Extending Eclipse in Haskell

Leif Frenzel (himself@leiffrenzel.de)

I am the founder of an Open Source project (called eclipsefp, [2]) that develops language support for the programming language Haskell [1]. Haskell is statically typed, purely functional, with lazy evaluation. Code can be compiled and linked into a native executable on Windows, Linux and other platforms, or run in an interpreter.

I have experimented with various approaches to use Haskell as implementation language for Eclipse plugins. I'd like to exchange some experiences, point out some problems I am stuck with, and I also would like to demonstrate a working prototype of an Eclipse extension implemented in Haskell.

Motivation

Some people that are interested in participating in the eclipsefp project have asked if it would be possible to implement part of the Eclipse-based Haskell IDE in Haskell itself, because:

- the users of the IDE are Haskell programmers, not necessarily Java programmers, so contributors could be more easily found if they could work in their favorite language
- almost all tools that could be used for supporting Haskell development (Haskell parsers, tools for static code analysis, refactorer, documentation generator) are already there, but implemented in Haskell, and it should be possible to re-use them
- there are tasks where an implementation in a functional language plainly seems more apt than Java (e.g. parsing)

Approaches

In the project, we have tried out different approaches to integrate Haskell code:

1) Some tools are compiled and linked into an executable and just called as an external process (Haskell compiler, doc generator), although they have an API that would allow to call them directly from a Haskell program.

2) We are using a parser implemented in Haskell to parse the object model that fills the outline view. The parser is compiled and linked into a `.dll` and integrated via a JNI bridge [3].

We have also discussed the possibility of:

3) compiling Haskell code to Java bytecode and running it in the JVM that runs Eclipse (but there is no Haskell > Java bytecode compiler, and it is not likely that there will be one in production quality in the future)

4) running Haskell code in an interpreter (but that is slow and it would be an external process, therefore there is no real advantage over calling an executable in an external

process).

The first two worked out, the latter two did not.

Scope

Although people would like to implement some things in Haskell, they still have to learn to use PDE and they have to understand the plugin model in Eclipse, manifest files etc. It is likely that only the implementation part (i.e. what is currently done in Java) of the extensions is replaced by something in Haskell.

It would be only a set of the extensions of the Eclipse platform that are likely to be usefully implemented in Haskell, e.g. builders. Other things (e.g. UI extensions) may still be better done in Java (or at least an object-oriented language). This means that there may be plugins that contain a mix of extensions with implementations in different languages, or alternatively, sets of plugins where the functionality that is written in Haskell is separated from functionality written in Java. Since Haskell code is compiled into different binaries on different platforms, this means that also usually platform-specific fragments are involved.

Requirements for a general approach

- There must be a protocol for implementing Eclipse APIs in Haskell. E.g for a builder extension, the abstract class `IncrementalProjectBuilder` must be extended and the abstract `build()` method must be implemented. An implementation in Haskell must provide a native implementation of that method. This means there is a Java class that extends `IncrementalProjectBuilder` and implements `build()` by linking it against a native function, which is implemented in Haskell.

- There must be a protocol for calling Eclipse API methods inside Haskell code. E.g. the `build()` method gets an `IProgressMonitor` object passed on which progress state is supposed to be updated. In the Haskell code, there must be a way to call the methods of `IProgressMonitor` on this object.

- For this, it is necessary to model the Eclipse APIs in Haskell. Since Haskell has a type system too, this means some type conversion/mapping that may prove difficult. Basically, all interfaces that are referenced in the API must be modeled in Haskell code (i.e. `IProgressMonitor`, because it is a method parameter, `IResource`, `IResourceDelta` etc., because they are used in a typical builder implementation, and also static entry points like `ResourcesPlugin.getWorkspace().getRoot()`).

- There is likely to be some bridging code, and most probably a common shared framework of which all plugins that implement something in Haskell will depend. The latter is needed because Haskell code has to run in a runtime environment (that takes care of memory management, garbage collection ...). Such functionality could be put into a core Haskell runtime plugin that manages the Haskell runtime and provides libraries and interfaces to link the Haskell code against. (There has been some work on the question of loading Haskell code dynamically without losing type safety, see [4].)

From this it follows that implementors of an extension in Haskell would expect that the Eclipse APIs are accessible from the Haskell code. The provider of a core Haskell runtime plugin must therefore generate API interfaces that can be called from Haskell functions.

API interfaces should be completely generated

When Eclipse APIs change, the change should look for a client in Haskell as it looks for a client in Java. The interfaces have changed, he has to re-compile and adjust his code against the new API. To make it look like this, the core Haskell runtime plugin must be delivered with the newly generated API interfaces that reflect the new Eclipse API.

(Side note: since Haskell is statically typed, this involves a huge mass of API interface code, but it seems the process can be automated. Dynamic languages won't have to have such large libraries of API interface code, which looks like an advantage. On the other hand, API changes will probably be harder to notice, because changes in the types of symbols, i.e. parameters or functions, may not be reflected in the API interfaces in such a language; which looks like a disadvantage.)

Problem: different programming language paradigms

The trouble with the completely generated API interfaces approach is that it will not work in some cases. For example, take an object action (extension to the extension point `org.eclipse.ui.popupMenus`). Extenders must implement `IObjectActionDelegate`, which has three methods:

```
void setActivePart(IAction action, IWorkbenchPart targetPart);
void run(IAction action);
void selectionChanged(IAction action, ISelection selection);
```

The workbench calls the `run()` method when the action should be executed. Before that, it updates state on the action delegate, that is, it calls the `setActivePart()` and `setSelection()` methods. A typical implementation would store a reference to the selection in a field and use it later in the `run()` method:

```
public class MyDelegate implements IObjectActionDelegate {

  private ISelection selection;

  public void run(IAction action) {
    // usually use this.selection here to find out on which
    // objects the action was triggered
  }

  public void selectionChanged(IAction action, ISelection selection) {
    this.selection = selection;
  }
}
```

This is typical for an object-oriented language, but problematic if the API is used from a purely functional language (like Haskell). It would not make sense to have a native

implementation (in Haskell) for each of the interface methods. Rather, the state would have to be collected and passed into a single function call:

```
public class MyDelegate implements IObjectActionDelegate {

  private ISelection selection;

  public void run(IAction action) {
    performRun( action, this.selection );
  }

  public void selectionChanged(IAction action, ISelection selection) {
    this.selection = selection;
  }

  // implemented in Haskell
  private native void performRun( IAction action, ISelection selection );
}
```

But if such patterns are involved, it is much harder to generate the entire API interfaces (and it is hard to figure out automatically where such a pattern would be usefully applied, if we only know the API interface - we would need a description of a typical use).

In general, it may be difficult to connect to the Eclipse APIs where they assume typical object-oriented programming idioms which cannot be used in the extension language. Even if they could be used, they may be considered an inappropriate style in the extension language.

References

[1] http://haskell.org
The main Haskell website.

[2] http://eclipsefp.sf.net
The homepage of the eclipsefp project.

[3] http://www.cin.ufpe.br/~tbas/eclipsefp-parser.techarticle/build/article.html
An article that describes how we have integrated a parser implemented in Haskell into the eclipsefp plugins.

[4] http://www.cse.unsw.edu.au/~dons/hs-plugins/paper/
Describes an approach to build a dynamic structure out of plugins (not unlike Eclipse) out of Haskell modules, without losing type safety. A core Haskell runtime plugin would probably have to use this framework.