

Experience Report: Building an Eclipse-based IDE for Haskell

Leif Frenzel

himself@leiffrenzel.de

Abstract

This paper summarizes experiences from an open source project that builds a free Haskell IDE based on Eclipse (an open source IDE platform). Eclipse is extensible and has proved to be a good basis for IDEs for several programming languages. Difficulties arise mainly because it is written in Java and requires extensions to be written in Java. This made it hard to reuse existing development tools implemented in Haskell, and turned out to be a considerable obstacle to finding contributors. Several approaches to resolve these issues are described and their advantages and disadvantages discussed.

1. Introduction

In April 2004, I started an open source project (EclipseFP 2007) to build a free Haskell IDE based on Eclipse (Eclipse 2007).¹ For a project that is based on volunteer work, reuse of existing functionality is vital. The goal was from the beginning to mostly *integrate* existing development tools (compilers, interpreters, refactorer etc.) on top of a suitable IDE platform. I chose Eclipse, primarily because I was already familiar with its concepts and APIs. There are, however, a number of further reasons that make Eclipse an obvious candidate.²

Eclipse is free, developed by an open source project, has a lively community and is backed by large software companies who are members of the Eclipse Foundation and thus committed to supporting maintenance and further development. It has well-defined APIs, and modularity and extensibility are built right into its core. Moreover, language-neutral functionality and language-specific functionality are separated by design, so that an IDE can be built upon the language-neutral layer (the ‘Eclipse Platform’) without need to keep any specific support for other languages (such as Java). There is also a large number of reusable UI components.

¹ Since then, several developers have joined. The project is hosted at sourceforge.net and has now 5 developers; the development mailing list has currently 47 subscribers. I will use the first person plural when referring to our work as a group; I use the first person singular only for some initial decisions made by me at the very beginning of the project.

² A number of other projects exist which implement Haskell IDEs, some from scratch, some based on other platforms than Eclipse. Since this is an experience report focused on a particular project, none of these are discussed here.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICFP’07, October 1–3, 2007, Freiburg, Germany.

Copyright © ACM, 2007 This is the author’s version of the work. It is posted here by permission of ACM for your personal use. The definitive version was published in Proceedings of the 2007 ACM SIGPLAN international conference on Functional programming, Freiburg, Germany, <http://doi.acm.org/10.1145/1291151.1291186>

Extending Eclipse works by implementing Java classes (conforming to interfaces specified by the Eclipse APIs) and declaring them in an XML manifest file. There is no support for extending Eclipse in other languages than Java.³ While this poses no problem for communities that want to extend Eclipse with support for Java-related development tools (e.g. the popular CheckStyle code auditing tool), there is a tension in the case of communities around general-purpose programming languages, such as Haskell. This tension has both technical and social aspects: existing tools and libraries are written in the target language - not in Java; and similarly, potential contributors are more likely to be interested in (and competent in) the target language - not in Java.

2. Integrating development tools

Modern IDEs include source code editors that go far beyond mere syntax coloring and context-independent code completion. For example, Eclipse users are used to have an *Outline View* available for their source files, where top-level elements are presented; it can be used to quickly navigate to the location of these language elements in the source code. Code portions such as comments or blocks can be *folded*, i.e. collapsed to a single line in the editor; this helps to focus the user’s attention. With the text cursor located over a language element (e.g. a function name), the IDE *highlights other locations* in the source code where the same language element appears, and it provides context options to *navigate directly to its declaration*, or to initiate an *automated renaming refactoring*. To be able to provide such functionality, the IDE must be language-aware; in particular, it must analyze the source text to a certain degree (which goes beyond mere text matching). Thus it needs to be able to parse syntax trees, infer types etc. (Performance often matters in this context, since editors must react quickly to changes that result from the user’s typing.) The Haskell community has already built the necessary tools and libraries to do this, but of course these are written in Haskell. For an Eclipse-based Haskell IDE thus the question arises how to integrate these tools and make their functionality available to the Plug-Ins that extend the Eclipse user interface.

2.1 Execute as external program

Some tools can be integrated by requiring users to install them on their system independently of the IDE, and then have the IDE call the executable of the tool, passing necessary information as command line arguments. A typical example would be a documentation generator, such as Haddock (Marlow 2007). The IDE just provides some user interface for options that users may want to set, and it runs the tool in an external process over the source files managed in the IDE. With this approach, all of the integration code is writ-

³ Although the possibility has been extensively discussed at the Eclipse Languages Symposium in 2005, and at several informal meetings during the annual Eclipse developer conferences, no such support has been built into the Eclipse Platform so far, and there are no immediate plans to do so.

ten in Java.⁴ There is no need for the two languages to integrate directly.

This approach does not help in all cases, however. More often, a deeper integration with language-aware tools is needed for the IDE to build its user interface presentation. This situation arises when implementing many coding aids such as those described in the previous section. Calling an external program is not an option in such cases. This is not only because of performance requirements, but also because there is much more, and more structured, information that needs to be marshalled (e.g. ASTs), compared with the case of integrating tools such as Haddock, where the necessary information is restricted to a list of file names and a few option values. Furthermore, some tools, such as refactoring programs, require interaction with users and therefore have to work closer with the IDE and its user interface.⁵

There are several options for providing such a deeper integration. We have investigated and (with one exception) implemented all of them at different stages in our project.

2.2 Call native Haskell via JNI/FFI

To call compiled Haskell code (e.g. in dynamically linked libraries) makes it possible to directly use existing Haskell APIs; and additional code may be written in Haskell.

We used this approach in our first attempt to implement the source code outline and code folding features in the Haskell editor of our IDE. The Haskell parser from `Language.Haskell` in the Haskell standard libraries was used together with some additional logic to collect information from the AST, also implemented in Haskell. To call it from Java code, this was exported via Haskell's Foreign Function Interface (FFI), compiled and linked together with some C code into a dynamically linked library (dll) and then accessed from an Eclipse Plug-In via the Java Native Interface (JNI).

This enabled us to display the top-level language elements in a Haskell source file on Eclipse's *Outline View* and map them to source locations so that users could quickly navigate to them in the editor, and to compute source code regions for code folding (e.g. collapse the entire region of import declarations in the editor).

The main reason for dropping this approach was that it was not possible to make it work on Linux systems, which was important to some of our contributors and users. We were only able to create a library suitable for dynamically loading it from inside an Eclipse Plug-In on Windows operating systems.⁶

We noted some additional disadvantages when taking this option: first, it requires to manage platform-specific binaries. Although this is supported by Eclipse, compared with platform-independent Java bytecode there is an overhead for building and release engineering.

Secondly, unhandled program errors in the native code cause the entire Java Virtual Machine (JVM) to crash. This brings the entire IDE down, which is annoying for users. Moreover, since Eclipse Plug-Ins are normally written in Java, the IDE is able to handle program failures that occur in them. The behavior that users expect

⁴The actual implementation of the Haddock integration in EclipseFP comprises 23 Java source files with about 2850 lines of code and about 100 lines of auxiliary code in manifest files (plus some use of common functionality provided by other parts of the project).

⁵The expectations of Eclipse users play a role here, too. Refactoring tools in Eclipse-based IDEs provide detailed previews of the changes which are applied during the automated refactoring. There is a standard user interface in Eclipse for such situations which users have come to expect.

⁶There is now support in GHC for position independent code on PPC architectures, and there are plans to provide such support on Intel architectures in one of the next versions of the compiler. Once that is in place, this obstacle will be removed.

is that the Plug-In causing the problem is disabled (possibly writing details to the error log), but the rest of the IDE remains operating. With an error in a native library which was loaded by the JVM, this robustness cannot be warranted by Eclipse.

2.3 Re-implement in Java

This makes the language-specific functionality very easy to call from an Eclipse Plug-In, and improves independence from specific operating systems considerably.

However, the limits of this option are obvious:

- It does not allow to reuse existing Haskell APIs at all.
- Implementing all the functionality we are interested in would involve efforts (and most probably also formidable technical challenges) far beyond the means of our project.
- It means using a language that is arguably less convenient than Haskell for tasks such as analyzing and manipulating source code.
- There is a risk of lagging behind the evolution of the language itself.

Still, for lack of a better option, we have replaced the parser described in the previous section with an all-Java one that is generated using ANTLR, a parser generator (Parr 2007). Source code outline and code folding are now based on this parser, and some additional editor features (e.g. code completion for function calls) have been added.

The gravest problem with this approach is that it discourages interested contributors, who are typically either Haskell programmers or Java programmers learning Haskell - both groups want to code in Haskell, not Java.

2.4 Compile Haskell to JVM bytecode

An option that we have investigated is to implement all language-aware functionality in Haskell (and access existing Haskell APIs), and compile all Haskell code involved to Java bytecode that can be executed in the same JVM that runs the Eclipse IDE.

Although this would help with Haskell code written by ourselves, we could not use existing libraries (such as the GHC API or HaRe, the Haskell Refactorer (Thompson et al. 2007)) unless we would also recompile them and bundle them with our Eclipse Plug-Ins. Consequently there would be considerable maintenance issues. For most of the language-aware libraries it is very unlikely that they can be compiled without code modification into a Java bytecode version that runs on all platforms. But if we would have to maintain a source branch of these libraries, we would commit to a huge maintenance effort. (Even if we were able to do this, we would still be in the questionable situation of duplicating almost all of the code from these libraries.)

We have found that none of the Haskell-to-JVM-bytecode compilers (Meijer and Finne 2000; Alliet 2007) available to date is sufficiently mature to support this approach, or to support a general framework that allows to write Eclipse extensions in Haskell. However, if such a compiler would become available at some future time, we would prefer this option over those described so far.⁷

⁷An alternative we discussed is to use some language closely related to Haskell which does compile to JVM bytecode and is already stable (candidates would be the CAL language (Business Objects Labs 2007) or Jaskell (Jaskell 2007)). But it would be equally impossible to reuse much of the existing libraries then - unless we would port them to the chosen language. This seems unpractical, even for small libraries, and there would again be a huge maintenance overhead in keeping these ports current with respect to developments in their original versions.

2.5 Haskell code as Eclipse extension

All options for integrating existing development tools that have been discussed above have the additional drawback that they don't work well with Eclipse's concept of modularity and extensibility - they don't work together with Eclipse's Plug-In model. (Put differently: these options are approaches to integrate *Java* and Haskell, but we're looking for a way to integrate *Eclipse* and Haskell.)

We have recently started to develop, and experiment with, a framework that addresses this issue. Unlike the unsatisfying solutions discussed so far, which tend to concentrate the native code inside one Eclipse Plug-In, it allows us to put Haskell code into arbitrary Plug-Ins, while there is a runtime that knows how to collect, load and execute that code. This framework is fully integrated with Eclipse's plugin architecture and therefore allows structuring applications as it is usually done there. In order to load and execute Haskell code, it uses *hs-plugins* (Stewart 2007), a Haskell library that supports dynamically loading and executing Haskell code (both source code and compiled object code). This new approach is designed to provide a general mechanism to declare extensions written in Haskell in a way that fits into Eclipse's Plug-In model, but not does not impose any restrictions resulting from the integration into Eclipse on the Haskell code itself. The goal of this subproject is to find a way to enable access to Haskell code while avoiding the obstacles described in this paper.

3. Conclusion

Building an IDE for any programming language is a huge effort; to be able to build it on top of an existing platform is therefore in many cases the only road to success. The Eclipse Platform provides such a foundation, but since there is no support for extending it in other languages than Java, this complicates the task for small open source projects such as the one described here.

Without the possibility to write Eclipse extensions in Haskell, many of the advanced (and attractive) IDE features that require a language-aware user interface cannot be implemented. Apart from technical reasons (the necessary functionality is only available as Haskell code, code duplication should be avoided, etc.), there is also a problem to find contributors for the implausible task to implement a development environment for a language using a different, unrelated language. This applies both to developers with advanced knowledge and to beginners - they all have a better motivation to write Haskell code than they have to write Java code.

Yet over the recent years I have consistently received positive feedback and encouraging notes from users who are interested in a free, Eclipse-based Haskell IDE. The conclusion to draw from this should be to strive for finding a way to make it possible to use existing Haskell code (and preferably to implement all of the Haskell-specific functionality in Haskell) from within an Eclipse extension. The developments mentioned in the previous sections are intended to lead into this direction.

References

- [Alliet 2007] Alliet, Brian. LambdaVM: The Haskell to Java Translator. <http://www.cs.rit.edu/~bja8464/lambdavm/>
- [Business Objects Labs 2007] Business Objects Labs. The CAL programming language. <http://labs.businessobjects.com/cal/>
- [EclipseFP 2007] The EclipseFP Project. EclipseFP: Haskell development support in the Eclipse IDE. <http://eclipsefp.sourceforge.net>
- [Eclipse 2007] The Eclipse project. <http://eclipse.org/eclipse/>
- [Jaskell 2007] The Jaskell Project. <http://jaskell.codehaus.org/>
- [Marlow 2007] Marlow, Simon. 2007. Haddock: A Haskell Documentation Tool. <http://haskell.org/haddock/>

[Meijer and Finne 2000] Meijer, Erik, and Finne, Sigbjorn. 2000. Lambda: Haskell as a Better Java, Proc. Haskell Workshop 2000.

[Parr 2007] Parr, Terence. ANTLR: ANOther Tool for Language Recognition. <http://antlr.org/>

[Stewart 2007] Stewart, Donald Bruce. *hs-plugins: Dynamically Loaded Haskell Modules*. <http://www.cse.unsw.edu.au/~dons/hs-plugins/>

[Thompson et al. 2007] Thompson, Simon, Reinke, Claus and Li, Huiqing. 2007. HaRe: the Haskell Refactorer. <http://www.cs.kent.ac.uk/projects/refactor-fp/hare.html>