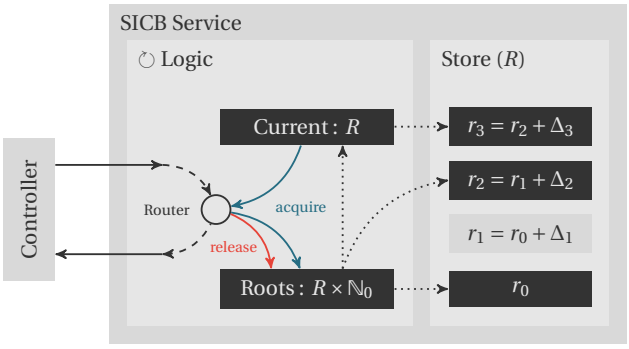


Figure 3.3: SICB Service Overview

When the SICB service is started, it initializes by requesting all necessary files from the editor. Necessary are those files which are needed to successfully build the code base. After the initialization is complete, the editor sends incremental updates to ensure that the SICB service is eventually in sync with the editor.

In order to ensure that the IDE state does not diverge from the editor's state and that the micro-services do not influence one another in unpredictable ways, the code base must be immutable. In order to combine continuous change with immutability, the concept of revisions is introduced. Revisions are immutable snapshots of the code base at a specific point in time.

The first revision (r_0) is acquired during the initialization of the SICB service, as it is described above. Subsequent revisions (r_n) are the sum of the previous revision (r_{n-1}) and the incremental update (Δ_n).

$$\begin{aligned} r_0 &= \text{initial clone of code base} \\ r_n &= r_{n-1} + \Delta_n \end{aligned} \tag{3.1}$$

To avoid space leaks, each revision has a reference counter ($c : R \times \mathbb{N}_0$) and if it reaches 0, the revision is disposed of.

$$\begin{aligned} &C : \text{set of reference counters} \\ &P(r_n) = (r_n, 0) \in C \end{aligned} \tag{3.2}$$

The top is always referenced (*Current* in Figure 3.3) The services which need to access the code base must ac-

quire a reference. They can only acquire a reference to the newest revision. If a service acquires a reference to a revision, that revision is kept alive until the service releases it. To avoid space leaks in the case of defective services, these services must be killed if they do not release their references in a certain amount of time and their open references must be released forcibly.

3.4 Compiler Service

The compiler service has two responsibilities: (1) it informs the user/editor of errors and warnings and (2) it provides a syntactic and semantic model of the code base to other services.

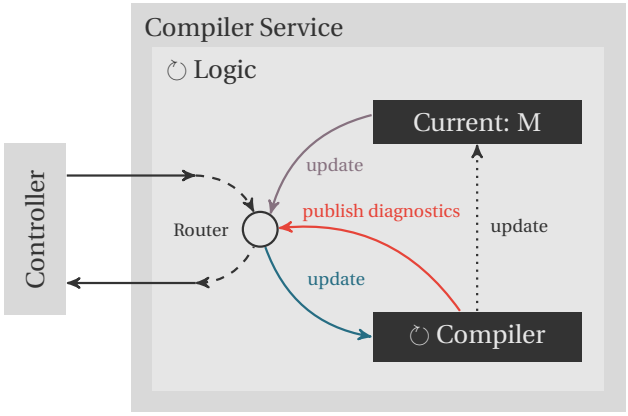


Figure 3.4: Compiler Service Overview

In order to prevent confusion of the user, no syntactic model shall be provided, if the code does not parse, and no semantic model shall be provided when the typechecker fails.

The syntactic model is an annotated syntax tree which contains all information needed to analyze but also transform said tree. This means it must contain all pre-processor statements, compiler pragmas, comments, definitions and so forth, including their exact positions. This information is needed to implement refactoring (UC-09 and UC-10), online documentation (UC-06), auto formatting (UC-02 and UC-11), etc.

The semantic model must contain all information about the types, preferably of all spans of the source code, which are valid expressions.[4] For identifier renaming (UC-09), jumping to the definition of an identifier, organizing imports (UC-10) and other functionality, the semantic model must also contain the locations of all usages of all qualified names and the scope of each qualified name.

[4] This is exactly how the *type-at* feature of GHCi is implemented.[19]

Chapter 4

Proof of Concept

Haskell IDE (HIDE) was developed as a proof of concept of the proposed architecture.[1] VS Code[2] was chosen as the editor component, because it encourages to add new programming language support by using a client server architecture. It uses the *language server protocol (LSP)*[11] for communicating with so called *language servers*.

LSP, however, lacks some of the functionality, that is needed by the proposed architecture. Specifically it does not include access to the contents of the files in the workspace, that are not opened at the moment. It also does not allow transferring binary files.

To account for this deficiency in LSP, a new component, the *LSP proxy*, is added to the proposed architecture. The LSP proxy acts as a facade for the editor. The HIDE controller, which looks exactly like it is specified in chapter 3, communicates with the LSP proxy as if it was the editor itself. The IDE protocol (IDEP) is an extended variant of LSP, which, to avoid confusions, is called E-LSP from here onwards.

E-LSP also includes all the request, notification, response and error types, including their parameter and result objects, which are yielded or consumed by any micro-service.[3] The set of valid E-LSP messages is therefore congruent with the set of valid JSON-RPC 2.0 messages and consequentially infinite.

Figure 4.1 shows the extended form of the original architecture proposal. As can be seen, neither the controller nor any micro-service needs to access the workspace directly.

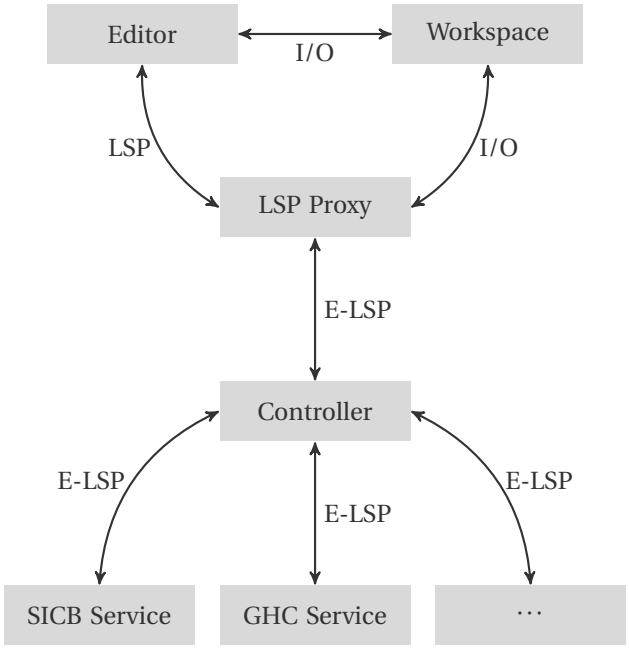


Figure 4.1: HIDE Architecture Overview

4.1 Project Structure

The implementation is structured as a stack project with multiple packages. This approach should provide modularity and code reuse. There are two libraries that contain code which is common between multiple packages. For these packages a naming convention was used where 'common' is appended to the greatest common prefix of the packages, that use it. All packages that are specifically part of the HIDE implementation are prefixed with 'hide'. The 'lsp' package is the only one, which does not have a 'hide' prefix, and its module hierarchy also does not start with 'HIDE'. The 'lsp' package ought to be a generic implementation of the LSP protocol. It, however, also contains data types for messages which are only contained in E-LSP but not LSP.

Each package has a 'src' directory which contains the source code of the library and a 'test' directory which contains the tests. Some packages also have an 'app' directory which contains the source code of the executable. No package has more than one executable target.

The 'clients' directory contains a directory with the client side code of every supported client. At the moment this is only VS Code.

4.2 Libraries

There is a subset of Haskell libraries which is used by a large number of projects.[3] These libraries — namely: *async, bytestring, containers, directory, mtl, monad-control, stm, text, transformers, filepath, process, network-uri, unix and Win32* — are not discussed here. The *lifted-base* library is used for exception handling in some places. The *lifted-async* library is used for concurrency.

[1] See chapter 3

[2] Visual Studio Code (VS Code) is an open source code editor that was developed by Microsoft.

[3] The JSON-RPC 2.0 terminology is used here.[6]

Both have their functions lifted from the IO monad to *MonadBaseControl IO m*.

The *aeson* library is used for JSON serialization and deserialization. *Aeson* is primarily used by the *Data.LSP.Types* module in the 'lsp' package. The *FromJSON* and *ToJSON* instances are, in most cases, generated during compile time by Template Haskell.

```
data Range = Range
    { rangeStart :: Position
    , rangeEnd   :: Position
    } deriving (Eq, Show)

$(deriveJSON defaultOptions
    {fieldLabelModifier = dropPrefix 5} ''Range)
```

Listing 4.1: Aeson Example (Data.LSP.Types)

Yaml, which is based on *aeson* is used for parsing the configuration files. Template Haskell is used to read the default configuration during compile time.

Most strings are represented using *text*. Some libraries also expect *ByteString* representations. *ByteString* is elsewhere used for reading, writing and processing binary data. File paths are represented using the *FilePath* type. *FilePath* is an alias for *String*. It is widely used by libraries such as *filepath*, *directory* and *process*. For parsing and processing URIs, the *network-uri* library is used. For encoding and decoding binary data from *ByteString* to *Text*, the *sandi* library is used. *Sandi* supports Base85 (a.k.a. Ascii85) which is a binary data exchange format. Base85 is more compact than Base64, but still fits into the printable range of ASCII. It can therefore be transferred as a JSON string without difficulty, since JSON supports UTF-8. The *utf8-string* and *bytestring-conversion* packages are used for string conversions.

For regular expression matching, the *text-icu* library is used. This library is based on the International Components for Unicode (ICU) libraries and provides regular expression matching support for unicode strings. In some places the *regex-base* and *regex-posix* libraries are used. The usage of those libraries should be replaced by *text-icu*.

The *conduit* and *conduit-combinator* libraries are used in most components of HIDE to represent the message flow in and between the components.

Monad Logger is used for logging.

For the communication between the components, *json-rpc* is used. This library is not available through stackage and therefore needs to be listed in the 'extra-deps' section of the 'stack.yaml'-file. *Json-rpc* is a simple implementation of the JSON-RPC protocol.

4.3 Proxy

The proxy component extends the LSP with the functionality needed to implement the proposed architecture. To achieve this, the proxy must be able to directly respond to the editor and to the IDE. This allows it to transform the message flows in any possible way. Messages can be dropped, answered directly or transformed and redirected.

The default behaviour of the proxy is to redirect all messages unchanged. It however may extract some information before redirecting a message to synchronize its state with the editor.

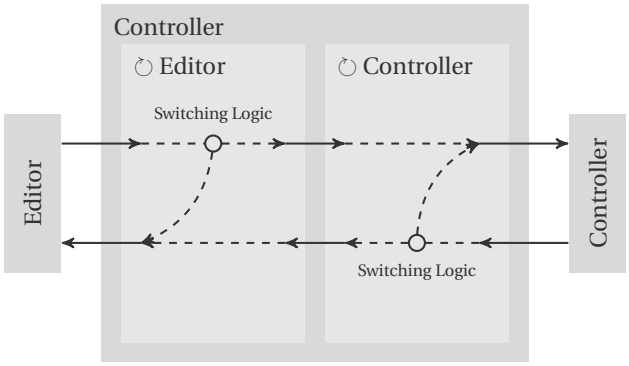


Figure 4.2: Proxy Overview

Figure 4.2 shows how the messages flow through the proxy. The two switches (labeled as *Switching Logic* in Figure 4.2) allow the proxy to do the aforementioned transformations on the message flows. Each switch has its own switching logic which decides how the messages are processed.

The 'HIDE.Common.Conduit' module implements a conduit based switch with its 'conduitSwitch' function. This switch implementation is sequential. This means that the switching logic must yield a result for a message before the next one can be processed. The LSP proxy solves this problem by using a non-blocking switch logic.

At first the type signature[4] of the switch logic was:

$$\text{Monad } m \Rightarrow a \rightarrow m \text{ (Either } b\ c)$$

But this did not support non-blocking switch logics. Because of that the type signature of the switch logic now is as follows:

$$\text{Monad } m \Rightarrow (\text{Either } b\ c \rightarrow m\ ()) \rightarrow a \rightarrow m\ ()$$

The first argument is a callback which redirects the message. The callback redirects the *Left* values to the first (left in Figure 4.2) producing conduit of the switch and *Right* values to the second (right in Figure 4.2) one.

Currently the proxy provides a way for the IDE to retrieve all files from the workspace which match a specific filter. Filters are specified as whitelist patterns. Patterns may contain wildcards as specified in Table 4.1.

Wildcard	Meaning
*	Matches any characters except '/'
**	Matches multiple directories

Table 4.1: File Path Pattern Wildcards

As mentioned before, binary files are encoded using Base85. Binary files are files which cannot be successfully decoded using UTF-8. This is necessary so that all file contents are encoded as valid JSON.

[4] The type constraints are simplified for the purpose of brevity.

4.4 Controller

The controller is implemented by using two subcomponents. The hub component (HIDE.Controller.Hub) and the service component (HIDE.Controller.Service). The former is responsible for routing messages to the right services or the editor. The latter is responsible for the management of service processes.

The hub subcomponent has no knowledge of services. It instead sees the services as *plugs*. Plugs are endpoints of bidirectional message flows. They consist of a source, a sink and a message filter. Plugs are defined as follows:

```
module HIDE.Controller.Types where

import           Data.Conduit (ConduitM)
import           Data.Void

data BidirectionalConduit i o m = BidirectionalConduit
    { source :: ConduitM () o m ()
    , sink   :: ConduitM i Void m ()
    }

data Plug p t m = Plug
    { conduit      :: BidirectionalConduit p p m
    , plugPFilter :: t -> Bool
    }
```

Listing 4.2: Definition of Plug (HIDE.Controller.Types)

The 'plugPFilter' predicate is true when the 'conduit' supports messages with the given tag. The *p* type-variable represents the message type and the *t* type-variable the tag type. In order to run a 'HubT' computation one must provide a 'TagSelector *p t*' which is a function from any message *p* to its tag *t*.

The main advantage of the plug abstraction is, that the hub does not need to make any distinction between editor and services.

A JSON-RPC endpoint may never receive two requests with the same id. As long as the communication takes place between two parties, the json-rpc library takes care of this. In the case of the controller, however, the hub must ensure the uniqueness of ids. Otherwise a situation could occur, where services A and B each send a request with the same id to service C. This yields two problems: service C would not know which of the two requests is valid or if both of them are. If service C answers one of the requests the hub would not know towards which service that response is directed. To solve this problem, each request is assigned an id which is unique on the hub. The hub memorizes which original id belongs to which hub id. When the hub receives a response, it looks up the original id of the corresponding request, replaces the unique id with the original id and redirects the response to the originator of the request.

The service management subcomponent provides a simple interface to launch services. For each service a service control and a plug is constructed. The purpose and structure of plugs is explained above. A service control allows the user (in this case the controller) to start, stop, restart and dispose services. Using this interface, the lifecycle of each service can be easily managed.

The controller reads the configuration using 'hideconf'. The configuration contains a list of all services, their supported messages, their parameters and the transport protocol/medium that is to be used.

```
data ServiceControl m = ServiceControl
    { start   :: m ()
    , stop    :: m ()
    , restart :: m ()
    , dispose :: m ()
    }
```

Listing 4.3: Service Control (HIDE.Controller.Service)

4.5 SICB Service

The SICB service initializes by requesting the workspace tree as described above. As soon as it receives a valid response, it replicates the directory structure and files it has received in a temporary directory.

The store directory contains a directory for each SICB process with the PID of that process as its name. The process directory in turn contains one directory for each revision that is alive (has references to it).

In order to avoid excessive disk usage and to cut down on I/O time, files are not copied for every new revision if they did not change. Instead they are hardlinked. This means that the filesystem must support hardlinks, since no fallback is implemented yet. Because of this, Fat32 would not work, for instance.

4.6 Compiler Service

The HIDE compiler service uses the GHC library to compile single files. With some minor improvements it would also work with multi-module projects.

In this prototype the modules are compiled using the 'load' function. The 'load' function compiles all modules, which are currently targeted, and their dependencies. In order to retrieve the necessary syntactic and semantic information for other services, the 'parseModule' and 'typecheckModule' functions should be used directly.

GHC has its own logging infrastructure. Unfortunately, there is no clean and easy way to extract the errors, warnings and other log messages from GHC. In this prototype this was solved by specifying a custom *log_action* in the *DynFlags* data structure. *DynFlags* holds all the configuration options for GHC. The type of *log_action* is specified as follows.

```
type LogAction = DynFlags
               → WarnReason
               → Severity
               → SrcSpan
               → PprStyle
               → MsgDoc
               → IO ()
```

Since the function yields a computation in the IO monad, there is no clean and easy way of adding context. But it

can be worked around by using 'IORef'.

With LSP, errors and warnings are reported as lists of diagnostics. Multiple sets of diagnostics for the same file override each other. This could be a problem when other services than the compiler service need to report diagnostics. This issue could be resolved by extending the compiler service. All diagnostics need to be aggregated by the compiler service. This means that other services need to report their diagnostics to the compiler service and never directly to the editor.

4.6.1 Rationale for using the GHC API

There are other options for implementing IDE features for Haskell than using the GHC API directly. Tools like *hlint* and *stylish-haskell* use *ghc-src-exts*, for instance.

One problem of parsers and typecheckers, that are external to the compiler, is compatibility. There is no external parser which supports all language extensions the latest GHC does. Even if there was such a parser, there would not be one for every GHC version there ever was. This might lead to the IDE displaying results, which are not equivalent to the ones the compiler gives.

Such a behaviour conflicts with NFR-05 (no clutter) in that it distracts. The IDE should never get in the way of the developer, so that he never needs to consider to change his code for the IDE.

Real world Haskell bears no resemblance to the Haskell, which is specified by the Haskell language report[9]. Instead it continuously evolves with the development of GHC.

4.6.2 Implications of using the GHC API

Using the GHC API means that all required compiler options must be passed to the API. This is more of a challenge than it seems at the first glance. More often than not, GHC is called by tools such as Stack and Cabal. To accurately reflect the result of standalone compilation, the IDE therefore needs to retrieve the excact arguments, Stack or Cabal would use.

Furthermore, in order to ensure the best possible compatibility and by extension user experience, the compiler service must be dynamically recompiled, whenever the compiler version needs to be changed because the Stack configuration changed in such a way or the natively installed compiler changed.

Solving these issues is no small feat. It probably requires to rewrite some parts of Stack and Cabal while coping with GHC API changes and a lot of edge cases, such as GHCJS.

4.7 Shortcomings

HIDE falls short on some points. Before its code could ever be used in practice, these have to be addressed. None of the shortcomings, which are outlined below, are due to the general architecture outlined in chapter 3.

Watch for file changes The proxy needs to watch the workspace for changed files and directories. Currently the editor and IDE states lose sync if the workspace is changed by any tool that is not the editor.

Rewrite URIs and paths With the current implementation, messages are directly passed from the editor to the micro-services. This breaks the isolation of the services from the workspace. It also makes it harder to implement services correctly, because the service developer needs to convert all URIs and Paths in the service code.

This problem should be solved by rewriting all URIs and paths at the proxy level, to generic paths which abstact from the specific workspace location.

Handle service crashes correctly If any service crashes, it needs to be automatically restarted by the controller. The *ServiceControl* data structure should already contain most of the functionality which is needed to implement this.

Write protect store Revisions, which are commited, should be write protected so that services can not interfere with one another.

Handle reference timeout SICB does not kill misbehaving services. Services misbehave, when they do not release their code base references within a certain time frame. This may lead to space leaks.

Make SICB more robust The SICB service can get stuck in a erroneous state. With the current implementation this can occur when changes are not received in order. This is a bug, because JSON-RPC does not garantee that messages arrive in order. The problem is not limited to the order of updates, but can also occur in the case of a timeout or an I/O exception.

If SICB is in such a state, it must be reinitialized.

Use SPECIALIZE for 'sendNotif' In several places 'sendNotif' is defined as 'sendRequest' to specialize its type signature. This can probably be solved globally by using the SPECIALIZE pragma.

Add Windows support Most of the code is plattform independent. But there are some problems.

The first step is to figure out how HIDE can be compiled on Windows. It needs the ICU library for *text-icu* to work.

Ensure that all the paths are handled correctly. This should be done after implementing, the rewriting of URIs and paths. Because that solves most of the issues with drive letters etc.

Chapter 5

Conclusion

IDE support for Haskell needs improvement. This was found by a study by FPComplete in 2015[2] and was confirmed by interviews with three students and four professional Haskell developers.

Several existing IDEs and other tools for functional programming were examined and compared with the interview results. From this comparison a list of functional and non-functional requirements was composed. Most of the deficiencies of existing Haskell IDEs are of the non-functional kind.

Most of the existing Haskell IDEs are built on top of GHCi. GHCid and Intero are two of the most commonly used Haskell IDEs and fall under this category. This approach has two fundamental problems.

First, it does not scale well in relation to the code base size. This goes especially for multi-package projects. When code from multiple packages is edited at the same time, there has to be one IDE instance per package.

Second, IDEs using such an approach need to be restarted quite frequently. If one changes module B on which module A depends, the IDE instance for module A needs to be restarted, so that the changes made to B are applied. There are other such cases. If a dependency is added to the Cabal-file or if any change is made to the stack file, the IDE needs to be restarted as well.

To solve these problems, a new architecture is proposed. This architecture is based on the results of the IDE comparison mentioned above and the requirements.

The proposed architecture relies on micro-services to provide the actual functionality. This makes the architecture very extensible.

As a proof of concept, a Haskell implementation of the architecture is provided. During the implementation, multiple issues arose but none of them is directly related to the proposed architecture. They might, however, be the reasons why IDE support for Haskell is lacking. Unlike many of the mainstream programming languages, Haskell is not developed by a standardization body. Instead all compilers develop their own extensions to the last language report from 2010[9]. Other projects which provide Haskell parsers, struggle to keep up with new language extensions. The GHC API itself is very young and also changes very frequently.

Future efforts should be concentrated on the most active Haskell IDE project there is, which does not pursue a fundamentally flawed approach. At the moment, this would be the Haskell IDE engine (HIE). That project represents the joint effort and experience of people who previously worked on other Haskell IDE projects.

As soon as a solid foundation for an IDE exists, the other tools, which rely on different parser backends, need to be ported over to that infrastructure.

Acknowledgements

I wish to express my sincere thanks to Farhad Mehta, for supporting this project from its inception. I also thank Brett Davidson for his invaluable advice on structuring this paper, for reviewing and for his general advice on language.

I am also grateful to Matteo Patisso, Davide De Giorgio, David Loosli, Simon Meier, Jasper van der Jeugt, Nicolas Gagliani and Joris Morger, for taking part in the interviews.

I also thank David Loosli and Claudia Saxer for reviewing this report, Simon Meier for his great advice and for agreeing to be co-examiner, Mirko Stocker for his advice and for being proofreader and Mario Meili for his advice.

I want to thank my parents for supporting me and to Florian Bitterlin for helping me with the interviews and the early stages of the project, as well as providing some of his infrastructure for me. Finally I thank Jan Aeberli and Jürg Schleutermann for their encouragement and help with debugging.

Appendix A

Interview Protocols

A.1 Interview with Matteo Patisso and Davide De Giorgio

Patisso and De Giorgio started out programming in various Languages (C, VB ṄET), but now they primarily use Java (with Eclipse or InetlliJ IDEA) and C# with Visual Studio.

Matteo once needed to program in PHP in a textarea in the browser. The thing he disliked the most about that experience was that he didn't even have syntax highlighting. On the other hand he often doesn't use most of the utilities provided by the IDEs he uses.

They both used either Sublime or Atom to perform small programming or scripting tasks and to program in Haskell for school assignments. For programming in Haskell they used a terminal window running GHCi beside their editor. In GHCi they used the ':load' command to recompile their source files.

While learning to use Haskell most of the problems they had were due to the editor not giving them any information about the syntax errors they made and because of the cryptic errors spit out by GHCi. Other than that both found it difficult to switch from loops to recursions. A visualization of the execution of a recursion would help them, they said. Manually reloading their code after every change slowed their development process down.

Their favourite features of their current IDEs are:

- Refactoring (especially renaming)
- Automatically import module dependencies
- Git integration
- Integration of build systems

The features they'd like to have in their Haskell IDE are:

- Syntax highlighting
- Inline highlighting of errors
- Auto completion of symbols
- Visualisation of recursions
- Automatically detect where foldr, foldl, fmap, etc. could be used instead of explicit recursion
- Typical IDE layout (file browser, code and console)
- Integration for testing
- Translation from imperative constructs to similar functional code (for beginners)
- Debugging support
- Type introspection

A.2 Interview with David Loosli

Loosli programmed in various different programming languages with different paradigms. He used C, C++ and ASM for low level programming and Java, C#, Prolog and Haskell for high level programming. As an IDE he primarily uses Eclipse IDE. Other than that he sometimes uses Emacs, other Editors and XCode. For programming in Haskell he used XCode (without syntax highlighting or any assistance at all) and a terminal window running GHCi.

Switching between XCode, the terminal and a browser window, which he needed to read API documentation, got especially cumbersome on his notebook. At home he just used multiple screens to work around this issue. However the feature he missed the most while programming in Haskell was syntax highlighting.

In his opinion needing to load ones code manually in GHCi is educational. If he however needed to work on a large project, he said, he would prefer automatic recompilation and reloading.

He often needed to look up the type signatures and descriptions of functions in his book and on the internet.

When asked what he'd like to see in a Haskell IDE, he said, that it should be very easy to install. The installation process should be as easy as installing the haskell tool stack, the IDE and the language plugin and may not require more than these three steps.

As for features he'd also like to see basic refactoring, specifically the renaming of identifiers. Also some debugging feature would help him out.

A.3 Interview with Simon Meier

Meier works in a software development company and uses Haskell for a commercial project. He uses Vim as an editor and Intero as IDE. The project team he is a part of consists of around 30 people which maintain a Codebase of around 350 modules and 30 Cabal files. They pro-

gram for all three major platforms (OS X, Windows and GNU/Linux).

They run into problems which are not likely to be witnessed by developers who are not working with a codebase that large. This is partly due to their single repository approach. Intero doesn't share sessions among multiple packages and thus uses a lot of memory. For Meier scalability across large code bases is very important.

Meier gives a few advices on how to approach the development of a new IDE for Haskell. To allow for better extensibility, the IDE must not access the file system directly. Editor and IDE should run in separate processes.

Meier deems it useful when tests are automatically detected and run when they probably fail, because of code changes. It should be possible to enable and disable specific tests. But in the best case scenario the IDE correctly decides on its own which tests should be run.

A list of features he thinks could be useful for a Haskell IDE follows:

- Show documentation
- Display error messages
  - Error messages should be displayed in full size for improved readability
  - They should be displayed in < 0.5s of code change
- Syntax aware editing
- Auto formatting
- Type introspection

A.4 Interview with Jasper van der Jeugt

Van der Jeugt started programming with C++ and Java but switched to Haskell after his first year in university. He uses Vim to write Haskell code and also tried to use Intero but could not get it to work and therefore doesn't use it. Before Stack he used Cabal and simple make files to build his projects. To look up the documentation of packages he uses haddock and a web-browser to surf Hackage directly.

Van der Jeugt thinks the biggest problem with learning Haskell for beginners is the step between writing snippets of code, writing simple functions and writing real world software. He thinks this is probably a architectural problem of Haskell. He also thinks using the REPL to try new libraries and algorithms is inconvenient.

Van der Jeugt works as part of a team of four people on a compiler for a domain specific language (DSL). In his team coding style is up to the developers. If someone works on the module of another developer, they try to imitate the style of that developer.

A list of Haskell IDE features he deems useful:

- Display online documentation, search documentation (by codomains of functions)
- Auto completion
- Breakpoints for debugging is nice to have

Van der Jeugt also mentioned that he only tries to prove his code if he thinks he could be wrong. He also said, that he doesn't need an integrated testing environment like Eclipse or Visual Studio offer for unit testing.

A.5 Interview with Nicolas Gagliani and Joris Morger

In the past Gagliani and Morger used a big variety of different programming languages. They worked with C, C#, Java, PHP, JS and Ruby and also expermiented with other more exotic languages. For developing in Java they used Netbeans and Eclipse. For C# they used Visual Studio which they liked a lot.

Both program in Haskell professionally and both use VS Code to do so. The ghc-mod plugin for VS Code does not seem to work well. Nicolas also tried to use Haskero, but that does not work with his current project. One of the problems of intero is, that if one's working on projects with multiple subprojects an Intero session needs to be maintained for each subproject. Hlint in VS Code seems to work quite good.

Their biggest grievance with Haskell IDE plugins is, that most of them don't work out of the box, they are difficult to install and are very unstable.

The following is a list of features they'd like to see in their Haskell IDE:

- Display types on hover (high priority; type of symbol under cursor or type of selected expression)
- Jump to documentation of symbol (and local hoogle)
- Auto-format code (optionally with custom rules)
  - Fix stylish haskell or hindent
  - Uniformity is more important than custom style
- Refactoring (nice to have; type on hover more important)
- Automatic organisation of imports
- Profiler output in IDE window
- Interactive scratch buffer with visualization (see iPython, swift playground, etc.)
- Better error message when package is missing in Cabal-file

Gagliani also mentioned that IDE support is a major concern for their project, because they find it difficult to persuade the Java developers in the company to work on the Haskell code, because there is no proper IDE support.

Appendix B

Task Description

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

Task Description for Bachelor Thesis

Modern IDE Support for Functional Programming

Spring Semester 2017

1 Supervisor

Prof. Dr. Farhad Mehta, HSR Rapperswil

2 Students

- Cyrill Schenkel <cschenke@hsr.ch>
- Florian Bitterlin <florian.bitterlin@hsr.ch>

3 Background

The functional programming (FP) paradigm offers a number of advantages over the imperative object-oriented style of programming that is currently mainstream. Although concepts originating from the functional programming world are increasingly finding their way into mainstream languages such as Java and C#, the use of functional programming languages as such in industry (i.e. outside some niche areas and academia) has been sluggish. It is assumed that one significant reason for this sluggish adoption of functional programming languages is the current lack of tools available to the programmer to make the job of programming in functional languages more efficient and convenient. Integrated development environments (IDEs) for functional languages do not currently offer the same degree of aid as compared to their imperative object-oriented counterparts. Furthermore, the concepts currently used within IDEs to aid the programmer have been built with the imperative object-oriented style of programming in mind. It is therefore unclear if a direct mapping of existing IDE concepts is as effective in the FP setting.

4 Goal

The main goal of this thesis is to *envisage* and *realise* improved modern IDE support for functional programming and thereby work towards a more widespread use of functional programming in industry.

The resulting improvements are to be relevant to the beginner learning how to program in the functional style, as well as the expert working on large and complex projects. In cases where this is not possible, priority will be given to features that aid the beginner over features that aid the expert.

The functional programming language Haskell and programming in Haskell will be used as the concrete object of study in this thesis. Where possible, generalisations to other functional programming languages will be made.

5 Tasks

The following tasks have currently been planned in order to achieve the goals of this thesis. The tasks are organised into two stages of work. In the first (conceptional) stage, the possibilities and requirements for the target IDE will be explored. This includes the study of the currently used development process in Haskell, as well as the identification of inefficiencies and difficulties during this process. In the second (realisation) stage, a selected number of proposed improvements will be implemented within a modern IDE and released.

5.1 Conceptual Stage

Experienced Haskell developers as well as beginners in the language will be consulted in order to identify and analyse the current perceived difficulties encountered during program development in Haskell. These consultations will be used to arrive at requirements for the proposed IDE features. The requirements will be prioritised by their estimated effectiveness.

With this basis, possible solutions for satisfying the proposed requirements will be explored. An estimated implementation cost will be assigned to each proposed solution.

The result of the conceptual stage will be a list of proposed solutions, prioritised using their estimated effectiveness and implementation cost.

5.2 Implementation Stage

The previously envisaged solutions will be implemented, taking into account their priority as stated in subsection 5.1.

The implementation approach (IDE/Editor plugin, language server[4], etc.) will be evaluated before starting implementation. The *language server* approach is currently very promising since it is already supported by several IDEs (VS Code, Eclipse Che and Eclipse IDE) and support can easily be implemented in other IDEs and editors.

6 Deliverables

The following artefacts will be delivered as part of this thesis. All documentation will be delivered as LaTeX code and PDF. All software will be delivered as buildable source code and binaries.

6.0.1 Project Plan

The project must be planned before the work on any other deliverables than the project plan begins. The code quality criterions and the time plan will agreed in advance.

6.0.2 Analysis and Solution Concept

This section of the report shall consist of the following:

- Analysis of the state of the art of Haskell IDEs.
- Description of the development workflows of different Haskell users.
- Well defined, structured and prioritised SMART goals and requirements in the form of use cases.
- Descriptions of the proposed solutions (implementation concepts), traceable to the aforementioned requirements. The implementation concepts must consist of an appropriately detailed description and a justified estimate of implementation cost (complexity).
- The proposed solutions, with assigned priorities.

6.0.3 Implementation of Solutions

This deliverable consists of the following four artefacts:

Source Code of Solutions The actual implementation of the selected solutions. A predefined standard of code quality will be maintained for all the code.

Developer Documentation All public APIs as well as the architecture of the solution shall be documented with the goal of allowing easy contributions from developers who are not familiar with the code base of the project. The amount of documentation must be appropriate for this purpose.

User Documentation This part of the documentation is aimed at the users of the solution. It shall be as precise, complete and concise as possible.

Implementation Report There will be a section in the report explaining the implementation difficulties and decisions which need further elaboration.

6.0.4 Conclusion

The conclusion shall contain the following contents:

- What has been accomplished? Which features were implemented?
- What conclusions can be drawn for future projects in this domain? (lessons learned)
- What are the possibilities for future projects in this domain?

7 Licensing of Deliverables

The chosen licenses are meant to lower the hurdle for contributions and follow up projects. Additionally they enforce sharing of the works based upon the results of this thesis and therefore emphasise a collaborative project culture. The intention behind this choice is to encourage distribution and provide developers with full freedom over their tools, as long as they share this intention. Experience shows that this is very important for a tool to become accepted by developers.

Based on the aforementioned premises the one following licenses will be used for every deliverable.

- All documentation will be licensed under *CC BY-SA 4.0.* [1]
- All code will be licensed under *GNU GPLv3.* [2]

8 Important Dates

Start of Project: 20.02.2017
Submission of Final Report: 16.06.2017

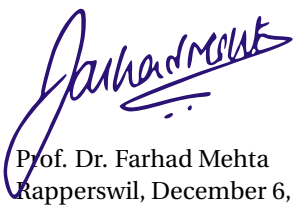
9 Guidelines

The students and the supervisor will plan weekly meetings to check and discuss progress. All meetings are to be prepared by the students with an agenda. The agenda will be sent at least 24h prior to the meeting. The results will be documented in meeting minutes that will be sent to the supervisor.

A project plan must be developed at the beginning of the thesis to promote continuous and visible work progress. For every milestone defined in the project plan, the temporary versions of all artefacts need to be submitted. The students will receive a provisional feedback for the submitted milestone results. The definitive grading is however only based on the final results of the formally submitted report.

A successful Bachelor thesis project results in 12 ECTS credit points per student. One ECTS point corresponds to a work effort of 30 hours. All time spent on the project must be recorded and documented.

All additional guidelines, deadlines and formal requirements for bachelor theses at HSR [3] must be met.



Prof. Dr. Farhad Mehta
Rapperswil, December 6, 2016

References

[1] Creative Commons. *Attribution-ShareAlike 4.0 International*. URL: https://creativecommons.org/licenses/by-sa/4.0/ (visited on 10/16/2016).

[2] FSF. *GNU General Public License*. June 29, 2007. URL: https://www.gnu.org/licenses/gpl-3.0.en.html (visited on 10/16/2016).

[3] Claudia Furrer. *Abläufe und Regelungen Studien- und Bachelorarbeiten im Studiengang Informatik*. Sept. 14, 2016.

[4] Microsoft. *Language Server Protocol*. URL: https://github.com/Microsoft/language-server-protocol (visited on 10/16/2016).

List of Figures

Listings

List of Tables

Bibliography

[1] Mathieu Boespflug. *Why is stack not cabal?* June 2015. URL: https://www.fpcomplete.com/blog/2015/06/why-is-stack-not-cabal (visited on 05/24/2017).

[2] Aaron Contorer. *What do Haskellers want? Over a thousand tell us.* May 2015. URL: https://www.fpcomplete.com/blog/2015/05/thousand-user-haskell-survey (visited on 06/01/2017).

[3] Stephen Diehl. *What I Wish I Knew When Learning Haskell 2.3.* Mar. 2016. URL: http://dev.stephendiehl.com/hask/#what-should-be-in-base (visited on 06/13/2017).

[4] Chris Done et al. *intero.* URL: https://github.com/commercialhaskell/intero (visited on 05/19/2017).

[5] Iulian Dragos et al. *Scala IDE for Eclipse.* URL: http://scala-ide.org/ (visited on 05/19/2017).

[6] JSON-RPC Working Group, ed. *JSON-RPC 2.0 Specification.* Jan. 2013. URL: http://www.jsonrpc.org/specification (visited on 06/10/2017).

[7] Isaac Jones et al. *The Haskell Cabal, A Common Architecture for Building Applications and Tools.* Aug. 2004. URL: https://www.haskell.org/cabal/proposal/pkg-spec.pdf (visited on 05/24/2017).

[8] Kodowa, Inc. *Light Table.* URL: http://lighttable.com/ (visited on 05/19/2017).

[9] Simon Marlow, ed. *Haskell 2010 Language Report.* July 2010. URL: https://www.haskell.org/onlinereport/haskell2010/ (visited on 06/02/2017).

[10] Eric Marsden, Luke Gorrie, Helmut Eller, et al. *SLIME: The Superior Lisp Interaction Mode for Emacs.* URL: https://common-lisp.net/project/slime/ (visited on 05/19/2017).

[11] Microsoft, ed. *Language Server Protocol.* Feb. 2017. URL: https://github.com/Microsoft/language-server-protocol/blob/master/protocol.md (visited on 06/05/2017).

[12] Neil Mitchell and Niklas Hambüchen. *ghc-make.* URL: https://github.com/ndmitchell/ghc-make (visited on 05/25/2017).

[13] Neil Mitchell, JP Moresmau, et al. *ghcid.* URL: https://github.com/ndmitchell/ghcid (visited on 05/19/2017).

[14] Christophe Rhodes. *swankr: SWANK (and SLIME) for R.* 2014. URL: https://common-lisp.net/~crhodes/swankr/ (visited on 05/19/2017).

[15] Ivan Shvedunov et al. *swank-js.* URL: https://github.com/swank-js/swank-js (visited on 05/19/2017).

[16] Unknown. *The Superior Lisp Interaction Mode for Emacs.* 2003. URL: https://github.com/slime/slime (visited on 05/19/2017).

[17] The Glasgow Haskell Compiler Wiki. *Deterministic Builds.* 2016. URL: https://ghc.haskell.org/trac/ghc/wiki/DeterministicBuilds (visited on 05/25/2017).

[18] The Glasgow Haskell Compiler Wiki. *Shaking up GHC.* 2015. URL: https://ghc.haskell.org/trac/ghc/wiki/Building/Shake (visited on 05/25/2017).

[19] Alan Zimmerman, Herbert Valerio Riedel, et al. *GHCi.UI.Info.* June 2017. URL: https://github.com/ghc/ghc/blob/ghc-8.2/ghc/GHCi/UI/Info.hs#L298 (visited on 06/08/2017).