Title

Daniel Turner

April 2012

1 Introduction

Today, we'll be trying to use the plugins-1.5.2.1 package from Hackage to build a Haskell application which can load functionality from an external Haskell file.

We make our task relatively easy by defining a particular interface that our plugins will implement, and a default implementation.

For our purposes, we will assume we have several files:

- pluginTest.hs
- GreetAPI.hs
- Hello.hs

Each of these files has their own specific purpose. The file pluginTest.hs is the core of our application, it loads the module and evaluates the appropriate function with some input. GreetAPI.hs defines the API of the plugin, and Hello.hs is a simple plugin which we have implemented. We will go through the implementation of pluginTest.hs first.

2 pluginTest.hs

2.1 Automatically compiling a plugin

If we assume our plugins are all uncompiled Haskell files, then we need a way to compile the files to an object file which can be used.

Luckily, System.Plugins provides a make function.

```
make :: FilePath -> [Arg] -> IO MakeStatus
```

MakeStatus is nice and simple, it defines failure and success, not too disimilar to the Either monad.

```
data MakeStatus
        = MakeSuccess MakeCode FilePath    -- ^ compilation was successful
        | MakeFailure Errors               -- ^ compilation failed
        deriving (Eq,Show)
```

We can clearly use this to load an appropriate module file. We will not concern ourselves with serious error handling in this article, and simply stick to the poor practice of directly calling error. This is to keep our code as simple as possible.

```
makePlugin plugName = do
  makeStatus <- make plugName []
  case makeStatus of
    MakeFailure err -> error $ "MakeFailure: " ++ show err
    MakeSuccess code obj -> return obj
```

2.2 Loading an object file

2.3 Calling methods on a loaded plugin